A Sequentially Consistent Multiprocessor Architecture for Out-of-Order Retirement of Instructions

R. Ubal†, J. Sahuquillo‡, S. Petit‡, P. López‡, and D. Kaeli†

†Electrical and Computer Engineering Department
Northeastern University, Boston, MA (USA)
{ubal, kaeli}@ece.neu.edu

‡Dept. of Computer Engineering (DISCA)
Universidad Politécnica de Valencia, Spain
{jsahuqui, spetit, plopez}@disca.upv.es

Abstract— **Out-of-order retirement of instructions has been shown to be an effective technique to increase the number of in-flight instructions. This form of runtime scheduling can reduce pipeline stalls caused by head-of-line blocking effects in the reorder buffer (ROB). Expanding the width of the instruction window can be highly beneficial to multiprocessors that implement a strict memory model, especially when both loads and stores encounter long latencies due to cache misses, and whose stalls must be overlapped with instruction execution to overcome the memory latencies.**

Based on the Validation Buffer (VB) architecture (a previously proposed out-of-order retirement, checkpoint-free architecture for single processors), this paper proposes a cost-effective, scalable, out-of-order retirement multiprocessor, capable of enforcing sequential consistency without impacting the design of the memory hierarchy or interconnect. Our simulation results indicate that utilizing a VB can speed up both relaxed and sequentially consistent in-order retirement in future multiprocessor systems by between 3% and 20%, depending on the ROB size.

Index Terms— **Out-of-order retirement, multicore processors, Validation Buffer, sequential consistency.**

I. Introduction

Multicore processors are now the current norm in both the general purpose and embedded systems processor markets. The move to multicore has mainly been prompted by the thermal issues associated with superscalar architectures and the difficulty of high frequency designs to exploit limited amounts of instruction-level parallelism. To address some of these issues, wider instruction windows are needed to attempt to hide long memory delays with computation, which in turn require large non-scalable microarchitectural structures (e.g., reorder buffers). Thus, further sources of concurrency must be obtained with the help of explicit parallelism.

There are still many factors present in single-thread performance that remain challenges in multicore designs. On one hand, the continued growth in chip integration allows complex designs to be considered that better balance the trade-off between the number of cores and increased core complexity. On the other hand, the intrinsic difficulties of parallel programming and the sequential nature of many existing applications limit the potential of parallel architectures that sacrifice single-thread performance.

One approach that can be explored to increase performance in single-threaded architectures is to utilize out-of-order retirement. This class of processors were originally proposed [1][2][3] as an effective solution to increase the instruction window size by relaxing the conditions under which instructions are retired from the pipeline. In this context, the Validation Buffer (VB) architecture has been proposed and evaluated for superscalar microarchitectures [4]. This solution works effectively while reducing the complexity of some major microarchitecture structures, such as the reorder buffer (ROB) or the register file. Since this technique can be considered a cost-effective mechanism, it becomes suitable for multicore environments, where energy dissipation is a major concern. However, out-of-order retirement of memory instructions raises a number of issues, specifically in terms of the memory consistency model used on parallel architectures.

A memory consistency model defines ordering of memory operations on shared memory multiprocessors and multi-core systems. Sequential consistency (SC) is the most restrictive memory consistency model. SC forces memory operations to be viewed by all processors in the same overall global order, which removes any complexity from the programming interface. This model is widely accepted and has been implemented in some commercial or prototype microprocessors [5][6][7].

The VB architecture supports relaxed consistency by design, but the implementation of an efficient checkpoint-free, sequentially consistent, out-of-order retirement multiprocessor is still an open problem. In this paper, we propose an implementation of sequential consistency on top of a VB-based multiprocessor. While the resulting architecture enforces strict global ordering of memory operations, it relaxes conditions to release pipeline resources, providing wider instruction windows that lead to performance gains over ROB-based multiprocessors. Experimental results show that i) a sequentially consistent VB multicore processor achieves speedups between 3% and 20% over a ROB-based design (depending on the ROB size assumed), and ii) a relaxation of the memory model (i.e., a multiprocessor based on unmodified baseline VB uniprocessor components) increases this speedup up to 10% for large ROB sizes.

The remainder of this paper is structured as follows. Sections II and III describe the baseline multiprocessor and the sequential consistency implementation used in this paper, respectively. Section IV presents the proposed out-of-order retirement multiprocessor architecture. Section V shows a performance evaluation, and finally Sections VI and VII discuss related work and provide concluding remarks, respectively.

II. Background

A. Out-of-Order Retirement

A traditional out-of-order execution processor uses a Reorder Buffer (ROB) to track all instructions in flight. The main aim of this structure is to provide the capability of recovering the machine state during the execution of any instruction in the case of a branch misprediction or an exception. Thus, the first condition for an instruction to exit the ROB is that its speculative state is resolved, that is, it becomes either non-speculative or mispredicted. The out-of-order execution and recovery from any of the in-flight instructions is additionally supported by a register renaming strategy that tracks all intermediate states inside the instruction window. Most register renaming strategies impose a second condition that instructions must be completed before exiting the ROB.

Our proposed out-of-order retirement architecture uses a structure called the *Validation Buffer* (VB). This structure has a similar function and implementation as the ROB, but instructions can leave it earlier. Specifically, the VB architecture implements an aggressive register renaming strategy (detailed below) which does not require instructions to be completed at the time they exit the VB. Thus, an instruction at the VB head can leave it as soon as its speculative state is resolved. An instruction leaving the VB is said to be *validated* when we can confirm that it has been correctly executed. On the contrary, it is said to be *invalidated* when it leaves the VB as mispredicted, invoking recovery to a previous processor state. As shown in previous work [4], the main benefits of the VB architecture are workload dependent, and come from a reduction of pipeline stalls. In a ROB-based processor, most pipeline stalls are due to long-latency instructions blocking the ROB head.

B. Register Renaming

Typically, modern microprocessors free a physical register when the next instruction renaming the corresponding logical register commits. At this point, it is known that all instructions reading this 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 processor pipeline, which makes it unsuitable for the VB microarchitecture.

Instead, the VB architecture uses a register reclamation strategy based on the counter method [8]. The hardware components used in this scheme, as shown in Fig. 1, are the Register Alias Table (RAT) and the Register Status Table (RST).

The RST is a table indexed by a physical register identifier. Each entry in the table contains three fields, called *pending*, *unmapped*, and *completed*, respectively. 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). The *unmapped* field is a bit that is set when the instruction that renames the corresponding logical register is retired from the VB as non-speculative. Finally, the *pending* field is a counter that tracks 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 the consumer instructions are dispatched, and decremented when they are issued (i.e., after they read their source registers)[1]. With this representation, a physical register can be considered free, and thus allocatable on new reclamations, when its producer has finished execution (i.e., *completed*=1), it has been remapped by a non-speculative instruction (i.e., *unmapped*=1), and there is no pending reader awaiting execution (i.e., *pending*=0).

The RAT is indexed by a logical register identifier and returns the associated physical register. There are two copies of this structure, named front-end RAT (FRAT) and retirement RAT (RRAT), respectively. At the retirement of an instruction, the RRAT matches the FRAT at the time this instruction was renamed.

Recovering the system from a misprediction involves restoring both the FRAT and the RST. A simple method to recover the FRAT is to wait until the offending instruction reaches the VB head, and then copy the contents of the RRAT into the FRAT. The RST 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 FRAT has been already recovered. Therefore, the draining overhead can be overlapped with new processor operations[2].

Additional details of the VB can be found in [4]. This paper focuses on an extension of the baseline out-of-order retirement architecture to be integrated into a multiprocessor environment. When reading the next sections, the VB architecture can be abstracted as a processor pipeline with an aggressive register renaming strategy that enables instructions to exit the VB before being completed, as soon as their speculative state is known.

C. Memory Instructions

In the VB architecture, memory instructions are internally split into two operations: address computation, which is inserted in the instruction queue (IQ), and memory access, which is placed in the load-store queue (LSQ). In the LSQ devised for the VB architecture, stores cannot be issued before older memory instructions [4]. However, loads are allowed to speculatively bypass

[1]A possible implementation for the pending counters is based on shift registers with 2 *inc* signals per decode slot and 2 *dec* signals per issue slot. The contents of the counters need not be routed out of the register file, and thus these signals are less costly than standard read/write ports. Moreover, the *dec* signal is raised at the same positions where source operands are read through standard ports in the issue stage. The *pending* counters take usually small values, and thus a limited number of bits can be used for the shift registers, stalling decode before their value overflows (4-bit shift registers have been found to cause negligible performance loss in practice).

[2]Due to the slight head-of-line blocking effect in the VB, a detected mispredicted branch takes an affordable amount of time (around 2 cycles on average) to reach its head. However, this time would cause a considerable penalty in a ROB-based machine if recovery was triggered at the commit stage. Thus, performance has been compared against an aggressive state-of-the-art ROB-based model capable of triggering recovery just after the mispredicted branch is resolved in the writeback stage, with a constant penalty of 1 clock cycle.

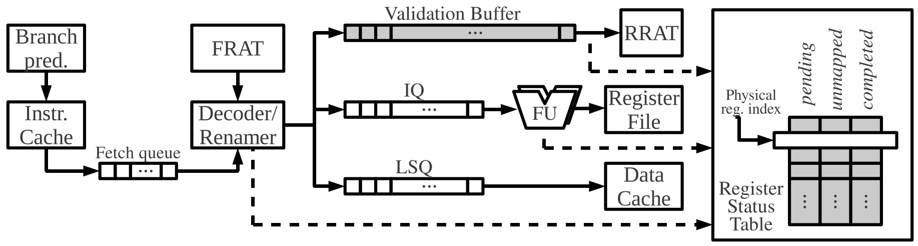


Fig. 1. VB architecture block diagram.

both loads and stores, and can get their values forwarded from a previous matching store. These optimizations, known as *load bypassing* and *load forwarding* respectively, use an additional structure, called *finished load buffer*, which tracks speculatively issued loads.

III. Dealing with Sequential Consistency

Due to the LSQ optimizations, memory accesses, as seen by other processors in a multiprocessor system, can be reordered in the VB architecture. This means that the originally proposed VB architecture is suitable when only a relaxed memory consistency model (RC) is imposed. In contrast, parallel programmers intuitively assume stricter models, such as sequential memory consistency (SC) [9], which, by definition, precludes the freedom of the system to arbitrarily alter the order of memory accesses. This conflict is solved in the RC models by providing programmers with mechanisms that can override the reordering of memory accesses when the semantic of the parallel program is compromised by this reordering. For example, it is known that if a parallel program is *data-race-free* [10] and *correctly labeled* [11] by synchronization operations, any reordering is allowed between synchronizations without affecting the program semantics.

Nevertheless, supporting the SC model is encouraged, since it allows us to reason about parallel programs assuming a simple behavior where memory operations are executed atomically one at a time and in program order, which is what most programmers expect. Therefore, aggressive implementations of this consistency model have been devised. Below, we present a formal description of the SC model and the aggressive implementation used in this paper as a baseline.

In a multiprocessor environment, a store is *globally performed* if no load to the same address in the system can return a value prior to it. A load is *globally performed* when its value is bound (it can be used by consumers of the same thread) and the store producing it is globally performed. Based on these definitions, two sufficient conditions have been presented [12] for a system to be sequentially consistent: i) every thread must issue memory accesses in program order, ii) after a memory instruction (load or store) is issued, the issuing thread waits for it to be globally performed before issuing its subsequent access. In-order (i) and atomic (ii) execution of memory instructions greatly hinders our ability to hide memory latency, and severely impacts performance.

Several techniques have been proposed to improve the performance of sequential consistency implementations (see Section VI). In this paper, *speculative retirement of loads* [13] is used as baseline implementation. This technique is based on the fact that both conditions just discussed can be speculatively ignored as long as the results of the speculative computations appear as if they were observed. In a modern multiprocessor system, a simple way to prevent remote processors from observing a local speculative computation is to monitor the state of the speculatively accessed cache blocks, and trigger a rollback whenever one of these blocks receives a remote invalidation (assuming an invalidation-based coherence protocol) or is evicted. In such cases, the processor must be able to recover its state prior to retiring the offending memory instruction.

Therefore, speculative memory instructions and subsequent ones must be recoverable until they are globally performed. A straightforward solution is to hold the ROB entries associated with the pending memory instructions until they are globally performed. This solution holds critical resources of the execution pipeline for recovering from memory consistency violations. For instance, traditional register renaming disallows physical registers to be released until instructions leave the ROB. However, as shown in Section IV, these misprediction events are rare. Moreover, critical resources are kept busy for long periods of time, since globally performing a memory instruction may involve long-latency coherence actions.

Speculative retirement of loads alleviates this waste by splitting the instruction window into two FIFO queues: the ROB and the History Buffer (HB). In this scheme, memory instructions and following operations can commit (releasing execution resources as they leave the ROB) even if they are still subject to memory consistency violations. After being committed, instructions enter the HB, where they remain until they are globally performed.

Each entry of the HB contains information so that we can undo the modifications performed on the processor state by its corresponding instruction. To support this ability, any instruction renaming (i.e., writing to) a given logical register holds in its HB entry two values. The first entry is the identifier l of its destination logical register; the second is the value of l before it was rewritten, that is, the contents of physical register p' (i.e., $RF[p']$) that were produced by the nearest previous (in program order) instruction renaming l, being p' the previous mapping of l[3]. Following this implementation, whenever a cache block accessed by a non-globally performed memory instruction is being remotely accessed or evicted, execution is recovered by squashing the contents of the ROB and undoing the changes logged in the HB.

Finally, store instructions need not be inserted into the HB, because they do not modify the processor state, and they are forbidden to write to the cache until they can be globally performed, that is, until the next instruction in program order

[3]In a ROB-based processor, the value of $RF[p']$ is always available at the commit stage. At this point in the execution, the state of register p' is changed to free by means of an RF write port. Notice that the construction of the undo-log at commit needs this port to support a read/write access in order to additionally retrieve the contents of p'.

TABLE I
FREQUENCY OF MISPREDICTION EVENTS.

Committed instructions	Branch mispredictions	Arithmetic exceptions
4,041,943,035	33,498,322 (0.8%)	0

Page faults	Load replay trap	Memory consistency
57,425 (< 0.01%)	331,180 (< 0.01%)	561,070 (0.01%)

is located at the HB head. Nevertheless, to prevent long-latency stores (i.e., those that involve remote block invalidations) from blocking the HB, the baseline SC implementation also includes *store prefetching* [14]. This technique allows stores to perform a *read-exclusive prefetch* before they are globally performed, thus reducing the odds of needing to invalidate remote copies when they exit the HB.

IV. Out-of-Order Retirement Multiprocessor Architecture

A. Architecture Description

To be able to extract an instruction from the VB in a single-core environment, the instruction must have its speculative state resolved, that is, either a completed branch or an instruction that is already known not to raise an exception. Table I lists different causes of misprediction that should be checked for before an instruction's speculative state is considered resolved. *Branch misprediction* refers to the resolution of a branch target address and direction that does not match the branch predictor outcome. *Arithmetic exception* and *page fault* refer to the resolution of the respective traps. *Load replay trap* refers to the resolution of the address of a store which was bypassed by a local load to the same address. Finally, *memory consistency* refers to the eviction/invalidation of an L1 cache block accessed by an in-flight memory instruction.

Table I attaches the total frequency of occurrence of each misprediction event during the execution of the SPLASH2 benchmark suite on a machine with the configuration shown in Section V. The table also includes the total number committed instructions. As observed, the frequency of occurrence is negligible —or even null for correctly written programs— relative to the number of committed instructions, except for branch mispredictions. Based on these results, the following sequentially consistent out-of-order retirement architecture is proposed.

After being dispatched, instructions enter the VB. This structure allows fast recovery on misprediction, but it prevents critical resources such as physical registers from being released by those instructions holding their VB entry. For this reason, the VB is responsible for resolving only those mispredictions that are likely to occur (i.e., branch mispredictions). After exiting the VB, instructions enter the HB. This structure provides a less efficient recovery mechanism, but it is decoupled from any other processor structure, such as the register file. Thus, infrequent misprediction events (i.e., all except branch mispredictions) can be resolved in the HB.

Fig. 2 lists the conditions to retire instructions at the head of each FIFO queue, both for the original uniprocessor ROB-based and VB-based models (Figs. 2a and 2b), and for the sequentially consistent ROB-based and VB-based multiprocessor architectures (Figs. 2a and 2b). Focusing on the sequentially consistent multiprocessors, the key difference observed when transitioning from a ROB-based scheme into the VB architecture

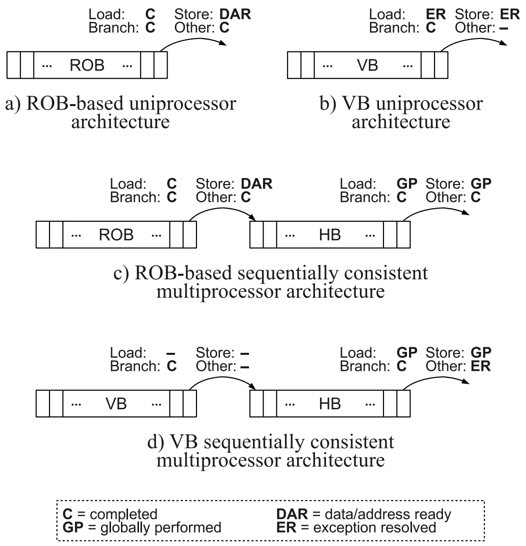


Fig. 2. Conditions for instructions to be retired from the ROB/VB and HB.

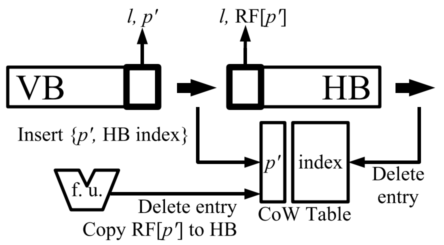


Fig. 3. Implementation of delayed writebacks

is a relaxation of retirement conditions both from the ROB and the HB, which leads to the following potential benefits:

- First, the only reason to stall an instruction at the VB head is that it is an unresolved branch. This relaxation removes most of the head-of-line blocking effects in the VB, enabling physical registers to be released much earlier. This effect is referred to hereafter as *enhanced register usage*.
- Second, instructions other than branches, loads, and stores can leave the HB in the VB architecture as soon as their speculative state is resolved, even if they are not completed or issued. As a consequence, the sum of the VB and the HB sizes does not limit the number of instructions in flight. This effect is referred to as *extended instruction window*.

B. Hardware Support

As instructions can leave the VB uncompleted, the contents of the physical register p' corresponding to the previous mapping of l (also RF[p']) may not be available to be copied to the HB. This problem can be solved either by blocking the VB exit until the contents to be copied are ready (thus adding an additional condition for instructions to leave the VB), or by allocating an entry in the HB whose contents will be written later. We choose the second option because it only requires little additional hardware support (explained below) to handle delayed writebacks to the HB. Likewise, instructions may leave the HB uncompleted, so this mechanism must consider that the writebacks should only occur as long as the corresponding HB entry is valid.

Fig. 3 shows a possible implementation of the supporting hardware, which uses a small CAM called *Copy-on-Writeback*

(CoW) table. Each valid entry in this table contains a pair $\{p'$, HB entry$\}$, which indicates that any result generated by the functional units for physical register p' should be forwarded to the corresponding HB entry. Considering that VB entries contain by design the fields l and p' (previous mapping of l) for each instruction, the mechanism works as follows.

If the contents of p' are ready when an instruction enters the HB (i.e., RST[p'].*completed*=1), they are straightforwardly copied to the allocated HB entry, with the same procedure as in the baseline architecture. Otherwise, a new entry in the CoW table is created, using p' and the index of the next free HB entry. When a functional unit generates a value for a destination physical register, the identifier of this register is associatively searched in the CoW. On hit, the value is also written back in the corresponding HB entry[4]. Instructions that leave the HB remove their associated entries in the CoW if present, avoiding overly delayed writebacks to affect the HB.

Notice that the random access at CoW positions is deadlock-free. Let's assume a processor state with all CoW entries occupied and the VB head stalled, waiting for a new delayed writeback to be scheduled in the CoW. Each current CoW entry is linked with an instruction in the HB, which will be drained after its final speculative state is resolved. Since the completion of in-flight instructions will eventually happen regardless of the VB stall, there is no cyclic dependence between events associated with CoW accesses, and thus, no potential for deadlocks.

When recovering from a mispredicted instruction in the HB, a sufficient condition to retrieve all the recovery information is to wait until the CoW table is empty (i.e., all delayed writebacks have been performed). Alternatively, the recovery process could only wait for the contents of the HB entries of canceled instructions. Finally, although the CoW table can be implemented as a direct-mapped table, an associative implementation is chosen, since a small CoW table suffices to prevent it from becoming a bottleneck for performance (see Section V).

V. Performance Results

A realistic multicore processor with 8 cores has been modeled to evaluate the proposed architectures. Fig. 4 shows a block diagram of the modeled system, whose characteristics are summarized in Table II. The memory subsystem consists of three levels of cache, where coherence adheres to the MOESI protocol. Separate L1 instruction and data caches are modeled, whereas L2 and L3 caches are unified (i.e., contain both code and data). This memory hierarchy and core interconnect is based on the recent commercial quad-core AMD Opteron 8350 processor [15]. In the figure, the gray boxes represent replicas of this same chip. Parallel workloads with shared data, i.e., the SPLASH2 benchmark suite [16], are used for our performance evaluations to stress shared components involved by coherence actions and long-latency memory operations. Programs are run until completion.

A. Out-of-order Retirement and Memory Consistency Model

This section presents performance results for the VB multiprocessor architecture implementing both relaxed and sequential consistency (referred to as VB-RC, and VB-SC, respectively), and compares them to ROB-based multiprocessors (ROB-RC and ROB-SC). In Figs. 5a, 5b and 5c, the results are presented as performance speedups over the ROB-SC architecture for 2-, 4-, and 8-core systems, respectively. The bar height represents the speedup achieved by VB-SC, whereas the circle- and triangle-ended lines represent the ROB-RC, and VB-RC architectures, respectively. Each bar/line in a group belongs to a different ROB/VB size, ranging from 8 to 128 entries, both for the represented architecture and the baseline. The implementation of the RC designs is modeled with the absence of an HB, as the order of memory operations can be safely altered.

The following observations can be made from the average results shown in the last groups of bars. Performance speedups are especially high in the VB architecture for small VB sizes, regardless of the memory consistency model used. The reason is that an 8 or 16-entry ROB is a very restrictive bottleneck in most applications, and is the main source of pipeline stalls, which are effectively avoided by the VB architecture.

Regarding the memory consistency model, results show that VB-SC outperforms both the ROB-SC and the ROB-RC in most cases (exceptions are when we consider large 64 and 128-entry ROBs for 8 processors). Compared to ROB-RC, the VB-SC architecture introduces the history buffer, which can serve as

[4]Additional accesses to the HB in the writeback stage require extra HB write ports in the VB multicore architecture. However, one single additional HB port can be shared by all functional units to dump their occasional delayed writebacks. Some tests have shown a negligible performance loss when temporarily stalling functional units that rarely contend for the shared HB port.

TABLE II
Baseline multicore processor parameters.

Processor Cores	
Machine width (decode, dispatch, issue, commit/ validate)	4 inst./cycle
Ld/st issue width	2 inst./cycle (limited by L1 ports)
Storage resources	40-entry IQ, 20-entry LSQ, 64-entry RF, 64-entry ROB, 64-entry HB
Functional units and latency (total/issue)	4 Int. add (2/1), 1 Int. mult. (5/2), 1 Int. div (20/10) 2 FP add (5/2), 1 FP mult. (10/5), 1 FP. div. (30/15)
Branch predictor	Hybrid (2-level + bimodal) 2-level pred.: 8-bit history, 1-entry L1, 1K-entry L2. Bimodal pred.: 1K 2-bit counters. Choice pred.: 1K entries.
Memory Hierarchy	
L1 caches	32KB, 2-way, 64-byte block, 2-cycle latency
L2 caches	512KB, 8-way, 64-byte block, 10-cycle latency
L3 caches	8MB, 16-way, 64-byte block, 50-cycle latency
Main memory	200-cycle access time

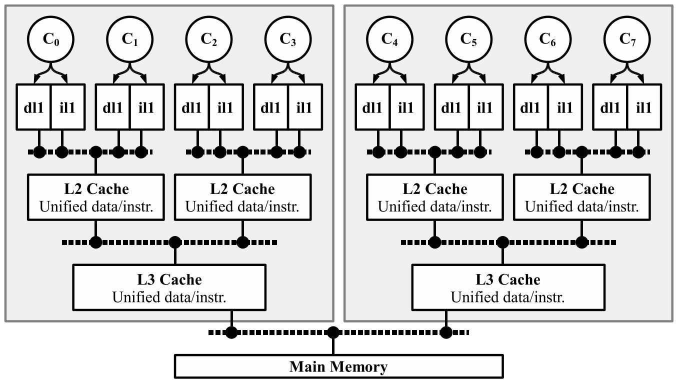
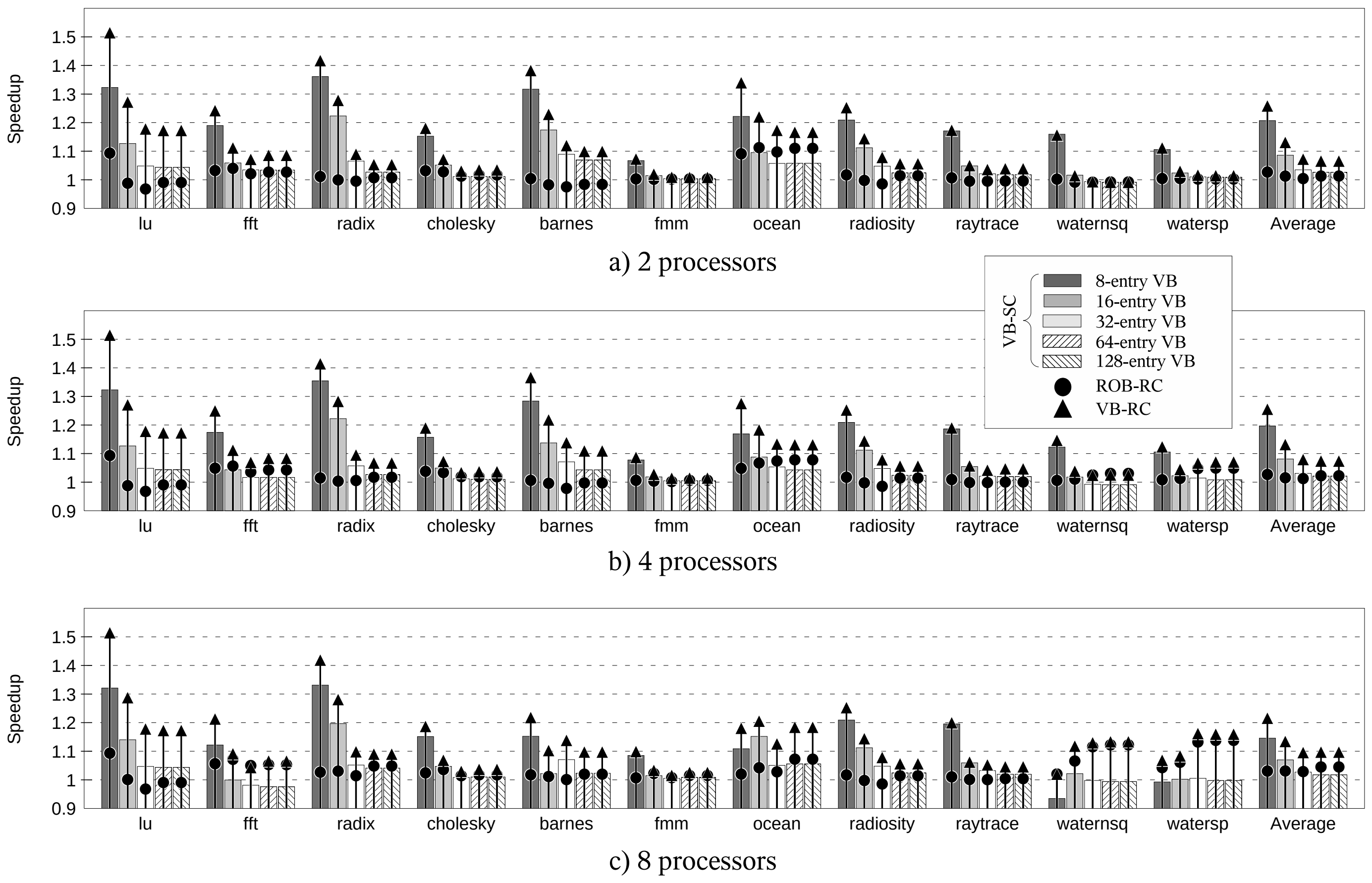


Fig. 4. Block diagram of the modeled multicore system.



a) 2 processors

b) 4 processors

c) 8 processors

Fig. 5. Performance speedups relative to ROB-SC (with variable ROB size).

an additional bottleneck; nevertheless, VB-SC speeds up the instruction flow from the VB into the HB, and allows instructions to be extracted early from the HB. As results show, this fact allows SC to be enforced while maintaining better performance.

Finally, notice that best performance results are shown by the VB-RC model, formed of individual VB-based uniprocessors disregarding memory consistency restrictions. This model reaches speedups of about 5% for large VB sizes, and up to 25% for small VB sizes. In this architecture, both loads and stores can leave the processor pipeline without being globally performed. Since these instructions are the main source of pipeline stalls, significant speedups are achieved, especially in memory-intensive applications.

VI. Related Work

This paper merges two mainstream research topics in the field of processor architecture: sequential consistency (SC) implementations and out-of-order retirement of instructions. There is a significant body of previous work in sequentially consistent multiprocessors. Performance enhancements such as *store prefetching*, and *speculative execution of loads* are considered in [14], and *speculative retirement of loads* is discussed in [13]; we have implemented these features in our baseline architecture due to their low complexity and effectiveness.

More sophisticated SC implementations can be found in the literature. In [17], speculative retirement of loads is improved by also retiring stores before their speculative state is confirmed. The fact that stores may commit stale values to the memory hierarchy forces a *history buffer* (in this context called *SHiQ*) to store the previous value at the written memory address. To get this value, SC++ requires stores to be implemented as read-modify-write operations in the cache. On misprediction, SC++ performs a burst of cache writes to roll back to a previous valid state. In this work, we avoid to impose this complexity in the cache hierarchy for the sake of generality.

SC++lite [18] is an improvement of SC++, based on the observation that the SHiQ is usually underutilized, although its storage is fully required during small periods of execution. To avoid its hardware overhead, the SHiQ is implemented directly in the memory hierarchy. On misprediction, SC++lite recovers at a slower rate than SC++, but consistency mispredictions are rare enough to afford it.

Finally, BulkSC [19] is another SC implementation where memory instructions are grouped in chunks, and appear to execute atomically and in isolation. The hardware enforces SC at a coarser grain (i.e., chunks), obtaining performance close to relaxed consistency implementations, by enabling optimizations in the execution of memory instructions.

Regarding out-of-order retirement of instructions in multiprocessor systems, a number of key papers have addressed this subject. In the Cherry-MP architecture [20], resources (e.g., physical registers) are released speculatively, entering into the so-called *cherry* mode by checkpointing previous valid machine states, which processors roll back to on future mispredictions. A Cherry-MP system supports both release and sequential consistency by setting up the conditions to release processor resources.

Unfortunately, Cherry-MP involves modifications in the cache hierarchy, such as the adaptation of the MOESI protocol, and also needs storage history about data shared among processors in *cherry* mode. This reduces its adaptability to generic memory systems, and checkpoints needed for managing speculation involve a considerable amount of hardware to be added to the processor pipeline.

The Kilo-Instruction Multiprocessors (or KIMPs) [21] have been also proposed as out-of-order retirement multiprocessor architecture. A KIMP enables many instructions to be in-flight by checkpointing the processor state when long-latency instructions block the pipeline and requires an aggressive register renaming mechanism. These checkpoints commit globally by locking a shared snoopy bus and broadcasting memory write accesses. SC is enforced by making remote processors roll back to previous checkpoints when an address match is snooped on the shared bus. This architecture imposes harsh restrictions on the system architecture, such as the presence of a shared bus (constraining scalability) and again the cost of several checkpoints in the processor pipelines.

Other out-of-order retirement uniprocessor architectures have been devised, including work by Cristal et al. [3], Akkary et al. [1], and Bell et al. [2]. However, to the best of our knowledge, no checkpoint-free out-of-order retirement architecture has been adapted and evaluated in a multiprocessor environment, especially when memory consistency is also considered.

The VB multiprocessor architecture is presented as an out-of-order retirement approach that uses no checkpoints, and imposes no restrictions on the underlying memory hierarchy, coherence protocol, or interconnection network. These features allow it to serve as a flexible and scalable approach. The differences with a traditional ROB-based multiprocessor are found in the register renaming strategy and the conditions for instructions to be retired from the VB. These structures are an integral part of most processor pipelines. The VB architecture is orthogonal to other optimizations in the multiprocessor system, including all of the SC implementations cited in this paper.

VII. Conclusions

In this paper, we have proposed a sequentially consistent (SC), out-of-order retirement, multiprocessor architecture. In this system, instructions are retired from processor pipelines out of program order by using a Validation Buffer approach, while still enforcing SC by using the *speculative retirement of loads* technique.

Our proposal focuses on three centralized processor components. First, conditions for instructions to leave both the VB and HB are relaxed, with no additional hardware cost. Second, renaming tables are handled with an alternative register renaming strategy, which decouples register release from the commit stage by means of little additional storage per physical registers. Finally, a small table (CoW) is introduced to support a delayed write back of destination operands into the HB. Since no loss of generality is incurred regarding the memory hierarchy or interconnects, the VB multiprocessor architecture remains highly scalable with the number of processors.

Our results provide three main conclusions: *i*) a sequentially consistent VB-based multiprocessor outperforms, in general, a ROB-based system implementing release consistency, regardless of the number of processors, *ii*) ROB, register file, and history buffer sizes can be reduced in the VB multiprocessor architecture, maintaining performance and lowering complexity, and *iii*) relaxation of instruction retirement conditions and memory model strictness can coexist without significantly impacting the performance of the VB architecture.

References

[1] H. Akkary, R. Rajwar, and S.T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. of the 36th Int'l Symposium on Microarchitecture*, Dec. 2003.

[2] G.B. Bell and M.H. Lipasti. Deconstructing Commit. In *Proc. of the The Int'l Symposium on Performance Analysis of Systems and Software*, Mar. 2004.

[3] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-Order Commit Processors. In *Proc. of the Int'l Symposium on High Performance Architecture*, Feb. 2004.

[4] S. Petit, J. Sahuquillo, P. López, R. Ubal, and J. Duato. A Complexity-Effective Out-of-Order Retirement Microarchitecture. *IEEE Transactions on Computers*, 58(12), Dec. 2009.

[5] K. C. Yeager. The Mips R10000 superscalar microprocessor. *IEEE Micro*, 16, 1996.

[6] A. Kumar. The HP PA-8000 RISC CPU. *IEEE Micro*, 17, March 1997.

[7] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The Bulk Multicore Architecture for Improved Programmability. *Communications of The ACM*, 52, 2009.

[8] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: an Alternative Approach. In *Proc. of the 26th Int'l Symposium on Microarchitecture*, Dec. 1993.

[9] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, Sept. 1979.

[10] Sarita Vikram Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Madison, WI, USA, 1993.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th Int'l Symposium on Computer Architecture*, May 1990.

[12] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *Proc. of the 14th Int'l Symposium on Computer Architecture*, June 1987.

[13] P. Ranganathan, V.S. Pai, and S.V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proc. of the 9th ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[14] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. of the Int'l Conference on Parallel Processing*, Aug. 1991.

[15] *AMD Opteron 8350 Quad-Core Processor – http://multicore.amd.com/us-en/quadcore/.*

[16] S.C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Int'l Symposium on Computer Architecture*, June 1995.

[17] C. Gniady, B. Falsafi, and T.N. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Int'l Symposium on Computer Architecture*, 1999.

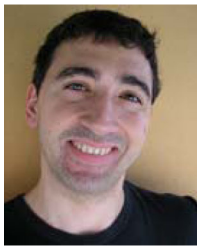
[18] C. Gniady and B. Falsafi. Speculative Sequential Consistency with Little Custom Storage. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[19] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proc. of the 34th Int'l Symposium on Computer Architecture*, 2007.

[20] M. Kirman, N. Kirman, and J.F. Martínez. Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors. In *Proc. of the Int'l Symposium on Microarchitecture*, Nov. 2005.

[21] E.Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J.E. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. In *Proc. of the IEEE Int'l Conference on Pervasive Services*, July 2005.

Rafael Ubal obtained his PhD Degree in Computer Engineering in 2010 from Universidad Politécnica de Valencia (Spain). He works currently as a Lecturer in the Electrical and Computer Engineering Department at Northeastern University (Boston, MA), and Postdoctoral Associate Researcher in the NUCAR group conducted by Prof. David Kaeli. His research topics of interest include power-aware cache designs, automatic parallelization of code, clustered/multithreaded/multicore architectures and GPGPU. He is the main developer of the Multi2Sim simulation framework, a CPU-GPU simulation platform for heterogeneous computing.



Salvador Petit received his PhD degree in Computer Engineering from Universidad Politécnica de Valencia (Spain). He is currently an Associate Professor in the Computer Engineering Department at the same university. His research topics include multithreaded and multicore processors, as well as memory hierarchy designs. He is a member of the IEEE Computer Society.

Julio Sahuquillo received his BS, MS, and PhD degrees in Computer Engineering from the Universidad Politécnica de Valencia (Spain). Since 2002 he is an Associate Professor at the Department of Computer Engineering. He has taught several courses on computer organization and architecture. He has published over 70 refereed conference and journal papers. His research topics include multiprocessor systems, cache design, instruction-level parallelism, and power dissipation. An important part of his research has also concentrated on the web performance field, including proxy caching, web prefetching, and web workload characterization. He is a member of the IEEE Computer Society.

Pedro López is a Full Professor at the Department of Computer Engineering at Universidad Politécnica de Valencia (Spain). He received his BS degree in Electrical Engineering and his MS and PhD degrees in Computer Engineering from the same university in 1984, 1990, and 1995, respectively. He has taught several courses on computer organization and architecture. His research interests include high performance interconnection networks for multiprocessor systems and clusters and networks on chip. He has published more than 100 refereed conference and journal papers. He is a member of the editorial board of Parallel Computing journal. He is a member of the IEEE Computer Society.

David R. Kaeli David Kaeli received a BS and PhD in Electrical Engineering from Rutgers University, and an MS in Computer Engineering from Syracuse University. He is the Associate Dean of Undergraduate Programs in the College of Engineering and a Full Processor on the ECE faculty at Northeastern University, Boston, MA where he directs the Northeastern University Computer Architecture Research Laboratory (NUCAR). Prior to joining Northeastern in 1993, he spent 12 years at IBM, the last 7 at T.J. Watson Research Center (Yorktown Heights, NY). He has co-authored more than 200 critically reviewed publications. His research spans a range of areas including microarchitecture to back-end compilers and software engineering. He leads a number of research projects in the area of GPU Computing. He presently serves as the Chair of the IEEE Technical Committee on Computer Architecture. He is an IEEE Fellow and a member of the ACM.