# The Impact of Out-of-Order Commit in Coarse-Grain, Fine-Grain and Simultaneous Multithreaded Architectures

R. Ubal, J. Sahuquillo, S. Petit, P. López, and J. Duato
Dept. of Computer Engineering (DISCA)
Universidad Politécnica de Valencia, Spain
raurte@gap.upv.es

## Abstract

*Multithreaded processors in their different organizations (simultaneous, coarse grain and fine grain) have been shown as effective architectures to reduce the issue waste. On the other hand, retiring instructions from the pipeline in an out-of-order fashion helps to unclog the ROB when a long latency instruction reaches its head. This further contributes to maintain a higher utilization of the available issue bandwidth. In this paper, we evaluate the impact of retiring instructions out of order on different multithreaded architectures and different instruction fetch policies, using the recently proposed Validation Buffer microarchitecture as baseline out-of-order commit technique. Experimental results show that, for the same performance, out-of-order commit permits to reduce multithread hardware complexity (e.g., fine grain multithreading with a lower number of supported threads).*

## 1. Introduction

Superscalar processors effectively exploit instruction level parallelism of a single thread. To this end, multiple instructions can be issued in an out-of-order fashion in the same cycle. Nevertheless, issue ports are usually wasted because of instruction dependencies (i.e., the available parallelism), thus, adversely impacting the performance. Two kinds of waste are distinguishable [16]: vertical waste, when no instruction is issued, and horizontal waste, when some instructions are issued but without completely filling the issue width.

Resource utilization can be improved by providing support to the execution of multiple threads, that is, by exploiting both instruction and thread level parallelism. There are three main multithreading models implemented in current processors: fine grain (FGMT), coarse grain (CGMT), and simultaneous multithreading (SMT). All of them reduce vertical waste but only SMT reduces horizontal waste [16] by issuing instructions from multiple threads in the same cycle, achieving the best performance gains. Nevertheless, this is done at the expense of adding complexity to the issue logic, which is a critical point in current microprocessors. Multithreaded architectures represent an important segment in the industry. For instance, the Alpha 21464, the Intel Pentium 4 [7], the IBM Power 5 [9], the Sun Niagara [11], and the Intel Montecito [3] are commercial microprocessors included in this group.

On the other hand, recent research has focused on the retirement of instructions in an out-of-order fashion (hereafter OOC processors) [1][2][12][17]. Unlike typical processors implementing a ROB, where instructions are committed in program order (from now on IOC processors), OOC architectures do not force an instruction to be the oldest one in the pipeline in order to be retired. In this way, long latency operations (e.g., a L2 cache miss) do not block the ROB when they reach the ROB head; instead, long memory latencies are overlapped with the retirement of subsequent instructions which do not depend on the memory operation. Thus, these architectures mainly attack the vertical waste, although horizontal waste is indirectly also improved. In addition, it has been shown that OOC processors can make better use of resources and are more performance-cost effective than IOC processors [17]. More recently, out-of-order retirement has been investigated on superscalar processors and chip multiprocessors (CMPs), but, to the best of our knowledge, only slight tips about OOC multithreading have been published in [19].

In this paper, we deeply analyze the impact of retiring instructions in an out-of-order fashion in the three main models of multithreading: FGMT, CGMT, and SMT. To this end, we selected the Validation Buffer microarchitecture (VB), detailed in Section 2, as the OOC base architecture, and extended it to support the execution of multiple threads. Experimental results provide three main conclusions:

- First, an OOC SMT processor requires in most cases

half the amount of hardware threads than an ROB-based SMT processor to achieve similar performance. In other words, performance can be maintained in VB-based SMT when reducing the number of hardware threads, thus saving all hardware resources to track their status.

- Second, an OOC FGMT processor outperforms an ROB-based SMT processor. In this case, performance can be sustained while simplifying the issue logic, which can be translated in shorter issue delays or lower power consumption of instruction schedulers.

- Third, existing fetch policies for SMT processors provide complementary advantages to the out-of-order retirement benefits. A high-performance SMT design could implement both techniques if area, power consumption and hardware constraints allow it.

The remainder of this paper is organized as follows. Section 2 describes background work. Section 3 summarizes the VB-MT microarchitecture. Section 4 cites and adapts previous existing techniques to VB-MT. Section 5 shows experimental results, and finally Sections 6 and 7 present related work and some concluding remarks, respectively.

## 2. Background: The Validation Buffer Microarchitecture

In [17], Petit et al proposed a new out-of-order retirement microarchitecture in which the classical ROB structure is substituted by a shorter FIFO structure: the VB (Validation Buffer). This microarchitecture, referred to as the VB microarchitecture, allows instructions to be retired early, as soon as it is known that they are nonspeculative even if these instructions are not completed yet. Once they are completed, they update the machine state and free the used resources. Therefore, instructions may exit the pipeline in an out-of-order fashion. In order to make this paper self-contained, this section details this microarchitecture.

The necessary conditions to allow an instruction to be committed out-or-order are [2]: a) 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 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 is straightforwardly met by any proposal at the writeback stage. The second condition is fulfilled by a register reclamation method based on the counter method [10] described below. Finally, the last three conditions are handled by the validation buffer, which replaces the ROB and contains the instructions whose conditions are not known yet.

To deal with the last three conditions, the VB microarchitecture decomposes code into *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.

The VB microarchitecture can support a wide range of epoch 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) and exceptions related with address calculation (e.g., page faults, invalid addresses) are supported by design. It is also possible enable and disable the support of more epoch initiators, e.g., user definable flags or specific machine instructions could dynamically enable or disable a precise handling of more exceptions.

### 2.1. Register Reclamation

Typically, modern microprocessors free a physical register when the instruction that renames the corresponding logical register commits, since at this point of the execution, it is known that all the instructions reading the physical register have already committed. Then, the physical register index is placed in the free physical register list. 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 the VB microarchitecture.

To overcome this problem, the VB microarchitecture uses a register reclamation strategy based on the counter method. The hardware components used in this scheme, as shown in Figure 1 are the Register Aliasing Table ($RAT$)

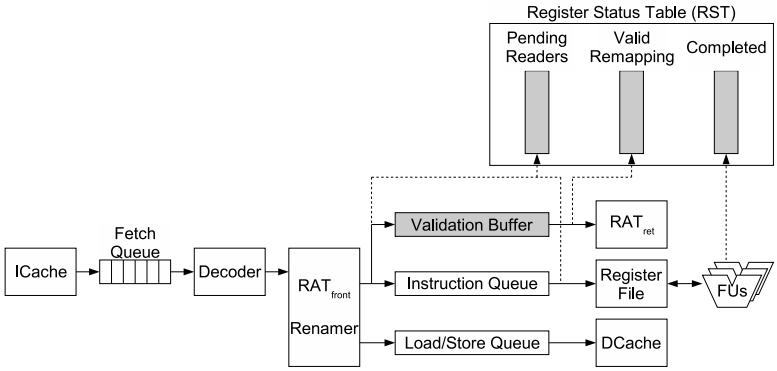


**Figure 1. VB microarchitecture block diagram.**

and the Register Status Table ($RST$).

The $RST$ is indexed by a physical register identifier and each entry contains three fields, $pending\_readers$, $valid\_remapping$ and $completed$. The $completed$ bit (also present in ROB-based microarchitectures) indicates that the instruction producing a value for the corresponding physical register has finished execution (i.e., has written back its result). $Valid\_remapping$ is a bit that is set when the instruction that renames the corresponding logical register is retired from the VB structure as non-speculative. Notice that this instruction may not be the oldest in the pipeline, so it is needed to account the number of pending readers of each physical register. To this end, the $pending\_readers$ field contains the number of decoded instructions that consume the corresponding physical register, but have not read it yet (i.e., located in the instruction queue). This counter is incremented when reading instructions are renamed, and decremented on issue (i.e, after they read their source registers). A register is free when its producer has finished execution (i.e, $completed = 1$), it has been remapped by a non-speculative instruction (i.e., $valid\_remapping = 1$), and there are not pending readers awaiting execution (i.e., $pending\_readers = 0$).

The $RAT$ is indexed by a logical register identifier and returns the associated physical register. There are two copies of this structure, named $RAT_{front}$ and $RAT_{ret}$, respectively. At the retirement of an instruction, $RAT_{ret}$ matches the $RAT_{front}$ at the time this instruction was renamed.

Recovering the system from a mispeculation involves restoring both the $RAT_{front}$ and the $RST$ tables. Since at the retirement of an instruction from the VB, the $RAT_{ret}$ matches the $RAT_{front}$ at the time the instruction was renamed, a simple method to recover the $RAT_{front}$ 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 [1]. Regarding the $RST$, it can be recovered by draining the canceled instructions from the VB and the instruction queue while undoing the modifications performed at the renaming stage on the $RST$. 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 draining can be overlapped with subsequent new processor operations.

### 2.2. Results

As the results shown in [17] demonstrate, the greatest performance improvement of the VB microarchitecture is achieved when the ROB acts as the main bottleneck, which corresponds to those situations where long-latency instructions (mainly floating-point operations or main memory accesses) increase the ROB occupancy causing it to get stalled. On average, a VB can be up to 4 times smaller than a ROB, while achieving a similar performance.

The VB microarchitecture can be considered a cost-effective approach, as it significantly reduces the size of a major processor structure. This fact makes it susceptible of being combined with other enhanced and orthogonal processor designs that attack other structures or pipeline stages to further increase performance, such as multithreading. The VB effectiveness, jointly with its low cost, and the wide spectrum of existing multithreaded processor designs motivate a deep study of the VB behavior on multithreaded processors.

## 3. VB-MT Microarchitecture

### 3.1. Providing Multithreading support

As in any multithreading model, the VB-MT architecture provides the illusion of having various logical processors, that is, various simultaneously active software contexts, one per hardware thread. Although most hardware structures

can be shared or private among threads [14], the logical state, defined by a *virtual memory image* and a *logical register file*, must be independently maintained per thread.

Concerning the *virtual memory image*, it makes sense for all threads to have a common physical address space. The MMU (Memory Management Unit), and the Translation Lookaside Buffers (TLBs) in particular, must be adapted to be indexed by pairs $\{$thread, virtual address$\}$, returning disjoint physical addresses to each thread.

In the case of the *register file*, the associated physical structures can be either shared or private per thread, but the subsets of physical registers bound to specific threads are always disjoint from each other. This implies a renaming mechanism that allocates a new physical register from a pair $\{$thread, logical register$\}$, which can be straightforwardly implemented using a private RAT (Register Aliasing Table) —including both $RAT_{front}$ and $RAT_{ret}$— per thread.

### 3.2. Resource sharing

A multithreaded design offers the view of having multiple logical processors in a single chip. A straightforward way of implementing such a system consists of replicating all hardware structures per thread, while maintaining a common functional resource pool. In the opposite approach, hardware structures can be shared among threads, using policies that dynamically allocate resource slots. Between these limits, there is a gradient of solutions to be explored in order to find the optimal sharing strategy of resources.

Shared resources are often more costly than private, as the allocation/deallocation of resource entries may get more complex. Moreover, shared resources need to increase read/write ports in order to enable parallel access to various threads, which may increase both their area and latency. The advantage of shared resources, as one would expect, lies in the fact that one single thread can compete for all its entries.

Processor resources can be classified as *storage resources* (reorder buffer, issue queue, load/store queue...) and *bandwidth resources* (fetch stage, issue logic...). As pointed out in [13], and in contrast to the comments above, shared storage resources can cause a performance degradation if no appropriate resource allocation policy is used. The reason is that over-allocations to stalled threads can cause starvation to active ones, and wrong allocation decisions can affect several future cycles. The same does not happen with unwise allocation decisions of bandwidth resources, since they can be quickly compensated in the next cycle. On the other hand, the high demand variability of bandwidth resources, in contrast to storage resources, causes a shared design to be the most performance-efficient solution. The issue stage is the main bandwidth resource affected by this fact, which has caused SMT (Simultaneous Multithreading) to stand out from other multithreading designs.

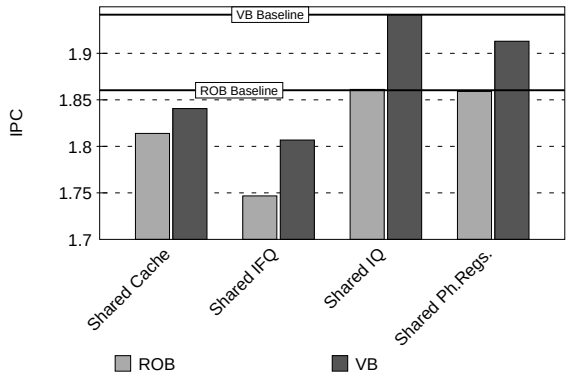


**Figure 2. Impact of resource sharing for ROB/VB architectures. Baseline IPCs refer to processors with no shared storage resource.**

This section focuses on the effect of sharing the main storage resources (cache, instruction fetch queue, issue queue, and register file) among hardware threads. Figure 2 shows the results of this study. In the case of the ROB-based architecture, results are consistent to those presented previously in [13], showing a performance degradation when using shared resources without a proper allocation policy. These results are compared with a VB multithreaded (VB-MT) architecture. To isolate the effect of sharing each resource, four configurations are evaluated on each architecture. In each configuration, only one storage resource (X-axis) is shared among threads. In all cases, an SMT processor with a round-robin fetch policy is simulated. The performance obtained with an ROB-based and a VB-based processor with all resources set as private is represented with two horizontal lines, labeled as *ROB baseline* and *VB baseline*, respectively.

Figure 2 reveals that conclusions stated in [13] remain valid in the VB-MT architecture: there is a performance loss when sharing any storage resource without an optimized allocation policy. Out-of-order retirement does not affect slow or stalled threads, which continue abusing of shared resources, reserving their entries for a long time, and preventing other threads from using them fruitfully. The effectiveness of previously proposed solutions to these problem, when applied on the VB-MT architecture, will be evaluated in Section 5.

The only component that does not show performance degradation when being shared is the ROB/VB (not shown in Figure 2). In this case, we can cite two possible mechanisms to handle the ROB entries that are assigned to threads. One approach consists of assigning disjoint ROB portions to threads [8], whose size can vary depending on the threads demand; each portion is handled as an independent ROB. As an opposite alternative, instructions can be inserted in

a FIFO manner into the shared ROB, as if all of them belonged to the same thread, and hence retired in the same order. Both approaches can take the advantages of a shared storage resource (also applying some resource allocation policy if necessary), but each one suffers of certain drawbacks. In the former case, ROB portions cannot grow arbitrarily to fill the whole ROB size; a portion size is constrained by the position of contiguous portions, and ROB portions can only be shifted when the associated *head* and *tail* pointers are properly aligned. In the latter case, instructions from different threads are intermingled across the ROB, so non completed instructions from a thread may prevent completed instructions from another from exiting the ROB. Moreover, the recovery process should cancel instructions selectively, forcing interleaved gaps to remain in the ROB until they are retired. A deeper study of the effects of the ROB/VB sharing strategies is planned as for future work, so we assume in this paper private ROBs/VBs for all experiments.

# 4. Extending VB-MT to support Simultaneous Multithreading Techniques

This section discusses the extensions of the VB architecture needed to implement existing techniques oriented to SMT processors. These techniques try to exploit the SMT potential by maintaining functional units utilization as high as possible. The techniques studied in this section are Instruction Count (ICOUNT), Predictive Data Gating (PDG) and Dynamically Controlled Resource Allocation (DCRA). Their impact on the VB architecture, compared with the ROB, will be evaluated in Section 5.

## 4.1. Instruction Count (ICOUNT)

ICOUNT is a fetch policy proposed by Tullsen et al [15], which assigns fetch priority to those threads with less instructions in the decode, rename and issue stages. The designation ICOUNT.$n_t$.$n_i$ means that at most $n_t$ threads can be handled at the fetch stage in the same cycle, taking at most $n_i$ instructions from each one. Experiments shown later use an ICOUNT.2.8 configuration, which provides the best results for the ROB-SMT architecture.

## 4.2. Predictive Data Gating (PDG)

PDG was proposed by El-Moursy et al [5], and can be classified as a fetch policy, too. This technique is aimed at avoiding a long permanence of non-ready instructions in the issue queue (IQ) due to dependences on long-latency instructions, long data dependence chains or contention for functional units or cache. To early detect long-latency events, a load miss predictor is placed in the processor front-end, and a per thread specific counter tracks the estimated number of pending cache misses. The values of these counters are used to stall threads fetch when a threshold is exceeded.

As suggested in [5], experiments assume a load miss predictor of 4K entries with 2-bit saturating counters. The predictor is indexed by the PC of the *load* instruction, and the prediction is given by the most significant bit of the saturating counter. Whenever a *load* misses the data cache, the corresponding counter is reset, while it is incremented when the associated *load* hits the cache.

## 4.3. Dynamically Controlled Resource Allocation (DCRA)

DCRA was proposed by Cazorla et al [6]. Unlike ICOUNT and PDG, DCRA does not only control the fetch bandwidth granted to each thread, but also manages the allocation of shared processor resources. DCRA classifies threads according to two criteria. A thread is considered slow or fast, depending on whether it has or not pending L1 cache misses. Likewise, a thread is considered active or inactive for a given resource depending on whether it has recently (e.g., in the last 256 cycles) demanded the resource.

The authors propose a mathematical formula to limit the number of entries in a shared resource that a thread can allocate depending on its previous classification. The additional hardware consists of resource occupancy counters and a lookup table to implement the computation of resource entries limit using the aforementioned formula.

# 5. Experimental Results

This section evaluates the impact of retiring instructions out of order using the main multithreading paradigms. To this end, the VB architecture has been used as OOC baseline processor, and both performance and complexity have been addressed. Experiments were run using Multi2Sim [18], a simulation framework which is able to model multicore-multithreaded processors, sharing strategies of processor resources, instruction fetch policies and thread priority assignment. Table 1 shows the baseline processor configuration.

For those experiments where a specific design has been evaluated with groups of benchmarks, we follow the criteria discussed in [8] to build these mixes using the SPEC2000 suite, as shown in Table 2. Benchmarks are classified as ILP (high instruction level parallelism) and MEM (memory-intensive), and three groups are created to evaluate possible combinations of two and four threads. In addition, two more groups (referred to as INT and FP) are analyzed, which

**Table 1. Baseline processor parameters**

| Parameter | Configuration |
|---|---|
| Machine width | 8-wide fetch, 8-wide issue, 8-wide commit |
| Storage resources size | 32 entry private IQs, 24 entry private LSQs, 32 entry private ROBs/VBs |
| Functional units and latency (total/issue) | 8 Int Add (2/1), 2 Int Mult (1/1), 2 Int Div (20/19), 4 Ld/St (2/1), 8 FP Add (4/2), 2 FP Mult (8/1), 2 FP Div (40/20) |
| Phys Registers | 128 entry private files |
| L1 I-cache | 32KB, 2-way, 64 byte line, per-thread private, 1 cycle hit time |
| L1 D-cache | 32KB, 2-way, 64 byte line, per-thread private, 2 cycles hit time |
| L2 Unified cache | 1MB, 8-way, 64 byte line, shared among threads, 10 cycles hit time |
| BTB | 1024 entry, 2-way |
| Branch Predictor | McFarling, private per thread, 4K entry gShare, 4K entry bimodal |
| Memory Latency | 200 cycles |
| D-TLB, I-TLB | 16K, 4-way, shared among threads |

**Table 2. Benchmark combinations**

| Classification | Mix Name | Benchmarks |
|---|---|---|
| ILP | Mix 0 | *wupwise, eon* |
| | Mix 1 | *apsi, eon, fma3d, gcc* |
| ILP+MEM | Mix 2 | *art, gzip* |
| | Mix 3 | *art, gzip, wupwise, twolf* |
| MEM | Mix 4 | *applu, ammp* |
| | Mix 5 | *applu, ammp, art, mcf* |
| INT | Mix 6 | *gcc, gzip* |
| FP | Mix 7 | *wupwise, mgrid* |

include only integer or floating-point benchmarks, respectively. These two groups are included due to the contrasting behavior of integer and floating-point applications on the VB architecture.

### 5.1. Basic Multithreading Models

As first experiment, we have evaluated the behavior of the VB microarchitecture implementing multithreading in its different forms (CGMT, FGMT, SMT). The FGMT design is modeled with a timeslice issue policy, while the SMT issue stage takes instructions from different threads in the same cycle (shared issue). Both FGMT and SMT are modeled with a timeslice, or round-robin, fetch stage (more complex fetch policies on VB-MT are evaluated in Section 5.3). Finally, the CGMT design establishes a thread quantum of 1000 cycles, and the thread switch penalty is equal to the number of cycles needed to drain the processor pipeline at a given time [14].

Figure 3 shows the obtained results. Comparing the behavior of the VB-MT architecture in its different variants with the ROB-SMT processor, we can observe that VB-CGMT is about 5% slower than ROB-SMT, while VB-FGMT and VB-SMT outperform ROB-SMT by about

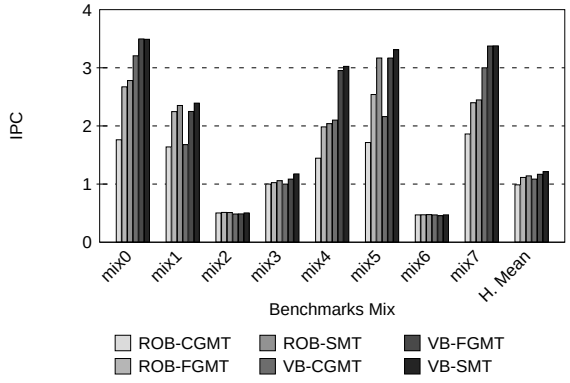


**Figure 3. Performance for different multithread designs in the ROB/VB architectures for different benchmark mixes.**

16.4% and 19.7%, respectively.

Mixes 2 and 6 show a flat behavior both when substituting the ROB by the VB and when improving the multithreading paradigm. This fact corroborates results of previous works [17] [2] [12], where it was shown that specific benchmarks do not obtain benefits neither from enlarging the ROB nor from retiring instructions out of order. This situation is caused by a lack of instruction level parallelism aggravated by a high L1 miss rate as well as by a high branch misprediction rate. Thread level parallelism is also affected by this fact, preventing SMT to outperform CGMT or FGMT.

An interesting observation is the average performance improvement of VB-FGMT, which is a simple design of multithreading, over ROB-SMT, which introduces more complex hardware in the issue stage to schedule instructions from different threads in the same cycle. The reason is that the benefits obtained by filling empty issue slots with instructions from various threads in ROB-SMT is compensated in VB-FGMT with the extra performance gained from

the efficient management of the VB structure, which prevents the pipeline from stalling so often.

### 5.2. Multithreading Scalability: Performance vs. Complexity

The complexity of a multithreaded processor varies across the different models of multithreading, as well as with the number of supported hardware threads $n$. This section trades off performance and complexity, by exploring how different VB-MT architectures behave when $n$ ranges from 1 up to 8. In each experiment, a specific benchmark is launched with as many instances as architected hardware threads.

Figure 4 shows the results using four different benchmarks, on the ROB and VB architectures implementing FGMT, CGMT and SMT. The top left graph contains the results of benchmark *gcc*, which shows a characteristic behavior of integer applications. As mentioned before, out-of-order retirement of instructions produces scarce benefits in performance for integer workloads, so the rest of this section will focus on results for floating-point benchmarks. The remaining graphs correspond to the floating-point benchmarks *mgrid*, *wupwise* and *art*, and show extremely contrasting results.

Looking at the *mgrid* graph, one can observe important effects as $n$ increases. On one hand, CGMT provides neither gain nor loss of performance up to 3 threads. In this case, the benefits of multithreading come from the fact of avoiding software context switches, which are not necessary in a system with $n$ logical processors executing $n$ software contexts. However, a further increase of $n$ has negative effects on the global IPC, shown more clearly in the case of VB-CGMT.

The FGMT and SMT curves belonging to the ROB architecture show well known effects of multithreading. A fine grain design reaches better performance up to 4 threads, while an SMT design is capable of exploiting thread level parallelism in a more scalable manner. Notice that this trend differs when instructions are retired out of order. In the VB architecture, performance grows abruptly for up to 4 threads on an SMT organization, and it stabilizes after this point. The reason is that a lower congestion in the VB with respect to the ROB allows other processor structures (e.g., issue queue or functional units) to be fed more aggressively, so a higher performance is reached with a lower value of $n$, and thus, a lower hardware overhead.

An interesting result can be observed by looking at the *mgrid* and *wupwise* curves: VB-FGMT provides better performance than ROB-SMT up to 4 threads. In other words, the fact of retiring instructions out of order makes a simple fine grain multithreaded organization outperform the simultaneous multithreading model with a complex *issue* stage in a ROB-based architecture. Although the ROB-SMT scalability is slightly imposed with values of $n$ higher than 4, the simpler VB-FGMT implementation can still be used reaching a higher performance up to approximately 4 threads.

### 5.3. Fetch Policies

This section studies the influence on the VB microarchitecture of several instruction fetch policies. All modeled processors follow the parameters shown in Table 1, except the one implementing DCRA, which has a shared instruction fetch queue, a shared issue queue and a shared load-store queue among hardware threads. The reason is that DCRA does not only assign different and variable fetch slots to threads, but also obtains benefits by dynamically assigning different number of entries of shared resources to threads.

Figure 5 depicts, on one hand, the pronounced advantage of a sophisticated instruction fetch policy in SMT processors. As results show, any fetch policy other than RR provides better benefits than the replacement of a ROB by a VB. On the other hand, Figure 5 illustrates that fetch policies advantages also apply to the VB architecture. If we compare the advanced fetch policies (the three right bars of the *Average* group) against the naive VB-RR policy, we obtain on average 28.4%, 32.9% and 40.2% benefits for VB-ICOUNT, VB-PDG and VB-DCRA, respectively. Comparing these three policies versus the ROB-DCRA policy (the best ROB-based policy), the performance speedup reaches 12%, 15.6% and 21.9%, respectively.

Although we need some improved fetch policy in the VB microarchitecture in order to outperform a ROB-DCRA architecture, there is no need to implement the most effective one (VB-DCRA), which might require more complex hardware. Instead, the instruction counters added by ICOUNT are sufficient to make the VB-based approach rise over ROB-DCRA. However, we can also see that the ability of retiring instructions out of order, combined with optimized fetch policies, allows the highest performance improvement, as both techniques contribute with their orthogonal potential.

### 5.4. Resources occupancy

As it is well known, SMT processors pursue to reduce the waste of issue slots, supplying a higher stress over functional units, and thus, a higher throughput of the execution stage. This section deals in depth with the reason why a VB-SMT design with a simple fetch policy outperforms an ROB-SMT with a complex one. To this end, it has been quantified how much these architectures stress the issue queue and functional units, by measuring the average issue slots used in each cycle.

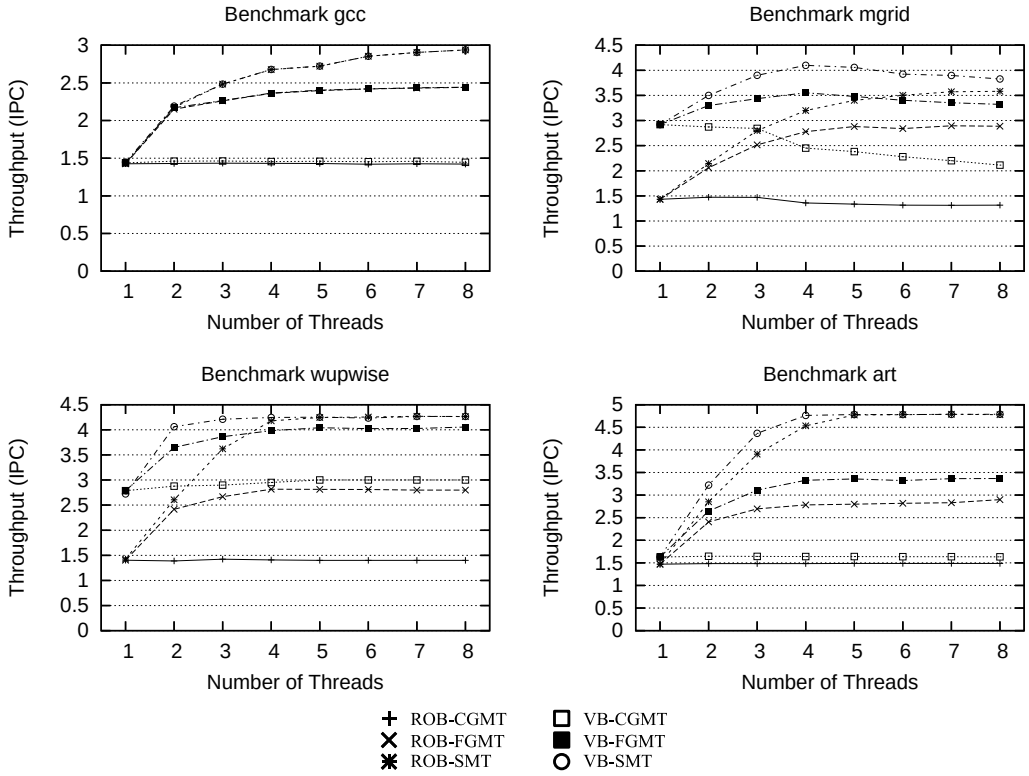


**Figure 4. Scalability of different multithread designs for ROB/VB architectures with a single replicated benchmark.**

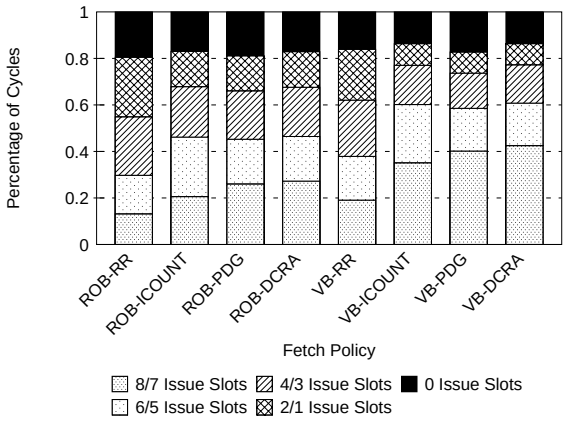


**Figure 6. Filled issue slots for different SMT architectures and fetch policies.**

Figure 6 represents the issue bandwidth utilization as a percentage of execution cycles in which a specific number of issue slots has been filled. Results clearly show how the VB architecture reduces the horizontal waste, since plotted regions corresponding to less than 7/8 issue slots are significantly smaller for VB-SMT. Vertical waste is also diminished for VB-SMT, which can be observed in the solid black regions, slightly smaller for the VB bars. These results also corroborate the relationship between filled issue slots and MT processor performance, when comparing Figures 5 and 6. The highest occupancy is achieved with VB-DCRA, which fills 7 or 8 issue slots in almost 50% of the execution time.

It is important to compare the bars corresponding to the ROB-DCRA and VB-ICOUNT designs, both in Figures 5 and 6. Figure 5 points ROB-DCRA as the most performance-effective ROB design, while Figure 6 shows that it exploits most efficiently the issue bandwidth. On the other hand, it is shown that a simple ICOUNT policy is enough to make VB-ICOUNT outperform ROB-DCRA (a naive RR fetch policy is not enough). This fact designates VB-ICOUNT as a cost-effective solution, reaching higher performance than the most complex fetch policy for ROB-

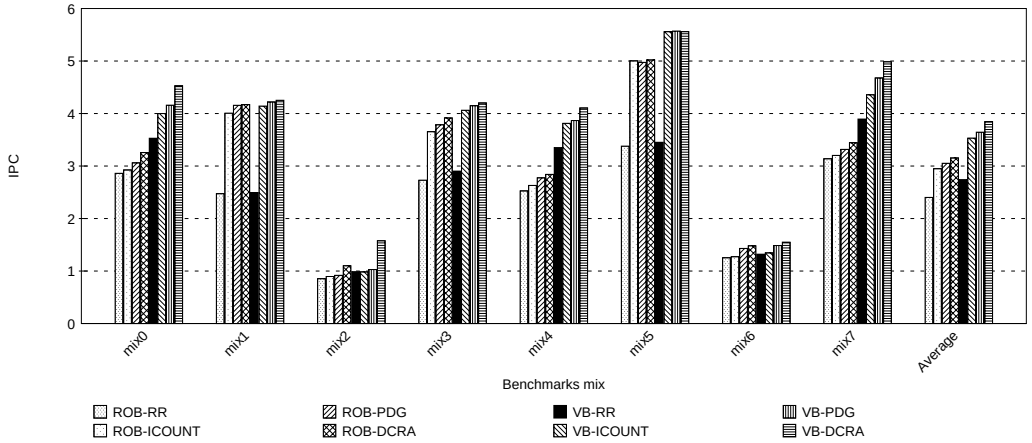


**Figure 5. Evaluation of fetch policies for the ROB and VB architectures**

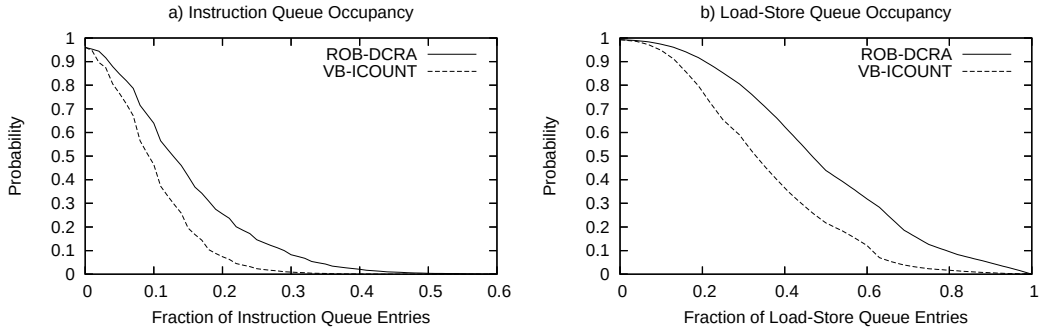


**Figure 7. Storage resources occupancy for ROB-DCRA and VB-ICOUNT.**

SMT, but implementing a simple fetch policy in VB-SMT.

Additionally to the issue slots, we have investigated the occupancy of storage resources, focusing on the most efficient ROB design (ROB-DCRA) and the design with the simplest efficient fetch policy on VB (VB-ICOUNT). Figure 7 shows the occupancy of the instruction queue (IQ) and the load-store queue (LSQ) for these designs. The curves in the graphs are to be interpreted as the probability (Y-axis) for a resource of having an occupation equal or greater than a specific fraction of its entries (X-axis).

As one can observe, VB-ICOUNT causes a lower occupancy both in the IQ and the LSQ. In the case of the IQ (Figure 7a), one can appreciate that only the 50% of the IQ entries are being used in ROB-DCRA, meaning that the IQ is over-dimensioned. However, the unused fraction of the IQ grows up to almost 70% in the case of VB-ICOUNT. Something similar happens with the LSQ, which could be implemented a 20% smaller, without practically affecting performance. As a consequence, VB-ICOUNT does not only outperform ROB-DCRA with a simpler fetch policy, but also permits a decrement of the main storage resources size, maintaining performance gains.

## 6. Related Work

Out-of-order instruction retirement proposals, like [1] or [4], replace the normal ROB with alternative structures to improve performance by speculatively retiring 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. This implies the enlargement of some major microprocessor structures, for instance, the register file [4] or the load/store queue [1], because completed instructions cannot free some critical resources until their associated checkpoint is released.

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 at 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).

A similar microarchitecture is proposed by Akkary et al in [1]. This proposal performs checkpoints selectively at low-confidence branches. In addition, to reduce resource requirements and improve scalability, this proposal uses an aggressive register reclamation scheme and a hierarchical store queue.

In [2], a checkpoint free, out-of-order retirement approach is presented. This proposal scans the $n$ oldest entries of the ROB to select instructions to be retired. 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. In this scenario, the ROB head may keep unaligned after the commit stage, and thus, the ROB must be collapsed to be aligned for the next cycle. Collapsing a large structure is costly in time and could adversely impact the microprocessor cycle. As a consequence, this proposal is unsuitable for large ROB sizes, which is the current trend. Moreover, the small number of instructions scanned at the commit stage significantly constrains the potential that this proposal could achieve.

In [20], a multiprocessor composed by out-of-order retirement processors, like the one presented in [4], is proposed. In this case, the checkpoint mechanism is extended to support transactional memory. Nevertheless, the authors do not include any discussion or analysis of a possible adaptation of their proposal into a multithreaded environment.

Finally, the Validation Buffer microarchitecture [17] is also a checkpoint free out-of-order retirement approach (see Section 2). This proposal has less resource requirements than a ROB-based processor and requires no additional complexity to manage the alternative structures. This fact makes it suitable for multiprocessor and multithreaded architectures.

## 7. Conclusions

Superscalar processors use a ROB structure to retire instructions in order. The way in which instructions in the ROB are handled has been proven to be poorly efficient when dealing with long latency instructions. This aspect has been addressed in recent research focusing in either single-threaded processors or multiprocessors, leading into alternative architectures in which instructions are retired out of order.

In this paper, we presented the extension of an out-of-order retirement architecture (the recently proposed Validation Buffer microarchitecture) with different models of multithreading. We also explored the behavior of different thread selection policies at the fetch stage (i.e., fetch policies) on the resulting multithreaded VB architecture. Results show that out-of-order retirement, as well as both multithreading and fetch policies, are techniques that contribute orthogonally to increase processor performance.

Our simulations are performed with a set of benchmark mixes, typically used for evaluation purposes on multithreading, and their results provide three main conclusions: (i) a fine-grain multithreaded VB-based processor outperforms, on average, a simultaneous multithreaded ROB-based processor; (ii) a simultaneous multithreaded VB-based processor reaches the maximum performance with about half the number of hardware threads than a simultaneous multithreaded ROB-based processor; (iii) benefits of fetch policies (such as DCRA) are orthogonal to the ones provided by the VB. These contributions justify the viability and cost-effectiveness of an out-of-order commit, multithreaded processor microarchitecture.

## Acknowledgements

This work was supported by Spanish CICYT under Grant TIN2006-15516-C04-01, by CONSOLIDER-INGENIO 2010 under Grant CSD2006-00046, and by Universidad Politécnica de Valencia under Grant PAID-06-07-20080029.

## References

[1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. of the 36th International Symposium on Microarchitecture*, Dec. 2003.

[2] G. Bell and M. Lipasti. Deconstructing commit. In *Proc. of the 2004 International Symposium on Performance Analysis of Systems and Software*.

[3] C. McNairy and R. Bhatia. Montecito: a Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, March-April 2005.

[4] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order Commit Processors. In *Proc. of the 31st International Symposium on High Performance Architecture*, Feb. 2004.

[5] A. El-Moursy and D. Albonesi. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proc. of the 9th International Conference on High Performance Computer Architecture*, Feb. 2003.

[6] F. J. Cazorla and A. Ramirez and M. Valero and E. Fernandez. Dynamically Controlled Resource Allocation in SMT Processors. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 171–182, 2004.

[7] G. Hinton, D. Sager, and M. U. et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal. Q1*, 2001.

[8] J. Sharkey and D. Balkan and D. Ponomarev. Adaptive Reorder Buffers for SMT Processors. In *Proc. of the 15 International Conference on Parallel Architectures and Compilation Techniques*, pages 244–253, 2006.

[9] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 Chip: a Dual-Core Multithreaded Processor. *IEEE Micro*, March-April 2004.

[10] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: an Alternative Approach. In *Proc. of the 26th International Symposium on Microarchitecture*, pages 202–213, Dec. 1993.

[11] P. Kongetira and K. Aingaran and K. Olukotun. Niagara: a 32-way Multithreaded Sparc Processor. *IEEE Micro*, March-April 2005.

[12] M. Pericás, A. Cristal, R. González, D. Jiménez, and M. Valero. A Decoupled KILO-Instruction Processor. In *Proc. of the 11th International Conference on High Performance Computer Architecture*, Feb. 2006.

[13] S. E. Raasch and S. K. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proc. of the* $12^{th}$ *International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[14] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Jul. 2004.

[15] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[16] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, Jun. 1995.

[17] R. Ubal, S. Petit, J. Sahuquillo, P. López, and J. Duato. The Validation Buffer Out-of-Order Retirement Microarchitecture. In *Proc. of the* $18^{th}$ *Jornadas de Paralelismo*, Sept. 2007.

[18] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *Proc. of the 19th International Symposium on Computer Architecture and High Performance Computing, www.gap.upv.es/˜raurte/multi2sim.html*, Oct. 2007.

[19] R. Ubal, J. Sahuquillo, S. Petit, P. López, and J. Duato. VB-MT: Design Issues and Performance of the Validation Buffer Microarchitecture for Multithreaded Processors. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2007.

[20] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenström, J. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. In *Proc. of the 2005 IEEE Conference on Pervasive Services*, Jul. 2005.