An Efficient Low-Complexity Alternative to the ROB for Out-of-Order Retirement of Instructions

S. Petit, R. Ubal, J. Sahuquillo, P. López, and J. Duato
Department of Computer Engineering (DISCA)
Universidad Politénica de Valencia, Spain

Abstract

Current superscalar processors use a Reorder Buffer (ROB) to support speculation, precise exceptions, and register reclamation. Instructions are retired from this structure in program order, which may lead to significant performance degradation if a long latency operation blocks the ROB head. In this paper, a checkpoint-free out-of-order commit architecture is proposed, which replaces the ROB with a small structure called Validation Buffer (VB) from which instructions are retired as soon as their speculative state is resolved. An aggressive register reclamation mechanism targeted to this microarchitecture is also devised. Experimental results show that the VB microarchitecture is much more efficient than a ROB-based microprocessor. For example, a 32-entry VB provides similar performance to a 256-entry ROB, while reducing the utilization of other major processor structures.

1 Introduction

Current high-performance microprocessors execute instructions out of program order to exploit instruction level parallelism (ILP). To support speculative execution, provide precise exceptions, and register reclamation, a reorder buffer (ROB) structure is used. After being decoded, instructions are inserted in program order in the ROB, where they remain while being executed and until retired at the commit stage. The key to support speculation and precise exceptions is that instructions leave the ROB also in program order, that is, when they are the oldest ones in the pipeline. Consequently, if a branch is mispredicted or an instruction raises an exception, there is a guarantee that, when the offending instruction reaches the commit stage, all the previous instructions have already been retired and none of the subsequent ones have done it. Therefore, to recover from that situation, all the processor has to do is to abort the latter ones.

However, this behavior is conservative. When the ROB head is blocked, for instance, by a long latency instruction (e.g., a load that misses in the L2 cache), subsequent instructions cannot release their ROB entries. This happens even if these instructions are independent from the long latency one and they have been completed. In such case, since the ROB has a finite size and instruction decoding may continue, the ROB may become full, thus stalling the processor for a considerable number of cycles. Register reclamation is also handled in a conservative way, because physical registers are mapped for longer than their useful lifetime. In summary, both the advantages and the shortcomings of the ROB come from the fact that instructions are committed in program order.

A naive solution to address this problem is to enlarge the ROB size to accommodate more in-flight instructions. However, as ROB-based microarchitectures serialize the release of some critical resources at the commit stage (e.g., physical registers or store queue entries), these resources should be also enlarged. This resizing increases the cost in terms of area and power, and it might also impact the processor cycle [13].

To overcome this drawback, some solutions that commit instructions out of order have been devised. These proposals can be classified into two approaches depending on whether instructions are speculatively retired or not. Some proposals falling into the first approach, like [9], allow the retirement of the instruction obstructing the ROB head by providing a speculative value. Others, like [2] or [7], replace the normal ROB with alternative structures to speculatively retire instructions out of order. As speculation may fail, these proposals provide checkpoints to recover the processor to a correct state. Again, this implies the enlargement of some major microprocessor structures (register file [7] or load/store queue [2]), because completed instructions cannot free some critical resources until their associated checkpoint is released.

Regarding the non-speculative approach, Bell and Lipasti [4] propose to scan a few entries of the ROB (as many as the commit width allows), and those instructions satis-
fying certain conditions are allowed to be retired. None of these conditions imposes an instruction to be the oldest one in the pipeline to be retired. Hence, instructions can be retired out of program order. However, in this scenario, the ROB head may become fragmented after the commit stage, so it must be collapsed for the next cycle. Collapsing a large structure is costly in time and could adversely impact the cycle time, which makes this proposal unsuitable for large ROB sizes. In addition, as experimental results will show, this proposal has performance constraints due to the low number of instructions that can be scanned at the commit stage.

In this paper we propose the Validation Buffer microarchitecture, based on the non-speculative approach. Similarly to a ROB-based processor, a FIFO structure, called Validation Buffer (VB), provides support for speculative execution, exceptions, and register reclamation. However, an instruction is allowed to leave the VB once all previous branches and exceptions are resolved, that is, when its speculative status is resolved, regardless of whether the instruction is completed, only issued, or just decoded and not issued. Results show that the VB microarchitecture improves performance while requiring less resources, compared to a ROB-based architecture and to other non-speculative out-of-order retirement architectures.

The remainder of this paper is organized as follows. Section 2 describes the VB microarchitecture. Section 3 explores the potential of the proposed microarchitecture and its performance in a modern processor. Section 4 summarizes the related work. Finally, Section 5 presents some concluding remarks.

2 The VB Microarchitecture

The commit stage is typically the latest one of the processor pipeline. At this stage, a completed instruction updates the architectural machine state, frees the used resources and exits the ROB. The mechanism proposed in this paper allows instructions to be extracted early, as soon as they become non-speculative. Notice that these instructions may not be completed. Once they are completed, they will update the machine state and free the occupied resources. Therefore, instructions will exit the pipeline in an out-of-order fashion.

The necessary conditions to allow an instruction to be committed out of order are [4]: i) the instruction is completed; ii) WAR hazards are solved (i.e., a write to a particular register cannot be committed before all prior reads of that architected register have been completed); iii) previous branches are successfully predicted; iv) none of the previous instructions is going to raise an exception, and v) the instruction is not involved in memory replay traps. The first condition is straightforwardly met at the writeback stage by any proposal. The last three conditions are handled by the Validation Buffer (VB) structure, which replaces the ROB and contains those instructions whose speculative state is still unknown. The second condition is fulfilled by the devised register reclamation method (see Section 2.1).

The VB deals with the speculation-related conditions (iii, iv and v) by decomposing code into fragments or epochs. The epoch boundaries are defined by instructions that may initiate an speculative execution, referred to as epoch initiators (e.g., branches or potentially exception raiser instructions). Only those instructions whose previous epoch initiators have completed and confirmed their prediction are allowed to modify the machine state. We refer to these instructions as validated instructions.

Instructions reserve an entry in the VB in program order when they are dispatched. Epoch initiator instructions are labeled as such in the VB. When an epoch initiator detects a mispeculation, all following instructions are canceled. Likewise, when an uncompleted epoch initiator reaches the VB head, it cannot be extracted until completion. When it completes, it leaves the VB and updates the machine state, if any. Non epoch-initiator instructions that reach the VB head can leave it regardless of their execution state. That is, they can be either dispatched, issued, or completed. However, only validated instructions update the machine state, while canceled instructions are drained to free the resources they occupy (see Section 2.2).

Let us study a typical problem where the out-of-order retirement can help to improve performance. Figure 1 shows the source and assembler codes to compute the product of vectors $v_1$ and $v_2$. The assembler code was generated by the SimpleScalar toolset [5] using the O3 optimization flag. When this code runs on a ROB-based microprocessor, the first load that misses in the data cache blocks the ROB head, stalling instruction decoding after the ROB fills up. The presence of many memory instructions with potential cache misses can worsen the problem —here two loads and one store per iteration. In particular, when running this loop on a modern ROB-based microprocessor (see Section 3.2) the decode is stalled by about 77% of the execution time, clearly dropping the performance. In contrast, this percentage is nearly negligible, by about 0.02%, when the program runs in a processor implementing the VB microarchitecture. By unclogging the ROB, the pressure is moved to the instruction queue and the load-store queue, which are the new, but less restrictive, bottlenecks. However, as the conditions applied to release these resources are more relaxed, the IPC raises from 0.70 to 1.05 on a processor with the characteristics shown in Table 2.

The proposed microarchitecture can enforce a variable number of epoch initiators. The smallest set of epoch initiators that must be enforced is composed by those instructions affecting the tree speculation-related conditions,
namely branches and memory reference instructions (i.e., the address calculation part). In other words, branch speculation, memory replay traps (see Section 2.4) and exceptions related with address calculation (e.g., page faults, invalid addresses) are supported by design. In order to support precise floating-point exceptions, all floating-point instructions should be considered epoch initiators. However, notice that a large set of epoch initiators might reduce the performance benefits of the VB microarchitecture, by incurring more pipeline stalls due to a full VB. To help this, user definable flags can be used to enable or disable support for precise exceptions. If the corresponding flag is enabled, instructions generating exceptions can force a new epoch when decoded.

The management of hardware interrupts does not vary from a ROB-based processor. On such an event, all instructions in the VB are canceled, then the recovery mechanism squashes them as explained in Section 2.2), and finally the interrupt service routine is launched.

2.1 Register Reclamation

Typically, modern microprocessors free a physical register when the instruction that renames the corresponding logical register commits [11]. Then, the physical register index is placed in a list containing the free physical registers available for new producers.

Waiting until the commit stage to free a physical register is easy to implement, but it constitutes a conservative approach; a sufficient condition is that all consumers have read the corresponding value. Therefore, this method does not efficiently handle registers, as they are usually mapped for longer than their useful lifetime. In addition, this method requires keeping track of the oldest instruction in the pipeline. As this instruction may have already left the VB, this method is unsuitable for our proposal.

For these reasons, we devise a register reclamation strategy based on the counter method [11], targeted for the VB microarchitecture. The hardware components used in this scheme, as shown in Figure 2, are:

- **Frontend Register Alias Table (RATfront)**. This table maintains the current mapping for each logical register and is accessed at the rename stage. The table is indexed by the source logical register to obtain the corresponding physical register identifier. Additionally, each time a new physical register is mapped to a destination logical register, the RATfront is updated. As in conventional ROB-based architectures, this table is updated in program order, thus the renaming mechanism avoids WAW hazards.

- **Retirement Register Alias Table (RATret)**. The RATret is updated by instructions exiting the VB. Like the RATfront, this table is also updated in program order, but only by validated (non-speculative) instructions. Therefore, the RATret contains a valid state of the register mappings, which are used on recovery to disable the updates performed by canceled instructions in the RATfront.

- **Register Status Table (RST)**. The RST is indexed by a physical register number and contains three fields, labeled as pending_readers, valid_remapping and completed, respectively. The pending_readers field contains the number of decoded instructions that consume the corresponding physical register, but have not read it yet. This value is incremented as consumers enter the decode stage, and decremented when they are issued to the execution units. The second and third fields are composed each by a single bit. The valid_remapping bit is set when the associated logical register has been definitively remapped to a new physical register, that is, when the instruction remapping the destination logical register leaves the VB as validated. Finally, the completed bit indicates that the instruction producing that value has completed execution, writing the result to the corresponding physical register.

With this representation, a free physical register p can be easily identified when the corresponding entry in the RST contains the triplet \( \{0,1,1\} \). A 0 in pending_readers guarantees that no instruction in the pipeline (i.e., after the...
rename stage) will read the content of p. Next, a 1 in valid_remapping implies that no new instruction will enter the pipeline (i.e., the rename stage) and read the content of p, because p has been remapped by a valid instruction. Finally, a 1 in the completed field denotes that no instruction in the pipeline is going to overwrite p. These conditions ensure that a specific physical register can be safely reallocated for a subsequent renaming. On the other hand, the triplet \{0,0,1\} is used for the initial mappings of the architectural logical registers. It denotes a busy register, with no pending readers, not unmapped by a valid instruction, and appearing as the result of a completed (and valid) operation.

Table 1. Renaming actions with no mispeculation.

<table>
<thead>
<tr>
<th>Event</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction l with physical register (p,r,p) as source operated, enters the rename stage.</td>
<td>RST[p] pending readers +</td>
</tr>
<tr>
<td>l enters the rename stage and claims a p,r to map an output logical register l.</td>
<td>Find a free p,r, say p, RAT_{old}[l] = p.</td>
</tr>
<tr>
<td>l is issued and reads p,r,p.</td>
<td>RST[p] pending readers --</td>
</tr>
<tr>
<td>l finishes execution, writing the result over p,r,p.</td>
<td>RST[p] completed = 1</td>
</tr>
<tr>
<td>l exits the VB as validated, l is the logical destination register, p,r,p and p,l are the current mapping of l, and p,r,p and p,l are the previous mapping of l.</td>
<td>RST[p] valid_remapping = 1 RAT_{old}[l] = p</td>
</tr>
</tbody>
</table>

2.2 Recovery Mechanism

The recovery mechanism always involves restoring both the RAT_{front} and the RST tables.

Register Alias Table Recovery. Current microprocessors employ different methods to restore the renaming information when a mispeculation or exception occurs. The method presented in this paper uses the two renaming tables, RAT_{front} and RAT_{ret}, explained above, similarly to the Pentium 4 [8].

RAT_{ret} contains a delayed copy of a validated RAT_{front}. That is, it matches the RAT_{front} at the time the validated instruction was renamed. So, a simple method to implement the recovery mechanism (restoring the mapping to a precise state) is to wait until the offending instruction reaches the VB head, and then copy the RAT_{ret} into the RAT_{front}. Alternative implementations can be found in [2]. Since only validated instructions update the RAT_{ret}, the updates performed by canceled instructions to the machine state are effectively undone. In other words, new instructions from the correct path will not be renamed to consume the values produced by canceled instructions, even if those canceled instructions remain in the instruction queues and the functional units.

Register Status Table Recovery. The recovery mechanism must also undo the modifications performed by the canceled instructions in any of the three fields of the RST.

Concerning the valid_remapping field, we describe two possible techniques to restore its values. The first technique squashes from the VB those entries corresponding to instructions younger than the offending instruction when the latter reaches the VB head. At that point, the RAT_{ret} contains the physical registers identifiers used to restore the correct mapping. The remaining physical registers must be freed. To this end, all valid_remapping entries are initially set to 1 (necessary condition to be freed). Then, the RAT_{ret} is scanned looking for physical registers whose valid_remapping entry must be reset.

The second technique relies on the following observation. Only those physical registers allocated (i.e., mapped to a logical register) by instructions younger than the offending one (i.e., the canceled instructions) must be freed. All these instructions are located in the VB when the offending instruction reaches the VB head. Thus, we only need to drain them from the VB. Drained instructions must set to 1 the valid_remapping entry of their current mapping. In this case, the valid_remapping flag is used to free the registers allocated by the current mapping, instead of the previous mapping like in normal operation. While the canceled instructions are being drained, new instructions can enter the renaming stage, provided that the RAT_{front} has been already recovered. Therefore, the VB draining can be overlapped with new processor operations.

Regarding the pending_readers field, it cannot be just reset, as there can already be valid pending readers in the issue queue. Thus, each pending_readers entry must be decremented as many times as the number of canceled pending readers for the corresponding physical register. To this end, the issue logic must allow detection of those instructions younger than the offending instruction, that is, the canceled pending readers. This can be implemented by using a bitmap mask in the issue queue to identify which instructions are younger than a given epoch initiator [17]. The canceled instructions must be drained from the issue queue to correctly handle (i.e., decrement) their pending_readers entries. Notice that this logic can be also used to handle the completed field, by enabling a canceled instruction to set the entry of its destination physical register. Alternatively, it is also possible to simply let the canceled instructions exe-
2.2 Validation Process

The validation process starts with the decoupled store issue stage, when the commit logic issues a signal that the store has been completed. The validation process consists of two stages: first, the machine waits until the physical register currently mapped to its destination logical register has been resolved. Then, the processor recovers to a correct state by copying the ROB entry of the VB head. During the recovery process, the processor must restore the status of the ROB, the register renaming table, and the invalidation bitmap. In addition, the processor must restore the status of the ROB even if any store address is unresolved yet. As speculation may fail, processors must provide some mechanism to detect and recover from load mispeculation. For instance, loads issued speculatively can be placed in a special buffer called finished load buffer (FLB). The entry of this buffer is released when the load commits. On the other hand, when a store commits, the FLB is associatively searched for aliasing loads which were speculatively issued. Like in a ROB-based implementation, all loads in the buffer are younger than the store commits, both the load and subsequent instructions must be re-executed.

2.3 Working Example

Figure 3 shows an example that illustrates different scenarios of register renaming and mispeculation recovery. The example represents a validation buffer with 12 instructions belonging to 3 different epochs. Instructions can be in one of the following states: dispatched but not issued, issued but not yet completed, and completed. Unlike normal ROBs, no control information about these states is stored in the VB. Instead, the only information required is whether the epoch is validated or canceled.

In the example, assume that the three epochs have just been resolved, epochs 0 and 1 as validated and epoch 2 as canceled. Thus, only instructions belonging to epochs 0 and 1 should be allowed to update the machine state.

Firstly, instructions belonging to epoch 0 leave the VB. As this epoch has been validated, each instruction will update the RATret and set the valid_remapping bit of the physical register previously mapped to its destination logical register. Since these instructions are completed, the VB is the last machine resource they consume.

Then, instructions of epoch 1 leave the VB, two completed, one issued and the other one dispatched but not yet issued. The RATret table and the valid_remapping bit are handled as above, regardless of the instruction status. However, the non completed instructions will remain in the pipeline until they are complete.

Finally, instructions belonging to epoch 2 leave the VB as canceled. These instructions must not update the RATret, but they must set the valid_remapping bit of the physical register currently mapped to its destination logical register. In addition, the processor must restore RATfront, re-execute the epoch initiator (if needed), and fetch the correct instructions. To this end, the machine waits until the epoch initiator that has triggered the cancellation reaches the VB head. Then, the processor recovers to a correct state by copying the RATret to the RATfront, and resumes execution from the correct path. The RST state is recovered as explained in Section 2.2.

2.4 Memory Model

To correctly follow the uniprocessor memory model, it must be ensured that load instructions get the data produced by the newest previous store matching its memory address. A key component to improve performance in such a model, is the load/store queue (LSQ). In the VB microarchitecture, as done in some current microprocessors, memory reference instructions are internally split by the hardware into two instructions when they are decoded and dispatched: the memory address calculation, which is considered as an epoch initiator, and the memory operation itself. The former reserves a VB entry when it is dispatched while the latter reserves an entry in the LSQ. To free its corresponding LSQ entry, any memory reference instruction must be validated and completed. In addition, a store must be the oldest instruction in the LSQ.

Load bypassing is the main technique applied to the LSQ to improve processor performance. This technique permits loads to early execute by advancing previous stores in their access to the cache. Load bypassing can be speculatively performed by allowing loads to bypass previous stores in the LSQ even if any store address is unresolved yet. As speculation may fail, processors must provide some mechanism to detect and recover from load mispeculation. For instance, loads issued speculatively can be placed in a special buffer called finished load buffer (FLB). The entry of this buffer is released when the load commits. On the other hand, when a store commits, the FLB is associatively searched for aliasing loads (note that all loads in the buffer are younger than the committing store). On a hit, and after the load commits, both the load and subsequent instructions must be re-executed.

An FLB can be quite straightforwardly implemented in the VB architecture with no additional complexity. In this case, a load will release its entry in the FLB when it leaves the VB. When a store leaves the VB, the address of all previous memory instructions and its own address have been resolved. Thus, it can scan the FLB for aliasing loads which were speculatively issued. Like in a ROB-based implementation, all loads in the buffer are younger than the store. On a hit, the recovery mechanism should be triggered as soon as the mispeculated load exits the VB. Notice that this implementation allows mispeculation to be early detected. Finally, store-load forwarding and load replay traps are also supported by the VB by using the same hardware available in current microprocessors.

Regarding multiprocessor environments, such as CMPs or SMPs, the memory consistency model specifies the global ordering of memory operations. Sequential consistency (SC) is a strict memory model that gives an intuitive interface to the programmer, but its implementation usually forces processors to keep very large instruction win-
dows in order to hide latencies of accesses to remote caches. This can harden other performance optimizations, so some commercial architectures implement weaker memory consistency models (e.g., IBM 360 or DEC Alpha). However, some efficient SC implementations on ROB-based processors (e.g., using history buffers or storing speculative results in the memory subsystem [6]) can be straightforwardly applied on top of a VB-based processor, while keeping the performance speedups from out-of-order retirement of instructions. The description and evaluation of a sequentially consistent VB-based multiprocessor is planned as future work, but is out of the scope of this paper.

3 Experimental Framework and Performance Evaluation

This section presents the simulation environment used to evaluate the performance of the VB microarchitecture. For comparison purposes, two ROB-based proposals without checkpointing have been modeled. The first one retires instructions in program order (hereafter the IOC processor) and the other commits instructions out of order as proposed in [4] (from now on, the Scan processor). As the VB microarchitecture cancels instructions as soon as they leave the VB, the recovery mechanism for the ROB-based architectures is also triggered at the writeback stage for fair comparison purposes. In addition, the recovery penalty has been assumed to be equal in all the simulated microarchitectures, regardless of the ROB/VB size. Notice that this assumption provides conservative results for the VB microarchitecture, since small VB sizes would have a shorter recovery time [2].

The analyzed architectures are modeled on top of an extensively modified version of the SimpleScalar toolset [5] with separated ROB, instruction queues, and register file structures. The pipeline is also enlarged with separated decode, rename, and dispatch stages. Both load speculation and store-load replay are modeled in all the evaluated approaches. Table 2 summarizes the architectural parameters used through the experiments. Performance is analyzed using a moderate value of memory latency (200 cycles) because processor frequency is not growing at the same rate as in the past. However, as it can be deduced from the results, if longer memory latencies were considered, performance gains provided by the VB would be higher, as ROB stalls would be longer.

Experimental results are obtained using the Alpha ISA and running the SPEC2000 benchmark suite [11]. Both integer (SpecInt) and floating-point (SpecFP) benchmarks are run with the ref input sets and statistics are gathered by using single simulation points [16].

3.1 Exploring the Potential of the VB Microarchitecture

To explore how the VB size impacts on performance, the remaining major processor structures (i.e., instruction queue, register file, and load/store queue) are firstly assumed to be unbounded. Figure 4 shows the average IPC (i.e., harmonic mean) for the SpecFP and SpecInt benchmarks when varying the ROB/VB size. Results show that the improvements provided by the VB microarchitecture are much higher for floating-point benchmarks than for the integer ones. The reason is that the ROB size is not the main performance bottleneck in integer applications, mainly because of the high percentage of mispredicted branches (as also stated in previous work [9, 4]). Thus, hereafter, performance analysis will focus on floating-point workloads. Concerning floating-point benchmarks, the highest IPC difference appears with the smallest VB/ROB size (i.e., 32 entries), and these differences get smaller as the VB/ROB size increases.

3.2 Exploring the Behavior in a Modern Microprocessor

This section explores the behavior of the VB microarchitecture while dimensioning the four major microprocessor structures closely resembling the ones implemented in a modern microprocessor, namely the Intel Pentium 4 processor: a 32-entry instruction queue, a 64-entry load/store queue, 128 physical registers, and a 128-entry ROB (IOC and Scan models). The VB size ranges from 8 to 128 entries.

As shown in Figure 5, the VB microarchitecture is much more efficient since it achieves with only 16 entries higher IPC than the other models, on average. On the other hand, the use of VBs larger than 32 entries provides minor performance benefits.

---

Table 2. Machine parameters.

<table>
<thead>
<tr>
<th>Microprocessor Core</th>
<th>Out of order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue policy</td>
<td>Hybrid gShare/bimodal</td>
</tr>
<tr>
<td>Branch predictor type</td>
<td>gShare: 16-bit global history + 64K 2-bit counters</td>
</tr>
<tr>
<td></td>
<td>Bimodal: 2K 2-bit counters</td>
</tr>
<tr>
<td></td>
<td>Choice predictor: 1K 2-bit counters</td>
</tr>
<tr>
<td>Fetch, issue, commit bandwidth</td>
<td>4 inst/cycle</td>
</tr>
<tr>
<td>Integer ALUs / multipliers-dividers</td>
<td>4 / 1</td>
</tr>
<tr>
<td>FP ALUs, multipliers-dividers</td>
<td>4 / 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main memory</td>
</tr>
<tr>
<td>1.1 data cache</td>
</tr>
<tr>
<td>1.2 data cache</td>
</tr>
</tbody>
</table>
Although the VB microarchitecture does not benefit to the same extent from integer and floating-point benchmarks, simulations for integer applications (not included here) show that a 32-entry VB provides the same performance than a 128-entry ROB. Thus, one can conclude that integer benchmarks performance is not significantly affected when reducing the VB size.

4 Related Work

Long latency operations constrain the output rate of the ROB, hence microprocessor performance. Several architectural optimization have been recently proposed to deal with this problem [12, 9, 7, 2]. All these proposals permit to retire instructions in a speculative way when a long latency operation blocks the ROB head. These solutions introduce specific hardware to periodically checkpoint the architectural state and guarantee correct execution. When a misprediction occurs, the processor rolls back to the checkpoint, discarding current computations.

In [12], Mutlu et al proposed the run-ahead architecture for out-of-order microprocessors. In this proposal, the state of the architectural register file is checkpointed every time a long-latency memory operation blocks the ROB head. When a checkpoint is performed, the processor enters in the run-ahead mode until the long latency instruction finishes. Meanwhile, a bogus value is distributed for dependent instructions to continue. However, this execution mode does not allow instructions to update the architectural state. When the long latency operation finishes, the processor rolls back to the checkpoint and re-executes the instructions in the normal mode, discarding previous results. The run-ahead execution provides useful prefetching requests (both instructions and data) as well as effective train for branch predictors.

In [9], Kirman et al propose the checkpointed early load retirement mechanism which has certain similarities with the previous proposal. To unclog the ROB when a long-latency load instruction blocks the ROB head, a predicted value is provided for those dependent instructions to allow them to continue. When the value of the load is fetched from memory, it is compared with the predicted one. On a misprediction, the processor rolls back to the checkpoint.

In [7], Cristal et al propose to replace the ROB structure with a mechanism to perform checkpoints at specific instructions. This mechanism uses a CAM structure for register mapping purposes, which is also in charge of freeing physical registers. Stores must wait in the commit stage to modify the machine state until the closest previous checkpoint has committed. In addition, instructions taking a long time to issue (e.g., instructions dependent from a load) are moved from the instruction queue to a secondary buffer, thus freeing resources that can be used by other instructions. These instructions must be re-inserted into the instruction queue when the instruction they are dependent on has completed (e.g., the load data has already been fetched). This problem has also been tackled by Akkary et al in [2].

In [10], Martinez et al propose an in-order retirement mechanism which identifies irreversible instructions for early release of resources. Unlike the VB microarchitecture, this proposal retires instructions in order. Also, checkpoints are required to roll back the processor to a correct state. In [4], a checkpoint-free approach is presented. The proposal scans the n oldest entries of the ROB to select instructions ready to be retired. However, scanning the ROB is unsuitable for large sizes, and other resources are handled as a typical in-order commit architecture, without any focus on improving their usage.

Finally, some proposals alleviate the performance degradation caused by ROB blocking by enlarging the major microprocessor structures or efficiently managing them [15, 3, 14].

5 Conclusions

In this paper, we have presented the Validation Buffer microarchitecture, a novel out-of-order retirement approach with support for speculation and precise exception handling. By substituting the ROB with a Validation Buffer (VB), this proposal retires instructions in a non-speculative
way, making checkpoints unnecessary.

On a machine with similar characteristics to a modern microprocessor, the VB architecture with a 32-entry VB achieves performance close to an in-order retirement architecture with a 256-entry ROB. The benefits of the proposed microarchitecture do not only apply to the complexity of the VB buffer, but also to the complexity of the remaining major processor structures. Specifically, results show that the utilization of resources other than the instruction queue is either maintained or decreased in the VB architecture, while performance increases.

In summary, the VB architecture behaves a complexity-effective approach, aimed at either increasing performance with the same hardware cost, or reducing hardware complexity for a given performance.

References


