Hiding the Misprediction Penalty of a Resource–Efficient High–Performance Processor

AMIT GOLANDER and SHLOMO WEISS
Tel-Aviv University

Misprediction is a major obstacle for increasing speculative out-of-order processors performance. Performance degradation depends on both the number of misprediction events and the recovery time associated with each one of them. In recent years a few checkpoint based microarchitectures have been proposed. In comparison with ROB-based processors, checkpoint processors are scalable and highly resource efficient. Unfortunately, in these proposals the misprediction recovery time is proportional to the instruction queue size.

In this paper we analyze methods to reduce the misprediction recovery time. We propose a new register file management scheme and techniques to selectively flush the instruction queue and the load store queue, and to isolate deeply pipelined execution units. The result is a novel checkpoint processor with Constant misprediction RollBack time (CRB). We further present a streamlined, cost-efficient solution, which saves complexity at the price of slightly lower performance.

Categories and Subject Descriptors: C.1 [**Processor Architectures**]

General Terms: Design, Performance

Additional Key Words and Phrases: Checkpoints, misprediction, out-of-order execution, rollback, scalable architecture

ACM Reference Format:

Golander, A. and Weiss, S. 2008. Hiding the misprediction penalty of a resource–efficient high–performance processor. ACM Trans. Architec. Code Optim. 4, 4, Article 25 (January 2008), 32 pages. DOI = 10.1145/1328195.1328201 http://doi.acm.org/10.1145/1328195.1328201

1. INTRODUCTION

Speculative execution is a key element in modern processors. Speculating on the direction and the target address of branch instructions [Lee and Smith 1984] helps reduce latencies involving control dependences. Speculating on load values [Lipasti and Shen 1996; Mutlu et al. 2005], memory dependences [Moshovos

Author's address: S. Weiss, Department of Electrical Engineering-Systems, Tel-Aviv University, Tel-Aviv, 69978, Israel.

New Paper, Not an Extension of a Conference Paper.

and Sohi 1999] and execution latency is used to overcome data dependences. Speculation has been proposed to eliminate locks in multithreaded applications [Rajwar and Goodman 2001], and for parallelizing applications in distributed shared-memory (DSM) multiprocessors [Zhang et al. 1999]. As a final example, the recent research in transactional memory [Hammond et al. 2004; Chung et al. 2006], motivated by the need to simplify parallel programming, is based on speculative execution of application-level transactions optimistically assuming [Kung and Robinson 1981] that most transactions do not interfere with each other.

Given the interest in speculative execution and the abundance of research on related topics, there is a need to develop methods for efficiently implementing speculation at the microarchitecture level. There are two well-known approaches for recovering the processor state after speculative execution following an incorrect guess: the reorder buffer [Smith and Pleszkun 1988] and checkpointing [Hwu and Patt 1987]. Although the reorder buffer (ROB) is used in most out-of-order execution microprocessors, the method does not scale well [Cristal et al. 2003]. Motivated by this problem, recent research has proposed ways to tolerate long memory latencies without scaling the instruction window [Mutlu et al. 2003], and techniques to extend the ROB [Cristal et al. 2004a; Ceze et al. 2006] or to entirely replace it by checkpointing.

Checkpoint microarchitectures are scalable [Akkary et al. 2003; Cristal et al. 2004b]. At least as important as scalability is another attribute of checkpoint microarchitectures, high resource efficiency. Resources are saved by using a register file substantially smaller than the register file in comparable, ROB-based microarchitectures, and by entirely removing the reorder buffer. Resource efficiency translates into reduced complexity and power, making checkpoint processors a valuable component in chip multiprocessors (CMP). Two approaches are followed in multicore chips [Rattner 2005; Kahle 2005]: symmetric cores [Davis et al. 2005; Spracklen and Abraham 2005; Kongetira et al. 2005] (which could be in-order, single-issue, identical processors with short pipelines) and asymmetric [Balakrishnan et al. 2005] cores. A chip multiprocessor with asymmetric cores integrates two (for example) cores of different sizes and processing power. One core could be an out-of-order superscalar machine, the other an in-order single-issue processor. Grochowski et al. [2004] provide evidence that the best approach to minimize the energy per instruction is a combination of asymmetric cores and voltage/frequency scaling. Morad et al. [2005] show that asymmetric CMPs can reduce power consumption by more than two thirds, in comparison with CMPs that achieve the same performance level using only simple processing cores. Checkpoint-based processors fit very well into the asymmetric CMP framework, they can be used to implement powerful out-of-order processing cores more efficiently and with fewer resources than comparable ROB-based cores.

1.1 Objective

Checkpointing is an important technique for speculative execution and various aspects of it have been extensively studied. Little has been done however on the topic of the penalty of recovering from mispredictions when the processor is forced to roll back to a checkpoint. The primary objective of this paper is to present a detailed study focusing on the recovery mechanism in a ROB-free microarchitecture. Specifically, the paper makes the following contribution:

(1) In current proposals the misprediction recovery time is proportional to the instruction queue size. This work presents a novel mechanism that achieves Constant RollBack time (*CRB*). CRB completely hides the OOO core recovery penalty, regardless of the instruction queue size, program flow and behavior.
(2) It further presents a simplified implementation of CRB, which settles for upper bounded (*URB*) rather than constant rollback time, but achieves almost the same performance.
(3) Alternatively, it presents improvements of the wakeup and select logic, designed to reduce rollback time without changing the register management scheme.
(4) It analyzes CRB and URB implementation cost and tradeoffs related to the number of checkpoints, the front-end pipeline depth, and the branch prediction unit. It also reports results on the impact of two key parameters, the register file size and the memory access time.

1.2 Paper Overview

The rest of the paper is organized as follows. Section 2 describes the simulation methodology. Section 3 depicts the microarchitecture and analyzes it. Section 4 suggests overlapping the front end and the OOO core rollback time. Further reducing rollback time, Section 5 discusses the bounds of the register management scheme, and Section 6 presents novel mechanisms for achieving constant misprediction rollback time (CRB). Section 7 introduces CRB performance and Section 8 analyzes CRB implementation tradeoffs. Section 9 describes a streamlined method—upper bounded rollback time (URB). Finally, related work is surveyed in Section 10 and conclusions are drawn in Section 11.

2. EXPERIMENTAL METHODOLOGY

We have modified the sim-outorder version of SimpleScalar [Burger and Austin 1997] to implement checkpoints, variable length pipelines, and variable size register files. The microarchitecture simulation parameters are listed in Table I. These parameters are mostly along the line of Intel's CPR and follow work by [Akkary et al. 2004; Srinivasan et al. 2004]. Cache and memory latencies are derived from [Mutlu et al. 2005b]. Some parameters are intentionally tuned for the following reasons:

—The instruction set architecture is Alpha rather than X86. The front-end pipeline is shorter and no uOP translation is required. The level one instruction cache is larger, as there is no trace cache.

—To better reflect a core in a multiprocessor cluster, we model a smaller register file, and a memory which takes slightly longer to access. Performance

Table I. Microarchitecture Simulation Parameters

Branch prediction unit	Tournament: 16K bimodal, 64K gshare. BTB: 512 sets, 4-way. RAS: 32 entries. Confidence estimator: 8K entries, 4bit JRS [Jacobsen et al. 1996].
Front end recovery	Read next PC after misprediction: 2 cycles Fetch + decode + rename: 6 cycles min. ($FE_{Rollback} = 2 + 6$)
Resources	8-deep in-order checkpoint buffer. Processor width: 4. 128/128 register file (integer/FP). IQ: 256. LSQ: 256. Register file and scheduler are single cycle structures.
Execution unit latencies	Integer/FP: ALU: 1/2 cycles, multiplication 3/6 cycles. Division: 20 cycles.
Caches and memory	Inst.-L1: 64KB, 512 sets, 4-way, 2 cycles, 32B blocks. Data-L1: 64KB, 512 sets, 4-way, 2 cycles, 32B blocks. Unified-L2: 1MB, 8-way, 10 cycles, 64B blocks. Memory latency is 600 cycles

sensitivity to these parameter is later analyzed, so it can be seen that a bigger register file or a closer memory would only increase our constant rollback mechanism advantage.

—A large LSQ was chosen to hold uncommitted stores. In checkpoint-based processors a store instruction commits only when the checkpoint it belongs to is released. A reduced-cost implementation (SRL – Store Redo Log), using secondary buffers without CAM and search functions, is proposed in [Gandhi et al. 2005]. Currently our simulator does not support SRL and we do not evaluate its effect on the results. An indication of the effect that can be expected is provided by [Gandhi et al. 2005], who report that SRL is within 6% of the performance of an ideal store queue.

All the SPEC CPU2000 integer (INT) and floating point (FP) benchmarks were used. Results were measured on a 300-million instruction interval, starting after one-half a billion instructions. Each result was normalized separately per benchmark. Unless otherwise stated, harmonic mean was used to summarize performance rates.

3. ROB-FREE MICROARCHITECTURE

The microarchitecture, whose issue stage [Palacharla et al. 1997] is illustrated in Figure 1, does not use a reorder buffer and accordingly, there is no need for a commit pipeline stage whose function, in a ROB-based microarchitecture, is to determine which instructions are no longer speculative. In the ROB-free microarchitecture described in this section speculative execution is supported by the use of checkpoints.

A checkpoint saves the state of a program at a certain point during the execution of the program. An instruction I_k is speculative as long as there is still a checkpoint CP, taken on I_k or on an earlier instruction I_j that precedes I_k in the program execution order. Otherwise I_k is committed. If CP_j and CP_k are two consecutive checkpoints taken on I_j and I_k respectively, and all the checkpoints prior to CP_j have been released, releasing CP_j commits all the instructions I_j $\ldots I_{k-1}$. We refer to instructions $I_j \ldots I_{k-1}$ as *belonging* to checkpoint CP_j. This

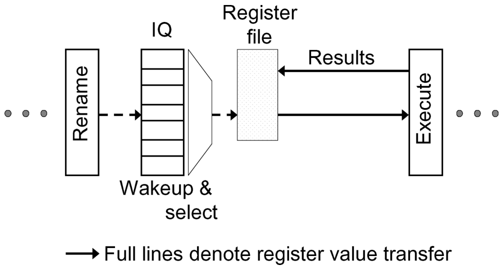


Fig. 1. The issue stage of the microarchitecture consists of an instruction queue holding renamed instructions, wakeup logic, and select logic. The wakeup logic flags instructions that are ready for execution. Ready instructions that are selected for execution read operands from the register file and proceed to the execute stage of the pipeline. The register file maintains both architectural registers (those currently pointed by the mapping table) and additional registers in a single file.

association is maintained by a tag $ChkpntTag_j$ which identifies CP_j and is part of the state of the instruction as it flows down the pipeline.

When an instruction belonging to CP_j is renamed, it increments a counter $NumInstr_j$. When that instruction finishes execution, if it is not a branch or it is a correctly predicted branch, it simply decrements $NumInstr_j$. If it is a mispredicted branch, a recovery procedure is initiated by rolling back to CP_j and restarting execution from that point. Finally, if all the instructions belonging to CP_j have finished execution without requiring any special handling, a situation reached when $NumInstr_j$ becomes again zero, CP_j can be released. A checkpoint is also released when the program rolls back to an earlier checkpoint.

Checkpoints are taken periodically according to a checkpointing policy. Assuming a free checkpoint exists, the microarchitecture will take a checkpoint at the following points in the program:

—at a branch that is predicted with a low confidence estimation level,
—at the first branch after a rollback,
—when the number of instructions after the last checkpoint exceeds a certain threshold, and
—when the number of store instructions after the last checkpoint exceeds another threshold.

A misprediction recovery example is illustrated in Figure 2. A branch misprediction (shown marked) was detected while the instruction window contains five branches and three checkpoints. The processor state is rolled back to the closest checkpoint. Rollback is followed by reexecuting the code from the checkpoint up to the mispredicted branch.

A checkpointing policy should be designed to reduce the use of resources, checkpoints and registers, while minimizing the impact on performance. The checkpointing policy affects the reclamation of registers—a register that was in use at the time when a checkpoint was taken, and hence is included in the checkpointed state, cannot be freed until the checkpoint itself is not freed. A register is deallocated and returned to the pool of free registers when all the following three conditions are satisfied: (1) the register has been removed from

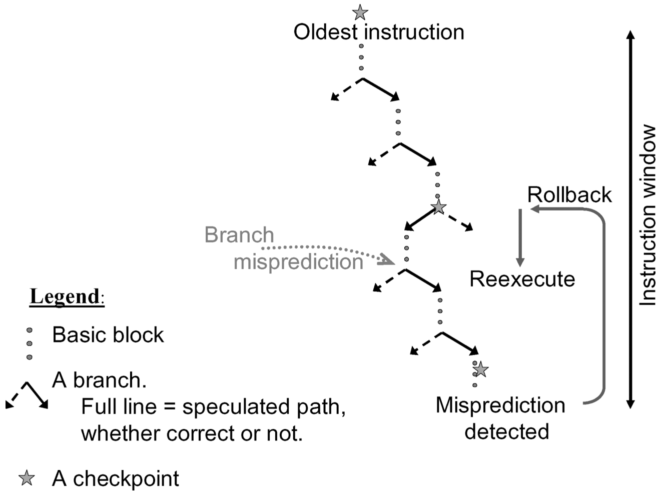


Fig. 2. Rollback and reexecution flow.

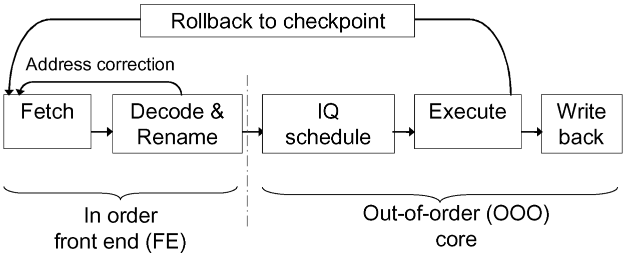


Fig. 3. Outline of the essential functions of the microarchitecture. The diagram is not intended to show the actual pipeline stages.

the mapping table, (2) the register is not included in any checkpointed mapping table, and (3) the register is not needed by any instruction waiting to be issued from the instruction queue (IQ).

A *RegUnmapped* flag and a *RegUse* counter, associated with each register, help determine when to deallocate a register. The counter is incremented when an instruction, which uses the register as a source operand is renamed, and is decremented when an instruction reads the value just before it is sent to an execution unit. The RegUse counter is also incremented when a checkpoint that includes the register in its mapping table is taken, and is decremented when that checkpoint is released. A register is deallocated when the RegUnmapped flag is set and the RegUse counter is zero.

3.1 Rollback Model

Figure 3 illustrates the operation of an OOO speculative processor that is relevant to this discussion. The pipeline has two logical parts: an in-order front end (FE) and an OOO core. When the processor is rolled back to a checkpoint, the FE refreshes the program counter (PC) and the register mapping table and

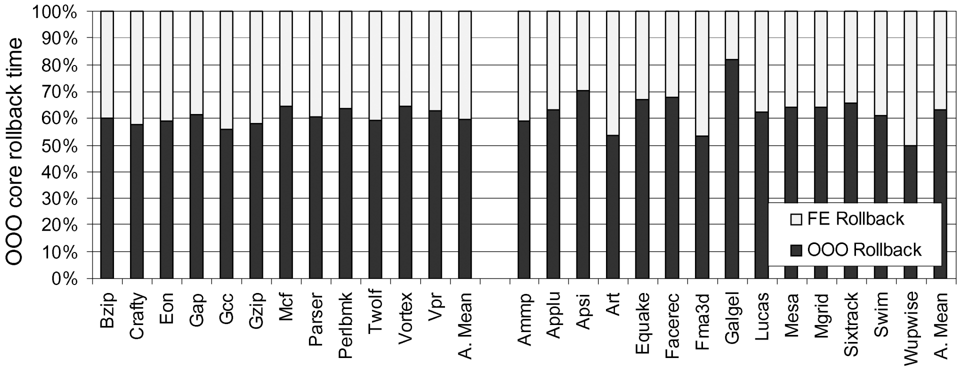


Fig. 4. Percentage of the OOO core rollback time. The total (100%) is the sum of $FE_{Rollback}$ and $OOO_{Rollback}$.

begins fetching new instructions. The OOO core drains the pipelines and the instruction queue and brings the RegUse counters to values consistent with the recovered processor state. Let $FE_{Rollback}$ denote the number of cycles needed to restore the FE, free the checkpoint and supply the first instruction to the IQ. $FE_{Rollback}$ consists of the time it takes to refresh the PC, fetch latency (assuming level one instruction cache hit), followed by decode, rename and dispatch latencies. $FE_{Rollback}$ depends primarily on the pipeline structure.

The OOO core recovery time ($OOO_{Rollback}$) depends on the instruction mix in the IQ and in the pipelines. The RegUse counters are the essential structure that supports aggressive register reclamation. During rollback, these counters are decremented according to the instructions still located in the IQ. Handling instructions on an individual basis makes the rollback time proportional to the number of instructions in the IQ.

3.2 Motivation

Figure 4 shows the proportion of the front end and the OOO core rollback times. The ratio of $OOO_{Rollback}$ to the sum of $FE_{Rollback}$ and $OOO_{Rollback}$ is 60% and 63% for the integer and floating point benchmarks respectively. Clearly $OOO_{Rollback}$ is a substantial component of the processor's recovery time.

Optimizing the rollback model is important as it holds a real impact on the overall performance. Comparing the optimal method (CRB), which entirely hides the OOO core recovery time ($OOO_{Rollback} = 0$) to the naive model (Non-overlapping), reveals an average speedup potential of 12.9% for the integer benchmarks. The measured speedup potential reflects the configuration in Table I and will grow to an average of 17%, if for example more registers are available and latency to memory is smaller. Finally, the measured speedup potential reflects a processor speculating only on branch direction and target address, and is expected to grow as other sources of speculation, such as load value prediction are applied.

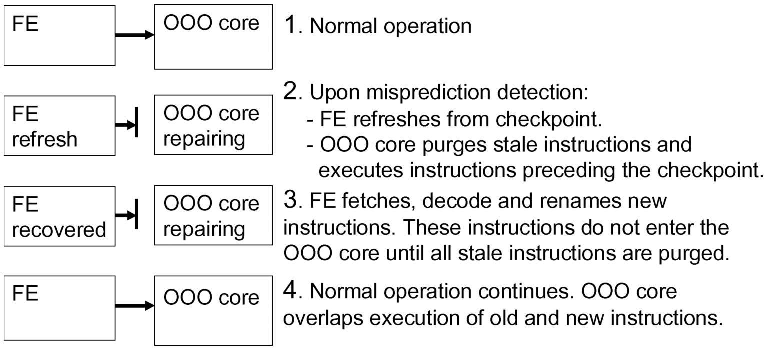

FE
OOO core
1. Normal operation
FE refresh
OOO core repairing
2. Upon misprediction detection:
- FE refreshes from checkpoint.
- OOO core purges stale instructions and executes instructions preceding the checkpoint.
FE recovered
OOO core repairing
3. FE fetches, decode and renames new instructions. These instructions do not enter the OOO core until all stale instructions are purged.
FE
OOO core
4. Normal operation continues. OOO core overlaps execution of old and new instructions.

Fig. 5. Overlapping model rollback scheme.

4. OVERLAPPING FE AND OOO CORE ROLLBACK TIME

Checkpoints hold the mapping table that should be refreshed, therefore the front end can theoretically start to fetch, decode and rename before the OOO core is done repairing. The challenge in overlapping the FE operation derives from what happens when the newly renamed instructions should be placed in the instruction queue. New instructions and valid instructions preceding the checkpoint are never associated with the same checkpoint and can co-exist in the IQ as well as in other parts of the OOO core. However this is not the case for stale instructions. We use the simplest checkpoint management that allocates and deallocates checkpoints in an in-order fashion, so during rollback the same checkpoint tag associated with stale instructions in the OOO core can be associated with new instructions in the FE. As a result we do not allow new instructions in the OOO core before purging stale instructions.

We define two rollback models: a simple *nonoverlapping* model that initiates the FE recovery after the OOO core is safe (clean from all stale instructions), and a more advanced *overlapping* model that is described in more detail in Figure 5. The idea of overlapping the FE and OOO core was implemented in Intel's P6 microarchitecture [Shen and Lipasti 2005, p.338]. In the rest of this paper we assume the *overlapping* model is the baseline for performance comparison and present results of the non-overlapping model only for completeness.

Figure 6 shows the normalized IPC of the overlapping model relative to the nonoverlapping model. The rightmost bar of each half of the figure is the harmonic mean. Overlapping the FE and OOO core improves IPC by 7.7% and 2.2%, on average, for the integer and the floating-point benchmarks respectively. The remainder of the paper describes two attempts to further improve the IPC by reducing the OOO core rollback time. From this point on, results are normalized to the overlapping model.

5. REDUCING THE ROLLBACK TIME: SPLIT QUEUE AND STALE FIRST

After rollback, all the RegUse counters must be consistent with the recovered state of the processor. To that end, when the microarchitecture is rolled back, all the instructions in the instruction queue are processed and the RegUse counters of their source registers are decremented. Keeping this in mind, our

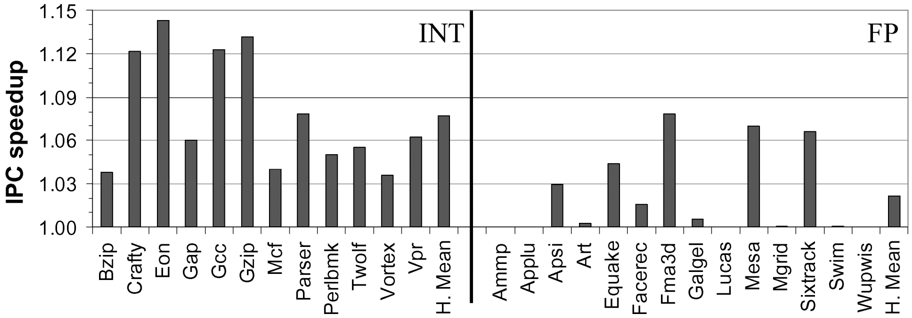


Fig. 6. Impact of the overlapping model.

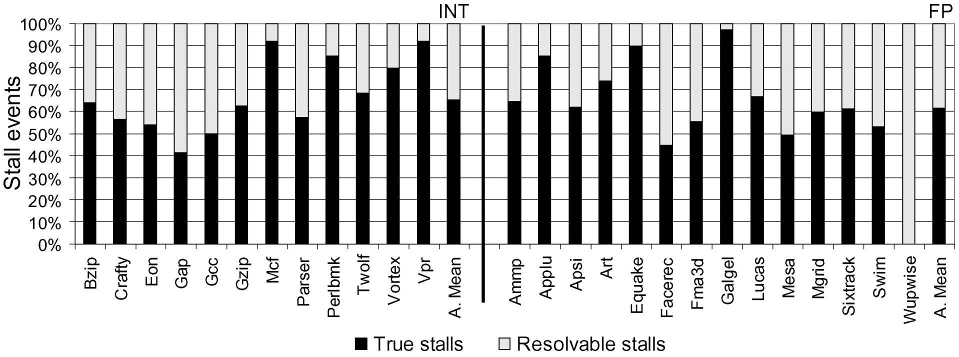


Fig. 7. Percentage of stalls that can be eliminated/resolved by reducing the number of instructions in the IQ and by giving priority to stale instructions. The total (100%) is the number of rollback events in which $OOO_{Rollback} > FE_{Rollback}$.

first attempt at reducing the rollback time is based on two complementary approaches: (1) reducing the number of instructions in the IQ, and (2) giving priority to stale instructions. After removing the stale instructions, the OOO core can be restarted immediately because the remaining instructions in the IQ and new instructions fetched after recovery can be mixed in the pipelines. Figure 7 shows that the approach introduced in this section is worth pursuing—by careful management of the instructions in the IQ, stalls exceeding $FE_{Rollback}$ can be entirely eliminated in over one-third of the recovery events in which $OOO_{Rollback} > FE_{Rollback}$.

5.1 Splitting the Instruction Queue Structure

The IQ holds instructions until they are issued for execution. Instructions are selected for execution when two requirements are met: operands are available in the register file and there is no structural hazard. The microarchitecture's IQ structure, which we denote as *Unified*, holds all instructions until they satisfy both requirements. On rollback, every instruction that has not read its values from the registers has to be taken care of on an individual basis. The Unified IQ structure is suboptimal, as it must hold any instruction that cannot be issued

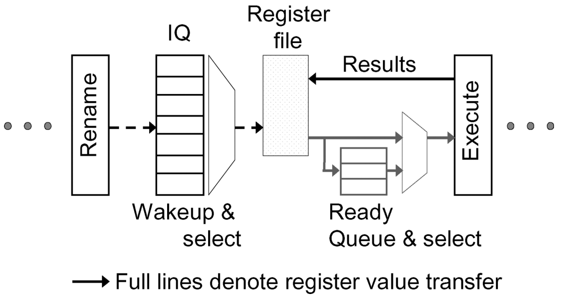


Fig. 8. A Split Queue structure.

due to a structural hazard, whether its operands are available in the register file or not.

To reduce rollback time and increase performance we introduce a *Split Queue* structure (Figure 8). If an instruction has the operands and resources it needs, it is transferred to execution, as in the Unified scheme. Otherwise, if an instruction cannot be issued because of a structural hazard but its operands are available, the instruction reads its source registers and is transferred to the *Ready Queue*. (A Ready Queue was proposed in a different context of simplifying the issue logic [Canal and Gonzalez 2001].) Since all instructions in the Ready Queue have decremented the RegUse counters of their source registers, rollback time can be made proportional to the number of instructions in the IQ structure alone.

5.2 Purging Stale Instructions First

Two types of instructions occupy the IQ:

—Valid instructions, preceding the checkpoint that the processor is recovering to.

—Stale instructions, belonging to one of the checkpoints about to be folded. (A checkpoint has to be *folded* if the mispredicting instruction belongs to it, or it was taken afterward.)

The microarchitecture's select logic (referred to as *Equal Priority*) does not give priority to one instruction type over the other at rollback. A different logic could select the stale instructions prior to others, because they are the bottleneck in repairing the OOO execution core. We refer to this latter method as *Stale First*.

Example 1: Equal Priority and Stale First

Consider a four- wide superscalar processor in which the select logic can handle up to four instructions during normal execution and also during rollback. On misprediction detection (refer to Figure 9) the IQ contains 43 instructions, 22 of which are stale. An Equal Priority select logic might be ready for reexecution after $\lceil \frac{43}{4} \rceil$ cycles, creating a rollback stall of three cycles (that is, beyond the rollback time of the front end, which is assumed to be eight cycles). However the same machine with a Stale First select logic will process all stale instructions in

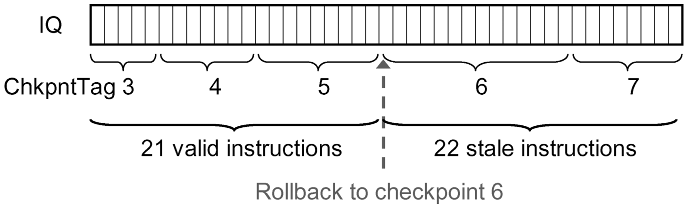


Fig. 9. The logical state of the instruction queue at a misprediction detection. Instructions with checkpoint tag six or seven are stale. Valid and stale instructions are mixed up in the OOO core.

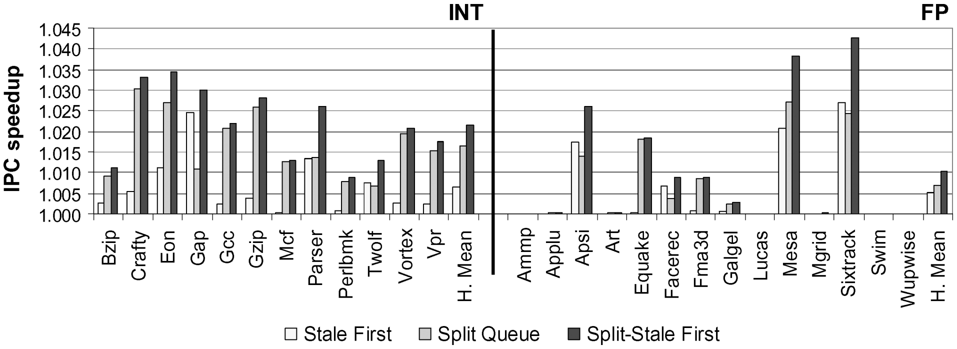


Fig. 10. Impact of the Split Queue and Stale First select logic. (IPC is normalized relative to the overlapping model).

$\lceil \frac{22}{4} \rceil$ cycles; after these six cycles the OOO core recovery is complete. In this scenario the OOO core recovery is completely hidden by the front end refresh time.

5.3 Performance of the Split Queue and Stale First Methods

Figure 10 presents the results that can be achieved by the Split Queue structure and the Stale First select logic. Several observations can be made regarding these results. The harmonic mean of the speedup of the integer and floating-point programs is 2.1% and 1% respectively. About one-half of the floating point benchmarks almost never mispredict branches (Figure 11), and, hence, are unaffected by the rollback scheme. Most of the performance contribution is because of splitting the instruction queue. The effectiveness of the Split Queue is limited, however, because when a rollback is started, some old instructions that precede the checkpoint are still in the IQ, waiting for operands.

The Split Queue and Stale First methods increase the complexity of the scheduler. Split Queue requires it to search both the IQ and the Ready Queue, and Stale First requires the scheduler to distinguish between stale and valid instruction types. To get the best results achievable with the Split–Stale First approach, we made the assumption that the increased complexity of the scheduler has no effect on the clock cycle. Furthermore, these results do not improve, even if we do not limit the size of the Ready Queue and the bandwidth of the bus that transfers instructions from the IQ to the Ready Queue. (The results shown in Figure 10 are based on these assumptions for the Ready Queue.) Having reached the maximum potential for the approach investigated in this

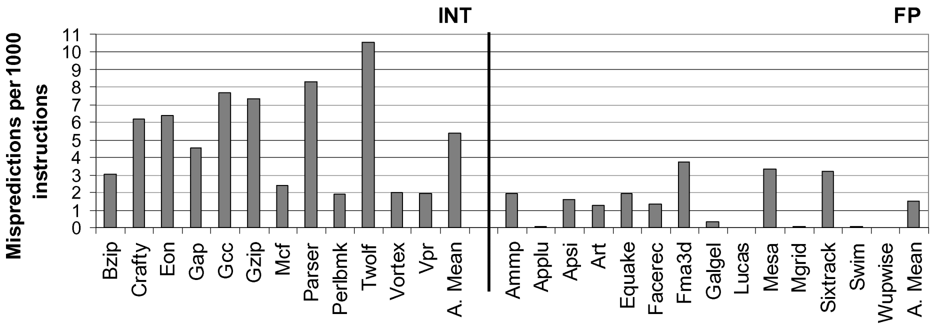


Fig. 11. Branch mispredictions per 1000 instructions.

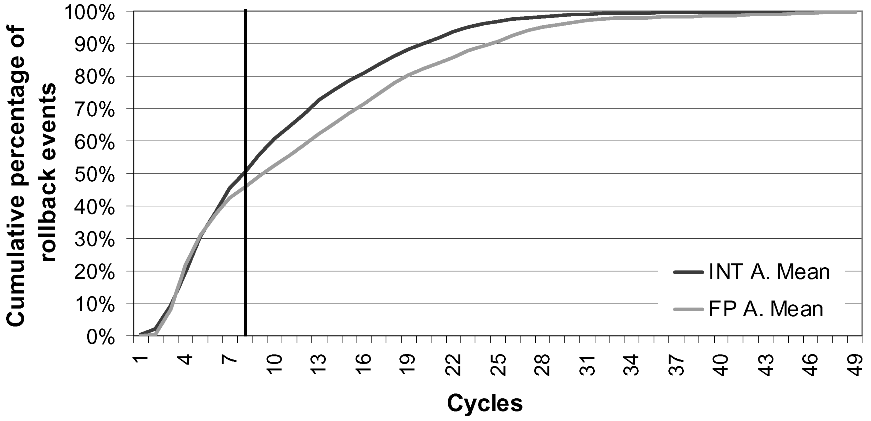


Fig. 12. Cumulative distribution of the OOO core repair time ($OOO_{Rollback}$). The vertical line shows the front-end repair time, which, in this study, is fixed at eight cycles (Table I). As long as the $OOO_{Rollback}$ does not exceed $FE_{Rollback}$, $OOO_{Rollback}$ can be completely hidden by the overlapping model. The overlapping method cannot hide, however, extra cycles needed by the OOO core, beyond $FE_{Rollback}$, as happens in more than one-half of the rollback events.

section, we make now another attempt at reducing the rollback time by trying an entirely different direction.

6. REDUCING THE ROLLBACK TIME: CRB

Both the nonoverlapping and the overlapping models, and the enhancements presented above, share the same weakness—their rollback time is proportional to the instruction queue size (refer to Figure 12). Even the more aggressive scheme, Split–Stale First, has a rollback time proportional to the number of stale instructions waiting for operands, which could be in the order of magnitude of the IQ. In our second attempt at reducing the rollback time, we address this problem directly by devising an approach that provides constant rollback time, independent of the number of stale instructions in the instruction queue. We refer to this method as *CRB* (constant rollBack time). CRB has two key elements: (1) a novel register management scheme, and (2) a method that handles the rollback of long-latency units.

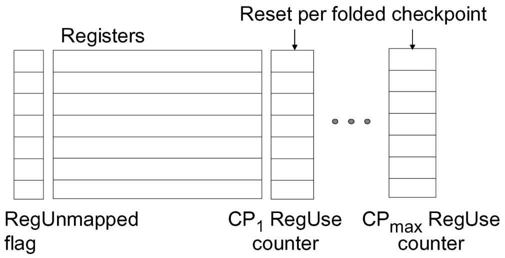


Fig. 13. Registers with per checkpoint RegUse counters.

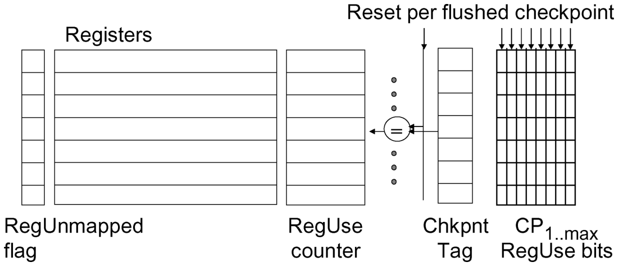
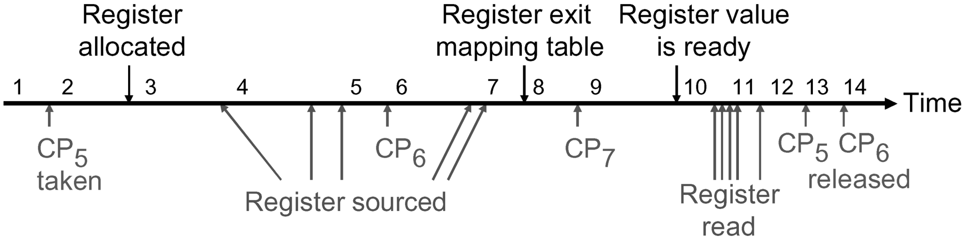


Fig. 14. Optimized register management. Associated with each register there is a single RegUse counter, a single ChkpntTag, and n RegUse bits, one per checkpoint.

6.1 Register Management

In the new register management scheme, illustrated in Figure 13, the RegUse counter is replaced by multiple counters, one per checkpoint. We also change the rules for allocating and freeing registers. Now a register is deallocated and returned to the pool of free registers when the following three conditions are satisfied: (1) the register has been removed from the mapping table, (2) the register is not included in any checkpointed mapping table, and (3) all the RegUse counters associated with the register are zero. Note that only the third rule has changed in order to support multiple RegUse counters per register. We further add an ability to reset all RegUse counters associated with a specific checkpoint. With these details in place, restoring the RegUse counters of a register to values consistent with the recovered state of the processor becomes a matter of resetting the counters of the folded checkpoints. If the last taken checkpoint is CP_n, rolling back to CP_{n-i} involves resetting the RegUse counters of checkpoints $CP_n, CP_{n-1}, \ldots CP_{n-i}$.

Using multiple RegUse counters per register creates a costly hardware structure. Fortunately, this logical structure can be optimized as illustrated in Figure 14. In this optimization a single RegUse counter is used. The RegUse counter is allocated to a single checkpoint—the one in which the register became part of the mapping table. Other checkpoints taken while the register is still part of the mapping table need only a single *RegUse bit*. Taking a checkpoint sets the corresponding RegUse bit, and folding the checkpoint resets the bit. A single bit is sufficient because all the instructions belonging to a checkpoint must read their registers and execute before releasing the checkpoint.

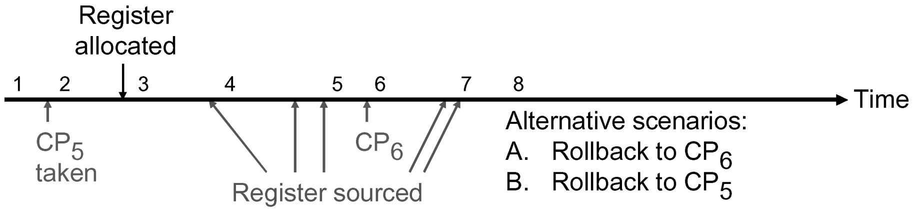


	RegUnmapped flag	RegUse counter	ChkpntTag	CP1..max RegUse bits
1	1	0	-	0, 0, 0, 0, 0, 0, 0, 0
2	1	0	-	0, 0, 0, 0, 0, 0, 0, 0
3	**0**	0	**5**	0, 0, 0, 0, 0, 0, 0, 0
4	0	**1**	5	0, 0, 0, 0, 0, 0, 0, 0
5	0	**3**	5	0, 0, 0, 0, 0, 0, 0, 0
6	0	3	5	0, 0, 0, 0, 0, **1**, 0, 0
7	0	3	5	0, 0, 0, 0, 0, 1, 0, 0
8	**1**	3	5	0, 0, 0, 0, 0, 1, 0, 0
9	1	3	5	0, 0, 0, 0, 0, 1, 0, 0
10	1	3	5	0, 0, 0, 0, 0, 1, 0, 0
11	1	**0**	5	0, 0, 0, 0, 0, 1, 0, 0
12	1	0	-	0, 0, 0, 0, 0, 1, 0, 0
13	1	0	-	0, 0, 0, 0, 0, 1, 0, 0
14	1	0	-	0, 0, 0, 0, 0, **0**, 0, 0

Fig. 15. Optimized register-management flow. The upper part includes a time axis with multiple events and 14 timestamps. The lower part of the figure describes the state of the register control fields at each timestamp. Changes are marked in bold.

Example 2: Optimized Register-Management Flow

Figure 15 is an example of how the register-management mechanism operates, demonstrating why a single RegUse bit is equivalent in functionality to a full-size RegUse counter. The register at hand (R_i) is allocated by an instruction belonging to checkpoint five (CP_5), saved in the mapping table of checkpoint six, and removed from the mapping table before taking checkpoint seven. CP_6 is taken before timestamp six and released after timestamp 13. The two instructions belonging to CP_6 are renamed between timestamps 6 and 7 (after CP_6 is taken) and the register value is read between timestamps 10 and 12 (before CP_6 is released). Hence it is sufficient to track taking and releasing CP_6, any relevant events (renaming an instruction I_k that uses R_i as a source operand, and the read of R_i, which marks the end of the use of R_i by I_k) must have occurred between these two points. Note that the optimized management mechanism does not delay register deallocation compared to the baseline. In



	RegUnmapped flag	RegUse counter	ChkpntTag	$CP_{1..max}$ RegUse bits
7	0	3	5	0, 0, 0, 0, 0, 1, 0, 0
8A. Rollback to CP_6	0	3	5	0, 0, 0, 0, 0, 1, 0, 0
8B. Rollback to CP_5	1	0	-	0, 0, 0, 0, 0, 0, 0, 0

Fig. 16. Rollback with the optimized register-management logic. Two scenarios are described, A and B, rolling back to CP_6 and CP_5, respectively.

the example, although all the reads of R_i occur before time 12, the register is still locked in the mapping table of CP_6 and can be freed only after releasing CP_6. Delaying register deallocation because of a checkpoint release was evaluated by Akkary et al. [2004, Section 4.3], in which they compare the aggressive register reclamation scheme (our baseline model) with an ideal scheme.

Example 3: Rollback

Figure 16 shows how the optimized register-management logic is modified during a rollback. During the first seven timestamps the same events occur as in Figure 15, the register, however, is not removed from the mapping table by time eight. CP_6 is the last taken checkpoint, and rolling back to it (scenario A) has no effect on the status of the register. After the rollback we begin again execution from CP_6, and thus we leave the RegUse bit of CP_6 set, as it was before. Rolling back to CP_5 is different. The register is not contained in the mapping table of CP_5. Hence the register is freed, its RegUnmapped flag is set, and the RegUse counter and bits are all cleared.

Having restored the RegUse counter and bits to consistent values after a rollback, we are now ready to take care of the instructions in the instruction queue. As explained earlier in Section 3, every instruction carries a ChkpntTag as it flows down the pipeline. For clarity, this tag, which is really part of the instruction's state, is shown as a separate field in Figure 17. As shown in the figure, the tag of checkpoint CP_n is loaded into a register and simultaneously compared with the checkpoint tags of all the instructions in the IQ. All matching instructions are flushed. If the last taken checkpoint is CP_n, to roll back to CP_{n-i+1} the "Checkpoint to be flushed" register is loaded in consecutive clock cycle with the tags of CP_n, CP_{n-1}, $\ldots CP_{n-i+1}$ until all the i checkpoints are folded. Purging i checkpoints takes i clock cycles. The time required to flush the instruction queue can be shortened. In Subsection 8.1 we show why this is not necessary.

Note that this simple and fast procedure of selectively flushing the instructions in the IQ is made possible by the new register management scheme

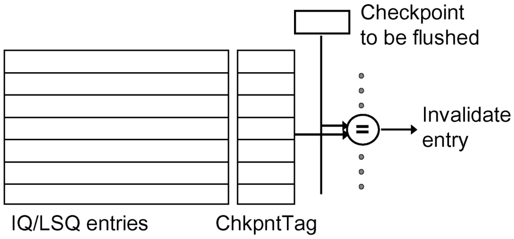


Fig. 17. Selective flushing of the IQ and LSQ structures.

described above. Now that the instructions in the IQ are no longer needed to update counters, stale instructions can be removed from the IQ as fast as possible. This is done by matching checkpoint tags and eliminating instructions belonging to folded checkpoints. The same selective-flushing method described for the IQ is also used for the LSQ. The LSQ contains either instructions that have already read their operands or store instructions in which the operand is not yet ready. In the latter case, the store is also kept in the IQ until the store operand is ready, it is read from the register file and the RegUse counter is decremented. Instructions in the LSQ do not decrement the RegUse counters and they can be flushed by matching checkpoint tags.

6.2 Handling Long-Latency Units

Recall that the overlapping model (Section 4 and Figure 5, in particular), which we use as the baseline model in this work, supports overlapping of the front-end and the out-of-order engine, but does not support selective flushing of the execution units. Without selective flushing, new and stale instructions cannot mix in the execution units. Thus instructions are not allowed to enter the OOO core until all stale instructions are purged. This model has been implemented in the Intel P6 microarchitecture [Shen and Lipasti 2005, p. 338]. In the rest of this section, we propose a simple and inexpensive solution that, functionally, has the same effect as selective flushing of the execution units. This solution allows new instructions to enter the OOO core as soon as they become available, after the rollback latency of the front-end. Those pipelines whose latency is no longer than $FE_{Rollback}$ do not pose a problem, as they have already been drained. Hence, we only have to handle long-latency units whose execution requires more than $FE_{Rollback}$ cycles.

Our goal is to allow the OOO core process new instructions, while stale instructions still drain in long-latency pipelined execution units. For each such unit, we add control logic as shown in Figure 18 that marks the unit as busy until it completes all the operations associated with folded checkpoints. The structural hazard that occurs when a unit is busy prevents new instructions from mixing with stale instructions inside the long latency unit. In this situation, the same ChkpntTag$_j$ can be used to identify an old stale checkpoint within the long latency unit and a new checkpoint in the rest of the OOO core. Long-latency execution units are those that require more than FE$_{Rollback}$ cycles. In our simulation model, this applies only to division instructions, but other processors might support other long latency instructions. Finally, long-latency

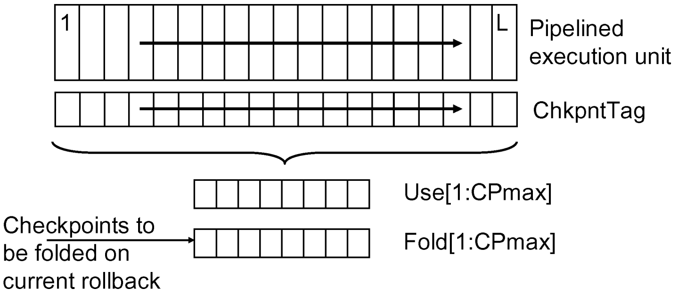


Fig. 18. An implementation of the control logic associated with an L deep pipelined execution unit. We add to the execution unit and accompanying the ChkpntTag structure, two bit vectors each, with a length equal to the total number of checkpoints. In the *Use* vector, a bit is on if the corresponding tag is used by at least one instruction currently in the pipeline. In the *Fold* vector, a bit is on if the corresponding checkpoint is to be folded, which is determined when a rollback is started. The control logic blocks the pipeline input if, for any k, $Use[k] * Fold[k] = 1$. The logic purges a result if $Fold[ChkpntTag(L)] = 1$.

execution units that are not pipelined hold a single instruction, so they are either entirely valid or entirely stale, and the additional control logic could be optimized to a single ChkpntTag and comparator flushing the unit when the checkpoint is folded.

Example 4: Handling-Long Latency Units

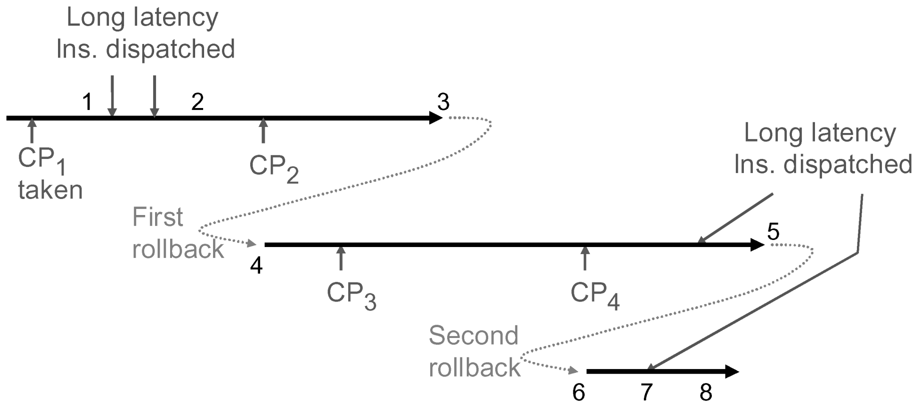
Figure 19 demonstrates how CRB maintains consistent tag naming by isolating long-latency units that contain stale instructions from the rest of the OOO core. Figure 19a illustrates the instruction flow. Figure 19b is a step-by-step trace of the example.

Figure 19a shows two rollbacks. In the first one, CP_2 is folded. The long-latency instructions were dispatched before CP_2 and none of them uses the CP_2 tag. After the rollback, new instructions carrying the CP_2 tag are allowed to enter the long pipeline, which is marked *not busy* (see Busy column in Figure 19b).

In the second rollback CP_4 is folded. The long instructions, which were dispatched after CP_4, are still in the pipeline. This time, after the rollback, new instructions are not allowed to enter the long pipeline, which is marked *busy* (Figure 19b). This prevents new and old instructions carrying the same checkpoint tag from getting mixed up.

7. CRB PERFORMANCE

Figure 20 illustrates the performance of the constant rollback method. The relevant IPC results are 2.1% and 4.8% for the floating-point and integer benchmarks, respectively. In the integer benchmarks, the speedup is in the range 2.3% (Bzip) to 8.3% (Eon). Results from running a specific benchmark on an overlapping model can be used to roughly estimate how efficient will a CRB model be for that benchmark. Eon has a large number of branch mispredictions (but not the largest in the set of the integer programs), a high IPC (but not the highest), and, on average, a large number of stale instructions that have to be eliminated during rollback. The combination of these three factors



(a) Instruction flow, showing checkpoints, dispatch of long-latency instructions and two rollbacks.

Time

	Use [1:CP4]	Fold [1:CP4]	Busy	Explanations
1	0, 0, 0, 0	0, 0, 0, 0	0	Initial state.
2	**1**, 0, 0, 0	0, 0, 0, 0	0	Long latency instructions were issued.
3	1, 0, 0, 0	0, **1**, 0, 0	0	First rollback starts.
4	**0**, 0, 0, 0	0, **0**, 0, 0	0	First rollback ends (execution unit drained during rollback).
5	0, 0, 0, **1**	0, 0, 0, **1**	**1**	Second rollback starts.
6	0, 0, 0, 1	0, 0, 0, 1	1	Second rollback ends (the stale long latency instruction is still in the execution unit pipeline)
7	0, 0, 0, 1	0, 0, 0, 1	1	The newly dispatched long latency instruction waits due to a structural hazard (Busy==1).
8	0, 0, 0, **0**	0, 0, 0, **0**	**0**	The stale long latency instruction was purged. The long latency instruction waiting since timestamp 7 can be issued.

(b) Values of the control fields at eight timestamps. Changes are marked in bold.

Fig. 19. An example of the CRB control logic associated with a long-latency execution unit.

contributes to Eon's speedup. As noted earlier, seven of the floating-point programs have low branch misprediction rates and any reduction in the recovery time is bound to have a minor effect, if any, on the speedup.

7.1 CRB Implementation Cost

Power, area, and latency costs of the CRB hardware structures were estimated using version 4.2 of Cacti [Tarjan et al. 2006] with 70-nm technology, the most advanced technology the tool was verified on. Dynamic energy was translated to power consumption using a 2-GHz clock frequency and switching data from the performance simulation. Leakage power was also taken into account using CACTI. The figures in Table II are reported separately for leakage and dynamic power.

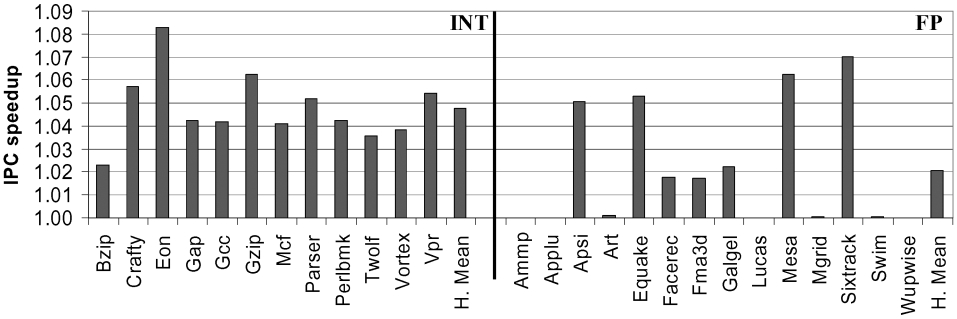


Fig. 20. Impact of constant recovery time. (IPC is normalized relative to the overlapping model)

Table II. Area and Power Consumption of the Storage and Combinational Logic Required by the CRB Microarchitecture

		Optimized Register Management	Selective Flushing of IQ and LSQ	Total
Area	[mm^2]	0.100	0.024	1.124
Leakage power	[mW]	2.101	0.496	2.597
Dynamic power	[mW]	10.464	3.009	13.473
Total power	[mW]	12.565	3.505	16.070

Table II shows the area and power consumption of the storage and combinational logic required by the CRB microarchitecture. These numbers do not include any control associated with the long-latency execution unit. The reason for this is that the logic added to the long-latency execution unit is expected to be small and, to a large extent, depends on parameters that vary between processors, such as ISA, the unit's latency relative to $FE_{Rollback}$, and the degree of pipelining. The access time is not specified as it is either not on the critical path, or was measured to be well below a single cycle.

7.2 Register File Size

Adding more registers should improve performance by reducing the number of events that block instruction rename. This is indeed true for CRB, as shown in Figure 21A. Interestingly enough, this is not always true for the other microarchitectures. As expected, the extra registers are used to handle more instructions in parallel. This is beneficial when speculation is correct, but could be harmful when incorrrect. Handling more instructions in parallel introduces additional stale instructions, consequently, traditional rollback models like the nonoverlapping and overlapping require longer periods of time to clean the pipelines and resume execution following mispredictions. The balance of these two opposite effects changes from one benchmark to another. However, for most integer benchmarks it is a moderate slowdown.

On the other hand, the rollback time in CRB is not affected by the number of instructions that have to be purged and the performance improves with the addition of registers. As a result, in the integer benchmarks (Figure 21A) the gap between CRB and the overlapping model grows from 4.8% at 128 registers to 7.4% at 256 registers. A similar trend can be observed when

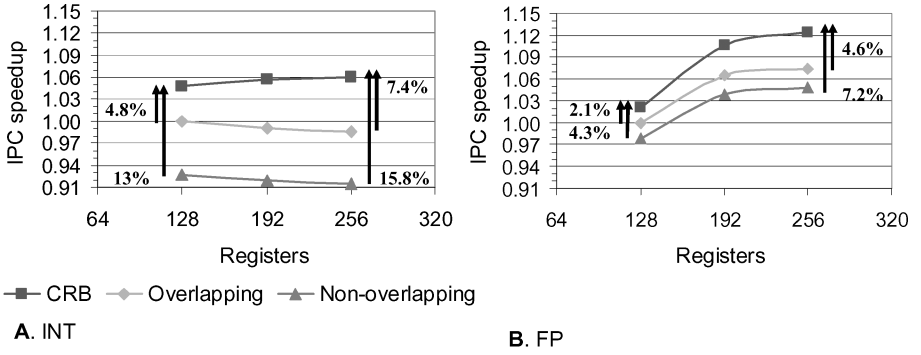


Fig. 21. Effect of the register file size.

Table III. Effect of the Memory Access Latency

Benchmarks	Memory Latency	Overlapping Model (%)	CRB Model (%)	CRB Advantage (%)
Integer	600	100.0	104.8	4.8
	450	107.0	112.5	5.1
	300	115.1	121.4	5.5
	150	123.9	131.4	6.0
FP	600	100.0	102.1	2.1
	450	111.3	113.8	2.3
	300	124.7	127.9	2.6
	150	139.4	143.4	2.9

comparing CRB to the nonoverlapping model. Figure 21B shows the data for the floating-point benchmarks, and again, the gap between CRB and overlapping or nonoverlapping models grows with the number of registers. The floating point benchmarks have fewer branch mispredictions, however, and none of the models show a performance slowdown.

7.3 Memory Latency

Overlapping and CRB performance was measured for four memory access latencies: 600, 450, 300, and 150 cycles. Table III shows the normalized IPC relative to the overlapping model with a 600 cycle memory latency, for both models and for both the integer and floating point benchmarks. The performance of the models improves substantially when memory latency is shorter.

Looking closer at the rightmost column, it appears that the performance of CRB improves at a slightly greater rate. To understand this effect consider the following equation:

$$CPI = CPI_{Ex} + CPI_{MemStalls} + CPI_{RollbackStalls}.$$

For the purpose of this explanation, we assume that there is no overlap between the three parts of the sum (that is, any cycles during which a nonblocking memory access occurs are counted in CPI_{Ex}, not in $CPI_{MemStalls}$). CRB reduces $CPI_{RollbackStalls}$ and we assume it does not affect the other two parts of the sum, a reasonable assumption for a first-order approximation. The extra cycles spent

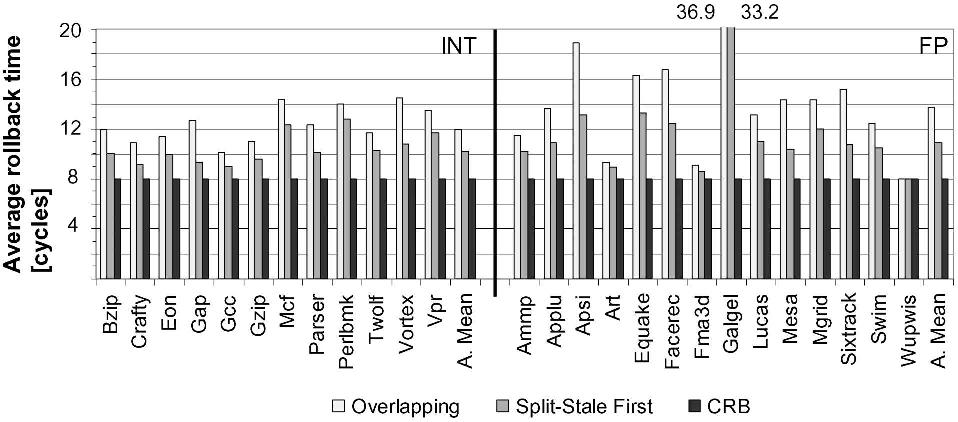


Fig. 22. Average rollback time in clock cycles.

without CRB are

$$CPI_{NoCRB} - CPI_{CRB} = \Delta CPI_{RollbackStalls},$$

and the speedup of CRB is

$$speedup = \frac{\Delta CPI_{RollbackStalls}}{CPI_{Ex} + CPI_{MemStalls} + CPI_{RollbackStalls}}.$$

The speedup improves with decreasing memory latency because the $CPI_{MemStalls}$ part of the denominator decreases.

7.4 Rollback Time: Clock Cycles

Throughout the paper, so far, we have mostly examined the performance aspects of the proposed techniques by looking at the IPC. In this section, we offer a more local view by focusing on the rollback time, in clock cycles, of the three main methods that we have considered: overlapping, Split–Stale First, and CRB. Figure 22 shows that for almost all the benchmark programs, with the exception of Art, Fma3d, and Wupwise, CRB substantially reduces the rollback time. As expected, on average, the reduction is larger in the floating point programs. As we have already seen, however, in many floating programs this gain does not translate into IPC gains because mispredictions are infrequent.

8. CRB IMPLEMENTATION TRADEOFFS

In this section we describe implementation tradeoffs related to the number of checkpoints to be folded, front-end latency, checkpoint buffer depth, and branch prediction unit size.

8.1 Number of Checkpoints to be Folded per Rollback Event

The latency of the selective flushing method described in Section 6 depends on the number of checkpoints which need to be folded. The time required to flush the instruction queue can be shortened by making the "Checkpoint to be flushed" register wider and adding comparators. This, however, is not needed.

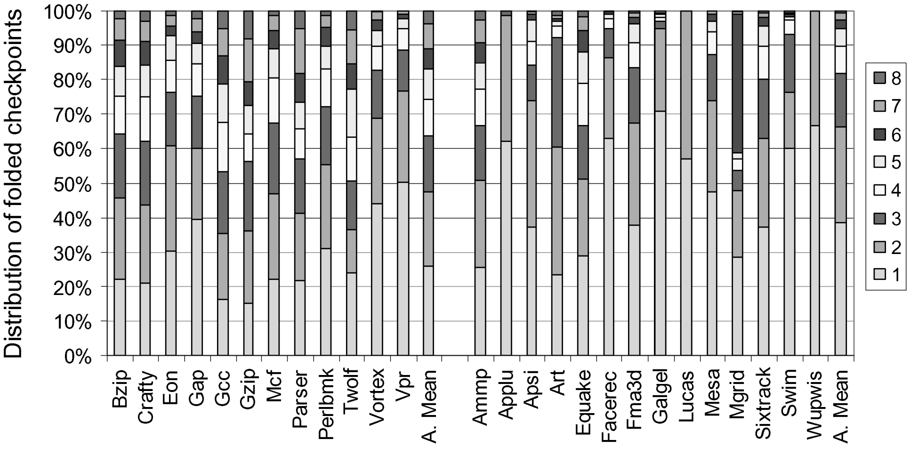


Fig. 23. Folded checkpoints. The total (100%) is the total number of recovery events. The percentages are the number of recovery events in which $i = 1, 2, \ldots 8$ checkpoints were folded. We maintain an 8-deep in-order checkpoint buffer (Table I).

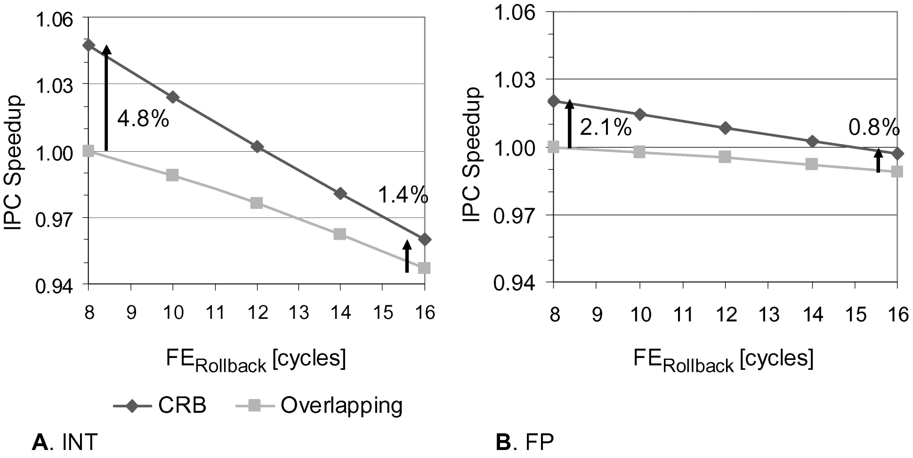


Fig. 24. Effect of the front end rollback time.

As shown in Figure 23, the number of folded checkpoints is small—in 64% and 82% of the rollbacks of the integer and floating-point benchmarks, respectively, only one two or three checkpoints have to be flushed. Reducing the time needed to process the IQ below the front-end recovery time is useless, as the OOO core will wait until the FE supplies new instructions.

8.2 Front-End Latency

As expected, increasing the front-end latency reduces performance. Figure 24 shows that the performance of the overlapping and the CRB models decreases almost linearly as the front-end rollback time increases. The linear dependence shown is only valid for the range presented. This is the range of interest, because the latency of reasonable implementation alternatives differ only by a

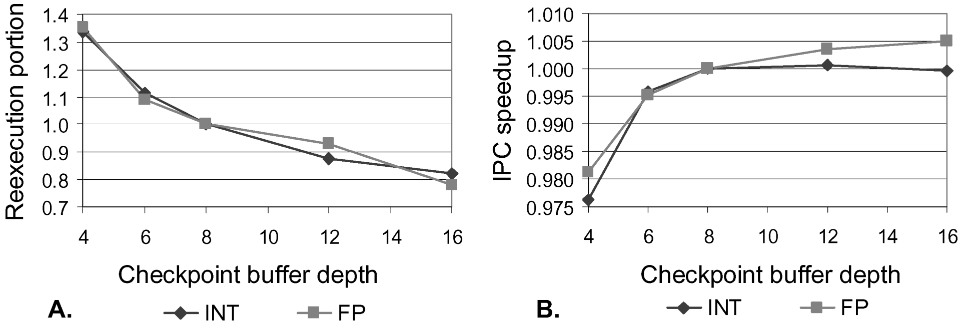


Fig. 25. Number of reexecuted instructions and speedup as a function of the checkpoint buffer depth. All measurements are normalized to the results of a CRB model with parameters as presented in Table I (checkpoint buffer depth of eight).

few cycles. The difference in the slope of the lines in parts A and B of the figure is related to the frequency of mispredictions, which is higher in the integer benchmarks.

The advantage of CRB over the overlapping model is that it always repairs the OOO core by the time a new instruction reaches the IQ after recovery. Naturally, if the FE pipeline is very deep, this advantage is worn out. (Dual-path processing can be used to reduce the effective FE pipeline depth.) This effect is demonstrated in Figure 24 by the diminishing gap between the CRB and overlapping models.

8.3 Checkpoint Buffer Depth

Situations in which the checkpoint buffer is full may prevent the processor from taking checkpoints. As a result more instructions may have to be reexecuted when a mispredicted branch is detected. Figure 25A shows how the number of reexecuted instructions changes as a function of the checkpoint buffer depth. The reexecution portions of the integer and the floating-point benchmarks respond in a similar way, both decreasing as long as additional checkpoints are available.

Reexecution affects overall IPC but is not always the most dominant factor. As shown in Figure 25B, the integer benchmarks slow down after 12 checkpoints, although the number of reexecuted instructions keeps decreasing. One factor that may reduce the performance of a processor with a deeper buffer is register availability. Every checkpoint prevents registers that are part of its mapping table from being reused until the checkpoint is freed. Unnecessary checkpoints might cause renaming to halt because of lack of registers. This analysis is backed by the fact that slowdown occurs mostly in benchmarks that tend to use the free registers, and the average slowdown vanishes when a larger register file is available.

Probability of executing on the wrong path is another factor decreasing the advantage of a deeper checkpoint buffer. As branch misprediction is the only cause for rollback in our simulated model, checkpoints are mostly taken because of low confidence branch prediction estimations, especially in integer

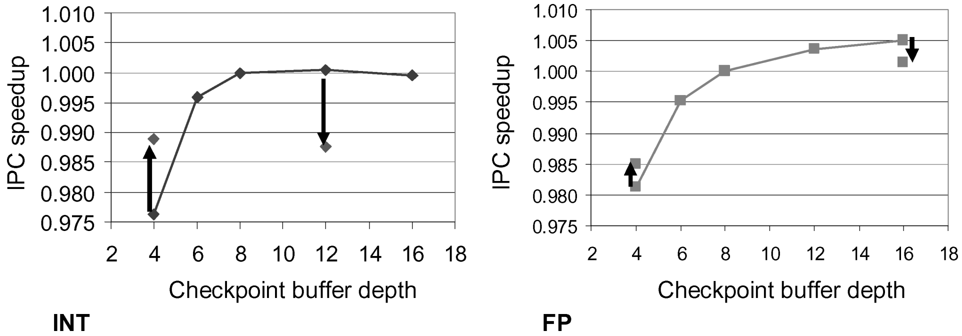


Fig. 26. IPC speedup sensitivity to the time it takes to read the NPC field of a checkpoint. The basic IPC speedup curve is using a constant $FE_{Rollback}$, so it is identical to the one in Figure 25. Arrows point to where the IPC speedup would be if $FE_{Rollback}$ would be dependent on checkpoint buffer depth. In this example, $FE_{Rollback}$ is a cycle shorter for a checkpoint buffer with four entries and a cycle longer for 12 entries or more. The results are normalized to a buffer of depth eight.

benchmarks. When more checkpoints are used, the probability of still executing on the correct path is lower and the advantage of having a deeper checkpoint buffer diminishes. The same observation is also the motivation for pipeline gating [Manne et al. 1998].

So far we have analyzed CRB sensitivity to checkpoint buffer depth disregarding the fact that the number of checkpoints affect $FE_{Rollback}$ time. Each checkpoint contains a next PC (NPC) field, used when the processor rolls back. Reading the NPC field directly affects $FE_{Rollback}$ time, it is a serial action preceding the fetch. In processors with short clock cycles the time to read the NPC depends on the buffer depth. Figure 26 illustrates how IPC speedup changes when $FE_{Rollback}$ time is proportional to the checkpoint buffer depth. For the integer benchmarks the result is an almost flat, though an irregular curve showing that the performance of CRB remains within a narrow range when the buffer depth changes from four to 16 entries. The performance of the FP benchmarks remains almost unaffected.

8.4 Branch Prediction Unit Size

All rollback models experience a slowdown when a smaller branch predictor unit is used. In this section, we look at the dependence on the branch prediction unit size, and show that CRB is less affected relative to the other rollback models. Figure 27A illustrates the impact of reducing the branch predictor size on the nonoverlapping, overlapping, and CRB models.

For all the models, reducing the branch predictor table size causes aliasing that decreases the branch predictor accuracy. Nevertheless, the IPC slowdown is dependent not only on the branch predictor accuracy, but also on the average misprediction penalty, which varies between the models. For this reason, CRB with the smallest misprediction penalty, is less affected relative to the two other models.

Although integer benchmarks are characterized by many branches and mispredictions in comparison to floating point benchmarks, their performance

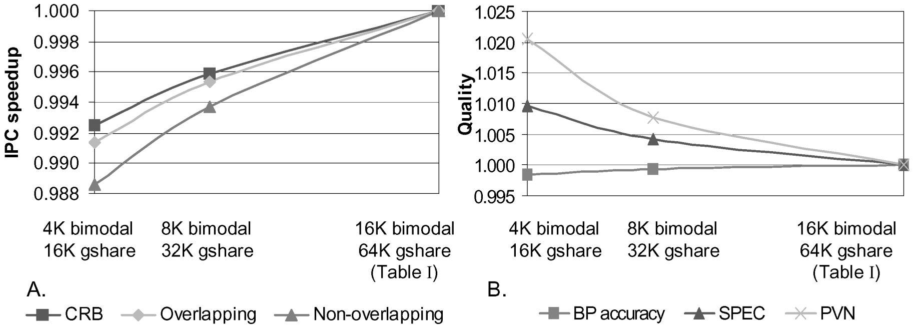


Fig. 27. (A). IPC speedup as a function of the direction predictor unit size, for floating-point benchmarks. (B). Branch predictor (BP) accuracy and quality of the branch prediction confidence estimator as a function of the branch predictor size. The quality of the confidence estimator is measured using two indicators, *SPEC* (specificity), which is the fraction of branch prediction misses that were estimated as low confidence predictions, and *PVN* (predictive value of a negative test), which is the fraction of branch predictions estimated as low confidence estimations that were actual mispredictions. The values of the rightmost configuration are an average accuracy of 94.5%, an average SPEC of 87.3% and an average PVN of 20.9% for the integer benchmarks. For clarity of comparing the trends on a single scale, all results are normalized to the results of rightmost configuration.

deteriorates slower, in all the models, when the branch predictor size is reduced. The reason for this is that predicting the branch direction and estimating the associated confidence level are related issues. Thus, even though reducing the branch predictor size decreases the branch predictor accuracy, it improves the confidence estimator quality [Grunwald et al. 1998], as shown in Figure 27B. IPC depends on both structures. Better confidence estimations translate to re-executing fewer instructions and to lower use of checkpointing resources. This tradeoff helps to reduce the performance slowdown associated with reducing the branch predictor size on both benchmark types, but has greater value for integer benchmarks, which are more sensitive to the quality of the confidence estimator.

We have used TAGE [Seznec and Michaud 2006] to check the effect of the branch prediction algorithm. Compared with the tournament branch predictor used in the remainder of this paper, TAGE improved the average branch prediction accuracy from 94.5% to 95.7%, achieving an IPC speedup of 3.2% for both CRB and the overlapping model. Reinforcing the tradeoff discussed in this section, when TAGE is used the confidence estimation accuracy deteriorated from an average SPEC of 87.3% to 66.5% and from an average PVN of 20.9% to 10.9%. TAGE did not significantly change the CRB advantage over the overlapping model and showed a very moderate slowdown when too many checkpoints were being used, resembling the one presented in Figure 25A.

9. A STREAMLINED IMPLEMENTATION: URB

The CRB approach provides constant rollback time. In this section, we use the fact that long-latency instructions are usually infrequent to present a streamlined version of CRB in which the OOO core repair is stalled until all the execution units are drained. In this scheme, the rollback time is upper bounded by the execution time of the slowest execution unit. We refer to this simplified

Table IV. Usage of Long Latency Division Execution Unit and URB Slowdown

Benchmarks		Unit is not Idle at Rollback Event (%)	Unit is not Idle After $FE_{Rollback}$ Cycles (%)	IPC Slowdown $1 - \frac{IPC_{URB}}{IPC_{CRB}}$ (%)
INT	Eon	7.31	4.74	0.61
	Gap	4.75	3.33	0.14
	Twolf	10.84	6.42	0.38
	Vortex	2.21	1.43	0.05
FP	Applu	8.82	8.79	0.00
	Apsi	1.42	0.48	0.01
	Facerec	0.74	0.37	0.00
	Mesa	15.71	7.87	0.49
	Sixtrack	27.78	7.37	0.41
	Swim	6.42	6.39	0.00

method as *URB* (Upper-bounded rollback time). URB is based on CRB's register management and selective flushing methods. URB, however, has simpler control logic for the long-latency pipelined execution units.

A stale instruction in an execution unit pipeline is associated with a stale checkpoint. After rollback, this checkpoint becomes valid and new instructions will reuse it. CRB maintains consistent checkpoint tagging by isolating the execution units containing stale instructions from the rest of the OOO core. Thus, the same ChkpntTag$_j$ can be stale within the execution unit, but valid in the rest of the OOO core. URB avoids inconsistency by adding stalls. It will not allow new instructions to propagate into the IQ until all execution units are drained.

URB's simpler control logic also causes false dependencies, in which recovery is stalled even if none of the instructions in the execution unit pipeline is stale, or if the stale instructions were already drained. There are several "enhanced URB" design alternatives that analyze checkpoint tags, to some degree, and can reduce or eliminate the false dependencies. We do not quantify these options, because the difference in the performance between URB and CRB is insignificant.

The difference between URB and CRB shows up only for long-latency instructions, such as division. Table IV tracks the use of the division unit during rollback events. SPEC2000 benchmarks which are not specified in the table do not use the execution unit frequently (less than 0.1% occupancy). It can be seen that on the vast majority of rollback events the execution unit is idle. Furthermore, when checked after $FE_{Rollback}$ cycles, the execution unit was found idle for at least 90% of the time for all the benchmarks. Hence, we expect URB to come very close to CRB, which is indeed shown by the results presented in the rightmost column. Only a few benchmarks show a noticeable performance slowdown (at most 0.6%).

In order to estimate the URB slowdown for a specific benchmark, we have found that knowing the instruction mix and the number of rollback events is insufficient. The URB slowdown is hard to estimate as it is highly dependent upon two additional code characteristics. The first is that instruction mix is sometimes different in the vicinity of mispredicted branches than it is in the rest of the code. For example, 0.61% of the Swim benchmark instructions were measured to be division instructions, while only-half that rate (0.33%) was measured when only reexecuted instructions were considered. The second code characteristic concerns the average time a long-latency execution unit takes to drain. There are large differences between benchmarks regarding the average location of the long-latency instructions within the execution unit pipeline. While in the Applu benchmark, for example, almost every ($\frac{8.79}{8.82} = 99.66\%$) event in which the execution unit is busy results in URB adding stalls; in Sixtrack benchmark, only a quarter of these events do so ($\frac{7.37}{27.78} = 26.50\%$, values are taken from Table IV).

As was done for CRB, URB sensitivity to register file size, memory access time, number of checkpoints, front-end pipeline depth, and the branch prediction unit size was also measured. In all cases, URB's average performance was found to be within 0.1% of CRB.

10. RELATED WORK

This section describes previous work on two related topics, misprediction recovery and ROB-free microarchitectures.

Misprediction recovery can be divided into three phases. First, the processor returns to a correct state, then the first new instruction is issued, and finally new instructions are processed until the next misprediction event occurs. Returning the processor to the correct state includes mainly restoring the register mapping table. In ROB-based microarchitectures, a restored copy is usually available either when the mispredicted instruction is committed or earlier, by incrementally reconstructing a copy on misprediction detection. Mapping table incremental reconstruction can be made from either the commit-stage mapping table, or the front-end mapping table [Hinton et al. 2001; Akkary et al. 2004]. A second approach, exemplified by the MIPS R10000 and Alpha 21264 processors, is to use checkpoints in addition to the ROB to refresh rather than reconstruct the mapping table [Yeager 1996; Kessler 1999]. Cristal et al. [2004b] aim at supporting a large instruction window. They too use a checkpoint to refresh the mapping table, but also introduce a method to prevent the instruction queues from blocking new instructions. The method converts stale instructions to special no-operations which issue (maintain correctness of source operands management counters), but do not execute. Postponing the act of updating the counters prevents stalls at the cost of a less efficient register file and instruction queue. This method achieves 82% and 63% of the CRB speedup for the integer and floating-point benchmarks, respectively, but is sensitive to the size of the register file, for example, when the processor has 96 registers (instead of 128) it achieves only 65% and 16% (integer and floating-point) of the CRB speedup. A third approach for returning the processor to the correct state was suggested by Zhou et al. [2005]. They allow renaming new instructions before the mapping table is fully reconstructed. In this scheme, the mapping table structure differentiates between restoration-dependent and -independent instructions. Overlapping the mapping table reconstruction and the execution of independent instruction increases performance but requires complex control, especially when a subsequent misprediction occurs and the mapping table has not as yet been restored.

Selective dual-path execution was proposed by Heil and Smith [1996]. This scheme reduces the recovery time of mispredicted branches by executing instructions from both paths of the branch if the branch is predicted with a low confidence level. Aragon et al. [2002] process instructions from the alternative path also, but wait for the branch to be resolved before actually executing them. This simplifies the processor implementation while still reducing the average time required for new instructions to reach the execution units. Dual-path processing optimizations are complementary to our work.

In certain situations the results of speculative instructions, executed following the mispredicted path, can be reused. Chou et al. [1999], Gandhi et al. [2004] and others accelerated control independent subsets of new instructions that were already executed following the branch misprediction. Similarly, Sarangi et al. [2005] accelerated data independent subsets of new instructions following a false load value speculation. Filtering the reusable instructions that are both control and data independent with wrong path instructions preceding them is the main difficulty. Checkpoint based microarchitectures have a unique subset which can always be accelerated, as every instruction between the checkpoint and the mispredicting instruction will be reexecuted. Mutlu et al. [2005b] examine accelerating reexecution. They conclude that this is not a promising direction, but their study was limited to reexecution on infrequent events (last level cache misses). Faster processing of new instructions is complementary to rollback time, the topic we are exploring in this paper.

The work closest to the research presented in this paper is the work of Akkary, Rajwar, and Srinivasan on CPR (Checkpoint processing and recovery) [Akkary et al. 2003, 2004]. CPR is a ROB-free checkpoint microarchitecture with several novel features. The register-management scheme in which there is no need for a reorder buffer to determine when registers can be freed, follows earlier work by Moudgill et al. [1993]. The register management of CPR includes an enhancement that prevents freeing registers pointed by the mapping table saved in a checkpoint. Other features of CPR include confidence based checkpoints, a concept derived from the work of Moshovos [2003], and a novel hierarchical store queue structure. A property that makes CPR attractive is scalability. Moreover, the CPR misprediction recovery scheme was shown to outperform ROB based microarchitectures. Our microarchitecture is derived from CPR.

In later studies, CPR was optimized for long-latency memory accesses. This was done by freeing registers and instruction queue slots populated by long-latency cache miss-dependent instructions [Srinivasan et al. 2004] and exploiting CPR bulk commits to simplify the LSQ design [Gandhi et al. 2005]. Although we do not model these enhancements, based on the study we present in Subsection 7.3, we expect our CRB mechanism advantage to be even greater if these enhancements were modeled.

11. CONCLUDING REMARKS

Beginning with a checkpoint-based, ROB-free microarchitecture, conceived for highly efficient speculative processing, we have investigated several alternative designs aimed at reducing the recovery time of speculative events. The motivation for this research was provided by results showing that the OOO core rollback time is a significant factor in the recovery process. Our initial effort, based on the observation that the recovery time is proportional to the number of instructions in the instruction queue, led to two complementary enhancements. The first one, the Split Queue method, reduces the IQ population. The second enhancement, Stale First, gives priority to stale instructions. We have presented performance results for a microarchitecture integrating the overlapping, Split Queue, and Stale First methods.

Our second attempt at reducing the rollback time follows a different route. We introduce a novel register-management scheme that supports bulk retirement of instructions from the instruction queue. All the instructions belonging to a folded checkpoint are flushed in a single clock cycle. This constant rollback time (CRB) method can be easily extended to support multiple checkpoints per cycle, but the extension is not necessary and satisfactory results are obtained by the checkpoint-per-cycle scheme.

An optimized implementation of CRB, using a single RegUse counter and per checkpoint RegUse bits, substantially lowers the implementation cost. This optimization is based on the insight that counting the use of a register is only necessary until a new checkpoint is taken. All the register use events occur between taking and releasing the checkpoint, hence, a single bit monitoring the use of the checkpoint is sufficient.

We have presented performance results for CRB and for a closely related method, upper-bounded rollback (URB). We have shown that the area and power consumption of the additional logic required by these methods is low, and also presented a detailed performance study of CRB and reported results obtained by varying key parameters—the register file size, the memory latency, the front-end recovery time, the checkpoint buffer depth, and the branch prediction unit. These results demonstrate that a CRB processor core can be made smaller by reducing certain resources with a minimal performance cost.

The checkpoint microarchitecture investigated in this paper is a resource–efficient machine [Akkary et al. 2004] that uses early register release mechanisms to attain satisfactory performance with a reduced-size register file. The exclusion of the reorder buffer further reduces both complexity and power. In this paper, we have proposed several methods that reduce the rollback time of the resource–conscious microarchitecture, and by doing so further enhance its efficiency. Higher efficiency can be translated to higher performance, or if a certain performance level is desired, it can be also translated to lower power by proportionally reducing the machine's activity level or its clock frequency.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers, Haitham Akkary, Daniel Citron, and Ron Gabor for helpful feedback on this work. Also thanks to Andreas Moshovos for tips on building the simulation model, and to Noam Jungmann for assistance in estimating the cost of CRB.

REFERENCES

Akkary, H., Rajwar, R., and Srinivasan, S. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proc. of the 36th Annual Int'l Symp. on Microarchitecture*. 423–434.

Akkary, H., Rajwar, R., and Srinivasan, S. 2004. An analysis of a resource efficient checkpoint architecture. *ACM Trans. on Architecture and Code Optimization 1,* 4 (Dec.), 418–444.

Aragon, J., Gonzalez, J., Gonzalez, A., and Smith, J. 2002. Dual path instruction processing. In *Proc. of the 16th Annual Int'l Conf. on Supercomputing*. 220–229.

Balakrishnan, S., Rajwar, R., Upton, M., and Lai, K. 2005. The impact of performance asymmetry in emerging multicore architectures. In *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*. 506–517.

Burger, D. and Austin, T. 1997. The simplescalar tool set. *SIGARCH Computer Architecture News 25,* 3 (June), 13–25.

Canal, R. and Gonzalez, A. 2001. Reducing the complexity of the issue logic. In *Proc. of the 15th Annual Int'l Conf. on Supercomputing*. 312–320.

Ceze, L., Strauss, K., Tuck, J., Torrellas, J., and Renau, J. 2006. Cava: Using checkpoint-assisted value prediction to hide l2 misses. *ACM Trans. on Architecture and Code Optimization 3,* 2 (June), 182–208.

Chou, Y., Fung, J., and Shen, J. 1999. Reducing branch misprediction penalties via dynamic control independence detection. In *Proc. of the 13th Annual Int'l Conf. on Supercomputing*. 109–118.

Chung, J., Chafi, H., Minh, C., McDonald, A., Carlstrom, B., Kozyrakis, C., and Olukotun, K. 2006. The common case transactional behavior of multithreaded programs. In *Proc. of the 12th IEEE Int'l Symp. on High-Performance Computer Architecture*. 266–277.

Cristal, A., Martinez, J., and Valero, M. 2003. A case for resource-conscious out-of-order processors. *IEEE Computer Architecture Letters 2*.

Cristal, A., Ortega, D., Llosa, J., and Valero, M. 2004. Out-of-order commit processors. In *Proc. of the 9th IEEE Int'l Symp. on High-Performance Computer Architecture*. 48–59.

Cristal, A., Santana, O., Valero, M., and Martinez, J. 2004. Toward kilo-instruction processors. *ACM Trans. on Architecture and Code Optimization 1,* 4 (Dec.), 389–417.

Davis, J., Laudon, J., and Olukotun, K. 2005. Maximizing CMP throughput with mediocre cores. In *Proc. of the 14th Int'l Conf. on Parallel Architectures and Compilation Techniques*. 51–62.

Gandhi, A., Akkary, H., Rajwar, R., Srinivasan, S., and Lai, K. 2005. Scalable load and store processing in latency tolerant processors. In *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*. 446–457.

Gandhi, A., Akkary, H., and Srinivasan, S. 2004. Reducing branch misprediction penalty via selective branch recovery. In *Proc. of the 10th IEEE Int'l Symp. on High-Performance Computer Architecture*. 254–264.

Grochowski, E., Ronen, R., Shen, J., and Wang, H. 2004. Best of both latency and throughput. In *Proc. of the Int'l Conf. on Computer Design*. 236–243.

Grunwald, D., Klauser, A., Manne, S., and Pleszkun, A. 1998. Confidence estimation for speculation control. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture*. 122–131.

Hammond, L., Wong, V., Chen, M., Carlstrom, B., Davis, J., Hertzberg, B., Prabhu, M., Wijaya, H., Kozyrakis, C., and Olukotun, K. 2004. Transactional memory coherence and consistency. In *Proc. of the 31st Annual Int'l Symp. on Computer Architecture*. 102–113.

Heil, T. and Smith, J. 1996. Selective dual path execution. Tech. Rep. ECE, University of Wisconsin-Madison. Nov.

Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., and Roussel, P. 2001. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal* 1 (Feb.), 1–12.

Hwu, W. and Patt, Y. 1987. Checkpoint repair for high-performance out-of-order execution machines. *IEEE Transactions on Computers 36,* 12 (Dec.), 1496–1514.

Jacobsen, E., Rotenberg, E., and Smith, J. 1996. Assigning confidence to conditional branch predictions. In *Proc. of the 29th Annual Int'l Symp. on Microarchitecture*. 142–152.

KAHLE, J. 2005. The Cell processor architecture. In *Proc. of the 38th Annual Int'l Symp. on Microarchitecture*. 3.

KESSLER, R. 1999. The Alpha 21264 microprocessor. *IEEE micro 19,* 2 (Apr.), 24–36.

KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. 2005. Niagara: A 32-way multithreaded Sparc processor. *IEEE micro 25,* 2 (Mar.), 21–29.

KUNG, H. AND ROBINSON, J. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6,* 2 (June), 213–226.

LEE, J. AND SMITH, A. 1984. Branch prediction strategies and branch target buffer design. *IEEE Computer Magazine 17,* 1 (Jan.), 6–22.

LIPASTI, M. AND SHEN, J. 1996. Exceeding the dataflow limit via value prediction. In *Proc. of the 29th Annual Int'l Symp. on Microarchitecture*. 226–237.

MANNE, S., KLAUSER, A., AND GRUNWALD, D. 1998. Pipeline gating: Speculation control for energy reduction. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture*. 132–141.

MORAD, T., WEISER, U., KOLODNY, A., VALERO, M., AND AYGUAD, E. 2005. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Computer Architecture Letters 4*.

MOSHOVOS, A. 2003. Checkpointing alternatives for high performance, power-aware processors. In *Proc. of the 2003 Int'l Symp. on Low Power Electronics and Design*. 318–321.

MOSHOVOS, A. AND SOHI, G. 1999. Read-after-read memory dependence prediction. In *Proc. of the 32st Annual Int'l Symp. on Microarchitecture*. 177–185.

MOUDGILL, M., PINGALI, K., AND VASSILIADIS, S. 1993. Register renaming and dynamic speculation: an alternative approach. In *Proc. of the 26th Annual Int'l Symp. on Microarchitecture*. 202–213.

MUTLU, O., KIM, H., AND PATT, Y. 2005. Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *Proc. of the 38th Annual Int'l Symp. on Microarchitecture*. 233–244.

MUTLU, O., KIM, H., STARK, J., AND PATT, Y. 2005. On reusing the results of pre-executed instructions in a runahead execution processor. *IEEE Computer Architecture Letters 4*.

MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proc. of the 9th IEEE Int'l Symp. on High-Performance Computer Architecture*.

PALACHARLA, S., JOUPPI, N., AND SMITH, J. 1997. Complexity-effective superscalar processors. In *Proc. of the 24th Annual Int'l Symp. on Computer Architecture*. 206–218.

RAJWAR, R. AND GOODMAN, J. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. of the 34th Annual Int'l Symp. on Microarchitecture*.

RATTNER, J. 2005. Multi-core to the masses. In *Proc. of the 14th Int'l Conf. on Parallel Architectures and Compilation Techniques*. 3.

SARANGI, S., W. LIU, J. T., AND ZHOU, Y. 2005. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proc. of the 38th Annual Int'l Symp. on Microarchitecture*. 257–270.

SEZNEC, A. AND MICHAUD, P. 2006. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction-Level Parallelism 8*.

SHEN, J. AND LIPASTI, M. 2005. *Modern Processor Design, Fundamentals of Superscalar Processors*. Mcgraw-Hill.

SMITH, J. AND PLESZKUN, A. 1988. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers 37,* 5 (May), 562–573.

SPRACKLEN, L. AND ABRAHAM, S. 2005. Chip multithreading: Opportunities and challenges. In *Proc. of the 11th IEEE Int'l Symp. on High-Performance Computer Architecture*. 248–252.

SRINIVASAN, S., RAJWAR, R., AKKARY, H., GANDHI, A., AND UPTON, M. 2004. Continual flow pipelines. In *Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. 107–119.

TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. 2006. Cacti 4.0. Tech. Rep. HPL-2006-86, HP Laboratories Palo Alto. June.

YEAGER, K. 1996. The MIPS R10000 superscalar microprocessor. *IEEE micro 16,* 2 (Apr.), 28–40.

ZHANG, Y., RAUCHWERGER, L., AND TORRELLAS, J. 1999. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proc. of the 5th IEEE Int'l Symp. on High-Performance Computer Architecture*.

ZHOU, P., ONDER, S., AND CARR, S. 2005. Fast branch misprediction recovery in out-of-order superscalar processors. In *Proc. of the 19th Annual Int'l Conf. on Supercomputing*. 41–50.

Received August 2006; revised December 2006; accepted May 2007