A Complexity-Effective Out-of-Order Retirement Microarchitecture

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

Abstract

Current superscalar processors commit instructions in program order by using a reorder buffer (ROB). The ROB provides support for speculation, precise exceptions, and register reclamation. However, committing instructions in program order may lead to significant performance degradation if a long latency operation blocks the ROB head.

Several proposals have been published to deal with this problem. Most of them retire instructions speculatively. However, as speculation may fail, checkpoints are required in order to rollback the processor to a precise state, which requires both extra hardware to manage checkpoints and the enlargement of other major processor structures, which in turn might impact the processor cycle.

This paper focuses on out-of-order commit in a nonspeculative way, thus avoiding checkpointing. To this end, we replace the ROB with a validation buffer (VB) structure. This structure keeps dispatched instructions until they are nonspeculative or mispeculated, which allows an early retirement. By doing so, the performance bottleneck is largely alleviated. An aggressive register reclamation mechanism targeted to this microarchitecture is also devised.

As experimental results show, the VB structure is much more efficient than a typical ROB since, with only 32 entries, it achieves a performance close to an in-order commit microprocessor using a 256-entry ROB.

I. Introduction

Current high-performance microprocessors execute instructions out-of-order to exploit instruction level parallelism (ILP). To support speculative execution, provide precise exceptions, and register reclamation, a reorder buffer (ROB) structure is used [1]. After being decoded, instructions are inserted in program order in the ROB, where they are kept 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.

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), 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 a case, since the ROB has a finite size, as long as instruction decoding continues, the ROB may become full, thus stalling the processor for a valuable 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 [2].

To overcome this drawback, some solutions that commit instructions out of order have been published. 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 [3], allow the retirement of the instruction obstructing the ROB head by providing a speculative value. Others, like [4] or [5], replace the normal ROB with alternative structures to speculatively retire instructions out of order. As speculation may fail, these proposals need to provide a mechanism to recover the processor to the correct state. To this end, the architectural state of the machine is checkpointed. Again, this implies the enlargement of some major microprocessor structures, for instance, the register file [5] or the load/store queue [4], because completed instructions cannot free some critical resources until their associated checkpoint is released.

Regarding the nonspeculative approach, Bell and Lipasti [6] propose to scan a few entries of the ROB, as many as the commit width, and those instructions satisfying 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, and thus it must be collapsed for the next cycle. Collapsing a large structure is costly in time and could adversely impact the microprocessor cycle, which makes this proposal unsuitable for large ROB sizes. In addition, as experimental results will show, this proposal has performance constraints due to the limited number of instructions that can be scanned at the commit stage.

In this paper we propose the Validation Buffer (VB) microarchitecture, which is also based on the nonspeculative approach. This microarchitecture uses a FIFO-like table structure analogous to the ROB. The aim of this structure is to provide support for speculative execution, exceptions, and register reclamation. While in the VB, instructions are speculatively executed. Once all the previous branches and previous exceptions are resolved, the execution mode of the instructions changes either to nonspeculative or mispeculated. At that point, instructions are allowed to leave the VB. Therefore, instructions leave the VB in program order but, unlike the ROB, they do not remain in the VB until retirement. Instead, they remain in the VB only until the execution mode of such instruction is resolved, either nonspeculative or mispeculated. Consequently, instructions leave the VB at different stages of their execution: completed, issued, or just decoded and not issued. In particular, a long latency memory reference instruction could leave the VB as soon as its memory address is successfully calculated and no page fault has been risen. This paper discusses how the VB microarchitecture works, focusing on how it deals with register reclamation, speculative execution, and exceptions.

The main contribution of this paper is the proposal of an aggressive out-of-order retirement microarchitecture without checkpointing. This microarchitecture decouples instruction tracking for execution and for resource reclamation purposes. The proposal clearly outperforms the existing one [6] that, like ours, does not perform checkpoints, since it achieves more than twice its performance using smaller VB/ROB sizes. On the other hand, register reclamation cannot be handled as done in current microprocessors [7], [8], [9] because no ROB is used. Therefore, we devise an aggressive register reclamation method targeted to this architecture. Experimental results show that the VB microarchitecture increases the ILP while requiring less complexity in some major critical resources like the register file and the load/store queue.

This paper presents the VB microarchitecture and its performance evaluation results. Indeed,

this paper analyzes in a deep and extensive way, complexity-performance tradeoffs, including two main studies: i) an analysis of the complexity of the major processor structures required to achieve a given performance level, and ii) an analysis of the impact on performance of different pipeline widths. Results show that the VB microarchitecture performs better using less resources, and that it can outperform in-order retirement architectures even when using narrower pipeline widths. The latter result brings important implications, since the pipeline width has a strong impact on processor complexity.

The remainder of this paper is organized as follows. Section II describes the VB microarchitecture. Section III explores the potential of the proposed microarchitecture and its performance in a modern processor. Section IV analyzes the performance obtained when varying the complexity requirements. Section V summarizes the related work. Finally, Section VI presents some concluding remarks.

II. The VB Microarchitecture

The commit stage is typically the latest one of the microarchitecture 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 retired early, as soon as it is known that they are nonspeculative. 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-or-order are [6]: i) the instruction is completed; ii) WAR hazards are solved (i.e., a write to a particular register cannot be permitted to commit 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 by any proposal at the writeback stage. The last three conditions are handled by the Validation Buffer (VB) structure, which replaces the ROB and contains the instructions whose conditions are not known yet. The second condition is fulfilled by the devised register reclamation method (see Section II-A).

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 when they are dispatched, that is, they enter in program order in the VB. Epoch initiator instructions are marked as such in the VB. When an epoch initiator detects a mispeculation, all the following instructions are canceled. When an instruction reaches the VB head, if it is an epoch initiator and it has not completed execution yet, it waits. When it completes, it leaves the VB and updates 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 those not canceled instructions (i.e., validated) will update the machine state. On the other hand, canceled instructions are drained to free the resources they occupy (see Section II-B).

Notice that when an instruction leaves the VB, if it is already completed, it is not consuming execution resources in the pipeline. Thus, it is analogous to a normal retirement when using the ROB. Otherwise, unlike the ROB, the instruction is *retired* from the VB but it remains in the pipeline until it is completed.

The proposed microarchitecture can support a wide range of epochs initiators. At least, epoch initiators according to the three speculative related conditions are supported. Therefore, branches and memory reference instructions (i.e., the address calculation part) act as epoch initiators. In other words, branch speculation, memory replay traps (see Section II-D) and exceptions related with address calculation (e.g., page faults, invalid addresses) are supported by design.

It is possible to include more instructions in the set of epoch initiators. For instance, in order to support precise floating-point exceptions, all floating-point instructions should be included in this set. As instructions are able to leave the VB only when their epoch initiators validate their epoch, a high percentage of epoch initiators might reduce the performance benefits of the VB microarchitecture. User definable flags can be used to enable or disable support for precise exceptions. If the corresponding flag is enabled, the instruction that may generate a given type of exception will force a new epoch when it is decoded. In fact, a program could dynamically enable or disable these flags during its execution. For instance, it can be enabled when the compiler suspects that an arithmetic exception may be raised.

The design and performance evaluation of a VB-based multiprocessor system is out of the scope of this paper. Nevertheless, as multicore processors are becoming an important segment of the market, we briefly discuss how the proposed microarchitecture could be adapted for supporting stricter memory consistency models, like the sequential memory consistency model [11] (see Section II-E).

A. Register Reclamation

Typically, modern microprocessors free a physical register when the instruction that renames the corresponding logical register commits [12]. Then, the physical register index is placed in the list that contains the free physical registers available for new producers.

Waiting until the commit stage to free a physical register is easy to implement but conforms to a conservative approach; a sufficient condition is that all consumers have read the corresponding value. Therefore, this method does not efficiently use registers, as they can be 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.

Due to these reasons, we devised a register reclamation strategy based on the counter method [13], [12] targeted for the VB microarchitecture. The hardware components used in this scheme, as shown in Figure 1, are:

Frontend Register Alias Table (RAT_{front}). 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 RAT_{front} is updated.

Retirement Register Alias Table (RAT_{ret}). The RAT_{ret} table is updated as long as instructions exit the VB. This table contains a precise state of the register map, as only those validated instructions leaving the VB are allowed to update it.

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

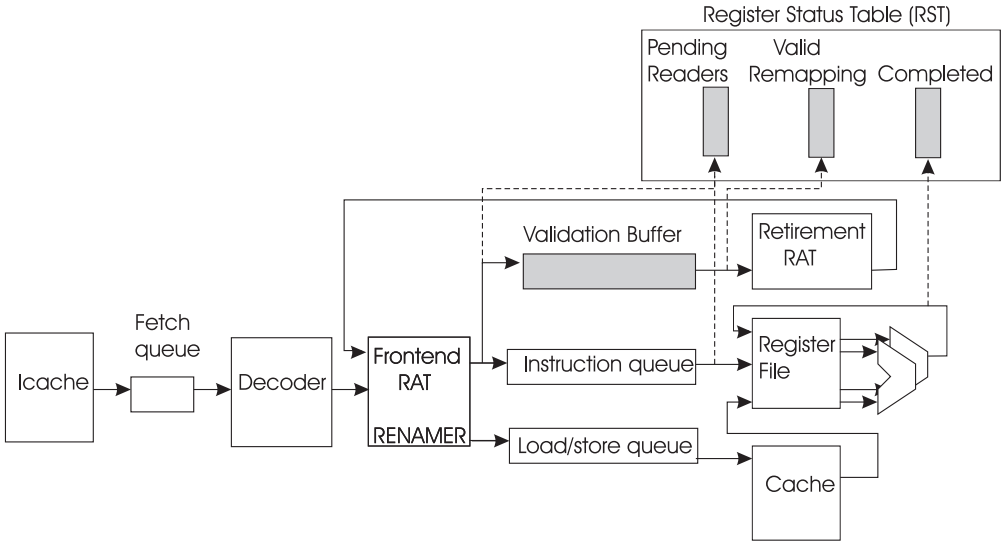


Fig. 1. VB microarchitecture block diagram.

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 that remapped the logical register has left the VB as validated. Finally, the *completed* bit indicates that the instruction producing its 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 already in the pipeline 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 unmapped 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, a triplet $\{0,0,1\}$ 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 I shows different situations which illustrate the dynamic operation of the proposed register reclamation strategy in a non-speculative mode, as well as the updating mechanism of RST, RAT_{front} and RAT_{ret} tables. In particular, the last row of the table points out the tasks

TABLE I

RST ACTIONS FOR DIFFERENT PIPELINE EVENTS.

Event	Actions
An instruction I enters the *rename* stage and has physical register (p.r.) p as source operand.	$RST[p].pending\_readers++$
I enters the *rename* stage and reclaims a p.r. to map an output logical register l.	Find a free p.r. p and set $RST[p] = \{0,0,0\}$, $RAT_{front}[l] = p$.
I is issued and reads p.r. p.	$RST[p].pending\_readers--$
I finishes execution, writing the result over p.r. p.	$RST[p].completed = 1$
I exits the VB as validated. l is the logical destination of I, p.r. p is the current mapping of l, and p.r. p' was the previous mapping of l.	$RST[p'].valid\_remapping = 1$, $RAT_{ret}[l] = p$.

to be performed when an instruction leaves the VB as validated. To perform these tasks, the VB must keep track of: i) the destination logical register l of the instruction, ii) the physical register p mapped to l by the instruction, and iii) the physical register p' that was previously mapped to l. Notice that this information is also stored in a typical ROB based architecture.

B. 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 [9].

RAT_{ret} contains a delayed copy of a *validated* RAT_{front}. That is, it matches the RAT_{front} table at the time the exiting (as valid) 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 copying RAT_{ret} into RAT_{front}. Alternative implementations can be found in [4].

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 this one reaches the VB head. At that point, the RAT_{ret} contains the physical registers identifiers that we use 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 the physical registers that were 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 will be inside the VB when the offending instruction reaches the VB head. Thus, we only need to drain them from the VB. These drained instructions must set to 1 the *valid_remapping* entry of their current mapping. Notice that 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 subsequent new processor operations.

Regarding to 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 as the number of canceled pending readers for the corresponding physical register. To this end, the issue logic must allow to detect 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 branch [14]. 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 execute -but without writing their result anywhere- to correctly handle the *pending_readers* and *completed* fields.

C. Working Example

To illustrate different scenarios that could require triggering the recovery mechanism as well as register reclamation handling, we use the example shown in Figure 2. This example shows 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 RAT_{ret} 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 RAT_{ret} table and the *valid_remapping* bit are handled as above, regardless 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 RAT_{ret}, but they must set the *valid_remapping* bit of the physical register currently mapped to its destination logical register. In addition, the processor must set the correct RAT_{front}, re-execute the epoch initiator (if needed), and fetch the correct instructions. To this end, the machine waits until the epoch initiator instruction that has triggered the cancellation reaches the VB head. Then, the processor recovers to a correct state by copying the RAT_{ret} to the RAT_{front} and resumes execution from the correct path. The RST state is recovered as explained in Section II-B.

D. Uniprocessor 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

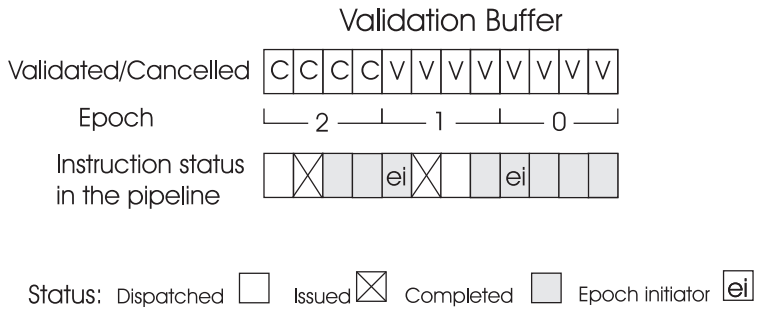


Fig. 2. Instruction status and epochs.

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, to free its entry, a store instruction must be the oldest memory reference 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 the finished load buffer [15]. The entry of this buffer is released when the load commits. On the other hand, when a store commits, a search is performed in this buffer looking for aliasing loads (note that all loads in the buffer are younger than the store). If aliasing is detected, when the mispeculated load commits, both the load and subsequent instructions must be re-executed.

A finished load buffer can be quite straightforwardly implemented in the VB, with no additional complexity. In this case, a load instruction will release its entry in the finished load buffer 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 already search the mentioned buffer looking for aliasing loads speculatively issued. As 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.

E. Memory consistency model

In the LSQ devised for the VB microarchitecture, stores cannot be issued before older memory instructions. However, as explained above, loads are allowed to bypass both loads and stores. That is, we natively support a weak memory consistency model, similar to the ones available in some commercial multiprocessors like the PowerPC, SPARC RMO, and Alpha.

Conservatively, to support a sequential memory consistency model, loads should not be allowed to be issued before older memory operations have completed. However, assuming an interconnection network like the split-transaction bus implemented in the Sun Microsystem's Gigaplane [16], a memory operation can be issued without violating the sequential memory consistency model provided that all previous instructions requiring a bus transaction have already been issued on the bus [17] (even if their data is still not ready). Similar enhancements are also possible when using directory-based coherence protocols [18].

A more performance-effective implementation of the sequential memory consistency model allows loads to speculatively bypass older local memory operations (loads or stores) without enforcing the above condition. In such a case, a mispeculation is detected when a remote store aliases with any local load issued speculatively. This can be done by using the hardware already available to support the uniprocessor memory model [19].

To recover from such mispeculations, the VB should not validate memory address calculations of loads until all previous memory operations result either in a cache hit or they generate a bus transaction. Notice that memory latency could be hidden by requesting a remote cache block even before the processor suffers a read miss in its cache for that block [18]. A deep study of these enhancements and their impact on performance is planned as for future work.

III. Experimental Framework and Performance Evaluation

This section presents the simulation environment and the benchmarks used to evaluate the performance of the VB microarchitecture. For comparison purposes, two ROB-based proposals without checkpointing have been modeled, one retiring instructions in program order (hereafter the *IOC* processor) and the out-of-order commit technique proposed in [6] (from now on, the

Scan processor). As the VB microarchitecture cancels instructions as soon as they leave the VB, to perform a fair comparison, the recovery mechanism for the ROB-based architectures is also triggered at the WB stage. In addition, the recovery penalty has been assumed to be equal in all the simulated microarchitectures, regardless the ROB/VB size. Notice that this assumption impacts negatively on the results obtained by the VB microarchitecture, since it requires a smaller validation buffer to obtain the desired performance (see Section IV-A), which can be recovered faster [4].

The analyzed architectures have been modeled on top of an extensively modified version of the SimpleScalar toolset [20] with separated ROB, instruction queues, and register file structures. The pipeline has been also enlarged with separated decode, rename, and dispatch stages. Both load speculation and store-load replay have been modeled in all the evaluated approaches. Table II summarizes the architectural parameters used through the experiments. Performance has been 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 the ROB would be blocked for longer.

Experiments were run using the SPEC2000 benchmark suite [21]. Both integer (SpecInt) and floating-point (SpecFP) benchmarks have been evaluated using the *ref* input sets and statistics were gathered using single simulation points [22]. The experimental study pursues two main goals: to evaluate the potential benefits on performance of the proposal, and to explore its complexity requirements in a modern processor.

A. 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) have been assumed to be unbounded. Figure 3 shows the average IPC (i.e., harmonic mean) for the SpecFP and SpecInt benchmarks when varying the ROB/VB size. As expected, performance improvements provided by the VB microarchitecture are much higher for floating-point benchmarks than for the integer ones. This is because the ROB size is not the main performance bottleneck in integer applications. One of the reasons is the high percentage of mispredicted branches. This result is in the line with the works [3], [6]. Thus, hereafter, performance analysis will focus on floating-point workloads.

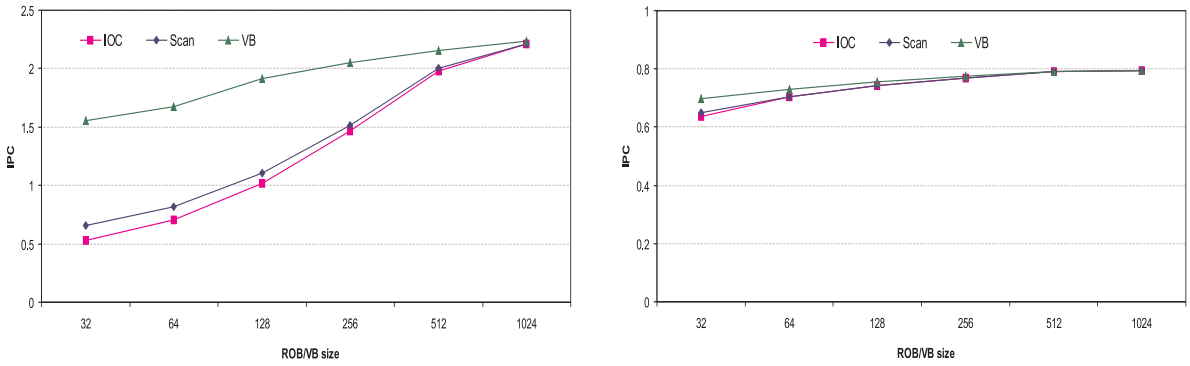
TABLE II

MACHINE PARAMETERS.

Microprocessor core	
Issue policy	Out of order
Branch predictor type	Hybrid gShare/bimodal: Gshare has 16-bit global history plus 64K 2-bit counters. Bimodal has 2K 2-bit counters, and the choice predictor has 1K 2-bit counters
Branch predictor penalty	10 cycles
Fetch, issue, commit bandwidth	4 instructions/cycle
# of Integer ALU's, multiplier/dividers	4/1
# of FP ALU's, FP multiplier/dividers	2/1
Memory hierarchy	
Memory ports available (to CPU)	2
L1 data cache	32KB, 4 way, 64 byte-line
L1 data cache hit latency	3 cycles
L2 data cache	512KB, 8 ways, 64 byte-line
L2 data cache hit latency	18 cycles
Memory access latency	200 cycles

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. In spite of this fact, it is required a large 1024-entry ROB for the IOC and Scan processors to match the IPC achieved by the VB microarchitecture.

Figure 4 presents the IPC achieved by each individual benchmark for a 32-entry ROB/VB. Loads and floating-point instructions are the main sources of IPC differences, since these instructions could potentially block the ROB for long. To provide insight into this fact, Table III shows the percentage of these instructions, the L1 miss rate, that the retirement of instructions from the ROB/VB is blocked. Results demonstrate that the VB microarchitecture effectively reduces the blocking time, therefore improving performance. This can be appreciated by observing that those applications showing high retirement blocking time differences also show high IPC differences (see Figure 4). Of course, applications having a low percentage of both floating-point instructions and a low cache miss rate would slightly benefit from our proposal.



(a) SpecFP (b) SpecInt

Fig. 3. IPC for SpecFP and SpecInt benchmarks.

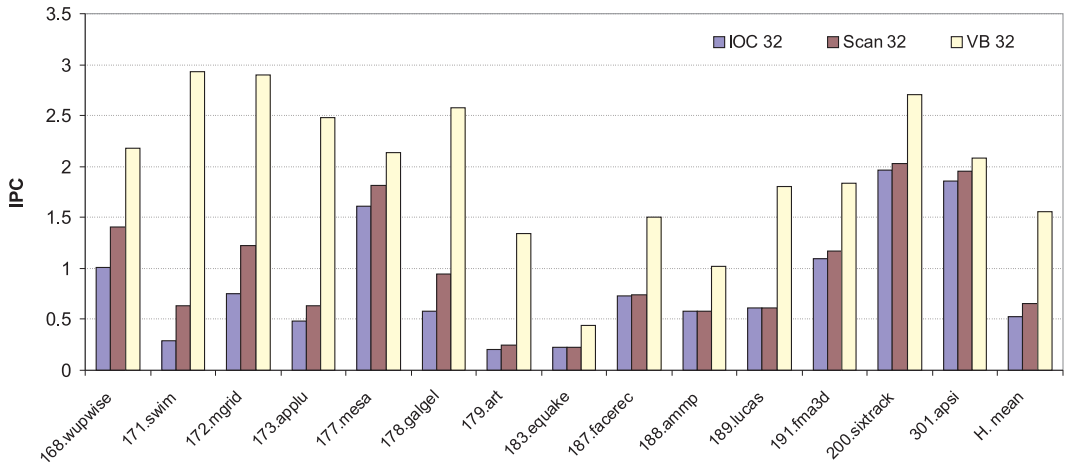


Fig. 4. IPC for SpecFP benchmarks assuming a 32-entry ROB/VB.

B. Exploring the Behavior in a Modern Microprocessor

This section explores the behavior of the VB microarchitecture while dimensioning the major microprocessor structures closely resembling the ones implemented in the Intel Pentium 4: 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 has been ranged from 8 to 128 entries.

Figure 5 shows the obtained IPC. As it can be seen, the VB microarchitecture is much more efficient since it achieves, with only 16 entries, on average, higher IPC than the other architectures. On the other hand, the use of VBs larger than 32 entries provides minor benefits

TABLE III

VB IMPROVED PERFORMANCE REASONS.

Workload	Instructions (%)		L1 miss	Blocked time (%)	
	f.point	load	rate (%)	ROB 32	VB 32
168.wupwise	30	23	1	75	35
171.swim	43	27	9	93	41
172.mgrid	59	32	3	84	47
173.applu	52	30	5	90	49
177.mesa	13	27	1	59	47
178.galgel	27	40	6	88	44
179.art	20	27	34	95	60
183.equake	35	41	11	96	89
188.ammp	35	27	5	86	74
189.lucas	66	13	10	91	29
191.fma3d	35	30	1	73	60
200.sixtrack	64	19	0	72	39
301.apsi	22	24	1	47	44

on performance.

Although the VB microarchitecture does not benefit in the same rate integer and floating-point benchmarks, we ran simulations for integer applications, and found that a 32-entry VB provides the same integer performance than a 128-entry ROB. Thus, we can conclude that integer benchmarks performance is not significantly affected when reducing the VB size.

To explore the complexity requirements of the four major microprocessor structures in the VB microarchitecture, we measured their occupancy in number of entries. As shown in Figure 6, in general, their occupancy is smaller in the VB than in the other architectures, regardless the VB/ROB size. The only exception is the instruction queue structure. This result was expected, as the decode stage is blocked for longer in the other architectures, which means that the VB architecture allows a higher number of instructions to be dispatched. In other words, the validation buffer structure has less pressure than the ROB, because part of this pressure moves to the instruction queue (see Section IV-A).

Regarding the VB occupancy, differences are really high. The VB occupancy is, on average, lower than one third the occupancy of the ROB. Notice that the highest IPC benefits appear

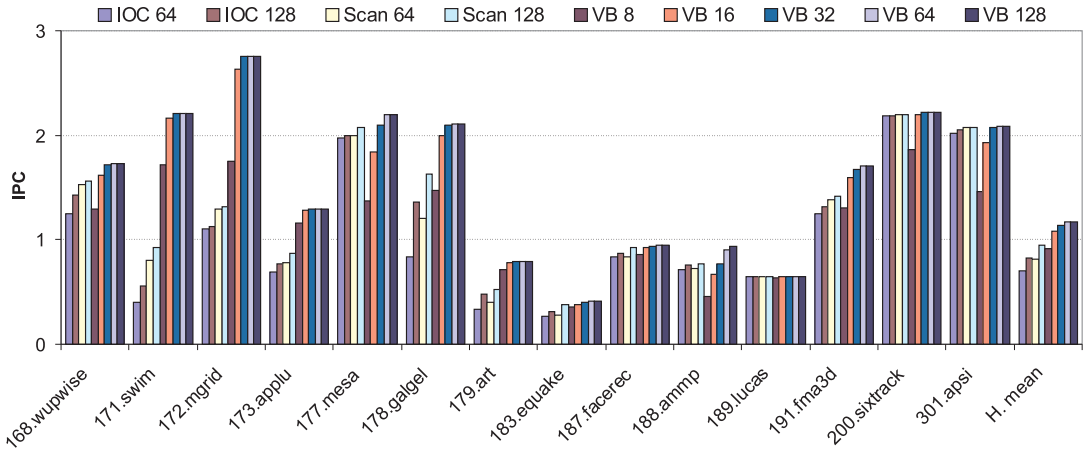
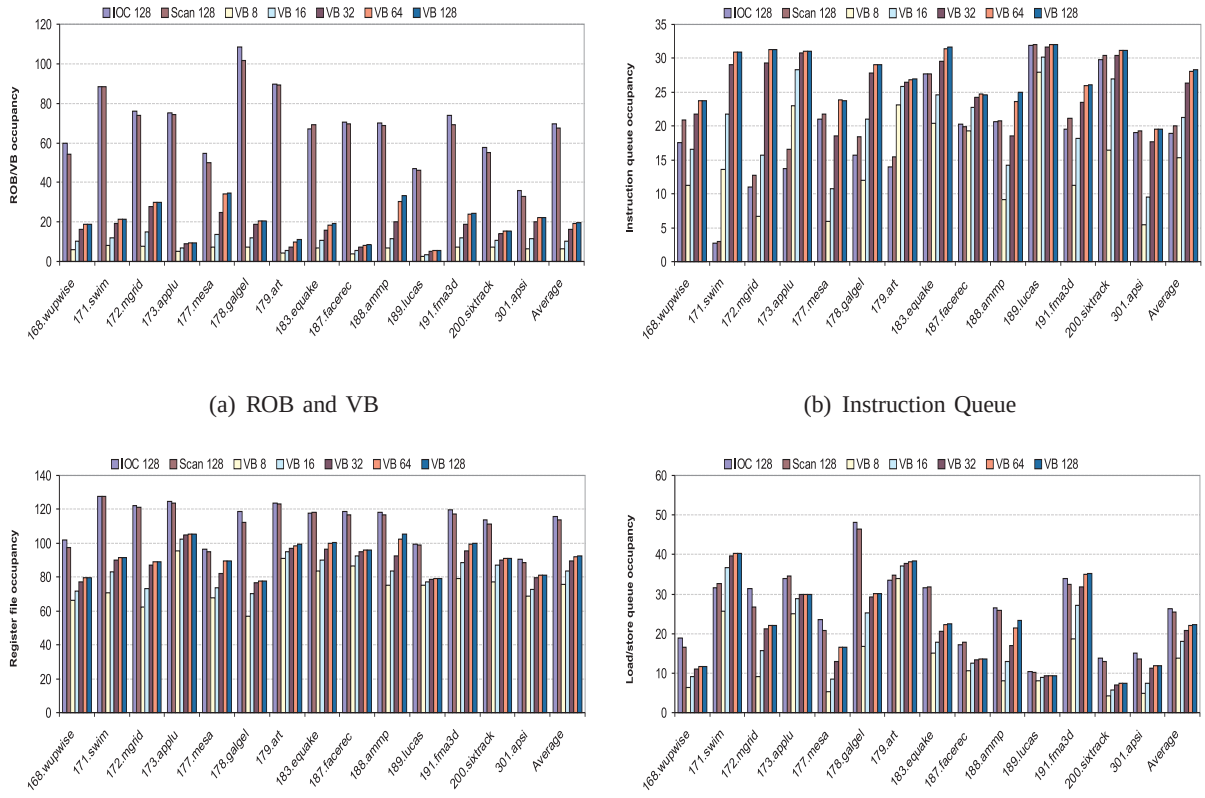


Fig. 5. IPC for SpecFP benchmarks in a modern microprocessor.

in those applications whose VB requirements are smaller than the instruction queue ones (e.g., *swim* or *mgrid*, see Figures 6(a) and 6(b)). In these cases, the VB microarchitecture effectively alleviates the retirement of instructions from the pipeline, allowing more instructions to be decoded and increasing ILP. On the contrary, when the VB requirements are larger than the ones of the instruction queue like happens when using a ROB (e.g., *mesa* or *equake*), the benefits are smaller, since the ROB is not acting as the main performance bottleneck. Results also show the effectiveness of the proposed register reclamation mechanism (see Figure 6(c)), which leads to a lower number of required registers. Finally, the LSQ occupancy is lower in the VB microarchitecture (see Figure 6(d)). This is because in ROB-based machines, a LSQ entry cannot be released until all previous instructions have been committed. In contrast, in the VB microarchitecture a LSQ entry only needs to wait until all the previous instructions have been validated, which is a weaker condition.

As the proposed architecture implements both out-of-order retirement and an aggressive register reclamation method, one might think that performance benefits may come from both sides. To isolate which part comes from the VB and which one from the register reclamation method, we ran simulations assuming an unbounded register file. Figure 7 shows the results. As observed, the register mechanism itself slightly affects the overall performance of the VB microarchitecture. Thus, one can conclude that almost all the benefits come from the fact that instructions are out-of-order retired. As opposite, IOC and Scan improve their performance with an unbounded



(a) ROB and VB

(b) Instruction Queue

(c) Register File

(d) Load/Store Queue

Fig. 6. Resource occupancy.

amount of physical registers, but even in these cases, the performance of a 16-entry VB is still better.

C. Impact on Performance of Supporting Precise Floating-Point Exceptions

Any floating-point exception can be supported by including the corresponding instructions in the set of epoch initiators. If all floating-point instructions were included, since some of these instructions may take tens of cycles to complete, the performance might be significantly impacted. To deal with this fact, such instructions could solve their epoch early in the pipeline. In this section, we assume that all the floating-point instructions are epoch initiators and their epoch is resolved when they are completed, which is over-conservative. For example, some floating-point exceptions can be easily detected by comparing the exponents (e.g., overflow) or checking one of the operands (e.g., division by zero). Figure 8 shows the obtained results.

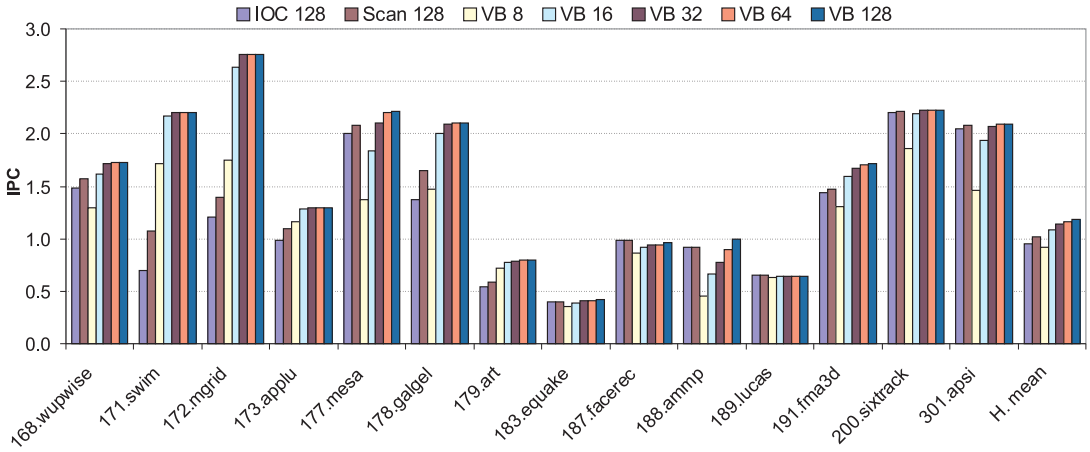


Fig. 7. IPC in a modern microprocessor assuming an unbounded amount of physical registers.

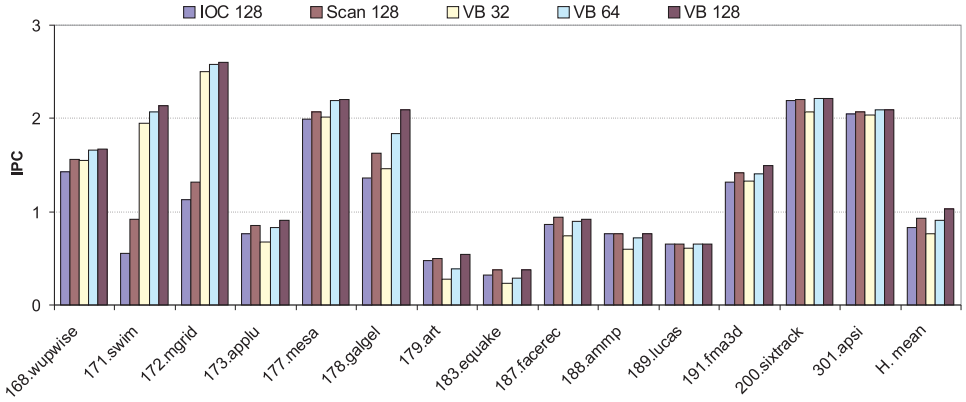


Fig. 8. IPC when supporting floating-point exceptions.

As expected, supporting all floating-point exceptions hurts the performance achieved by the VB microarchitecture. However, even in this case, a 32-entry VB achieves a performance close to the IOC processor with a 128-entry ROB. Moreover, a 64-entry VB achieves performance close to the Scan and IOC models while using a ROB twice as large.

From these results, we can conclude that, although supporting precise floating-point exceptions is encouraged by the IEEE-754 floating-point standard, the cost of enabling all the exceptions defined by the standard strongly impacts performance. As a matter of fact, most architectures allow disabling floating-point exceptions by software. In some architectures (e.g., Alpha) these exceptions are imprecise by default [7], and it is required the help of the compiler to detect which instruction raised the exception [23].

IV. Dealing with complexity

Superscalar processors have evolved towards complex out-of-order microarchitectures in order to exploit large amounts of instruction level parallelism. However, increasing the complexity of critical components may adversary impact on the processor cycle. Therefore, there is an important tradeoff, because increasing complexity may increase instructions per cycle, but at the same time, it also may reduce clock speed.

This section analyzes how in the VB architecture, retiring instructions in an out-of-order fashion, not only improves performance but also it can be implemented with less hardware complexity. For this purpose, two different approaches have been analyzed: i) the required size of the microprocessor components to reach a given performance, and ii) analyzing the impact on performance of the fetch, decode, and issue widths.

A. Dimensioning the major microprocessor components

When designing a microprocessor, its components must be correctly sized. If a component is too small dimensioned, it will incur in a performance degradation since it would act as a bottleneck. On the contrary, dimensioning a component larger than necessary will incur in area wasting.

As analyzed in Section III-B, the VB microarchitecture performs a more efficient use of the microprocessor components than the in-order commit architecture. This means that components should be dimensioned for the VB in a different way than for a in-order commit architecture.

The goal of this section is to identify the less complexity-effective configuration or mix of processor components to reach a given performance level (i.e., IPC). To this end, a wide range of complexities of the major processor structures have been analyzed: register file (64, 128, and 256 entries), issue queue (16, 32, 64, 128, and 256 entries), load-store queue (16, 32, 64, 128, and 256 entries), and ROB/VB size (16, 32, 64, 128, and 256 entries). The remaining processor parameters have been fixed as in Section III-B. This gives an amount of $5^3 \times 3 = 375$ configurations for each modeled processor, which multiplied by the number of benchmarks evaluated represents an important number of simulations.

Because of the large number of results, performance comparison becomes very difficult. To ease this analysis, we firstly sorted the different configurations in increasing order of performance. Then, for each group of mixes or configurations providing the same performance, we selected

the one with less hardware complexity. In many cases, several mixes showed similar complexity. However, for the sake of clarity, for each performance value, only one configuration was selected. Figures 9 and 10 show these results. Notice that resources are not labeled in the same order in both figures as their impact on performance differs depending on the modeled processor. As it can be seen, the register file is the main performance bottleneck in both architectures, but the second performance bottleneck differs. In the IOC processor, it is the ROB size while in the VB architecture it is the IQ (Instruction Queue) size.

Since the main performance bottleneck in both architectures is the register file size, for a given register file size, performance rises as long as the other resources are properly enlarged. Then, to further improve performance, the register file itself must be enlarged again. This sequence keeps on until the performance saturation point is reached.

To perform a fair comparison, we compare the complexity of the processors on the basis of the same performance. To this end, we selected four performance levels, namely A, B, C, and D (see Figures 9 and 10).

If VB and IOC processors are compared based on the D performance level, the IOC processor requires a minimum complexity of a 128-entry RF (Register File), a 128-entry ROB, a 64-entry IQ, and a 64-entry LSQ (Load-Store Queue), while the VB processor reaches similar performance with half the size of the IQ, and both a VB and a LSQ four times smaller. Notice also that the VB provides similar performance with only a 16-entry IQ.

Regarding the A performance level, the IOC processor requires a minimum complexity of a 256-entry RF, a 256-entry ROB, a 128-entry IQ, and a 128-entry LSQ, while the VB processor reaches similar performance with half the size of the IQ, and an VB four times smaller than the ROB (i.e., a 256-entry RF, a 64-entry VB, a 64-entry IQ, and a 128-entry LSQ). Notice also that in that case, negligible performance drops are observed in the VB when the size of the LSQ is reduced by a half (i.e., a 256-entry RF, a 64-entry VB, a 64-entry IQ, and a 64-entry LSQ). In addition, if the complexity of the IOC processor is increased, no significant performance gain is achieved above the A level. On contrast, in the VB processor performance might increase up to around 32%.

Finally, remark that in between the analyzed cases there is a wide gradient of performance levels (for instance, B and C in Figures 9 and 10). For any given performance level, it is always possible to find a VB hardware configuration with much less complexity than the required for

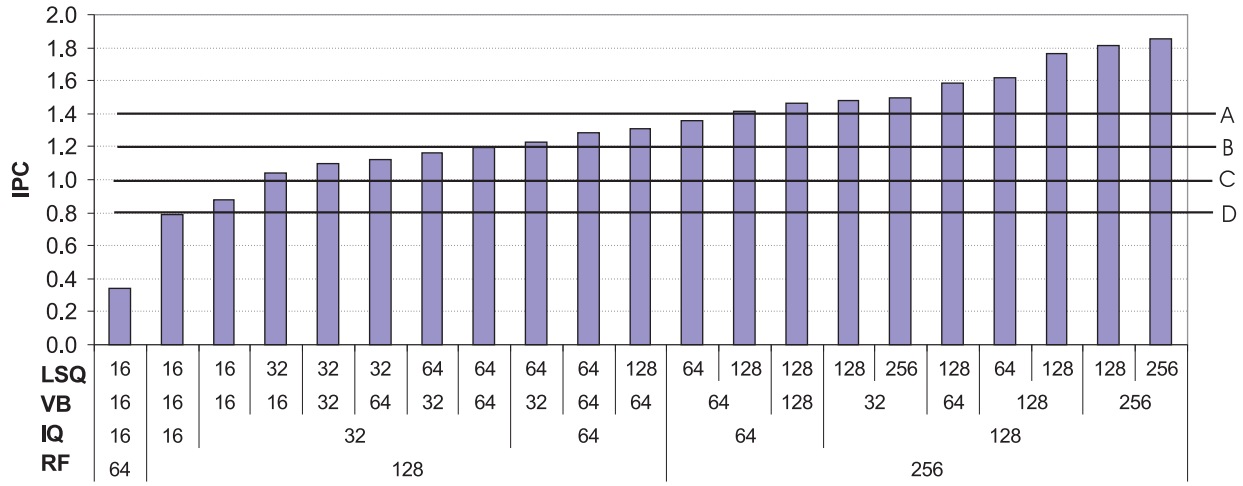


Fig. 9. Performance of the VB processor when varying the complexity of the RF (Register File), IQ (Issue Queue), ROB, and LSQ (Load-Store Queue).

the IOC processor.

B. Analyzing the pipeline width

Previous section dimensioned the major microprocessor components. This section goes further away exploring the complexity of the microprocessor pipeline by decreasing the fetch/decode/issue of the baseline processor down to 1 instructions per cycle.

This is much more aggressive than simply resizing the microarchitecture components, because modifying such widths has important implications in complexity. For instance, increasing the issue width also increases the delay of the rename logic since that width also determines the number of ports into the map table and the width of the dependence check logic. As another example, reducing the issue width from 4 to 2 ways means that the number of read ports in the register file will drop from 8 to 4. In addition, there are important area implications. For example, the number of bypass paths increases with the square of the issue width. More details on complexity issues can be found in [2].

Figure 11 shows the performance reached by the three compared architectures when varying the issue fetch/decode/issue width from 1 to 4 instructions per cycle. The remaining processor parameters have been fixed as in Section III-B. Results are remarkable, since a 2-instruction width

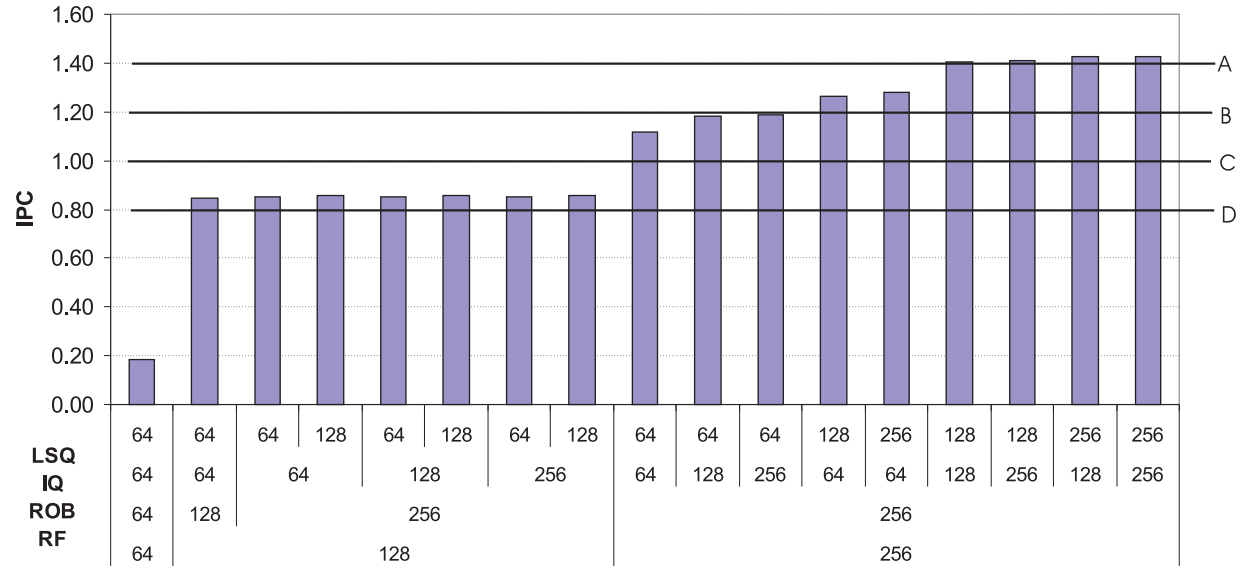


Fig. 10. Performance of the IOC processor when varying the complexity of the RF (Register File), IQ (Issue Queue), ROB, and LSQ (Load-Store Queue).

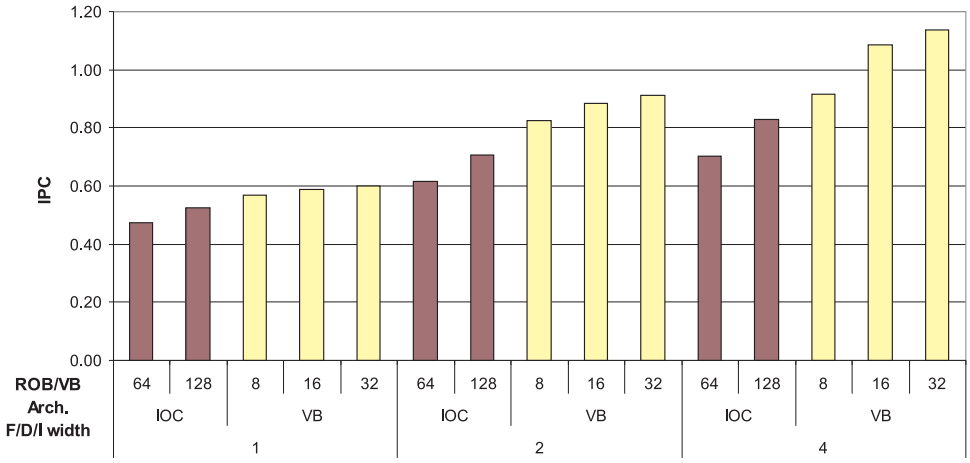


Fig. 11. IPC for different fetch/decode/issue widths.

VB architecture with a small 8-entry validation buffer reaches the same performance as a IOC processor with a 4-instruction width and a 128-entry ROB. Also, with just a single-instruction issue width and a 16-entry validation buffer, the VB microarchitecture reaches performance close to a much more complex IOC processor with a two-instruction width and a 64-entry ROB.

In summary, experimental results evidence that the VB microarchitecture not only can out-perform existing in order and out-of-order architectures, but also requires much less hardware

complexity.

V. Related Work

Long latency operations constrain the output rate of the ROB, and thus, microprocessor performance. Several microprocessor mechanisms have been recently proposed dealing with this problem [24], [3], [5], [4]. In summary, these proposals permit to retire instructions in a speculative mode when a long latency operation blocks the ROB head. These solutions introduce specific hardware to checkpoint the architectural state at specific times and guarantee correct execution. When a misprediction occurs, the processor rolls back to the checkpoint, discarding all subsequent computations. Some of these proposals have been extended to be used in multiprocessor systems [25], [26].

In [24], 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 each 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 [3], Kirman et al propose the checkpointed early load retirement mechanism which has certain similarities with the previous one. 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 against the predicted one. On a misprediction, the processor must roll back to the checkpoint.

In [5], 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 the 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., those 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 [4].

In [27] Martinez et al propose an in-order retirement mechanism which identifies irreversible instructions to early freeing resources. Unlike the VB microarchitecture, this proposal retires instructions in-order. Also, as well as the works discussed above, it needs checkpointing to roll back the processor to a correct state.

In [6], a checkpoint free approach is presented. However, it still uses a ROB. The proposal scans the *n* oldest entries of the ROB to select instructions to be retired. This fact constrains this proposal making it unsuitable for large ROB sizes. In addition, resources are handled as a typical processor using a ROB, without any focus on improving resource usage.

Finally, some proposals alleviate the performance degradation caused by ROB blocking by enlarging the major microprocessor structures or efficiently managing them [28], [29], [30].

VI. Conclusions

This paper has proposed the VB microarchitecture, which it is aimed at retiring instructions out of order while providing support for speculation and precise exception handling. The proposed microarchitecture does not perform checkpoints because out-of-order retirement is correctly performed by design.

Performance results have been compared against two ROB-based proposals, one retiring instructions in order and the other one in out-of-order fashion. Regarding the effectiveness of the VB buffer, results show that, with only a 32-entry validation buffer and assuming the remaining major processor structures unbounded, our proposal achieves performance similar to the other evaluated architectures 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. In fact, results show that resource usage is not higher in the VB while achieving better performance.

Concerning complexity versus performance tradeoffs, results show that to achieve a given performance level, the VB microarchitecture requires simpler hardware in the major microprocessor structures (i.e., register file, LSQ, and IQ) than an in-order retirement processor. Moreover, the VB achieves, with a narrower pipeline width, similar performance than conventional ROB-based

processors. This fact brings important implications in those architectures where the microarchitecture complexity is one of the most important issues, e.g., power-aware and SMT processor implementations.

In summary, the VB has been shown to behave as a complexity-effective microarchitecture, which makes the proposal attractive to either achieve better performance or reduce complexity for a given performance level.

VII. Acknowledgments

This work has been partially supported by the Generalitat Valenciana under grant GV06/326, by the Spanish CICYT under grant TIN2006-15516-C04-01, and by CONSOLIDER-INGENIO 2010 under grant CSD2006-00046.

References

[1] J. Smith and A. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985, pp. 36–44.

[2] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processor," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[3] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martínez, "Checkpointed early load retirement," in *Proceedings of the International Symposium on High Performance Architecture*, February 2005.

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

[5] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-order commit processors," in *Proceedings of the International Symposium on High Performance Architecture*, February 2004.

[6] G. Bell and M. Lipasti, "Deconstructing commit," in *Proceedings of the The International Symposium on Performance Analysis of Systems and Software*, March 2004.

[7] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[8] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture, technical white paper," *IBM Server Group*, 2001.

[9] G. Hinton, D. Sager, M. Upton, D. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Rousell, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, Q1, 2001.

[10] S. Petit, J. Sahuquillo, P. López, and J. Duato, "The validation buffer out-of-order retirement microarchitecture, Tech. Rep. DISCA/0069-2007, 2007.

[11] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[12] J. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proc. of the IEE*, vol. 83, no. 2, 1995.

[13] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register renaming and dynamic speculation: an alternative approach," in *Proceedings of the 26th International Symposium on Microarchitecture*, December 1993, pp. 202–213.

[14] K. Yeager, "The mips r10000 superscalar microprocessor," *IEEE Micro*, pp. 28–40, 1996.

[15] J. Shen and M. Lipasti, *Modern Processor Design*. McGraw-Hill, 2005.

[16] K. Gharachorloo, A. Gupta, J. H. Singhal, D. Broniarczyk, F. M. Cerauskis, J. Price, L. Yuan, G. Cheng, D. D. amd S. Fosth, N. Agarwal, K. Harvey, and E. Hagersten, "Two techniques to enhance the performance of memory consistency modelsgigaplane: A high performance bus for large smps," in *Proceedings of the Symposium on High Performance Interconnects IV*, 1996, pp. 41–52.

[17] D. J. Sorin, M. Plakal, M. D. Hill, and A. E. Condon, "Lamport clocks: Reasoning about shared memory correctness, Tech. Rep. CS-TR-1998-1367, 1998.

[18] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill, "Lamport clocks: Verifying A directory cache-coherency protocol," in *Proceedings of the 10th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'98)*, 1998, pp. 67–76.

[19] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proceedings of the 1991 International Conference on Parallel Processing*, vol. I, Architecture. Boca Raton, FL: CRC Press, 1991, pp. I–355–I–364.

[20] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0." *Computer Architecture News*, vol. 25, no. 3, 1997.

[21] "Standard performance evaluation corporation," *http://www.spec.org/cpu2000/*.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[23] *GCC online documentation*, Free Software Foundation, [Online]. Available:http://www.gnu.org/software/gcc/onlinedocs/, 2006.

[24] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: An alternative to very large instruction window for out-of-order processors," in *Proceedings of the International Symposium on High Performance Architecture*, February 2003.

[25] M. Kirman, N. Kirman, and J. Martínez, "Cherry-mp: Correctly integrating checkpointed early resource recycling in chip multiprocessors," in *Proceedings of the International Symposium on Microarchitecture*, November 2005.

[26] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero, "Implementing kilo-instruction multiprocessors," in *IEEE Conference on Pervasive Services, Invited lecture*, July 2005.

[27] J. Martínez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, "Cherry: checkpointed early resource recycling in out-of-order processors," in *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[28] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt, "A scalable instruction queue design using dependence chains," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[29] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *Proceedings of the 34th Int. Symp. on Microarchitecture*, December 2001.

[30] I. Park, C. Ooi, and T. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.