UNIVERSIDAD POLITÉCNICA DE VALENCIA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Out-of-Order Retirement of Instructions in Superscalar, Multithreaded, and Multicore Processors

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHYLOSOPHY
(COMPUTER ENGINEERING)

Author
RAFAEL UBAL TENA

Advisors
JULIO SAHUQUILLO BORRÁS
PEDRO LÓPEZ RODRÍGUEZ

VALENCIA, 2010

Agradecimientos

Para la realización de esta tesis ha sido necesario el esfuerzo conjunto de varias personas, así como el apoyo profesional y moral de muchas otras. En primer lugar, mis directores de tesis, Julio Sahuquillo y Pedro López, han sido fundamentales durante todo el proceso. Agradezco a Julio la confianza que depositó en mí cuando todavía era estudiante de Ingeniería Informática, instándome en todo momento a continuar con los estudios de doctorado y a unirme al mundo de la investigación. Sus conocimientos, su vocación y sus hábitos son un punto de referencia para mí. También agradezco a Pedro el tiempo que me ha dedicado en numerosas reuniones, sus aportaciones técnicas y su diligencia a la hora de orientar el enfoque del trabajo.

Además de mis directores de tesis, otros profesionales han contribuido de forma directa a dar solidez a este estudio. Salvador Petit ha sido una ayuda primordial, participando activamente en el desarrollo y la evaluación de las técnicas aquí propuestas. Su voluntad y tenacidad han favorecido la materialización y el progreso constante de las ideas iniciales. Asimismo, destaco la capacidad de José Duato para identificar líneas de investigación abiertas y susceptibles de ser explotadas. Sus sugerencias han sido de utilidad para configurar la idea original en que se basa esta tesis.

El esfuerzo diario se ha hecho más llevadero gracias a los que han sido o son actualmente mis compañeros de laboratorio: David, Noel, Héctor, Carles, Ricardo, Crispín, Paco, Samuel, Gaspar, Blas, José Miguel y Andrés. Ha sido una experiencia muy positiva el hecho de trabajar junto a tantas personas con las que intercambiar conocimientos y resolver dudas.

En la elaboración de esta tesis ha influido una estancia en Northeastern University (Boston) gracias al profesor David Kaeli, quien me acogió cálidamente en su grupo de investigación durante cuatro meses. Las reuniones semanales del grupo completo, en las que cada uno de sus miembros exponía artículos recientes y producciones propias de manera rotativa, fueron muy enriquecedoras. Tuve la suerte de trabajar en un ambiente multicultural, donde encontré excelentes compañeros. Especialmente, agradezco a Dana y Jenny su empeño en hacerme sentir integrado, en ayudarme a mejorar mi inglés, y en hacerme conocer muchos lugares interesantes de Estados Unidos.

Por último, doy gracias a mis padres, Francisca y Rafael, y a mis abuelos, Francisca y Miguel, por haber sido un apoyo incondicional y desinteresado en cualquier circunstancia. El interés que todos ellos han mostrado por mi labor ha sido esencial para mantener mi motivación e ilusión por el trabajo.

Contents

List of Figures

List of Tables

Abstract

Current superscalar processors use a reorder buffer (ROB) to track the instructions in flight. The ROB is implemented as a FIFO queue where instructions are inserted in program order after decoded, and from which they are extracted when they commit, also in program order. The use of this hardware structure provides a simple support for speculation, precise exceptions, and register reclamation. However, retiring instructions in program order may lead to a significant performance degradation if a long-latency operation blocks the ROB head. Several proposals have been published dealing with this problem. Most of them allow instructions to be retired out of order in a speculative manner, so they require checkpoints in order to roll back the processor to a precise state when speculation fails. Checkpoints management usually involves costly hardware and causes an enlargement of other major processor structures, which in turn might impact the processor cycle. This problem affects most state-of-the-art microprocessors, regardless of whether they are single- or multithreaded, or whether they implement one or multiple cores. This thesis spans the study of non-speculative out-of-order retirement of instructions in superscalar, multithreaded, and multicore processors.

First, the Superscalar Validation Buffer architecture is proposed as a processor pipeline design where instructions are retired out of program order in a non-speculative manner, hence without checkpoints. The ROB is replaced with a smaller FIFO queue, called Validation Buffer (VB), which can be left by instructions just after they are classified either as non-speculative or mispeculated, irrespective of their execution state. The management of the VB is complemented with an aggressive register reclamation technique that decouples physical register release from instructions retirement. The VB architecture largely alleviates the ROB performance bottleneck, and reduces complexity of other processor structures. For example, a ROB can be outperformed by a half as large VB, while decreasing its hardware cost.

Second, the Multithreaded Validation Buffer architecture is extended with different multithreading organizations, namely coarse-grain, fine-grain, and simultaneous multithreading. Multithreaded processors became popular as an evolution of superscalar processors to increase the issue bandwidth utilization. Likewise, out-of-order retirement of instructions contributes to reduce the issue waste by avoiding frequent

pipeline stalls due to a full ROB. The evaluation of the VB architecture on multithreaded processors shows again significant performance gains and/or a reduction of complexity. For example, the number of supported hardware threads can be reduced, or the multithreading paradigm can be simplified, without affecting performance.

Finally, the Multicore Validation Buffer architecture is presented as an out-of-order retirement approach on multicore processors, which define the dominant trend in the current market. Wide instruction windows are very 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 gap. The extension of the VB architecture to work on a multiprocessor environment allows core pipelines to retire instructions out of program order, while still enforcing sequential consistency. This proposal provides similar performance to ROB-based multiprocessor architectures implementing a relaxed memory model, and it outperforms in-order retirement, sequentially consistent multiprocessors.

Resumen

Los procesadores superescalares actuales utilizan un *reorder buffer* (ROB) para contabilizar las instrucciones en vuelo. El ROB se implementa como una cola FIFO (*first in first out*) en la que las instrucciones se insertan en orden de programa después de ser decodificadas, y de la que se extraen también en orden de programa en la etapa *commit*. El uso de esta estructura proporciona un soporte simple para la especulación, las excepciones precisas y la reclamación de registros. Sin embargo, el hecho de retirar instrucciones en orden puede degradar las prestaciones si una operación de alta latencia está bloqueando la cabecera del ROB. Varias propuestas se han publicado atacando este problema. La mayoría utiliza retirada de instrucciones fuera de orden de forma especulativa, requiriendo almacenar puntos de recuperación (*checkpoints*) para restaurar un estado válido del procesador ante un fallo de especulación. Normalmente, los *checkpoints* necesitan implementarse con estructuras hardware costosas, y además requieren un crecimiento de otras estructuras del procesador, lo cual a su vez puede impactar en el tiempo de ciclo de reloj. Este problema afecta a muchos tipos de procesadores actuales, independientemente del número de hilos hardware (*threads*) y del número de núcleos de cómputo (*cores*) que incluyan. Esta tesis abarca el estudio de la retirada no especulativa de instrucciones fuera de orden en procesadores superescalares, *multithread* y *multicore*.

En primer lugar, la arquitectura *Validation Buffer* superescalar se propone como un diseño novedoso del procesador en el que las instrucciones se retiran fuera de orden de manera no especulativa y, por tanto, prescindiendo de *checkpoints*. El ROB se remplaza por una cola FIFO de menor tamaño, llamada *validation buffer* (VB), que las instrucciones pueden abandonar en cuanto sean clasificadas como correctamente o incorrectamente especuladas, sea cual sea el estado de su ejecución. La gestión de la estructura VB se complementa con una estrategia agresiva de renombrado de registros, que desacopla la liberación de registros físicos de la etapa *commit*. La arquitectura VB alivia sensiblemente el cuello de botella impuesto por un ROB y reduce la complejidad de otras estructuras del procesador. Por ejemplo, un VB puede mejorar las prestaciones de un ROB con el doble de entradas, reduciendo por tanto el coste de su implementación.

En segundo lugar, la arquitectura *Validation Buffer Multithread* se propone y evalúa sobre procesadores *multithread* de grano grueso (*coarse-grain*), *multithread* de grano fino (*fine-grain*) y *multithread* simultáneo. Los procesadores *multithread* se popularizaron como una evolución de los procesadores superescalares para aumentar la utilización del ancho de banda de la etapa *issue*. Asimismo, la retirada de instrucciones fuera de orden contribuye a reducir el desperdicio de este ancho de banda impidiendo que instrucciones de alta latencia bloqueen el flujo de instrucciones cuando se llena el ROB. La evaluación de la arquitectura VB en procesadores *multithread* muestra de nuevo un incremento de las prestaciones y una reducción de la complejidad hardware. Por ejemplo, el número de hilos hardware soportados puede reducirse, o el paradigma de *multithreading* puede simplificarse, provocando en ambos casos una reducción del coste que no afecta a las prestaciones.

Por último, la arquitectura *Validation Buffer Multicore* se presenta como un entorno multiprocesador en el que las instrucciones se retiran fuera de orden. Los multiprocesadores, y en particular los chips *multicore*, constituyen la tendencia actual en el mercado de procesadores. Los multiprocesadores que implementan un modelo de memoria estricto se benefician sensiblemente de la incorporación de ventanas de instrucciones muy largas, especialmente en aquellas ocasiones en que las instrucciones de lectura o escritura en memoria producen fallos de *cache*, y cuyo tiempo de acceso es deseable solapar con otras instrucciones de cómputo. La extensión de la arquitectura VB a un entorno multiprocesador contribuye a este objetivo permitiendo que cada núcleo de cómputo retire sus instrucciones fuera de orden, mientras se continúa garantizando la consistencia secuencial en el acceso a memoria. Esta propuesta exhibe prestaciones similares a un multiprocesador basado en un ROB cuando este último implementa un modelo de memoria relajado, y las mejora si éste implementa un modelo de memoria estricto.

Resum

Els processadors superescalars actuals utilitzen un *reorder buffer* (ROB) per a comptabilitzar les instruccions en vol. El ROB s'implementa com una cua FIFO (*first in first out*) en la que les instruccions s'insereixen en ordre de programa després de ser descodificades, i de la que s'extrauen també en ordre de programa en l'etapa *commit*. L'ús d'aquesta estructura proporciona un suport simple per a l'especulació, les excepcions precises i la reclamació de registres. No obstant això, el fet de retirar instruccions en ordre pot degradar les prestacions si una operació d'alta latència està bloquejant la capçalera del ROB. Diverses propostes s'han publicat atacant aquest problema. La majoria utilitza retirada d'instruccions de forma especulativa, requerint emmagatzemar punts de recuperació (*checkpoints*) per a restaurar un estat vàlid del processador davant d'una fallada d'especulació. Normalment, els *checkpoints* necessiten implementar-se amb estructures hardware costoses, i a més requereixen el creixement d'altres estructures del processador, la qual cosa pot impactar en el temps de cicle de rellotge. Aquest problema afecta molts tipus de processadors actuals, independentment del nombre de fils hardware (*threads*) i del nombre de nuclis de còmput (*cores*) que incloguen. Aquesta tesi comprèn l'estudi de la retirada no especulativa d'instruccions en processadors superescalars, *multithread* i *multicore*.

En primer lloc, l'arquitectura *Validation Buffer* superescalar es proposa com un disseny del processador en què les instruccions es retiren fora d'orde de manera no especulativa i, per tant, prescindint de *checkpoints*. El ROB es reemplaça per una cua FIFO més petita, anomenada *validation buffer* (VB), que les instruccions poden abandonar en quant siguen classificades com correctament o incorrectament especulades, siga quin siga l'estat de la seua execució. La gestió de l'estructura VB es complementa amb una estratègia agressiva de nomenament de registres, que desacobla l'alliberament de registres físics de l'etapa *commit*. L'arquitectura VB alleuja sensiblement el coll de botella imposat per un ROB i redueix la complexitat d'altres estructures del processador. Per exemple, un VB pot millorar les prestacions d'un ROB amb el doble d'entrades, reduint per tant el cost de la seua implementació.

En segon lloc, l'arquitectura *Validation Buffer Multithread* es proposa i avalua sobre processadors *multithread* de gra gros (*coarse-grain*), *multithread* de gra fi (*fine-grain*) i *multithread* simultani. Els processadors *multithread* es van popularitzar

com una evolució dels processadors superescalars per a augmentar la utilització de l'ample de banda de l'etapa *issue*. Així mateix, la retirada d'instruccions fora d'ordre contribueix a reduir el malgaste d'aquest ample de banda impedint que instruccions d'alta latència bloquegen el flux d'instruccions quan s'ompli el ROB. L'avaluació de l'arquitectura VB en processadors *multithread* mostra de nou un increment de les prestacions i una reducció de la complexitat hardware. Per exemple, el nombre de fils hardware suportats pot reduir-se, o el paradigma de *multithreading* pot simplificar-se, provocant en els dos casos una reducció del cost que no afecta les prestacions.

Finalment, l'arquitectura *Validation Buffer Multicore* es presenta com un entorn multiprocessador en què les instruccions es retiren fora d'ordre. Els multiprocessadors, i en particular els xips *multicore*, constitueixen la tendència actual en el mercat de processadors. Els multiprocessadors que implementen un model de memòria estricte es beneficien sensiblement de la incorporació de finestres d'instruccions molt llargues, especialment en aquelles ocasions en què les instruccions de lectura o escriptura en memòria produeixen fallades de *cache*, i el temps d'accés de les quals és desitjable sobreposar amb altres instruccions de còmput. L'extensió de l'arquitectura VB a un entorn multiprocessador contribueix a aquest objectiu permetent que cada nucli de còmput retire les seues instruccions fora d'orde, mentre es continua garantint la consistència seqüencial en l'accés a memòria. Aquesta proposta exhibeix prestacions semblants a un multiprocessador basat en un ROB quan aquest últim implementa un model de memòria relaxat, i les millora si aquest implementa un model de memòria estricte.

Chapter 1

Introduction

This chapter introduces some concepts and presents the motivation for the work developed in this thesis. First, some basic notions about superscalar, multithreaded, and multicore processors are given. Then, it is shown how these architectures are affected by the problem of long-latency instructions stalling the processor pipeline. Finally, it is summarized how the rest of this dissertation deals with this problem by means of the Validation Buffer architecture proposal.

1.1 Background

1.1.1 Superscalar Processors

In the mid-to-late 1980s, superscalar processors began to appear as an attempt to break the bottleneck of executing a single instruction per cycle [1]. By initiating more than one instruction at a time, these processors can exploit the instruction-level parallelism (ILP) present in applications. The superscalar processing model has been implemented in commercial chips with different pipeline organizations, instruction queues, or storage structures. A possible block diagram of a superscalar architecture is shown in Figure 1.1. This model is used along this thesis as the baseline design, and its components are described next.

The processor front-end consists of an instruction cache from which instructions are fetched and placed into the fetch queue. The memory address of the instructions to be fetched is provided by the branch predictor fed by a program counter. Instructions at the head of the fetch queue are decoded, and name dependences introduced by the compiler are removed by a register renaming mechanism (detailed below). This mechanism is supported by the front and retirement register alias tables (FRAT and RRAT, respectively), as well as by the physical register file.

After decoded, instructions are dispatched into several queues, depending on the instruction kind. All instructions allocate an entry in the reorder buffer (ROB). This is a FIFO queue where instructions stay until the end of their lifetime in the processor pipeline. After the instruction at the head of the ROB is completed and its execution is confirmed, it leaves this structure and commits its result to the definitive machine state. Arithmetic operations allocate an entry in the instruction queue (IQ), from where they can be chosen for execution in the functional units at any time after their source operands become available. Memory instructions allocate an entry in the load-store queue (LSQ), from where they access the data cache when the memory address (and operand in the case of stores) is ready. Both the IQ and LSQ are associative queues, that is, entries are allocated and deallocated in random or-

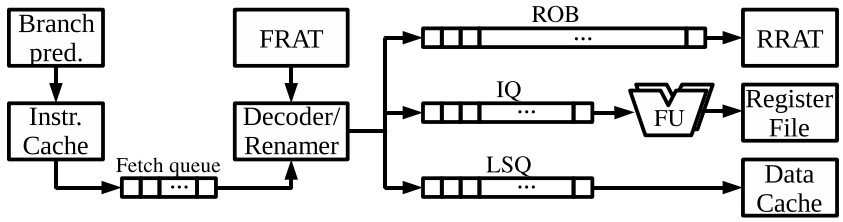


Figure 1.1: Block diagram of a superscalar processor pipeline (ROB = *reorder buffer*, IQ = *instruction queue*, LSQ = *load-store queue*, FU = *functional units*, FRAT = *front register alias table*, RRAT = *retirement register alias table*).

der. Arithmetic instructions finishing execution in a functional unit, as well as load instructions finishing the cache access, dump the obtained result into the physical register file.

Data dependences among instructions are tracked by a register renaming mechanism. Register renaming techniques distinguish two kinds of registers: logical and physical registers. Logical or architected registers refer to those used by the compiler, while physical registers are those actually implemented in the machine within the physical register file. Typically, the number of physical registers is quite larger than the number of logical registers. When an instruction that produces a result is decoded, the renaming logic allocates a free physical register, which is thereafter said to be mapped to the destination logical register of the instruction. After that, subsequent data dependent instructions rename their source logical registers so as to read this physical register. The logical-to-physical register mappings are stored in the FRAT. This table has as many entries as logical registers, and is accessed at the rename stage to obtain the current register mappings. Additionally, the RRAT contains a delayed copy of the FRAT, which is only updated by non-speculative instructions at the commit stage.

Control dependences are speculated by the branch predictor. The processor frontend provides the pipeline with a constant flow of instructions which are likely to belong to the correct path. However, speculation may fail, which is detected after a branch condition and target address are computed and any of them differ from the speculated value. On mispeculation, the processor state must be recovered to a previous consistent state, for example, just before the branch instruction was executed.

Processor recovery can be implemented either at the commit or at the writeback stage. In the first case, the processor waits for the mispeculated branch to reach the ROB head. At this time, the ROB contains only mispeculated instructions, whose state is exclusively held in the pipeline structures. Thus, all the processor has to do is squash the contents of the queues, and copy the RRAT into the FRAT to restore the valid mapping tables. Then, subsequent instructions start to be fetched from the correct path. Recovering mispeculation at the commit stage is a simple but not efficient approach, since mispeculation recovery might start a long time after its detection.

In the second case, recovery starts at the writeback stage as soon as mispeculation is detected just after the offending branch is resolved. Since the contents of the ROB, the FRAT, and other queues are only partially invalid, specific hardware must be devoted to selectively recover a valid processor state, by discarding mispeculated changes and maintaining correct in-flight instructions. Thus, recovering mispeculation at the writeback stage is more efficient and more costly at the same time. Since the proposal in this thesis changes the behavior of the commit stage, the recovery at writeback approach will be used for the baseline superscalar design for fair comparison purposes.

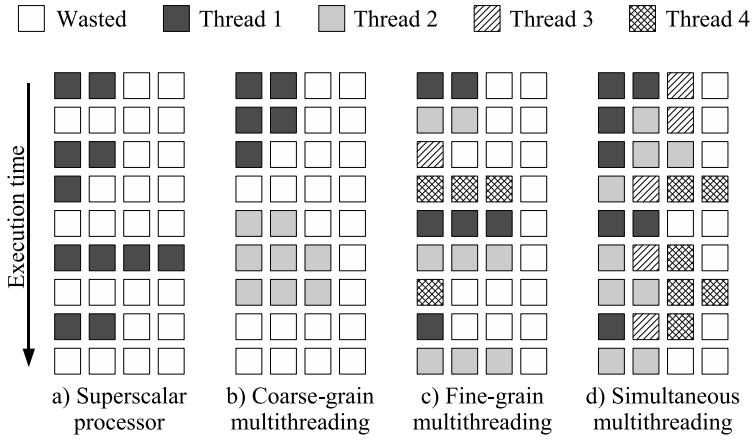


Figure 1.2: Issue bandwidth utilization in superscalar and multithreaded processors.

1.1.2 Multithreaded Processors

Until the early 1990s, a vast variety of sophisticated microarchitectural techniques were proposed to extract ILP. A higher integration scale allowed for wider processors with out of order execution [2], higher clock frequencies [3], or larger pipeline queues (IQ, LSQ). However, these techniques entailed an increasing complexity and power consumption, and the growing processor-memory gap made them less and less appealing. Consequently, researchers tried to develop new techniques based on the exploitation of the parallelism across multiple threads of control, or thread-level parallelism (TLP). Multithreaded architectures hold the state of different execution contexts in the same CPU. The key to increase processor performance is to increase the utilization of a shared pool of functional units (adders, multipliers, etc.), by providing them with a higher availability of instructions ready to be executed. Different multithreading paradigms can be distinguished depending on how execution resources are assigned to threads, namely, coarse-grain (CGMT), fine-grain (FGMT), and simultaneous multithreading (SMT).

To illustrate how each multithreading paradigm contributes to increase performance, Figure 1.2 plots the utilization of the issue bandwidth for a 4-way processor executing 4 threads. A white square represents a wasted issue slot, while a colored square represents an instruction issued from a specific thread's instruction queue. As observed in Figure 1.2a, a superscalar processor can incur two kinds of issue bandwidth waste, called *vertical waste*, which occurs in those cycles with all issue slots unused, and *horizontal waste*, which is incurred in those cycles when only a subset of the total issue bandwidth is used.

CGMT processors can only fetch and execute instructions from a single thread at a time. They partially alleviate the vertical waste by performing a thread switch

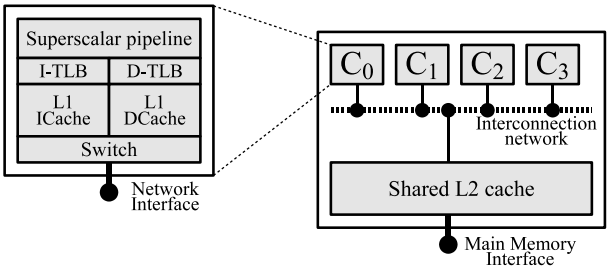


Figure 1.3: Example of a multicore processor.

whenever a long-latency event takes place, such as a cache miss. When this occurs, a not stalled thread starts to be fetched. The thread switch usually involves a penalty of some cycles until the stalled thread's instructions are drained from the processor. Thus, the vertical waste is not completely removed, as shown in Figure 1.2b.

FGMT processors fetch and issue instructions from a different thread in each cycle. The fetch logic usually works in a round-robin fashion among the available threads, skipping those stalled in a cache miss. The issue stage always works round-robin, skipping those threads in whose instruction queues no instruction is found ready to be executed. As observed in Figure 1.2c, the vertical waste is completely removed by avoiding thread switch penalties.

SMT processors increase the complexity of the issue stage by enabling instructions from different threads to be scheduled to the functional units in the same cycle. While a single thread may not have enough ready instructions to fill up the issue bandwidth in a given cycle, the ability to pick up instructions from different threads makes it more likely to take advantage of all issue slots. As Figure 1.2d shows, simultaneous multithreading mitigates both the vertical and horizontal waste.

1.1.3 Multicore Processors

After the success of multithreaded processors, technology of the 2000s kept on evolving and provided computer architects with a higher integration scale. Yet the power constraints remained, and the memory gap still needed to be bridged with the overlap of memory accesses and computations. Though a multithreaded processor allowed explicit parallelism to be exploited without requiring costly resources to hold long instruction windows, a scalable increase of the number of supported threads was no longer affordable to boost performance. This caused totally independent processing units (i.e., cores) to be integrated in a single chip.

In a homogeneous multicore processor, all elements of a superscalar processor pipeline are replicated per core. Each core usually owns its private L1 cache, and lower level caches may be shared or private among cores. When several caches are

present in the same level of the memory hierarchy, a cache coherence protocol is implemented to handle correct concurrent accesses to the same memory locations by different threads. Likewise, interconnection networks need to be introduced in the same die to communicate cores or caches sharing data. The topology of these interconnects may range from a simple bus to a complex point-to-point network, depending on the number of nodes they are giving service to and the traffic patterns observed among them. Figure 1.3 shows a block diagram of a possible multicore architecture with four embedded processing nodes with superscalar pipelines, private L1 caches, and a shared L2 cache.

1.2 Motivation and Challenges

Current high-performance microprocessors execute instructions out of order to exploit instruction level parallelism (ILP). In order to support speculative execution, provide precise exceptions, and register reclamation, a reorder buffer (ROB) is used [4]. 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. When a branch is mispredicted or an exception is raised, and the offending instruction reaches the commit stage, there is a guarantee that all previous instructions have already been retired, and no subsequent instruction has done it yet. At this time, all the processor has to do to recover a valid state is abort all instructions in flight.

However, this behavior is conservative. When the ROB head is blocked by a long-latency instruction (e.g., a load that misses in the L2 cache), subsequent instructions cannot release their ROB entries. This happens even if these instructions are independent from the long-latency one and they have been completed. In such a case, since instruction decoding continues, the ROB may eventually fill up, 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 [5].

To overcome this drawback, instructions can be committed out of program order, releasing resources early and causing a more efficient use of them. This approach raises a variety of challenges depending on the processor model considered.

These challenges are analyzed next for the three main processor architectures in the chronological order they showed up in research and industry, namely superscalar, multithreaded, and multicore processors.

1.2.1 Challenges in Superscalar Processors

Some solutions that commit instructions out of order have been published. These proposals can be classified in two approaches depending on whether instructions are speculatively retired or not. Some proposals falling into the first approach, like [6], allow the retirement of instructions obstructing the ROB head by providing a speculative value. Others, like [7] or [8], 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 a 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 [8] or the load-store queue [7], because completed instructions cannot free some critical resources until their associated checkpoint is released.

Regarding the non-speculative approach, Bell and Lipasti [9] propose to scan a few entries of the ROB, as many as allowed by the commit bandwidth, 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. Thus, instructions can be retired out of program order. In this scenario, empty intermingled slots may appear in the ROB after commit, so a defragmentation process is required to preserve its FIFO structure. Collapsing a large structure is costly in time and might adversely impact the microprocessor cycle, which makes this proposal unsuitable for large ROB sizes. In addition, the performance achieved by this proposal is constrained by the limited number of instructions that can be scanned at the commit stage.

Given the existing out-of-order retirement approaches, it remains a challenge to design an architecture that aggressively releases processor resources by neither performing checkpoints of the machine state nor collapsing a FIFO structure. In other words, we are looking for a non-speculative, out-of-order retirement approach where a sequential release of entries in hardware structures is decoupled from the commit stage.

1.2.2 Challenges in Multithreaded Processors

Multithreaded architectures represent an important segment in the industry. For instance, the Alpha 21464, the Intel Pentium 4 [3], the IBM Power 5 [10], the Sun Niagara [11], and the Intel Montecito [12] are commercial microprocessors included in this group. The utilization of processor resources is increased in multithreaded

processors by exploiting both instruction- and thread-level parallelism. While both coarse-grain and fine-grain multithreading contribute to reduce the vertical issue waste, only simultaneous multithreading tackles the horizontal waste, by issuing instructions from multiple threads in the same cycle. Nevertheless, this is done at the expense of adding considerable complexity to the issue logic, which is a critical stage in the processor pipeline.

On the other hand, the reduction of ROB stalls in an out-of-order retirement architecture allows instructions to pass more fluently through the processor pipeline. Specifically, the issue stage is affected with an income of instructions at a higher rate. Thus, these architectures mainly attack the vertical waste, although horizontal waste is indirectly also improved. Out-of-order retirement and multithreading orthogonally contribute reducing the issue waste, so it is a research opportunity to integrate these techniques and evaluate their joint behavior.

1.2.3 Challenges in Multicore Processors

Multicore processors are now the current norm in both the general purpose and embedded system 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, very wide instruction windows are needed to hide memory latency with computation, which in turn requires 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. In this sense, the continued growth in chip integration density 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.

Out-of-order retirement of instructions can increase processor performance by providing larger instruction windows, without the necessity of increasing the complexity of the major microprocessor structures, such as the ROB or the register file. With a suitable implementation, a cost-effective solution might be achieved, which would be of great interest in the multicore field, where energy dissipation is a major concern. New challenges arise when out-of-order retirement is introduced in a multiprocessor environment, specifically in terms of the memory consistency model used on parallel architectures.

A memory consistency model defines ordering of memory operations on shared memory multiprocessor and multicore 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 eases the intuitiveness of the programming interface. This model is widely accepted and has been implemented in some commercial microprocessors, such as MIPS R10000, but the implementation of an efficient, sequentially consistent, out-of-order retirement multicore architecture is still an open problem.

1.3 Objectives of the Thesis

The main objective of this thesis is to tackle the challenges highlighted in the previous section. Regarding superscalar processors, a non-speculative, out-of-order retirement architecture is pursued, which at the same time avoids costly operations on the major microprocessor structures, such as ROB collapsing, or checkpoints management. In the field of multithreaded processors, an integration of multithreading paradigms, resource-to-thread allocation policies, and out-of-order retirement is sought, which are three techniques that potentially increase the issue bandwidth utilization. And regarding multicore processors, we intend to design a multiprocessor architecture composed by several out-of-order retirement cores that enforces the well-known and widely accepted global sequential order of memory operations (i.e., sequential consistency).

1.4 Contributions of the Thesis

In this dissertation, the Validation Buffer (VB) architecture is proposed as a processing model where instructions are retired out of program order. This proposal is shown to behave efficiently in the processor organizations currently dominating the market, that is, it improves performance at a lower hardware cost. The contributions of this thesis can be summarized as follows:

- An energy-efficient out-of-order retirement architecture is devised, which, unlike previous proposals, needs no checkpoints to handle speculation.
- It is shown that the architecture of multithreaded processors can be extended to implement out-of-order retirement, causing an orthogonal increase of the total issue bandwidth utilization.
- Although out-of-order retirement makes a strict memory model hard to implement, an extension of the proposed architecture is devised, which enforces sequential consistency while releasing pipeline resources out of program order.

The VB architecture uses a FIFO structure analogous to the ROB, called Validation Buffer (VB), where instruction retirement conditions are relaxed. 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 previous branches and previous exceptions are resolved, the execution mode of an instruction changes either to non-speculative or mispeculated. At that point, instructions are allowed to leave the VB. Instructions leave the VB in program order but, unlike in ROB-based designs, they may not be held in the VB until they are retired from the processor pipeline (i.e., functional units, instruction queues, etc.). Instead, instructions remain in the VB only until their speculative state is resolved, so they can leave this structure while being in any state (completed, issued, or just decoded and not issued). For example, a long-latency memory instruction can leave the VB as soon as its memory address is computed without having raised a page fault.

Superscalar processors implement register renaming based on the fact that instructions leave the processor pipeline in program order [13][14][3]. Since the VB architecture does not track in-flight instructions as traditionally, an aggressive register renaming strategy and reclamation method are devised, which do not rely on a sequential release of pipeline resources at the commit stage. As evaluated in Chapter 3, the superscalar VB microarchitecture further exploits the available instruction-level parallelism, while requiring less complexity in some major critical resources, such as the register file or the load-store queue. It is also shown that the VB architecture can outperform in-order retirement architectures even when using narrower pipelines. This has important implications, since the pipeline width has a strong impact on the processor complexity.

Regarding multithreaded processors, the impact of out-of-order retirement of instructions is analyzed for three main models of multithreading, namely fine-grain (FGMT), coarse-grain (CGMT), and simultaneous multithreading (SMT). A pipeline design is proposed with proper resource sharing strategies, and existing allocation policies of shared resources are evaluated on top of it. As shown in the evaluation in Chapter 4, three important conclusions arise. First, an out-of-order retirement SMT processor requires in most cases half of the hardware threads than a 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 out-of-order retirement FGMT processor outperforms a 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, by orthogonally contributing to increase the issue bandwidth utilization. An SMT design can implement both techniques to provide higher performance if area, power consumption, and hardware constraints allow it.

In the multicore field, an implementation of an out-of-order retirement, sequentially consistent multiprocessor architecture is proposed, based on the speculative

retirement of load instructions. 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. The experimental evaluation presented in Chapter 5 shows important speedups for a sequentially consistent VB-based multiprocessor with respect to its ROB-based homologous. Likewise, an evaluation of the VB-based design with a relaxed memory consistency model is performed, showing higher speedups for large ROB sizes.

Finally, the work involved in this thesis includes the development of a simulation framework, called Multi2Sim, on top of which all performance evaluations have been carried out. This simulator models in a cycle-accurate manner the architecture of a superscalar, multithreaded, and multicore processor, as well as the underlying memory hierarchy and interconnection networks. The construction of this tool started with the necessity of modeling these complex systems working as a whole, and the unavailability of free and efficient simulators providing these characteristics. Currently, Multi2Sim has evolved into a formal open-source project aimed at being exploited by the research community in further works.

1.5 Thesis Outline

The rest of this dissertation is structured as follows. Chapter 2 presents the Multi2Sim simulator and the experimental framework. Chapter 3 describes and evaluates the Validation Buffer architecture for superscalar processors. This proposal is extended and evaluated for multithreaded and multicore processors in Chapters 4 and 5, respectively. Chapter 6 summarizes some published works related with this thesis, and finally, Chapter 7 presents some concluding remarks.

Chapter 2

The Multi2Sim Simulation Framework

Current microprocessors are based in complex designs, integrating different components on a single chip, such as processor cores with several hardware threads, a memory hierarchy with several cache levels, and interconnection networks. Complex simulation tools are required to model these systems, and to evaluate the global impact on performance of alternative designs in specific components. At the time this work started, there was no publicly available tool fulfilling the simulation necessities for the evaluation of the proposed techniques. The Multi2Sim simulation framework has been developed as an open-source project, with the aim of overcoming the drawbacks of existing simulation tools, and evaluating the proposals of this thesis. In this chapter, the simulation environment is described, including Multi2Sim's structure and main features, as well as the benchmarks run on top of it.

2.1 Overview

The evolution of microprocessors, mainly enabled by technology advances, has led to complex designs that combine multiple physical processing units in a single chip. These designs provide for the operating system the view of having multiple processors, and thus, different software processes can be scheduled at the same time. This processor model consists of three major components: the microprocessor cores, the cache hierarchy, and the interconnection network. A design modification on any of these components can affect the rest of them, and cause specific global behaviors. Therefore, the entire system should be modeled in a single tool that tracks the interaction between components.

An important part of the work of this thesis has focused on the development of the Multi2Sim simulation framework, which covers the limitations of other existing multiprocessor simulators. Multi2Sim integrates a model of the processor cores, the memory hierarchy, and the interconnection networks in a tool that enables their joint evaluation. Next, a brief comparative study is presented, focusing on Multi2Sim and other state-of-the-art simulation frameworks.

2.1.1 Existing Simulation Tools

Multiple simulation environments aimed at evaluating computer architecture proposals have been developed. The most widely used simulator in recent years has been SimpleScalar [15], which models an out-of-order superscalar processor. A vast amount of extensions has been applied on top of SimpleScalar to model in a more accurate manner certain aspects of superscalar processors. For example, the HotLeakage simulator [16] quantifies leakage energy consumption. SimpleScalar is quite difficult to extend to model new parallel microarchitectures without significantly changing its structure. In spite of this, various SimpleScalar extensions to support multithreading have been implemented, e.g. SSMT [17], M-Sim [18], or SMTSim [19], but they have the limitation of only executing a set of sequential workloads and implementing a fixed resource sharing strategy among threads.

Multithread and multicore extensions have been also applied on top of the Turandot simulator [20] [21], which models a PowerPC architecture. This tool has also been used with power measurement aims in an implementation called PowerTimer [22]. Some publications cite the use of Turandot extensions to model parallel architectures (e.g. [23]), but they are not publicly available.

Both SimpleScalar and Turandot are application-only tools, which directly simulate the behavior of an application without first running an operating system. Such tools have the advantage of isolating the workload execution, so statistics are not affected by the simulation of additional software. Moreover, the simulation time required to run benchmarks can decrease by about two orders of magnitude, in-

creasing the simulation capabilities and flexibility. Multi2Sim is also classified as an application-only simulator.

In contrast to the application-only simulators, a set of so-called full-system simulators are available. Similarly to virtual machines, these tools boot an unmodified operating system, on top of which applications are then run. Although this model provides higher simulation power, it involves a huge computational load and, depending on the goal of the study, unnecessary simulation accuracy. Simics [24] is an example of a generic full-system simulator, commonly used for multiprocessor systems simulation, but unfortunately not freely available. A variety of Simics derived tools has been implemented for specific research purposes in this area. This is the case of GEMS [25], which introduces a timing simulation module to model a complete processor pipeline, a memory hierarchy, and cache coherence. However, GEMS provides low flexibility to model multithreaded designs and does not integrate an interconnection network model, while still adding a sensible amount of computational overhead and sometimes prohibitive simulation times.

An important feature included in some processor simulators is the *timing-first* approach, provided by GEMS and adopted by Multi2Sim. In such a scheme, a timing module traces the state of the processor pipeline while instructions traverse it, possibly in a speculative state. Then, a functional module is called to actually execute the instructions, so the correct execution paths are always guaranteed by a previously developed robust functional simulator. The *timing-first* approach confers efficiency, robustness, and the possibility of performing simulations on different levels of detail. Multi2Sim adopts the *timing-first* simulation with a functional support that, unlike GEMS, need not simulate a whole operating system, but is still capable of executing parallel workloads, with dynamic threads creation.

The last cited simulator is M5 [26], which provides support for out-of-order SMT-capable CPUs, multiprocessors and cache coherence, and runs in both full-system and application-only modes. The limitations lie in the low flexibility of multithreaded pipeline designs. The next sections focus on the design and implementation of some components of the baseline Multi2Sim tool, and it is discussed how the architectural techniques proposed in this thesis are modeled on top of it.

2.1.2 The Multi2Sim Project

Three subprojects have been started and are currently maintained related with the Multi2Sim simulation framework: the Multi2Sim web site [27], the simulator source code, and the tool documentation. The web page is organized as a *wiki* page, and the rest of the projects reside in a publicly accessible SVN server. Anybody is allowed and encouraged to join any of these subprojects, and their format is intended to allow and manage access of concurrent developers. These are some properties of each subproject:

- **Multi2Sim web site**. This web site provides general information about the simulator, and includes links for interesting downloads. On one hand, both the simulator source code project and the documentation project can be directly downloaded. On the other hand, several sets of precompiled benchmarks are available. The provided binaries have been tested and their execution has been validated with Multi2Sim. Finally, a mailing list service is available for anybody to freely subscribe or unsubscribe, to share or ask for information to other users of the tool. The web site can be improved by anybody after requesting an account to the administrator.

- **Multi2Sim source code**. Contained in an SVN tree, the simulator source code is available for read-only access straightforwardly, and for read-write access for anybody who requests it by contacting the administrator. The code tree contains three first-level directories. The directory *trunk* holds the most recent copy of the simulator, which is usually unstable and in progress. The directory *tags* contains checkpoints of released versions. Finally, the directory *branches* contains copies of the trunk for different purposes, such as marginal modifications that are not intended to affect the central version.

- **Multi2Sim documentation**. The documentation is a PDF file, generated from a set of Latex source files, intended to be a user's and programmer's guide for Multi2Sim. It is currently in its first phase, and the structure and access modes of this subproject are the same as for the source code subproject.

In the following sections, the simulator structure and the processor model are described, including the superscalar architecture, the implementation of parallel architectures, and the memory subsystem. For each component, the associated set of Multi2Sim command-line options for its configuration is given.

2.2 The Superscalar Pipeline Model

Multi2Sim models a pipelined superscalar processor, capable of fetching, decoding, and executing Intel x86 instructions. In this model, a high-speed fetch stage is supported by a branch prediction mechanism, and a register renaming strategy is employed to track data dependences among instructions. Both the branch prediction and register renaming schemes are detailed in this section, followed by a description of the implemented pipeline stages.

2.2.1 Branch Predition

There are two different components involved in branch prediction: the *branch target buffer* (BTB) and the *branch direction predictor* (or simply *branch predictor*). The

BTB is a set-associative cache indexed by a macroinstruction address. A branch address is present in the BTB if any entry in the corresponding set contains a tag matching the address. In this case, the associated BTB entry contains the target address of the branch and the type of branch (conditional branch, inconditional jump, function call, or function return). The command-line option to specify the BTB organization is `-bpred:btb <sets>:<assoc>`. The argument `sets` is a power of 2 indicating the number of sets of the BTB, while `assoc` refers to the number of ways or associativity of the BTB, also a power of 2.

On the other hand, the branch predictor provides the direction of a branch located at a given address, i.e., whether it is taken or not. The branch predictor kinds modeled in Multi2Sim are *perfect*, *taken*, *not-taken*, *bimodal*, *two level adaptive*, and *combined*, which can be selected with option `-bpred <perfect>|<taken>|<nottaken>|<bimod>|<twolevel>|<comb>`. Each predictor type is described next.

- **Perfect branch predictor**. The *perfect* predictor (option `-bpred perfect`) is an ideal implementation that provides a totally accurate prediction. Accesses to an ideal BTB always return the correct target address even if the branch address was not inserted before, and accesses to the branch predictor always return the right direction. This implementation is unfeasible in hardware, but provides a useful upper bound for the performance achieved by other branch predictors.

- **Taken branch predictor**. The *taken* predictor (option `-bpred taken`) assumes that branches are always taken. However, instructions at the front-end are not decoded yet, and branches are identified as such only when their address is present in the BTB (see Section 2.2.3). Thus, a branch absent in the BTB is considered as a regular instruction, and is assumed not to jump, even with the *taken* branch predictor.

- **Not-taken branch predictor**. The *not-taken* predictor (option `-bpred nottaken`) assumes that conditional branches are never taken. However, it still predicts as taken those branches that are certainly known as such, that is, inconditional branches, function calls, and returns whose address is contained in the BTB.

- **Bimodal branch predictor**. A *bimodal* predictor (option `-bpred bimod`) is a table indexed by the least significant bits of the instruction address. The entries of the table are 2-bit up-down saturating counters. A counter represents the current prediction for a given branch. Values of 0 and 1 represent a not-taken prediction, while values 2 and 3 mean that the branch is taken. The number of entries in the table is a power of 2 given by options `-bpred:bimod <size>`.

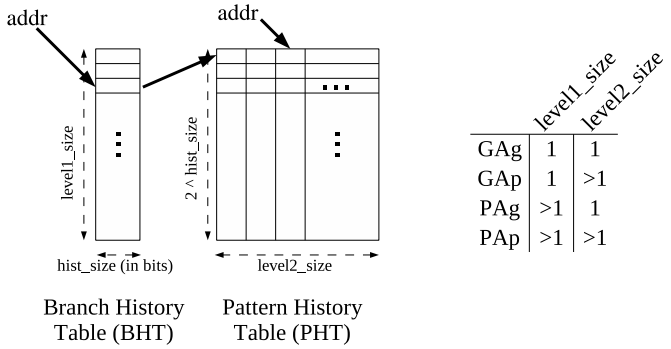


Figure 2.1: Two-level adaptive branch predictor.

- **Two-level adaptive predictor**. A *two-level adaptive* predictor (option `-bpred twolevel`) contains two tables, each corresponding to one prediction level. The additional command-line option involved with this predictor is `-bpred:twolevel <level1_size> <level2_size> <hist_size>`, which defines specific parameters for each predictor component.

  As shown in Figure 2.1, the first accessed table is the Branch History Table (BHT). This table is indexed by the least significant bits of the branch instruction address, and contains `level1_size` entries (power of 2). Each entry consists of a branch history register of `hist_size` bits that indicates the behavior of the last `hist_size` occurrences of the branch. Every time a branch commits, this register is shifted left, and the least significant bit is set or cleared according to whether the branch was actually taken or not.

  The content of the history register obtained from the BHT is used to index the row of a second two-dimensional table called Pattern History Table (PHT). Because the history register has `hist_size` bits, the PHT is forced to have 2^`hist_size` entries. The column of the PHT is also indexed by the least significant bits of the branch instruction address. The number of columns in the PHT is given by the `level2_size` parameter. Each entry in the PHT contains a 2-bit up-down saturating counter that gives the final prediction for the inquired branch.

  By properly tunning the parameters of option `-bpred:twolevel`, one can form the four two-level adaptive configurations commonly known as GAg, GAp, PAg, and PAp. See [28] for a more detailed description about these predictors. The table shown on the right of Figure 2.1 lists the restrictions that the predictor parameters should fulfill in order to be classified as each of the cited configurations.

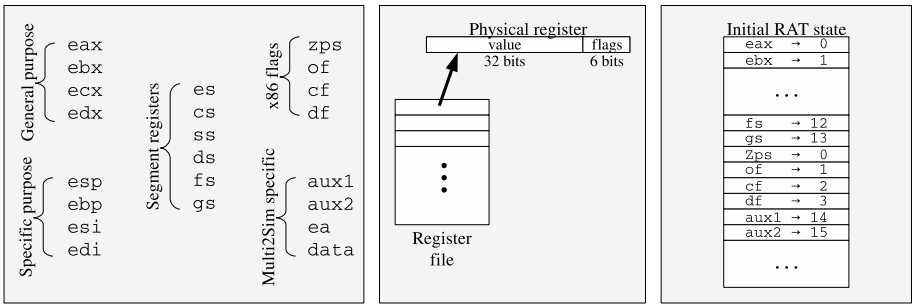


Figure 2.2: Register renaming.

- **Combined predictor**. The *combined* predictor (option `-bpred comb`) combines the bimodal and the two-level adaptive predictors. On an inquiry, both components are looked up, and their corresponding predictions are temporarily stored. Then, an additional table, called *choice predictor*, is accessed to decide whether to obey the bimodal predictor statement or the two-level predictor statement. Option `-bpred:choice` specifies the number of entries in the choice predictor (power of 2).

  Each entry contains a 2-bit saturating counter. If its value is 0 or 1, the statement of the bimodal predictor is considered. If its value is 2 or 3, the two-level predictor is used to give the final prediction. The choice predictor counters are updated at the commit stage only in the case that the bimodal and the two-level predictors gave a contradicting prediction.

2.2.2 Register Renaming

The register renaming mechanism implemented in Multi2Sim uses a simplification of the x86 logical registers. There are 22 possible name dependences between microinstructions, which are listed in Figure 2.2a. Logical registers `eax`...`edx` are general purpose registers used for computations and intermediate results. Registers `esp`...`edi` are specific purpose registers implicitly or explicitly modified by some microinstructions, such as the stack pointer or base pointer for array accesses. Registers `es`...`gs` are segment registers, while `aux1`...`data` are internally used by the macroinstruction decoder to communicate the generated microinstructions with one another.

The status of an x86-based processor includes a set of flags, which are written by some arithmetic instructions, and later consumed mainly by conditional branches to decide whether to jump or not. Flags `of`, `cf`, and `df` are the overflow, carry, and direction flags, respectively, and are tracked as separate dependences among instructions. On the other hand, flags `zf`, `pf`, and `sf` are the zero, parity, and sign flags,

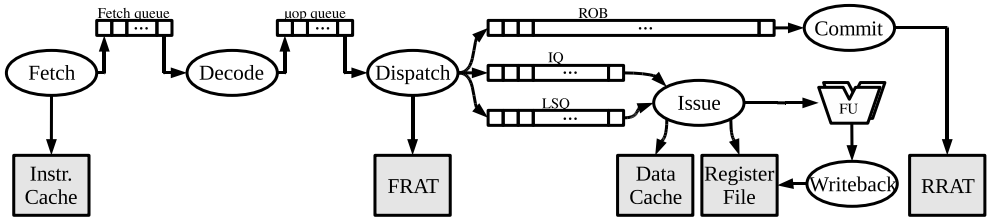


Figure 2.3: Multi2Sim model of the superscalar processor pipeline.

respectively, and any x86 instruction modifying any of these three flags is modifying all of them. Thus, they are tracked as a single dependence, called `zps`.

The value associated with each logical register, i.e., each potential input dependence for an instruction, is stored in the physical register file. As represented in Figure 2.2b, the register file (RF) consists of a set of physical registers that store operation results. Each physical register is formed of a 32-bit value, jointly with a 6-bit field storing the x86 flags. The number of physical registers can be established with the `-phregs_size` option.

At a given point, each logical register is mapped to a given physical register in the register file, containing the associated value. In the Multi2Sim renaming model, logical register and flags renaming works independently. This means, for example, that register `eax` and flag `cf` can be mapped to the same register file entry. In this case, the *value* field stores the contents of *eax*, while a specific bit in the *flags* field contains the value for *cf*. Each logical register is mapped to a different physical register, but x86 flags can be mapped all to the same physical register, even if the latter already has an associated logical register.

A Register Aliasing Table (RAT) holds the current mappings for each logical register. Its initial state is shown in Figure 2.2a. Additionally, a Free Register Queue (FRQ) contains the identifiers corresponding to free (not allocated) physical registers. When a new instruction writing into logical register l is renamed, a new physical register is taken from the FRQ and the new mapping for l is stored in the RAT. The previous mapping p' of logical register l will be needed later, and is stored in the ROB entry associated with the renamed instruction. When subsequent instructions consuming l are renamed, the RAT will make them consume the contests in p, where they will find the associated value.

When the instruction writing on l commits, it releases the previous mapping of l, i.e., physical register p', returning it to the FRQ if necessary. Notice that, unlike a classical renaming implementation ignoring flags, a physical register can have several entries in the RAT pointing to it (the maximum is the number of flags plus one logical register). Thus, a counter is associated with each physical register, which will only be freed and sent back to the FRQ in case this counter is 0.

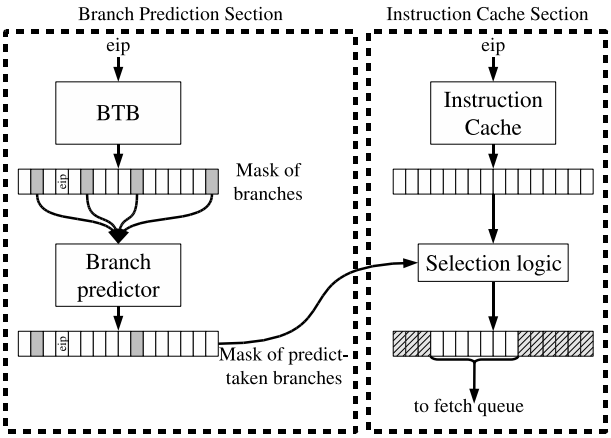


Figure 2.4: Block diagram of the fetch stage.

2.2.3 Pipeline Stages

Figure 2.3 shows a block diagram of the processor pipeline modeled in Multi2Sim. The gray-painted boxes represent hardware structures, whereas the round shapes represent pipeline stages. Six stages are modeled in Multi2Sim, named *fetch*, *decode*, *dispatch*, *issue*, *writeback*, and *commit*. The actions performed in each stage are described next.

- **The Fetch Stage**. The fetch stage is the first pipeline stage modeled in Multi2Sim. It is in charge of fetching instructions from the corresponding instruction cache at the addresses provided by the branch predictor. Then, these instructions are placed into the fetch queue. The instruction cache contains x86 macroinstructions, whose variable size is not known at this stage yet. Thus, the fetch queue is just a buffer of uninterpreted bytes, whose size can be specified with the `-fetchq_size` option. Figure 2.4 shows a block diagram of the fetch stage.

  The fetch stage is divided into two main sections, as shown in the figure, called *branch prediction* and *instruction cache fetching*, respectively. The branch prediction section provides information about the branches located within the fetched line. This information is then consumed by the instruction cache section. The modeled fetching mechanism works as follows:

  (i) First, the BTB is accessed with the current instruction pointer (*eip* register). As pointed out in Section 2.2.1, the BTB organization (capacity and associativity) can be configured by means of a specific command-line option. However, the BTB has a fixed number of banks (or interleaved

ways), namely as many as the size of the instruction cache block in bytes. In Figure 2.4, this value is set to 16. This means that 16 concurrent accesses to the BTB can be performed in parallel, as long as no pair of accesses matches the same interleaved way. This condition is true in the branch prediction section, as only contiguous addresses belonging to the same block are looked up.

(ii) The concurrent accesses to the BTB provide a mask of those instructions known to be branches, jointly with their corresponding target addresses. The branch predictor is next looked up to obtain the predicted direction for the branches, that is, whether they are taken or not. Since the BTB also provides the type of the branch (conditional branch, function call, etc.), the branch predictor will consider this information for its output. This means that an inconditional branch will always provide a predict-taken output, and function calls and returns will access a Return Address Stack (RAS) to obtain the actual target addresses. After the access to the branch predictor, the input mask is converted to an output mask that only tracks those taken branches.

(iii) In parallel with (i), the instruction cache is accessed in the instruction cache fetching section (right block in Figure 2.4). After a variable latency, dependent on whether there was a cache hit or miss, the cache block becomes available, and the mask provided by the branch prediction section is used to select the useful bytes. Specifically, a selection logic takes those bytes ranging from the address contained in register eip until the address of the first predict-taken branch, or until the end of the block if there is none.

The filtered bytes are then placed into the fetch queue, which communicates the fetch stage with the next pipeline stage. After fetching, the eip register is set either to the starting address of the next block if no predict-taken branch was found, or to the target address of the first taken branch, as provided by the BTB.

- **The Decode Stage**. In the decode stage, macroinstructions are taken from the fetch queue and decoded into the corresponding sequence of uops, which are then placed into the uop queue. In a single cycle, the decode stage can decode as many x86 instructions from the fetch queue as the decode bandwidth allows (option `-decode_width`).

- **The Dispatch Stage**. In the dispatch stage, a sequence of uops are taken from the uop queue. For each dispatched uop, register renaming is carried out by looking up the RAT for the current source and previous destination mappings, and allocating a new physical register for the current destination

operand. Then, the uop is inserted in the ROB, and either in the LSQ or the IQ, depending on whether the uop is or is not a memory instruction, respectively.

The number of instructions dispatched per cycle is specified with the `-dispatch_width` option. Instruction dispatching can get stalled for several reasons, such as the unavailability of physical registers, a lack of space in the ROB, or an empty uop queue. Since the dispatch stage acts as a bridge between the processor front- and back-end, a stall in this stage is a symptom of some processor bottleneck constraining performance. For each dispatch slot (i.e., for each uop susceptible of being dispatched in each cycle), the reasons for dispatch stalls are recorded, and shown after the simulation finishes in the `di.stall` statistics (`di.stall[rob]`, `di.stall[iq]`, etc.).

- **The Issue Stage**. The issue stage operates on the IQ and the LSQ. The uops placed in these queues are instructions waiting for their source operands to be ready, or for their associated processor resource to be available. The issue stage implements the so-called *wakeup logic*, which is in charge of selecting from each queue at the most `issue_width` uops that can be scheduled for execution. After selecting the proper candidates, instructions from the IQ are sent to the corresponding functional unit to be executed, whereas *load* instructions placed in the LSQ are sent to the data cache.

  Since *store* instructions irreversibly modify the machine state, they are handled in an exceptional manner both in the issue and the commit stage. On one hand, *stores* are allowed to access the data cache only after they are known to be non-speculative, which can be ensured after they have safely reached the ROB head. On the other hand, *stores* have no destination operand, so they need not perform any renaming action at the commit stage. Thus, they are allowed to leave the ROB as soon as they have been issued to the cache, without waiting for the cache access to complete.

- **The Writeback Stage**. The writeback stage is in charge of taking the results produced by the functional units or by a read access to the data cache, and store them into the physical register mapped to the logical destination of the executed instruction. If the executed instruction is a mispeculated branch, this is when mispeculation is detected, since both the branch condition and the target address are resolved at this time.

  Processor recovery on mispeculation can be performed either at the writeback or at the commit stage, as specified in parameter `-recover_kind`. If recovery is performed at the writeback stage, instructions following the mispeculated branch are drained from the ROB, IQ, and LSQ, the RAT is returned to a previous valid state, and instruction fetching is delayed as many cycles as specified by parameter `-recover_penalty`.

- **The Commit Stage**. The commit stage is the last stage in a superscalar processor pipeline, in which instructions commit their results into the architected machine state in program order. The oldest instruction in the pipeline is located at the head of the ROB. The condition for a *store* instruction to be extracted from the ROB is that it be issued to the cache, while the rest of instructions must be completed before committing.

  If the instruction at the head of the ROB is a mispeculated branch and the recovery process is specified to be triggered at the commit stage, the contents of the ROB, IQ, and LSQ are completely drained (only mispeculated instructions following the branch remain in the pipeline at this time), the RAT is recovered to a valid state, and `recover_penalty` cycles go by before instruction fetch resumes.

  When a completed, non-speculative uop commits, the register renaming mechanism frees the physical registers corresponding to the previous mappings of the uop's destination logical registers (see Section 2.2.2). Then, the branch prediction mechanism updates the appropriate tables, according to the observed behavior of the committed uop if it is a branch.

2.3 Support for Parallel Architectures

Multi2Sim provides a model for multicore and multithreaded processors. In order to evaluate these parallel architectures, they must be stressed with multiple tasks, which can be formed either by several applications with sequential code running in parallel, or by one parallel program spawning child tasks at runtime. To describe how a parallel architecture is modeled and evaluated in Multi2Sim, the following definitions are first given.

- A *context* is a software task (sometimes referred to as *software thread*) whose state is defined by a virtual memory image and a logical register file. Logical register values are exclusive for a context, whereas the memory map can be either exclusive or shared with other contexts. If an application contains sequential code, its state is represented by one single context. On the contrary, an application can also spawn child contexts at runtime when it contains parallel code, for example by using the OpenMP or POSIX threads libraries.

- A *(hardware) thread* is a hardware entity capable of storing the status of a single context and executing it. In order to store the logical register values, a thread has its own register aliasing table (RAT), which maps the logical registers into a physical register file. To store the state of a private memory image, a thread has its own memory map cached in a private translation look-aside

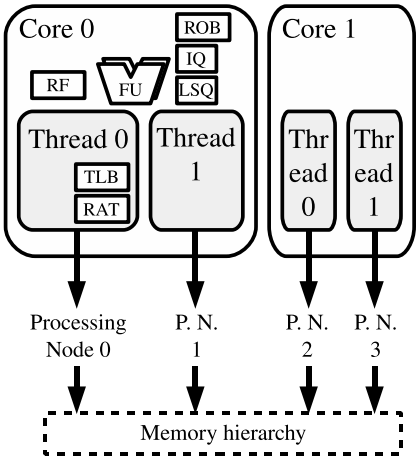


Figure 2.5: Parallel architecture scheme.

buffer (TLB), which maps virtual memory locations into physical memory pages. Functional units (adders, multipliers, FP execution unit...) are shared among threads, while other pipeline structures, stages, and queues (such as ROB, IQ, LSQ) can be private or shared.

- A *(processor) core* is formed of one or more threads. It does not share any pipeline structure, execution resource, or queue with other cores, and the only communication and contention point among cores is the memory hierarchy.

- A *processing node* is the minimum hardware entity required to store and run one context. In a multithreaded processor, each thread is one processing node. Likewise, each core in a multicore (and not multithreaded) processor is considered one processing node. Finally, an c-core, t-threaded processor (meaning that each core has t threads) has $c \times t$ processing nodes, since it can store and run $c \times t$ contexts simultaneously. Each processing node can have its own entry point to the memory hierarchy to fetch instructions or read/write data. The number of processing nodes limits the maximum number of contexts that can be concurrently executed in Multi2Sim. If an application spawns new contexts and this limit is exceeded at runtime, the simulation stops with an error message.

Based on these definitions, Figure 2.5 represents the structure of the parallel architecture modeled in Multi2Sim. The figure is an example of a processor with 2 cores and 2 threads per core, forming 4 processing nodes with independent entry points to the memory hierarchy.

2.3.1 Multithreading

A multithreaded processor is modeled in Multi2Sim using option `-threads`, and assigning a value greater than 1. In a multithreaded design, most execution resources can be either private or shared among threads. These resources can be classified as *storage resources* and *bandwidth resources*. The former refer to pipeline structures (such as the ROB or IQ), while the latter refer to the uops that a pipeline stage can handle in a single cycle (such as dispatch slots, issue slots, etc.). Multi2Sim options to configure each of these resources are given next.

- **Storage resources**. Regarding the configuration of storage resources, options `-rob_kind`, `-iq_kind`, `-lsq_kind`, and `-phregs_kind` specify the sharing strategy of the ROB, IQ, LSQ, and register file, respectively. The possible values for these options are `private` and `shared`. The parameter specifying the size of each structure always refers to the number of entries per thread. For example, when the ROB is shared in an n-threaded processor, the total number of ROB entries that can be occupied by a single thread is $n \times$ `rob_size`.

  The fact of sharing a storage resource among threads has several implications in performance and hardware cost. On one hand, private storage resources constrain the number of structure entries devoted to each thread, but they implement in a natural manner a guarantee for a fair distribution of available entries among threads. On the other hand, a shared resource allows an active thread to occupy resource entries not used by other threads, but a greedy thread stalled in a long-latency operation may penalize other active threads by hundreds of cycles if it is holding resource entries for too long.

- **Bandwidth resources** The options to specify how pipeline stages divide their slots among threads are `-fetch_kind`, `-dispatch_kind`, `-issue_kind`, and `-commit_kind`. The values that these options can take are `timeslice` and `shared`. The former means that a stage is devoted to a single thread in each cycle, alternating them in a round-robin fashion, while the latter means that multiple threads can be handled in a single cycle. The stage bandwidth always refers to the total number of slots devoted to threads. For example, a value of 4 for `issue_width` means that at the most 4 uops will be issued per cycle, regardless of whether the issue stage is shared or not.

  The fetch stage can be additionally configured with *long term* thread switches, by assigning the value `switchonevent` for the `-fetch_kind` option. In this case, instructions are fetched from one single thread either until a quantum expires or until the current thread issues a long-latency operation, such as a *load* instruction incurring a cache miss.

Option	Coarse-Grain MT	Fine-Grain MT	Simultaneous MT
-fetch_kind	switchonevent	timeslice	timeslice/shared
-dispatch_kind	timeslice	timeslice	timeslice/shared
-issue_kind	timeslice	timeslice	shared
-commit_kind	timeslice	timeslice	timeslice/shared

Table 2.1: Classification of multithreading paradigms depending on Multi2Sim options.

Depending on the combination of sharing strategies for pipeline stages, a multithreaded design can be classified as coarse-grain (CGMT), fine-grain (FGMT), and simultaneous multithreading (SMT). The combination of parameters for each stage and its classification are listed in Table 2.1. The main enhancement of FGMT with respect to CGMT is a round-robin fetch stage, which feeds the rest of the pipeline with uops from a different thread every cycle, thus increasing thread-level parallelism. The key improvement of SMT with respect to FGMT is the shared issue stage, which feeds functional units at higher rate with ready instructions, regardless of the thread they belong to.

2.3.2 Multicore Architectures

In Multi2Sim, a multicore architecture is modeled by assigning a value greater than 1 to option `-cores`. Since processor cores do not share any pipeline structure, there is no other option related with the multicore processor configuration. When the number of cores is greater than 1, all processor pipelines and their associated structures are simply replicated, and they work simultaneously in every execution cycle. As mentioned above, the only common entity for cores is the memory hierarchy.

2.4 The Memory Hierarchy

Multi2Sim provides a very flexible configuration of the memory hierarchy. Any number of cache levels can be used, and caches can be unified or separate for data and instructions, and they can be private or shared per groups of cores/threads. In this chapter, it is shown how the memory hierarchy is modeled, configured and implemented in Multi2Sim, including caches, main memory, and interconnection networks.

2.4.1 Memory Hierarchy Configuration

The configuration of the memory hierarchy is specified in an independent text file. If the file name is `<name>`, Multi2Sim is notified to use it as memory hierarchy config-

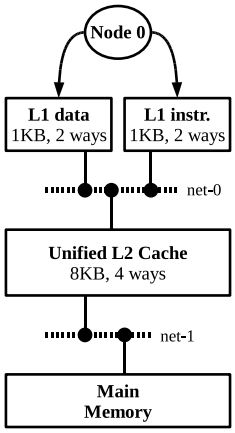


Figure 2.6: Example of a memory hierarchy configuration.

uration file with option `-cacheconfig <name>`. The configuration file is formed of sections and fields. Each section represents a component of the memory hierarchy, and is specified with a name enclosed in brackets (for example `[MainMemory]`). After the section header, a set of fields formed of pairs `<key>=<value>` follows, specifying the properties of the specific component.

The format of the memory hierarchy configuration file is illustrated by means of an example modeling the memory hierarchy represented in Figure 2.6. This example corresponds to a processor with one single core and one single thread, using two levels of cache, where L1 caches are separated for instructions and data, and an L2 cache is unified. The contents of the associated configuration file is listed in Table 2.2.

In this code, a set of sections named `[CacheTopology <name>]` defines cache organizations, that is, a set of characteristics used to later create caches. Two cache organizations are defined first, named `l1topo` and `l2topo`, which will be used to create the L1 and L2 caches, respectively. The former is defined with 8 sets, 2 ways, and 64-byte blocks (1KB of storage capacity in total), while the latter defines 32 sets, 4 ways, and 64-byte blocks (8KB in total). The total capacity of the cache in bytes is the product of the number of sets, the associativity, and the block size.

The next two sections `[Net <name>]` define the interconnection networks. In this case, two of them are created, called `net-0` and `net-1`, respectively. Both interconnects are defined with a bus topology and a link width of 32 bytes per cycle. Thus, a 64-byte cache block is transfered in two cycles.

The sections entitled `[Cache <name>]` create cache memories. The L1 cache is named `dl1`, has the topology `l1topo`, and is connected to the next cache level by means of the `net-0` network. The cache `il1` is the instruction cache, also with topology `l1topo`, and connected to network `net-0` below. Finally, the unified L2

```
[ CacheTopology l1topo ]
Sets = 8
Assoc = 2
BlockSize = 64
Latency = 2

[ CacheTopology l2topo ]
Sets = 32
Assoc = 4
BlockSize = 64
Latency = 20

[ Net net-0 ]
Topology = Bus
LinkWidth = 32

[ Net net-1 ]
Topology = Bus
LinkWidth = 32

[ Cache dl1 ]
Topology = l1topo
LoNet = net-0

[ Cache il1 ]
Topology = l1topo
LoNet = net-0

[ Cache l2 ]
Topology = l2topo
HiNet = net-0
LoNet = net-1

[ MainMemory ]
HiNet = net-1
BlockSize = 64
Latency = 200

[ Node 0 ]
Core = 0
Thread = 0
DCache = dl1
ICache = il1
```

Table 2.2: Example of a memory hierarchy configuration file.

Section	**Variable**	**Meaning**
`Topology <name>`	`Sets`	Number of sets.
	`Assoc`	Associativity (number of ways).
	`BlockSize`	Block size in bytes.
	`Latency`	Access time in cycles.
	`Policy`	Block replacement policy (`LRU`, `FIFO`, or `Random`).
`Net <name>`	`LinkWidth`	Link bandwidth in bytes per cycle.
	`Topology`	Network topology (`Bus` or `P2P`).
`Cache <name>`	`Topology`	Cache topology name, as defined in section `[Topology <name>]`.
	`HiNet`	Name of the network connected above, as defined in section `[Net <name>]`.
	`LoNet`	Name of the network connected below.
`MainMemory`	`HiNet`	Name of the network connected above.
	`Latency`	Access latency.
	`BlockSize`	Memory block size in bytes.
`Node <num>`	`Core`	Core identifier of the processing node.
	`Thread`	Thread identifier.
	`DCache`	Name of the cache for data access, as defined in section `[Cache <name>]`.
	`ICache`	Name of the cache for instruction fetch.

Table 2.3: Sections and variables in the memory hierarchy configuration file.

cache is named `l2`, has the topology `l2topo`, is connected to `net-0` above, and to `net-1` below.

Section `[MainMemory]` defines the main memory parameters. In this case, the main memory is connected to network `net-1` above, has a 64-byte block size, and a 200-cycle access time.

Finally, section `[Node 0]` specifies which is the entry point to the memory hierarchy for the computational node 0, corresponding to thread 0 in core 0. With the cache names assigned to variables `DCache` and `ICache`, it is stated that the node should access cache `dl1` when requesting data, and it should fetch code from cache `il1`. Table 2.3 lists all possible variables that can be used in each section of the memory hierarchy configuration file, including a brief description of their meaning.

2.4.2 Cache Coherence

Multi2Sim models a cache hierarchy with any number of cache levels, among which cache coherence is maintained. Coherence is enforced with the directory-based MOESI protocol at all caches connected to the upper link of an interconnect with respect to the single allowed cache connected to the lower link (see Figure 2.6). The upper level caches can belong to different threads, different cores, or a group of them, and they can contain instructions, data, or both. Even if one instruction cache and one data cache are connected to a unified cache at the lower level, coherence is maintained among them (which is required in the case of, for example, self-modifying code).

Figure 2.7 represents an example of four coherent L1 caches with respect to an inclusive L2 cache. Each block in the L2 cache, and in general each block in any lower level cache, contains the fields listed in the figure. Besides the data, tag, and status, a block has a directory entry containing two fields: the first one is an identifier of the single possible upper level cache being the owner of the block (i.e., an L1 cache with the block in state *exclusive*, *modified*, or *owned*). There is a specific value for this identifier to express that there is no owner in the upper level. The second field is a bitmap with as many bits as upper level caches, with those bits set to one corresponding to the caches having a copy of the block.

In fact, a single block can contain several directory entries in case the block size of an upper level cache is smaller. For example, two L1 caches with 16 and 32-byte block size connected to an L2 cache with 64-byte blocks, require 4 directory entries per L2 block. The reason is that there can be at the most 4 block portions or *subblocks* in an L2 cache block split among different cache lines in the L1 cache with the smallest block size.

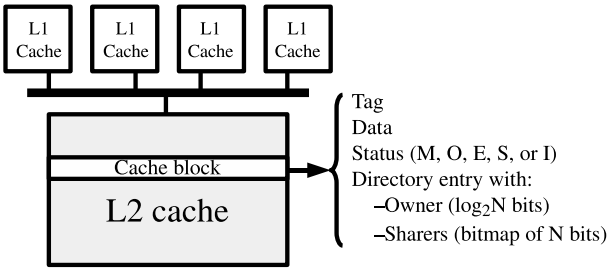


Figure 2.7: Enforcement of cache coherence.

2.5 Experimental Environment

The experiments in this thesis have been carried out on top of the Multi2Sim simulation framework. First, this section describes how the simulator is extended to model the proposed techniques. Then, the rest of the simulation environment and methodology is presented, including a brief description of the executed benchmark suites, and the performance metrics extracted from the simulation reports.

2.5.1 Multi2Sim Extensions

Version 2.1 of the Multi2Sim framework has been used to carry out the experiments in this thesis. The baseline simulator has been instrumented to implement the proposed techniques on top of the superscalar, multithreaded, and multicore processor models. The following extensions have been applied in each case:

- **Superscalar model.** The implementation of the out-of-order retirement architecture proposed in this thesis, referred to as Validation Buffer (VB) architecture, mainly involves an alternative management of the reorder buffer, and an aggressive register renaming mechanism. First, the commit stage has been modified to enforce a different set of retirement conditions for instructions. Then, the physical register file and the renaming tables have been redesigned to implement the new renaming scheme. Finally, the simulator has been instrumented with an additional option to switch between the ROB and VB architectures.

- **Multithreaded model.** The baseline simulator provides support for different multithreading paradigms and sharing strategies for the major processor structures. For the evaluation of out-of-order retirement in a multithreaded processor with shared structures, these resources have been instrumented with specific allocation policies that manage the assignment of their entries to active threads.

Benchmark	Type		Arguments
164.gzip	Int	CPU	input.random 60
168.wupwise	FP	CPU	–
171.swim	FP	MEM	< swim.in
172.mgrid	FP	CPU	< mgrid.in
173.applu	FP	MEM	< applu.in
175.vpr	Int	MEM	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2
176.gcc	Int	CPU	166.i
177.mesa	FP	CPU	-frames 1000 -meshfile mesa.in -ppmfile mesa.ppm
178.galgel	FP	MEM	< galgel.in
179.art	FP	MEM	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10
181.mcf	Int	MEM	< inp.in
183.equake	FP	MEM	< inp.in

Benchmark	Type		Arguments
186.crafty	Int	CPU	< crafty.in
187.facerec	FP	MEM	< ref.in
188.ammp	FP	MEM	< ammp.in
189.lucas	FP	MEM	< lucas2.in
191.fma3d	FP	CPU	–
197.parser	Int	CPU	2.1.dict -batch < ref.in
200.sixtrack	FP	CPU	< inp.in
252.eon	Int	CPU	chair.control.cook chair.camera chair.surfaces chair.cook.ppm ppm pixels_out.cook
253.perlbmk	Int	CPU	-I./lib diffmail.pl 2 550 15 24 23 100
254.gap	Int	CPU	-l ./ -q -m 192M < ref.in
255.vortex	Int	CPU	bendian1.raw
256.bzip2	Int	CPU	input.source 58
300.twolf	Int	MEM	ref
301.apsi	FP	CPU	–

Table 2.4: Classification and command-line arguments for the SPEC2000 benchmarks.

- **Multicore model.** For a configuration using multiple cores, the simulator has been instrumented to support the release and sequential memory consistency models. The sequential consistency model is based on speculative retirement of loads [29]. This implementation uses a history buffer, where instructions are inserted after leaving the reorder buffer. Additionally, L1 caches are instrumented to monitor memory blocks that are accessed by instructions located in the history buffer.

2.5.2 Benchmarks and Methodology

The benchmark suites executed on top of Multi2Sim in the experimental evaluations are SPEC2000 [30] and SPLASH2 [31]. A brief description of these programs is given next, including their combinations for multithreaded designs, and citing the command-line arguments for each benchmark.

- **SPEC2000**. This suite is formed of 26 single-threaded benchmarks, classified as integer (Int) or floating-point (FP). Integer benchmarks are written in C or C++, and include compression, compilation, artificial intelligence algorithms, or FPGA routing and placing, among others. Floating-point benchmarks are written in C, Fortran77, or Fortran90, and deal with physics simulation, image processing, numeric algorithms, etc. For each benchmark, three input data sets with different sizes are provided, named *test*, *train*, and *ref*.

  The SPEC2000 suite is used in this thesis to evaluate the baseline and proposed superscalar and multithreaded architectures. In the case of a superscalar model,

Benchmark	Arguments
lu	-p$NTHREADS -n2048 -b16
fft	-m18 -p$NTHREADS -n65536 -l4
radix	-p$NTHREADS -r4096 -n262144 -m524288
cholesky	-p$NTHREADS < tk14.O
barnes	$NTHREADS < input
fmm	$NTHREADS < input
ocean	-n258 -p$NTHREADS -e1e-07 -r20000 -t28800
radiosity	-batch -room -p$NTHREADS
raytrace	-p$NTHREADS balls4.env
waternsq	$NTHREADS < input
watersp	$NTHREADS < input

Table 2.5: Command-line arguments for the SPLASH2 benchmarks.

one single benchmark is run at a time to obtain one performance result, whereas multithreaded models use a mix of at least two benchmarks, each allocated to a hardware thread. To form these mixes, benchmarks are classified as CPU- or memory-intensive applications, depending on the amount of instruction level parallelism (ILP) they exhibit. Then, they are combined to build mixes with only CPU-intensive programs, only memory-intensive ones, or both kinds together. The specific configuration of SPEC2000 benchmark mixes is detailed in Section 4.2, where experiments with multithreaded architectures are shown.

As done in some previous works using this benchmark suite [6][9], most experiments are run by initially performing a functional simulation of the initial code, using here 500M x86 macroinstructions, and then running a detailed simulation until the next 500M uops commit. The objective is to skip programs initialization with fast simulation speed, which does not contribute to the computation of the final statistics. The files for the input data are taken from the *ref* set. For each benchmark, Table 2.4 lists the numerical computation kind (Int of FP), the classification as per ILP (CPU or MEM), and the command-line arguments used for its execution.

- **SPLASH2**. This suite consists of 11 parallel benchmarks, classified as kernels or applications. All of them provide command-line arguments or configuration files to specify the input data size. SPLASH2 benchmarks perform computations, synchronizations, and communication, stressing processor cores, memory hierarchy, and interconnection networks. Thus, they are used for the evaluation of the baseline and proposed multicore architectures presented in this thesis. Additionally, SPLASH2 benchmarks provide arguments to specify the number of contexts created at runtime, which allows the evaluation of systems

with different number of cores. Table 2.5 shows the arguments used for each benchmark.

The initialization code is not skipped in the SPLASH2 suite. Parallel programs do not necessarily execute the same instructions for the same input parameters and different hardware designs, because synchronization points may be reached at different times, causing threads to stop at different waiting loops. Thus, a fixed amount of assembler instruction has no direct correspondence with the same amount of job. Since SPLASH2 benchmarks can flexibly tune the problem size to provide reasonable simulation times, programs are run until completion, including initialization and finalization code.

2.5.3 Performance Metrics

After carrying out a simulation, Multi2Sim dumps a report with simulation statistics. In the experiments presented in this thesis, results of several simulations varying one or more input parameters are usually filtered and represented graphically so as to show a specific performance trend. By default, Multi2Sim provides a set of standard performance metrics, such as the number of committed instructions, the IPC (committed instructions per cycle), the branch prediction accuracy, cache hit ratio, etc. Although the default statistics suffice for most exhaustive evaluations, the simulator has been instrumented to provide the following additional performance metrics for a deeper comprehension of the modeled out-of-order retirement architectures.

- **Used dispatch slots**. The number of used dispatch slots in a cycle, which ranges from 0 up to the dispatch bandwidth, is the number of uops taken from the uop queue and placed into the ROB, among other structures. If the number of dispatched uops does not reach the dispatch bandwidth, it means either that the uop queue is empty, or that no additional uop could be placed into some other full structure. The specific reasons for dispatch stalls are recorded every cycle, and the final average values are considered in our experiments as a useful metric to analyze performance bottlenecks.

- **Used issue slots**. The number of issued uops, which ranges from 0 up to the issue bandwidth, is stored every cycle to form a probability distribution. This statistic is especially useful for the evaluation of multithreaded designs, where an improvement of the multithreading paradigm is usually reflected in a higher issue rate.

- **Resource occupancy**. This metric gives the fraction of busy entries for a given resource, either as an average value, or as a probability distribution. The simulator is instrumented to measure the occupancy of the reorder buffer (ROB), instruction queue (IQ), load-store queue (LSQ), and register file (RF). With the

obtained statistics, it may be possible to identify which processor structure is causing performance bottlenecks in a given scenario. For example, a simulation showing a fraction of 0.96 occupied entries in the ROB is suggesting that a slight increase of this structure will provide important performance gains.

2.6 Summary

This chapter has presented the simulation environment used in the remainder of this thesis for evaluation purposes. First, its main features have been cited and compared with other existing simulation tools. The modeled architecture for superscalar, multithreaded, and multicore processors has been detailed, including the branch prediction mechanism, register renaming scheme, and memory hierarchy configuration. Then, the SPEC2000 and SPLASH2 benchmark suites have been briefly presented, including the command-line arguments for their execution, and the simulation methodology. Finally, the main performance metrics have been cited and described.

Chapter 3

The Superscalar Validation Buffer Architecture

The commit stage is typically the latest in the processor pipeline. At this stage, a completed, non-speculative instruction updates the architectural machine state, frees the used resources, and exits the ROB. On the contrary, the Validation Buffer architecture implements a FIFO structure similar to the ROB, but instructions can leave it as soon as they are known to be non-speculative, and even though they are not completed yet. Once they complete, they update the machine state and free the used resources, hence exiting the processor pipeline in an out-of-order fashion. This chapter presents the Validation Buffer architecture for single-threaded superscalar processors.

3.1 Proposed Architecture

The necessary conditions to allow an instruction to be committed out-of-order are [9]: 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 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 3.1.1).

The VB deals with the speculation-related conditions (iii, iv and v) by decomposing code into fragments or *epochs*. The epoch boundaries are defined by instructions that may initiate speculative execution, referred to as *epoch initiators* (e.g., branches or potential 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, and thus, they enter this structure in program order. Epoch initiator instructions are marked as such in the VB. When an epoch initiator detects a mispeculation, all the following instructions must be canceled. Therefore, an epoch initiator reaching the VB head must wait for its execution to be completed before leaving the VB. After completion, it is allowed to update the machine state and release its entry at the head of the VB. Instructions other than epoch initiators reaching the VB head can leave it regardless of their execution state. That is, they can be either dispatched, issued, or completed. However, only those validated (i.e., not canceled) instructions update the machine state. On the other hand, canceled instructions are drained and free the resources they occupy (see Section 3.1.2).

Like in a ROB-based approach, a completed instruction that leaves the VB is not consuming any other execution resource in the pipeline. In contrast, an incomplete but validated instruction leaves the VB, but remains in the processor pipeline until it completes. Since execution time is variable among instructions, the VB architecture is said to retire instructions from the processor pipeline out of program order. In other words, the proposal in this thesis can be classified as an out-of-order retirement architecture.

The VB architecture allows a flexible set of epoch initiators to be established. At least, those epoch initiators are supported corresponding to the three speculation-related aforementioned conditions (iii, iv, and v). Therefore, branches and memory reference instructions (specifically its address computation) necessarily act as epoch initiators. This means that branch speculation, memory replay traps (see Section

a) C source code

```
for (i = 0; i < 100000; i++)
{
      result[i] = v1[i] * v2[i];
}
```

b) Target assembler code

```
$L11: l.d    $f0, 0($3)
      l.d    $f2, 0($4)
      mul.d  $f0, $f0, $f2
      addu   $6, $6, 1
      slt    $2, $7, $6
      addu   $4, $4, 8
      addu   $3, $3, 8
      s.d    $f0, 0($5)
      addu   $5, $5, 8
      beq    $2, $0, $L11
```

Figure 3.1: Vector product example.

3.1.3) and exceptions related with address calculation (e.g., page faults or invalid addresses) are supported by design.

It is possible to include more instructions in the set of epoch initiators. For instance, floating-point instructions should belong to this set in order to support precise floating-point arithmetic exceptions. However, as instructions are allowed to leave the VB only after the preceding epoch initiators validate their epoch, a large amount of epoch initiators might constrain the VB-based processor performance. To solve this, a user modifiable flag can be introduced in the ISA to dynamically switch the support for precise exceptions. This flag could be handled by the compiler, by being disabled in those program sections where exceptions are known not to be raised, or in which their precise handling is not required.

Hardware interrupts can be handled like in a ROB-based processor, without further modifications. In particular, the occurrence of such an event should cancel all instructions located in the VB. The recovery mechanism (see Section 3.1.2) will then automatically recover valid register mappings, squash canceled instructions from the pipeline, and resume execution at the interrupt service routine.

Figure 3.1 presents a typical scenario where the benefits of out-of-order retirement are clearly illustrated. A portion of source code (Figure 3.1a) and its corresponding sequence of MIPS assembler instructions (Figure 3.1b) are listed as a possible implementation of the product of vectors `v_1` and `v_2`. When this code is run in a ROB-based microprocessor, the first load missing in the data cache clogs the ROB head, and later stalls instruction dispatching when the ROB eventually fills up. When this occurs, the ROB contains plenty of memory instructions, namely two loads and one store per iteration, which are a potential source of additional cache misses aggra-

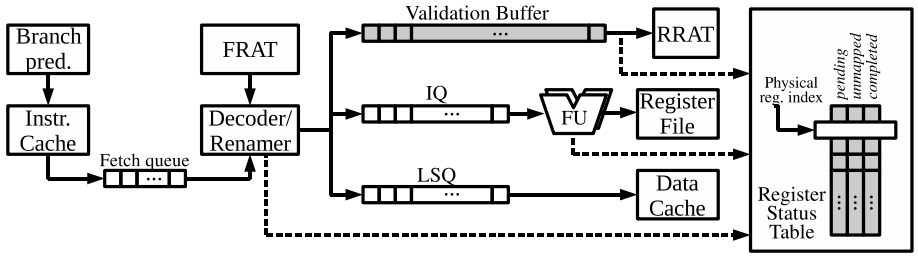


Figure 3.2: VB architecture block diagram.

vating the problem. A modern microprocessor running this code (see Section 3.3) is stalled during about 77% of the execution cycles. In contrast, this percentage is nearly negligible (about 0.02%) when the program runs in a VB-based processor, where the pressure is moved into the instruction queue and load-store queue. As more relaxed conditions are required to release these resources, IPC rises from 0.70 to 1.05.

3.1.1 Register Reclamation

Typically, modern microprocessors free a physical register when the instruction that renames the corresponding logical register commits [1]. 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 might be mapped for longer than their useful lifetime. In addition, this method requires to keep track of the oldest instruction in the pipeline. As this instruction may have already left the VB, this method is unsuitable for our proposal.

For these reasons, an alternative register reclamation strategy has been designed, based on the counter method [32][1], and targeted to the VB microarchitecture. The hardware components used in this scheme, as shown in Figure 3.2, are:

- **Front Register Alias Table (FRAT)**. This table maintains the current mapping for each logical register and is accessed at the renaming stage. It is indexed by a logical register to obtain the corresponding mapped physical register identifier. Whenever a new physical register is mapped to a destination logical register, the FRAT is updated.

- **Retirement Register Alias Table (RRAT)**. The RRAT is updated whenever an instruction exits the VB. This table contains a precise state of the register mappings, 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 identifier. It contains three fields: *pending*, *unmapped* and *completed*. The *pending* field tracks the number of decoded instructions that consume the corresponding physical register, but have not read it yet. This value is incremented when consumers enter the decode stage, and decremented when they are issued to the execution units. The second and third fields are composed each of a single bit. The *unmapped* 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, dumping its result into 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* guarantees that no instruction within the pipeline is going to read the contents of p. Next, a 1 in *unmapped* implies that no new instruction will enter the pipeline (i.e., the *rename* stage) and read the contents of p, because p has been unmapped by a valid instruction. Finally, a 1 in the *completed* field denotes that no instruction within the pipeline is going to write on p again. These conditions ensure that a specific physical register can be safely reallocated on 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 3.1 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 the RST, FRAT and RRAT.

Event	**Actions**
Instruction *I*, with physical register (p.r.) *p* as source operand, enters the rename stage.	RST[*p*].pending + +
I enters the rename stage and reclaims a p.r. to map an output logical register *l*.	Find a free p.r., say *p*. FRAT[*l*] = *p*. RST[*p*].completed = 0 RST[*p*].unmapped = 0
I is issued and reads p.r. *p*.	RST[*p*].pending – –
I finishes execution, writing the result over p.r. *p*.	RST[*p*].completed = 1
I exists the VB as validated, *l* is the logical destination register, p.r. *p* is the current mapping of *l*, and p.r. *p'* is the previous mapping of *l*.	RST[*p'*].unmapped =1 RRAT[*l*] = *p*

Table 3.1: Renaming actions for different pipeline events.

3.1.2 Recovery Mechanism

The recovery mechanism is triggered after a mispeculated branch finishes its execution, resolving its target address and branch direction. Among other actions, processor recovery involves restoring a previous valid state of both the FRAT and the RST structures.

Regarding the recovery of the mappings in the FRAT, different methods are employed in current microprocessors. The method presented in this work uses both FRAT and RRAT, similarly to the Pentium 4 processor [3]. The RRAT contains a delayed copy of a *validated* FRAT. That is, it matches the FRAT at the time the exiting (as valid) instruction was renamed. A simple method to implement the recovery mechanism (restoring the mapping to a precise state) is waiting until the offending instruction reaches the VB head, and then copying the RRAT contents into the FRAT. Alternative implementations can be found in [7].

On the other hand, the recovery mechanism must restore the RST by undoing the modifications performed by mispeculated instructions in any of its three fields. Concerning the *unmapped* field, two possible techniques are proposed to restore its values. The first technique squashes from the VB those entries corresponding to instructions younger than the offending one after the latter reaches the VB head. At that point, the RRAT contains the physical register identifiers used to restore the correct mapping. The remaining physical registers must be freed. To this end, all *unmapped* entries are initially set to 1, and the RRAT is scanned for physical registers whose *unmapped* entry must be cleared in the RST.

The second technique relies on the following observation: only those physical registers allocated by instructions younger than the offending one must be freed. Canceled instructions can leave the VB normally without being squashed, but they are processed differently by the validation logic. Specifically, these instructions must set to 1 the *unmapped* entry of their current mapping. Unlike validated instructions, mispeculated ones update the *unmapped* entry corresponding to the currently mapped destination physical register, instead of accessing the previous mapping, as done in non-speculative mode. While canceled instructions are being drained, new instructions can enter the pipeline, provided that the FRAT has been already recovered. Therefore, the VB draining can be overlapped with subsequent new processor operations.

The *pending* field cannot be just reset, because there might already be valid pending readers in the issue queue. Thus, each *pending* entry must be decremented as many times as the number of canceled pending readers for the corresponding physical register. To this end, the issue logic must allow the detection of those instructions younger than the offending instruction, that is, the canceled pending readers. This can be implemented by using a bit mask in the issue queue to identify which instructions are younger than a given branch [2]. The canceled instructions must be drained

from the issue queue to correctly handle (i.e. decrement) their *pending* entries. This logic can be also reused 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 follow their normal execution path to correctly handle the *pending* and *completed* fields, which avoids the previously described recovery logic.

3.1.3 Uniprocessor Memory Model

To correctly follow the uniprocessor memory model, it must be ensured that a load instruction read the data produced by the youngest previous store matching its memory address. A key component to improve performance in this 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 decoder into two uops: the memory address calculation, which is considered an epoch initiator, and the memory operation itself. When dispatched, the former reserves one IQ entry, the latter reserves an entry in the LSQ, and both occupy one entry each in the VB. A load can be issued to the data cache as soon as its effective address is ready. On the other hand, a store is issued to the data cache and retired from the VB when both the effective address and source operand are ready, and it is the last write access to memory in the pipeline. The latter condition is enforced by issuing stores only when they are placed at the head of the VB. Finally, any memory instruction frees its corresponding LSQ entry at the time it is issued to the data cache.

Load bypassing is an aggressive, widely used speculation technique applied on the LSQ to improve processor performance. This technique permits early execution of loads by making them advance previous stores, even though the address of any of the latter is not known yet. As speculation may fail, a mechanism must be provided to detect and recover from a load mispeculation. As proposed in [33], speculatively issued loads can be placed in a special buffer called *finished load buffer* (FLB). The entry of this buffer is released when the load commits. When a store commits, the FLB is looked up for aliasing loads (note that all loads in the buffer are younger than the store). If an address match is detected, the load in the FLB and all subsequent instructions must be canceled and re-executed.

An FLB is straightforwardly adapted into the VB architecture with no additional complexity. In this case, a load instruction releases its entry in the FLB when it leaves the VB. When a store is issued and retired from the VB, the addresses of all previous memory instructions and its own address have been resolved. Thus, it can already search the FLB for aliasing loads that were speculatively issued. As in a ROB-based implementation, all loads in the buffer are younger than the store. On a

hit, the recovery mechanism is triggered as soon as the mispeculated load exits the VB. Notice that this implementation allows mispeculation to be early detected.

Load forwarding is another optimization for the LSQ. This technique is applied whenever a load address matches a previous store address, and there is no unresolved nor matching store address between them. In this case, the load data is obtained from the previous store, and the cache access is avoided. This technique is also compatible with the VB architecture, and the hardware devoted to it is no different from a ROB-based processor.

3.1.4 Potential Benefits in Performance

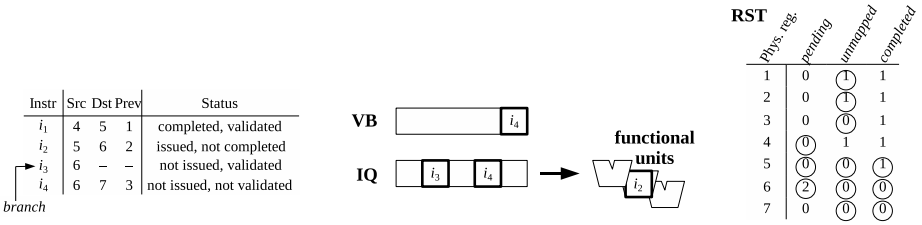
The VB architecture improves the management of instructions in the ROB architecture by means of an early retirement of instructions from the VB and an early release of unused physical registers. These features provide two main sources of potential benefits in performance, which can be described as follows:

- On one hand, an instruction placed at the VB head is only retained if it is an unresolved branch, an uncompleted memory address computation, or an instruction susceptible of raising an exception. The rest of instructions can leave the VB regardless of their execution state. Thus, the number of entries in the VB does not constrain the number of in-flight instructions in the processor pipeline. The potential benefit provided by the relaxation of retirement conditions is referred to hereafter as *extended instruction window*.

- On the other hand, an instruction leaving the VB tries to release the previous mapping of its destination logical register. If the associated physical register has moreover already been written, and there is no pending consumer in the pipeline, the register is indeed freed, even if older incomplete instructions are still in flight. The potential benefit that the register renaming technique provides will be referred to as *enhanced register usage*.

3.2 Working Example

Figure 3.3 illustrates a VB-based processor pipeline with four in-flight instructions, each in a different state. The table on the left (Figure 3.3a) lists these instructions, labeled from i_1 to i_4, with their source (*src*), destination (*dst*), and previous destination physical register (*prev*) —*prev* refers to the physical register mapped to the logical destination of an instruction before it was renamed. Though most instructions consume two source operands, they are assumed to have only one for the sake of simplicity. Instructions i_1, i_2 and i_4 are arithmetic operations that generate a result, while instruction i_3 is a branch. This means that instruction i_3 must be completed before the speculative state of the following instructions (i_4) is resolved.



a) Portion of code b) Pipeline state c) RST state

Figure 3.3: Working example for superscalar processors.

Figure 3.3b shows the VB, the IQ, and the functional units, while Figure 3.3c lists the contents of the RST in its current state. The circled identifiers correspond to those values affected by at least one of the four instructions. The following observations can be made:

- Instruction i_1 has completed execution and left the pipeline as validated. This makes both RST[5].*completed* and RST[1].*unmapped* be set, causing physical register 1 to become free.

- Instruction i_2 has exited the VB as validated, but it remains in execution in the corresponding functional unit. Thus, RST[6].*completed* is still clear, but RST[2].*unmapped* is already set, causing physical register 2 (the previous destination mapping) to be freed, even before the instruction completes. Likewise, RST[5].*pending* is set to 0, since no instruction in the IQ is going to read its contents.

- Instruction i_3 has also exited the VB as validated. However, it remains in the IQ because its source operand, which is produced by i_2, is not ready yet. Since both i_3 and i_4 will read register 6 in the future and they are located in the IQ, RST[6].*pending* is equals to 2.

- Finally, instruction i_4 has an unknown speculative state, since i_3 is an incomplete branch. Thus, it cannot leave the VB, and the RST[3].*unmapped* field remains clear. Since i_4 has not been issued yet, it is obviously not completed, and RST[7].*completed* is clear.

Based on this pipeline state, let us consider the two possible actions depending on the resolution of branch i_3. A correct prediction of the branch makes instruction i_4 non-speculative, and the RST management should continue normally. In contrast, a branch misprediction causes the following instructions to be squashed, and the renaming actions performed by i_4 should be undone. In the example, the final state after

all instruction have been issued, completed, and (in)validated would be the following depending on the prediction correctness:

- *Correct prediction*. All physical registers would be free (i.e., RST[i] = {0, 1, 1}), except register 5, 6, and 7 (i.e., RST[j] = {0, 0, 1}). The action performed by i_4 after leaving the VB is to set the RST[3].*unmapped* bit.

- *Misprediction of branch* i_3. All physical register would be free, except registers 3, 5, and 6. The action carried out by i_4 after leaving the VB is to undo its changes by setting the RST[7].*unmapped* bit. When i_4 completes, RST[7].*completed* is set, and physical register 7 becomes free.

3.3 Performance Evaluation

For comparison purposes, two ROB-based proposals without checkpointing have been modeled, one retiring instructions in program order (hereafter the IOC processor) and the other implementing the out-of-order retirement technique proposed in [9] (from now on, the Scan processor). To perform a fair comparison, the recovery mechanism for the ROB-based architectures is triggered at the writeback stage (see Section 1.1), to make it independent from the time instructions are retired. In addition, the recovery penalty has been assumed to take a constant value of 10 cycles for all simulated microarchitectures, regardless of the ROB/VB size. Notice that this is a conservative assumption for the VB, since it will usually have a smaller number of entries, which helps decrease the real recovery penalty [7].

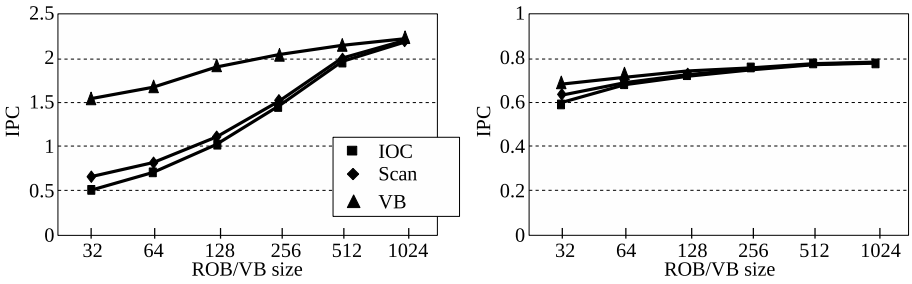
Table 3.2 summarizes the architectural parameters of the baseline modeled machine. Experiments have been performed by running the SPEC2000 benchmark suite, following the methodology described in Section 2.5.2. The experimental study pursues two main goals: first, it quantifies the potential of the proposed architecture, by modeling an unlimited number of entries for some structures, and making performance be limited only by the size of other specific storage elements; second, the study spans performance and complexity issues with realistic structure sizes, resembling a real commercial processor.

3.3.1 Quantifying the Performance Potential

The first step in our performance study is the evaluation of the maximum benefits that can be achieved by out-of-order retirement of instructions. For this aim, the size of the major processor structures other than the ROB/VB are assumed unbounded. These structures include the instruction queue (IQ), load-store queue (LSQ), and register file (RF). Figure 3.4 shows the average IPC for SpecFP and SpecInt when varying the ROB/VB size.

Processor Core	
Machine width (decode, dispatch, issue, commit/ validate)	4 uops/cycle
Storage resources	40-entry IQ, 20-entry LSQ, 64-entry ROB, 64-entry RF
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 type	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 cache	32KB, 4-way, 64-byte block, 2-cycle latency.
L2 cache	512KB, 8-way, 64-byte block, 20-cycle latency.
Main memory	200-cycle access time.

Table 3.2: Baseline superscalar processor parameters.



a) SpecFP benchmarks b) SpecInt benchmarks

Figure 3.4: Average potential performance for SpecFP and SpecInt benchmarks with unbounded IQ, LSQ, and RF.

As observed, performance improvements provided by the VB microarchitecture are much higher for floating-point benchmarks. The reason is that the ROB size is not the main performance bottleneck in integer applications, partially due to a higher percentage of mispredicted branches—this result complies with the observations made in [6] and [9]. Thus, performance analysis will focus on SpecFP hereafter. Concerning the latter, the highest IPC difference appears for the smallest VB/ROB size (i.e., 32 entries), and this difference gets smaller as the VB/ROB size increases. In spite of this, a large 1024-entry ROB is required for the IOC and Scan processors to match the performance achieved by the VB microarchitecture.

Figure 3.5 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 instructions taking long time to complete are the ones most likely

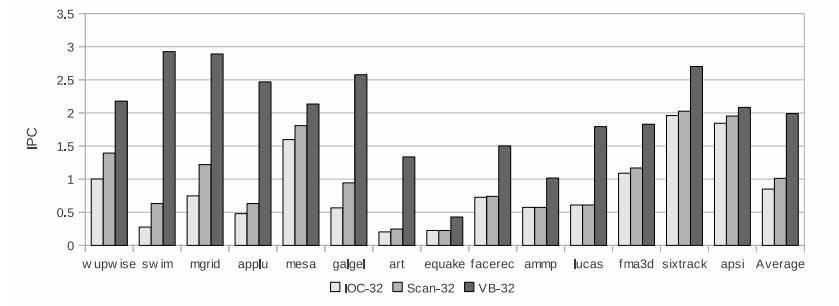


Figure 3.5: Potential performance for SpecFP benchmarks with a 32-entry ROB/VB and unbounded IQ, LSQ, and RF.

to be stalling the ROB head. To provide more insight into this fact, the specific impact on performance of long-latency instructions has been explored. Figure 3.6 shows the distribution of cycles where the decode stage was stalled due to a lack of space in the ROB. Stalled cycles are classified according to the type of the instruction that was blocking the ROB (memory, floating-point, and others). The category *None* represents those cycles where no stalls occurred (i.e., the fetched instructions were effectively decoded). Finally, decode stalls due to an absence of instructions to be consumed from the fetch queue (IFQ) are also quantified. Notice that the sum of all categories represents the total execution time of each benchmark.

As observed, memory instructions are the main cause for stalls. The second, though much more infrequent, cause of ROB blocking are floating-point instructions. Notice that the VB microarchitecture effectively deals with both causes in most of the benchmarks (e.g., *art*). In contrast, the VB provides minor benefits for those benchmarks where dispatching is stalled neither by memory nor by floating-point instructions (e.g., *apsi*).

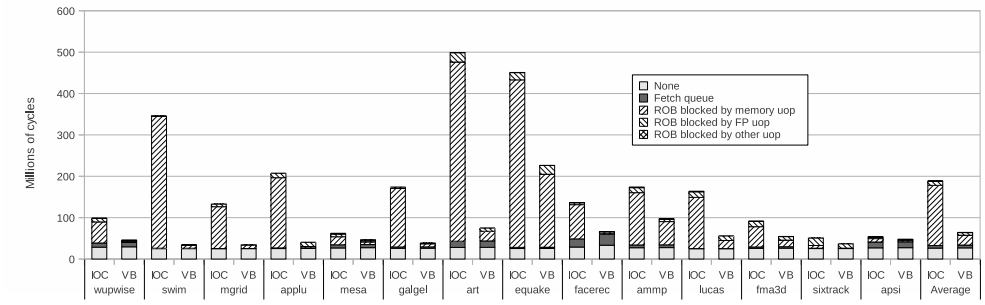


Figure 3.6: Execution time categorized at the dispatch stage in a machine with unbounded IQ, LSQ, and RF.

3.3.2 Exploring the Behavior in a Modern Microprocessor

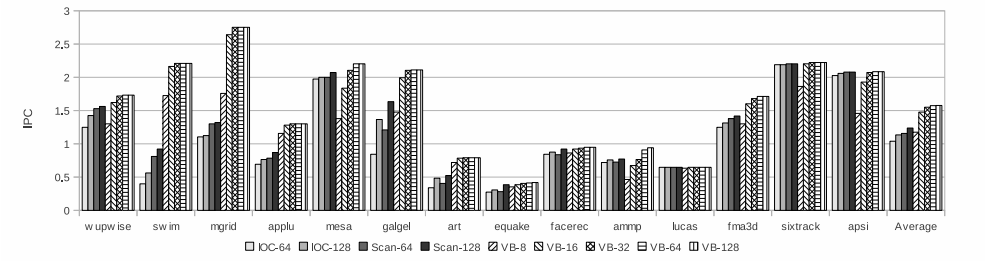


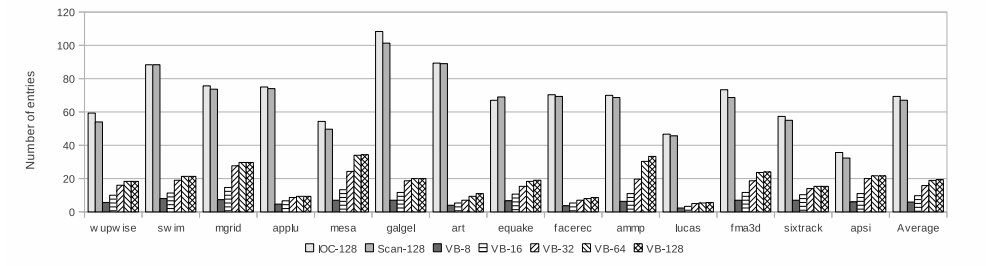
Figure 3.7: Performance for SpecFP in a modern microprocessor.

This section explores the behavior of the VB architecture with the major microprocessor structures closely resembling those implemented in the Intel Pentium 4 microprocessor, as specified in Table 3.2. The size of the VB has been ranged from 8 to 128 entries, and IPC values are shown in Figure 3.7 for SpecFP benchmarks. Results show that the VB microarchitecture is much more efficient, achieving with only 16 entries higher IPC than its counterparts, on average. On the other hand, VBs larger than 32 entries provide minor benefits on performance.

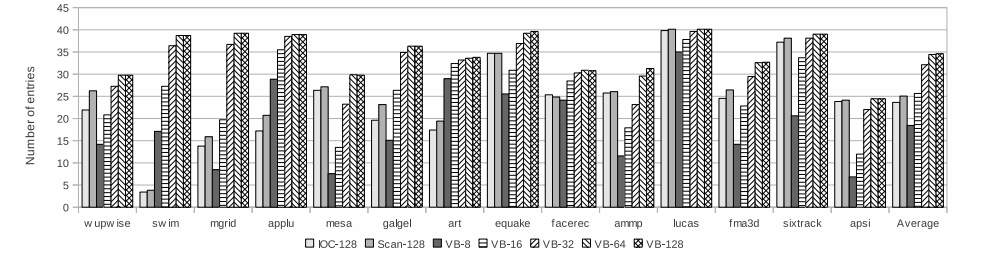
Although the VB microarchitecture does not stand out to the same extent for integer as for floating-point benchmarks, simulations for SpecInt have also been performed, showing that a 32-entry VB performs as well as a 128-entry ROB. Thus, it can be concluded that performance of integer benchmarks is not significantly affected when reducing the VB size.

To explore the complexity requirements of the four major microprocessor structures in the VB microarchitecture, their occupancy in number of entries has been measured. As shown in Figure 3.8, the VB architecture tends to reduce it, in general, regardless of the VB/ROB size, and with the exception of the instruction queue. An increase in the latter's occupancy is explained by the reduction in dispatch stalls due to a lack of space in the ROB, which shifts the pressure of the instruction flow into the instruction queue (see Section 3.4.1).

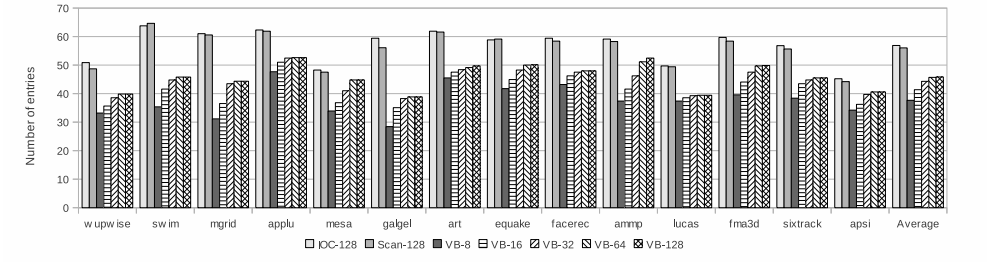
The VB occupancy is about one third of the ROB occupancy on average. The highest IPC benefits appear in those applications whose VB requirements are smaller than the IQ demand (e.g., *swim* or *mgrid*, Figures 3.8a and 3.8b). In these cases, the VB microarchitecture effectively increases ILP by enlarging the instruction window size, and allowing more instructions to be concurrently executed. Results also show the effectiveness of the proposed register reclamation mechanism (Figure 3.8c), which leads to a lower number of required registers. Finally, the LSQ occupancy is also lower in the VB microarchitecture (Figure 3.8d). The reason is that an LSQ entry cannot be released in ROB-based schemes until all previous instructions have



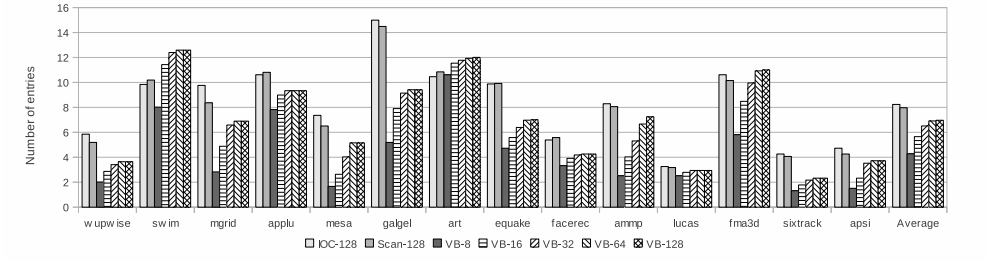
a) ROB and VB.



b) Instruction queue.



c) Register file.



d) Load-store queue.

Figure 3.8: Resources occupancy.

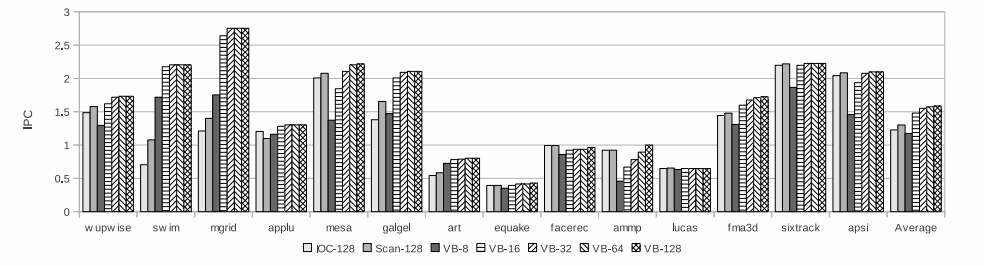
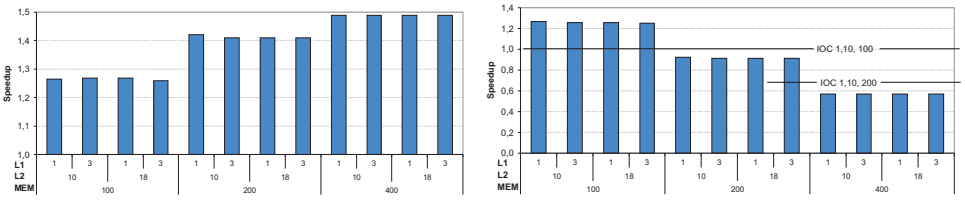


Figure 3.9: Performance with an unbounded RF.



a) Speedup over IOC.

b) Speedup over IOC with {1, 10, 100} memory configuration.

Figure 3.10: Impact on performance of memory latencies.

committed. In contrast, the VB microarchitecture can free an LSQ entry as soon as all previous instructions have been validated, which is a weaker condition.

The proposed architecture benefits both from out-of-order retirement and aggressive register reclamation. To distinguish the isolated contribution of out-of-order retirement, a set of simulations has been run by modeling a processor with an unbounded register file. As observed in Figure 3.9, the register renaming mechanism itself slightly affects the overall performance of the VB microarchitecture, so most benefits come from the fact that instructions are retired out of order. In contrast, IOC and Scan improve their performance with an unbounded amount of physical registers, yet these configurations are outperformed by a 16-entry VB.

3.3.3 Impact on Performance of Memory Latencies

To analyze the impact on performance of the access time of the memory hierarchy components, this section explores different realistic access times for L1 (i.e., 1 and 3 cycles), L2 (10 and 18 cycles) and main memory (100, 200, and 400 cycles). The processor core parameters remain as specified in Table 3.2. Each bar in Figure 3.10a shows the speedup achieved by the VB over the IOC architecture with exactly the same memory configuration. As observed, the main impact on performance comes

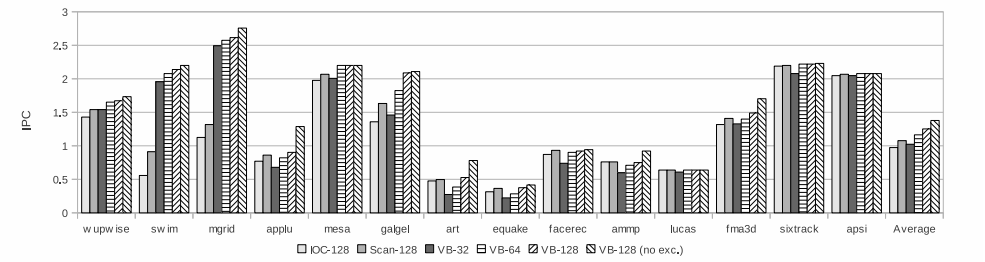


Figure 3.11: Performance with precise floating-point exceptions support.

from the main memory latency. When the latter increases, the wider instruction window provided by the VB architecture makes the speedup grow.

Figure 3.10b shows a cross-comparison version of the previous results. This figure plots relative speedups over the IOC processor with the fastest memory configuration (1, 10, 100). For the same main memory latency, the VB achieves a speedup of about 27%. When comparing the VB-based system with a 200-cycle memory latency against the IOC processor with the fastest main memory, performance is shown to drop by about 8%, while an IOC processor with the same memory latency makes it drop by about 43%.

3.3.4 Supporting Precise Floating-Point Exceptions

The VB architecture can support precise floating-point exceptions by including within the set of epoch initiators those instructions susceptible of raising them. However, the inclusion of all floating-point instructions damages performance, as many of them take tens of cycles to complete. In this section, an epoch initiated by a floating-point operation is assumed to be resolved when the instruction completes. Notice that this is a conservative approach, since some floating-point exceptions can be early detected by comparing the operator exponents (e.g., overflow) or checking one of the operands (e.g., division by zero).

Figure 3.11 shows the impact on performance of the precise floating-point exceptions support, compared with VB architecture with imprecise exceptions. As expected, the VB architecture performance is damaged, even though a 32-entry VB still behaves similarly to the IOC processor with a 128-entry ROB. Also, a 64-entry VB achieves performance close to the Scan and IOC models with a ROB twice as large.

Although precise floating-point exceptions are encouraged by the IEEE-754 floating-point standard, the performance penalty is considerable in many microarchitectures. As a matter of fact, most processors enable the deactivation of floating-point exceptions by software. In some of them (e.g., Alpha 21164), these exceptions are even imprecise by default [34], and the help of the compiler is required to de-

tect which instruction raised an exception [35]. This behavior is desirable for the proposed architecture to sustain the benefits provided by out-of-order retirement.

3.4 Hardware Complexity

Superscalar processors exploit instruction level parallelism by keeping long chains of instructions in flight. Usually, this requires an increase of complexity in some major hardware structures, which can in turn adversely impact the clock cycle. With this tradeoff in mind, this section shows how hardware complexity, in terms of structure sizes and pipeline width, varies for a given performance.

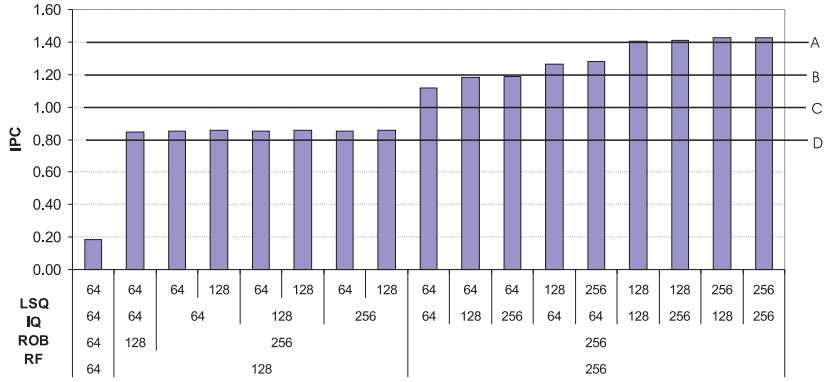
3.4.1 Size of the Major Processor Components

The microprocessor design involves correctly dimensioning hardware structures. If one component is too small, it becomes a bottleneck that incurs performance degradation. In contrast, an unnecessarily large component incurs power and area waste. The goal of this section is to identify the least complex configuration to reach a given performance level. To this end, a wide range of sizes for the major processor structures has been analyzed: 64, 128, and 256 entries for the register file (RF); 16, 32, 64, 128, and 256 entries for the instruction queue (IQ); 16, 32, 64, 128, and 256 entries for the load-store queue (LSQ); and 16, 32, 64, 128, and 256 entries for the ROB/VB. The rest of the parameters remain as specified in Table 3.2.

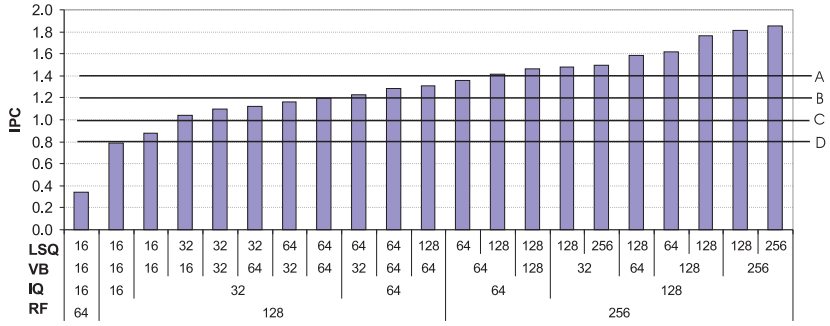
The combination of all tested structure sizes provides an amount of $5^3 \times 3 = 375$ configurations for each modeled architecture, which multiplied by the number of executed benchmarks represents an important number of simulations. To ease this analysis, results have been first ordered by increasing performance. Then, those configurations providing very similar performance have been filtered by discarding those with higher complexity.

Figure 3.12 shows the filtered results for both the in-order and the out-of-order retirement processor. In general, the RF presents itself as the main performance bottleneck in both architectures. The second bottleneck differs for each case, though. In the IOC processor, it is imposed by the ROB, while in the VB architecture it is the IQ. A complexity comparison is performed by selecting four performance levels, labeled from A to D.

Based on the lowest performance level (label D), it can be observed that the IOC processor requires a minimum complexity of a 128-entry RF, a 128-entry ROB, a 64-entry IQ, and a 64-entry LSQ, while the VB-based processor reaches similar performance with half the size of the IQ, and both a VB and an LSQ four times smaller. Regarding the highest labeled performance level (label A), the IOC processor requires a complexity of at least 256 RF entries, a 256-entry ROB, a 128-entry IQ, and a 128-entry LSQ, while the VB-based processor reaches similar performance with half the



a) ROB-based processor.



b) VB microarchitecture.

Figure 3.12: Performance for different RF, IQ, ROB/VB, and LSQ sizes.

size of the IQ, and a VB four times smaller. Performance levels B and C represent other intermediate IPC values in both architectures. For any given performance level, it is possible to find a hardware configuration in the VB architecture with much less complexity than that required for the IOC processor.

3.4.2 Impact of the Pipeline Width

In this section, several values for the pipeline width (i.e., fetch, decode, issue, and commit/validation bandwidth) are explored on top of the IOC and VB-based processor models with the baseline core parameters. A reduction of the pipeline width has important implications in complexity, which is usually more aggressive than just resizing a hardware structure. For example, the hardware cost incurred by the dependence check logic in the instruction queue depends over-linearly on the issue

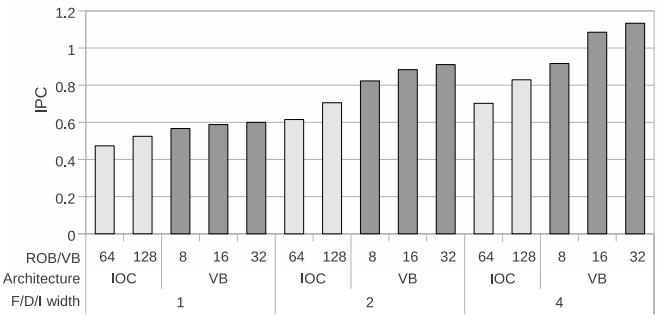


Figure 3.13: Performance for different pipeline widths.

bandwidth. Also, the number of ports in the register file is directly affected by the renaming logic, among others. More details on complexity issues can be found in [5].

Figure 3.13 represents the performance achieved by pipeline widths ranging from 1 to 4 instructions per cycle for the compared architectures. A remarkable result is the performance equivalence of a 4-way IOC design with a 128-entry ROB and a 2-way VB-based design with an 8-entry VB. Also, a VB-based processor with a pipeline width of 1 instruction per cycle reaches performance close to a much more complex IOC processor with a 2-way pipeline and a 64-entry ROB. In summary, experimental results evidence that the VB microarchitecture can not only outperform existing in-order and out-of-order retirement architectures, but also requires much lower hardware complexity.

3.5 Summary

In this chapter, the VB microarchitecture has been proposed for superscalar processors. This architecture retires instructions out of order, while providing support for speculation and precise exceptions. Moreover, checkpoints are not needed due to the non-speculative nature of the out-of-order retirement proposal. A performance evaluation study has been presented, where the VB architecture has been compared against a typical ROB-based processor and a previously proposed out-of-order retirement approach.

From this study, two significant results are highlighted: i) a 32-entry VB achieves performance similar to a 256-entry ROB, and ii) other major processor structures, or even the pipeline width, can be reduced, simplifying their hardware cost while sustaining performance. In summary, a VB-based superscalar processor has been shown to behave as a complexity-effective microarchitecture, since it can be viewed as a design aimed either at improving performance or at reducing complexity.

Chapter 4

The Multithreaded Validation Buffer Architecture

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, out-of-order retirement of instructions helps unclog the ROB when a long-latency instruction reaches its head, which further contributes to increase the utilization of the available issue bandwidth. In this chapter, a performance and complexity evaluation of a multithreaded, out-of-order retirement architecture is carried out, where different multithreading models and instruction fetch policies are analyzed.

4.1 Out-of-Order Retirement Multithreaded Architecture

As any multithreaded processor, the multithreaded Validation Buffer (VB) architecture provides for the operating system the illusion of having multiple logical processors working in parallel. The key idea behind multithreading is sharing a common functional unit pool to increase its utilization by feeding it with independent instructions from different threads. The main design issue when extending a single-threaded processor to support multithreading is deciding whether hardware components other than functional units should be private or shared among threads [9]. This aspect will be tackled in the following sections, ending with a discussion about sharing the VB structure.

4.1.1 Execution of Multiple Contexts

A multithreaded processor contains hardware to store the state of several software contexts and, optionally, execute them at the same time. As detailed in Section 2.3, the state of a context is represented by a virtual memory image and a logical register file, which are designed as follows:

- **Virtual memory image**. Hardware threads in a multithreaded processor access a common main memory pool with a shared physical address space, which is split among threads by the MMU (Memory Management Unit). The MMU includes a page table managed by the operating system (OS), which is indexed by a process identifier and a virtual address, and returns the associated physical address. To avoid a memory access for each address translation, a Translation Look-aside Buffer (TLB) is used within the processor as a cache for the page table, which is flushed every time a context switch is triggered by the OS.

  Since there are multiple active contexts in a multithreaded processor, the page table must be cached independently for each context using one of the following options. On one hand, a TLB shared among threads can be adapted to be indexed by a virtual address and a thread identifier. On the other hand, TLBs can be simply private per thread and indexed only by a virtual address. The advantage of a private approach is that no selective TLB flush needs to be implemented to support context switches; the whole private TLB is reset in this case. Even though no context switch is required in a model where the number of contexts matches the number of hardware threads, the private TLB approach is used for our simulations.

- **Logical register file**. The values associated with logical registers and status flags are stored in entries of the physical register file. This structure can be shared or private per thread. In both designs, it is guaranteed that the subset

of physical registers bound to specific threads are always disjoint, that is, no physical register is mapped to two logical registers of different contexts.

To keep disjoint register mappings for each context in a multithreaded processor, the register alias table (RAT) can be implemented in the following manners. On one hand, a common RAT can be modified to be indexed by a logical register and a thread identifier, returning the associated physical register. On the other hand, one private RAT can be used per thread, indexed just by a logical register. The latter approach is used for our simulations, since it avoids a selective RAT recovery after a branch mispeculation.

4.1.2 Resources Sharing

A straightforward way to extend a single-threaded processor to implement multithreading consists in replicating all hardware structures per thread, such as the reorder buffer (ROB), the instruction queue (IQ), the load-store queue (LSQ), or the register file (RF), while maintaining a common pool of functional units. 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 for resources.

Shared resources are often more costly than private ones due to a higher hardware complexity devoted for allocation or deallocation of individual entries. Moreover, shared resources need to increase read/write ports in order to enable parallel access from various threads, which may increase both their area and latency. In contrast, the potential advantage of shared resources is that one single thread can compete for all its entries. For example, if only one thread has instructions in its uop queue ready to be dispatched, it can use all entries of a shared IQ to dump them. If the IQ is private per thread, the active context is restricted to its assigned IQ portion, while the remaining IQs are unused.

But this does not mean that a shared approach performs always better. As pointed out in [36], shared storage resources can cause a performance degradation if they are abusively occupied by inactive threads. For example, if a thread is able to allocate all entries of a shared RF and then it stalls in a long-latency operation, no other thread can be dispatched until this operation finishes. This case is a sample of an unfair assignment of resources to a thread that cannot make an effective use of them.

The problem of fairly assigning resources to threads is solved with two different approaches, evaluated in this chapter. On one hand, a simple private approach provides a fair distribution of resources among threads, with simple hardware implementation but poor single-thread performance (Section 4.2.1). On the other hand, a shared approach can be instrumented with wise resource allocation policies to prevent stalled threads from greedily occupying resources (Section 4.2.4).

4.1.3 Resource Allocation Policies

To exploit the advantages of shared structures in a multithreaded design, several resource allocation policies have been proposed in other works. The aim of resource allocation policies is to make proper decisions about the assignment of structure entries to threads requesting them. In general, these decisions are based on the current distribution of resource entries and the recent history of running threads. The allocation policies used in our evaluated multithreaded architectures with shared structures are briefly described next.

- **Round-Robin (RR)**. This is a straightforward allocation policy, which consists in assigning resource entries to threads in a rotative way each cycle. If one thread has no ready instructions in a given cycle, it is skipped without consuming its allocation time slot.

- **Instruction Count (ICOUNT)**. This is a fetch policy proposed by Tullsen et al. [37], which assigns fetch priority to those threads with less instructions in the decode, rename, and issue stages. The notation 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. Experiments shown later use an ICOUNT.2.8 configuration, which provides the best results for the ROB-SMT architecture.

- **Predictive Data Gating (PDG)**. This policy was proposed by El-Moursy et al. [38], 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 data dependences with 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 stop instruction fetch when a certain threshold is exceeded.

  As suggested in [38], 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 incurs a data cache miss, the corresponding counter is reset, whereas it is incremented when the associated load hits in the cache.

- **Dynamically Controlled Resource Allocation (DCRA)**. This policy was proposed by Cazorla et al. [39]. Unlike ICOUNT and PDG, DCRA does not only control the fetch bandwidth fraction granted to each thread, but also manages the allocation of other shared resources. DCRA classifies threads according to

two criteria. First, a thread is considered slow or fast, depending on whether it has or not pending L1 cache misses. Second, a thread is considered active or inactive for a given resource depending on whether it has recently (e.g., in the last 256 cycles) requested its allocation.

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

4.1.4 Using Out-of-Order Retirement

A multithreaded processor can implement the proposed out-of-order retirement architecture by using a validation buffer (VB) and the aggressive register renaming technique presented in Section 3.1.1. Since it is a FIFO queue, the VB is a special case when it is designed as a structure shared among thread. Two mechanisms are devised to manage the assignment of VB entries to threads, referred to as *dynamic VB partitions* and *unified VB*. Both approaches enjoy the advantages of a shared resource, namely that one single thread can flexibly use any number of entries, and both of them can implement a resource allocation policy to avoid thread starvation. However, there are additional issues to be considered when sharing a FIFO queue. These are the discussed alternatives:

- **Dynamic VB partitions.** Similarly to the sharing strategy for the ROB proposed in [40], this mechanism consists in assigning disjoint VB portions to threads, whose size hinges upon the threads demand. Each portion is handled as an independent VB, where instructions are inserted and extracted in the local thread's FIFO order. The main drawback of this design is that VB portions cannot grow arbitrarily to fill the whole VB size. Specifically, a portion size is constrained by the position of the contiguous portions, and VB portions can only be shifted when the associated *head* and *tail* pointers are properly aligned.

- **Unified VB.** An alternative approach consists in inserting instructions in global FIFO order into a unified VB, as tough all of them belonged to the same thread. As a consequence, instructions are forced to be retired in the same global order. The problem presented by this implementation is that instructions from different threads are intermingled throughout the VB; if an instruction at the unified VB head cannot be validated, it will prevent instructions from other threads from leaving the VB, incurring unnecessary head-of-line blocking. Moreover, the recovery process should cancel instructions selectively, forcing interleaved gaps to remain in the VB until they are drained.

In a previous work [41], we have tackled the problem of sharing a ROB in a multithreaded, in-order retirement processor, where an efficient solution is proposed to trade off the advantages and drawbacks of each sharing strategy. However, this study is out of the scope of this thesis. For all experiments presented in this chapter, private ROBs and VBs have been assumed, eluding the problem of sharing a FIFO queue.

4.2 Performance Evaluation

Performance and complexity issues have been addressed on a multithreaded processor implementing the VB architecture. The main characteristics of the modeled machines are listed in Table 4.1. For those experiments running more than one benchmark simultaneously, the following criterion, based on previous works [40], has been used to construct workload mixes. First, SPEC2000 benchmarks have been characterized as memory- or CPU-bound, as described in Section 2.5.2, and then three different groups are created according to this classification. Two additional groups are analyzed including only integer or floating-point benchmarks, respectively. The specific benchmark mixes are listed in Table 4.2.

Processor Core / Hardware Threads	
Machine width (decode, dispatch, issue, commit/ validate)	8 uops/cycle
Storage resources	40-entry IQ, 20-entry LSQ, 64-entry ROB, 64-entry RF, all private per thread.
Functional units and latency (total/issue)	8 Int. add (2/1), 2 Int. mult. (5/2), 2 Int. div (20/10) 4 FP add (5/2), 2 FP mult. (10/5), 2 FP. div. (30/15)
Branch predictor type	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 cache	32KB, 4-way, 64-byte block, 2-cycle latency, private per thread.
L2 cache	1MB, 8-way, 64-byte block, 20-cycle latency, shared among threads.
Main memory	200-cycle access time.

Table 4.1: Baseline multithreaded processor parameters.

Classification	Mix name	Benchmarks
CPU	Mix 0	*wupwise, eon*
	Mix 1	*apsi, eon, fma3d, gcc*
CPU+MEM	Mix 2	*art, gzip*
	Mix 3	*art, gzip, wupwise, twolf*
MEM	Mix 4	*applu, ammp*
	Mix 5	*applu, ammp, art, mcf*
Integer	Mix 6	*gcc, gzip*
Floating-point	Mix 7	*wupwise, mgrid*

Table 4.2: Benchmark mixes.

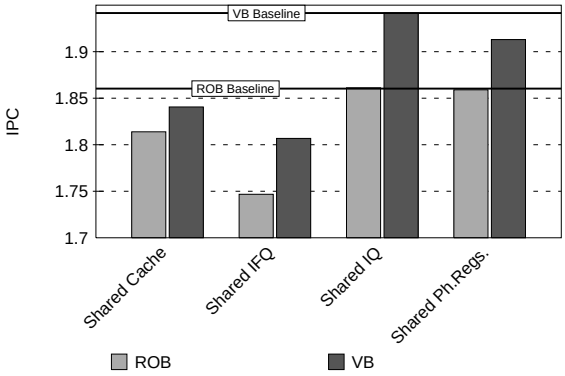


Figure 4.1: Storage resources sharing for the multithreaded ROB and VB architectures. Baseline IPCs refer to processors with no shared storage resource.

4.2.1 Sharing Strategies of Hardware Structures

This section studies the effect of sharing some storage resources among hardware threads, focusing on the L1 caches, the fetch queue (IFQ), the instruction queue (IQ), and the register file (RF). In Figure 4.1, the performance achieved by each design is plotted for both the ROB and the VB multithreaded architectures. 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, and each performance value is computed as the average IPC for the execution of all benchmark mixes. Additionally, the performance obtained with a 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.

Coherently with conclusions stated in [36] for a ROB-based processor, Figure 4.1 reveals that there is a performance loss when sharing any storage resource without a proper allocation policy. Out-of-order retirement does not affect slow or stalled threads, which continue to abuse of shared resources by reserving their entries for a long time and preventing other threads from using them fruitfully.

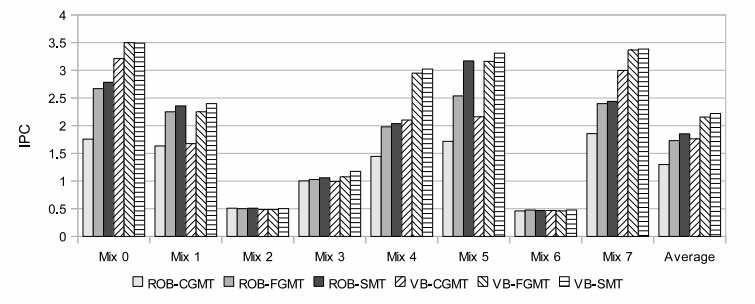


Figure 4.2: Performance for different multithreading paradigms in the ROB/VB architectures.

4.2.2 Comparison of Multithreading Paradigms

The VB architecture has been evaluated on top of the three major multithreading paradigms, namely coarse-grain (CGMT), fine-grain (FGMT), and simultaneous multithreading (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 either a timeslice or a round-robin instruction fetch policy. The CGMT design uses a thread quantum of 1000 cycles, and the thread switch penalty is equal to the number of cycles needed to drain all instructions of the previous thread from the processor pipeline [9].

Performance results are represented in Figure 4.2. Comparing the behavior of the multithreaded VB architecture in its different variants (i.e., VB-CGMT, VB-FGMT, and VB-SMT) with the ROB-SMT processor, it can be observed that the latter is outperformed by VB-FGMT and VB-SMT by about 16.4% and 19.7%, respectively, while only VB-CGMT performs worse with a 5% slowdown. Mixes 2 and 6 show a flat behavior both when substituting the ROB by the VB and when improving the multithreading paradigm. As observed in previous works [9][42], specific benchmarks, most of them integer benchmarks, do not obtain benefits from enlarging the ROB, nor from retiring instructions out of order. This situation is caused by a scarce 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 from outperforming CGMT and FGMT.

An interesting observation is the average performance improvement of VB-FGMT over ROB-SMT. The former is a simple multithreading paradigm with lightweight additional hardware devoted to bandwidth resources, while the latter 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

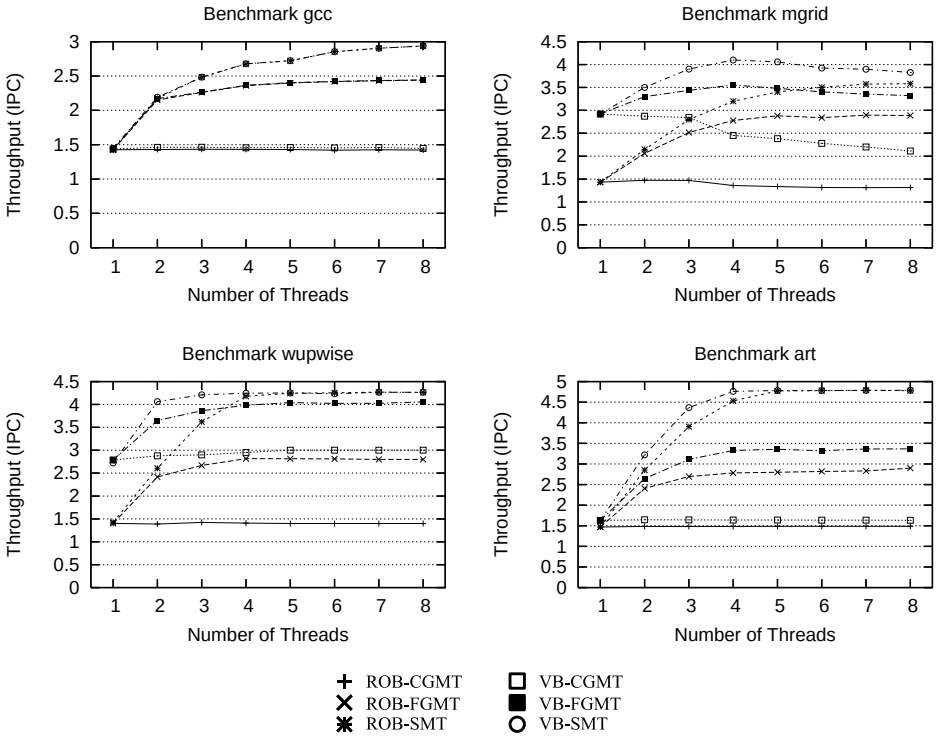


Figure 4.3: Scalability of different multithreading paradigms for ROB/VB architectures.

in VB-FGMT with the extra performance gained from the efficient management of the VB structure, which makes the pipeline stall less often.

4.2.3 Impact of the Number of Hardware Threads

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

Figure 4.3 shows results for four different benchmarks running on the ROB and VB architectures with FGMT, CGMT and SMT. The top left plot represents benchmark *gcc*, which shows a characteristic behavior of integer applications. As mentioned before, out-of-order retirement of instructions provides scarce performance benefits for integer workloads, so the rest of this section will focus on floating-

point benchmarks. The remaining plots correspond to the floating-point benchmarks *mgrid*, *wupwise* and *art*, and show extremely contrasting results compared to *gcc*.

The *mgrid* plot shows an important effect when n increases. On one hand, CGMT provides neither gain nor loss of performance up to 3 threads. In a real system, the benefits of CGMT in this case would come from the fact of avoiding software context switches by the operating system. However, this effect is not appreciated in an n threaded processor running n contexts. It can be observed that a further increase of n has even negative effects on the global IPC, shown clearly in the case of VB-CGMT. The reason is that the impact of the thread switch penalty grows faster than the benefits of a higher utilization of functional units when n is overly large.

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 higher performance is reached with a lower n, and thus, a lower hardware overhead.

An interesting result can be observed in 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 logic in a ROB-based architecture. Although the ROB-SMT scalability is slightly imposed for values of n higher than 4, the simpler VB-FGMT implementation still reaches higher performance up to approximately 4 threads.

4.2.4 Impact of Resource Allocation Policies on SMT processors

In this section, different fetch policies are evaluated on top of the ROB- and VB-based architectures. Except for those processors using the DCRA fetch policy, all of them are based on the parameters shown in Table 4.1. The DCRA model includes 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 from dynamically assigning different number of entries of shared resources to threads.

Results plotted in 4.4 show, on one hand, the pronounced advantage of a sophisticated instruction fetch policy in SMT processors. Any fetch policy other than RR provides higher benefits than the replacement of a ROB by a VB. On the other hand, Figure 4.4 illustrates that the benefits of fetch policies also apply to the VB architecture. Comparing the advanced fetch policies (the three right bars of the *Average*

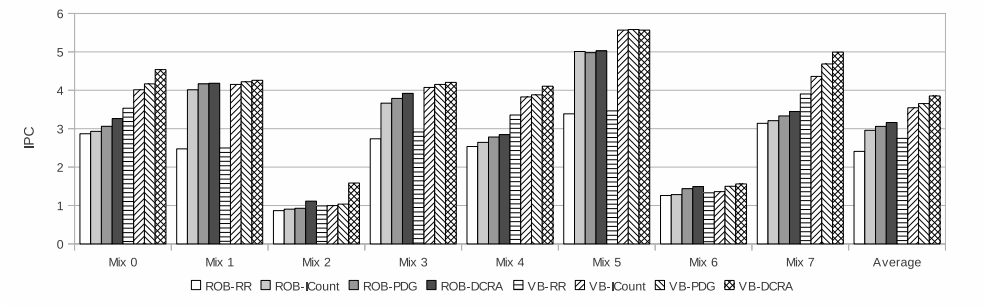


Figure 4.4: Evaluation of fetch policies for the ROB and VB architectures.

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 with the ROB-DCRA policy (the best ROB-based policy), the performance speedup reaches 12%, 15.6% and 21.9%, respectively.

Although an improved fetch policy is needed in the VB microarchitecture to outperform a ROB-DCRA architecture, there is no need to implement the most effective one (VB-DCRA), which requires more complex hardware. Instead, the instruction counters added by ICOUNT are sufficient to make the simple VB-based approach behave better than the best fetch policy for a ROB-based processor (ROB-DCRA). However, it can be appreciated that out-of-order retirement can be combined with the most effective fetch policy, contributing orthogonally to improve performance with respect to the remaining designs.

4.2.5 Resources Occupancy in SMT Designs

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

Figure 4.5 represents the issue bandwidth utilization as a percentage of execution cycles in which a specific number of issue slots has been filled. Results 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 architecture. These results corroborate the relationship between filled issue slots and the multithreaded processor performance, when comparing Fig-

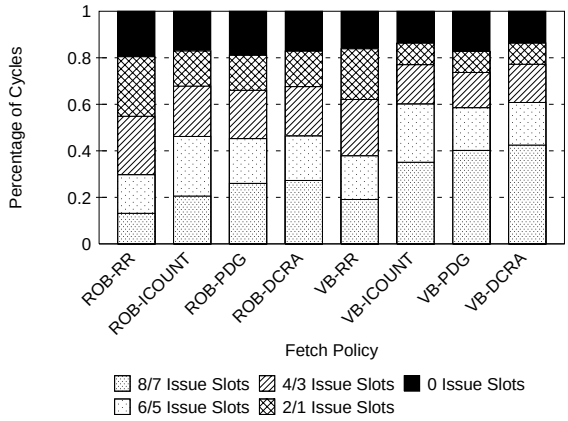


Figure 4.5: Issue slots for different SMT architectures and fetch policies.

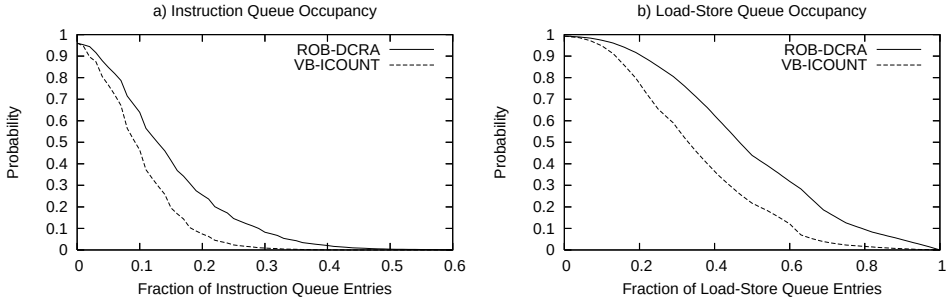


Figure 4.6: Storage resources occupancy for ROB-DCRA and VB-ICOUNT.

ures 4.4 and 4.5. The highest occupancy is achieved with VB-DCRA, which fills 7 or 8 issue slots in more than 40% of the execution time.

Let us compare the bars corresponding to the ROB-DCRA and VB-ICOUNT designs, both in Figures 4.4 and 4.5. Figure 4.4 points ROB-DCRA as the most performance-effective ROB design, while Figure 4.5 shows that it exploits most efficiently the issue bandwidth. On the other hand, it is shown that a simple ICOUNT policy suffices to make VB-ICOUNT outperform ROB-DCRA (a naive RR fetch policy is not enough). This fact makes VB-ICOUNT a cost-effective solution, reaching higher performance than the most complex fetch policy for ROB-SMT, while implementing a simple fetch policy in VB-SMT.

The occupancy of storage resources has also been measured, focusing on the most efficient ROB design (ROB-DCRA) and the design with the simplest efficient fetch policy on the VB (VB-ICOUNT). Figure 4.6 shows the occupancy of the instruction queue (IQ) and the load-store queue (LSQ) for these designs. The curves in the

figures should be interpreted as the probability (Y-axis) for a resource of having an occupancy equal or greater than a specific fraction of its entries (X-axis).

As observed, VB-ICOUNT causes a lower occupancy both in the IQ and the LSQ. In the case of the IQ (Figure 4.6a), only 50% of the IQ entries are being used in ROB-DCRA, which means 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 occurs in the LSQ, which could be implemented with 20% less entries without practically affecting performance. As a consequence, VB-ICOUNT does not only outperform ROB-DCRA with a simpler fetch policy, but also allows for a reduction of the main storage resources size, maintaining performance gains.

4.3 Summary

In this chapter, the VB architecture has been extended for multithreaded processors with different multithreading paradigms, sharing strategies of hardware resources, fetch policies, and resource allocation techniques. An exhaustive experimental evaluation has been performed, where multithreaded executions have been simulated using several mixes of single-threaded benchmarks.

The obtained 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 of the hardware threads than a simultaneous multithreaded ROB-based processor; and iii) benefits of fetch policies (such as DCRA) are orthogonal to those provided by the transition to the VB architecture. These contributions justify the viability and cost-effectiveness of an out-of-order retirement, multithreaded processor microarchitecture.

Chapter 5

The Multicore Validation Buffer Architecture

Multicore processors are now the current norm in the processor market, ranging from the general purpose to the embedded systems sectors. In these architectures, further sources of concurrency are obtained with explicit parallelism in the applications. Multicore systems provide a shared memory hierarchy whose access order is defined by the memory consistency model. In this sense, sequential consistency is the most convenient model, since it eases the system programming interface. However, it imposes ordering restrictions that may interfere with out-of-order retirement of instructions. In this chapter, an implementation of an out-of-order retirement, sequentially consistent, multiprocessor system is proposed, based on the Validation Buffer architecture.

5.1 Dealing with Sequential Consistency

Due to optimizations in the load-store queue, 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, like the sequential memory consistency model (SC) [43], 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* [44] and *correctly labeled* [45] by synchronization operations, any reordering is allowed between synchronizations without affecting the semantics of the parallel program.

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 thesis 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 [46] 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 prevents from hiding the memory latency, and strongly damages performance.

Several techniques have been proposed to improve the performance of sequential consistency implementations (see Chapter 6). In this thesis, the *speculative retirement of loads* technique [29] 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 obtained by obeying them. 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 a case, the processor must be able to recover its state prior to 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 of the memory instructions until they are globally performed. This solution holds critical resources of the execution pipeline for recovering from memory consistency mispeculations. For instance, traditional register renaming does not allow physical registers to be released until instructions leave the ROB. However, as shown in Section 5.2, these mispeculation 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 ones can commit (releasing execution resources as they leave the ROB) even if they are still subject to memory consistency violations. After committed, instructions enter the HB, where they stay until they are globally performed.

Each entry of the HB contains information to undo the modifications performed on the processor state by its corresponding instruction. To do so, any instruction renaming (i.e., writing to) a given logical register holds in its HB entry two values. On one hand, it stores the identifier l of its destination logical register; on the other, it attaches 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. Notice that in a ROB-based multiprocessor, the latter value is always available after the commit stage. With this implementation, whenever a cache block accessed by a non-globally performed memory instruction is being remotely accessed or evicted, execution is recovered from mispeculation 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 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* [47]. 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.

Committed instructions	Branch mispredictions	Arithmetic exceptions
4 041 943 035	33 498 322 (0.8%)	0
Page faults	**Load replay traps**	**Memory consistency**
57 425 (< 0.01%)	331 180 (< 0.01%)	561 070 (0.01%)

Table 5.1: Frequency of misprediction events.

5.2 Out-of-Order Retirement Multiprocessor Architecture

5.2.1 Architecture Description

The condition to extract an instruction from the VB in a single-core environment is that its speculative state is resolved, that is, it is a completed branch, or an instruction that is already known not to raise an exception. Table 5.1 lists different causes of misprediction that should be checked 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 statement. *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 5.1 also 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 5.4, as well as the number of total 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 mispeculation, 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 mispeculations that are likely to occur, that is, branch mispredictions. After exiting the VB in any of their possible execution states, 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, the infrequent misprediction events, that is, all except branch mispredictions, can be resolved in the HB.

With this specification, the conditions to retire instructions from the ROB/VB and HB queues become the ones listed in Figure 5.1. As observed, the difference between the ROB and VB architectures is the relaxation of the conditions for instructions to leave both the ROB/VB and the HB. As specified in Section 3.1.4, the sources of potential benefits in a VB-based single processor can be classified as *extended*

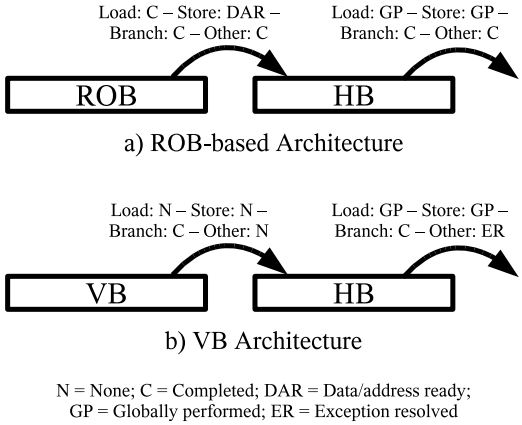


Figure 5.1: Conditions for instructions to be retired from the ROB/VB and HB.

instruction window and *enhanced register usage*. These features are maintained in the multicore VB architecture using the additional HB, as justified next:

- First, the only reason to stall an instruction at the VB head is that it is an uncompleted branch. This relaxation makes the VB alleviate most of its head-of-line blocking effects, enabling physical registers to be released much faster. Thus, the *enhanced register usage* effect is not only maintained, but also intensified with respect to the superscalar VB architecture, where validation conditions were slightly stricter.

- 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, which maintains the effect previously referred to as *extended instruction window*.

5.2.2 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.

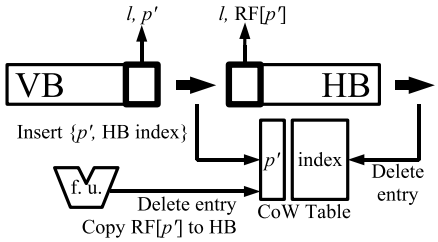


Figure 5.2: Implementation of delayed writebacks.

Figure 5.2 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. Instructions that leave the HB remove their associated entries in the CoW if present, avoiding overly delayed writebacks to affect the HB.

When recovering from a mispeculation at an instruction in the HB, a sufficient condition to retrieve all the recovery information is to wait until the CoW table is empty, that is, 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 5.4).

5.2.3 Working Example

Figure 5.3 represents the pipeline of a single processing unit of a VB-based multiprocessor in three consecutive cycles, focusing on the VB, the HB, and the CoW table. A piece of code formed of four instructions is listed in Figure 5.3a. Instructions i_1 and i_2 are exception-free arithmetic instructions, while i_3 and i_4 are long-latency memory instructions. Each instruction attaches its physical destination register (dst)

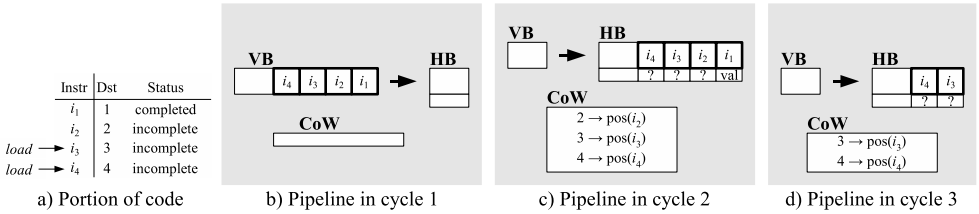


Figure 5.3: Working example for multiprocessors.

and its status, which is assumed invariant during the three cycles. When an instruction is labeled incomplete, it can be either not issued in the IQ, or issued into the corresponding functional unit. These are the observations for each cycle:

- **Cycle 1** (Figure 5.3b). All four instruction are located at the head of the VB, and are currently processed by the validation logic. No branch is found among them, so they are ready to be placed into the HB. The CoW table is empty so far.

- **Cycle 2** (Figure 5.3c). The four instructions are placed into the HB, and their completion is verified by the commit logic by testing the corresponding $completed$ bits in the RST. Completed instructions (i_1) copy the value of their destination physical register into the HB. For each incomplete instruction (i_2, i_3, and i_4), a delayed writeback into the HB is scheduled by allocating a new entry in the CoW. For example, i_2 records the identifier of its destination physical register (2), jointly with the HB position where the instruction was inserted.

- **Cycle 3** (Figure 5.3d). Instructions i_1 and i_2 exit the HB, because they are known to be free of exception, mispeculation, or memory consistency violation. However, i_3 and i_4 are long-latency loads that must remain in the HB. Instruction i_2 has released its HB entry while being still incomplete. Since there remains no HB entry associated with it, its eventual completion should no longer be propagated into the HB. Thus, the CoW entry holding the identifier 2 is removed to prevent an undesired delayed writeback.

After cycle 3, the completion of either instruction i_3 or i_4 will cause the CoW table to be looked up, and it will provide the indexes of the HB where the contents of physical registers 3 and 4 should be copied. In contrast, the CoW table will provide no valid position when searching physical register 2 after the completion of instruction i_2, and its result will be silently copied into the register file.

5.3 Analysis of Single-Thread Performance

This section evaluates the impact of the *enhanced register usage* and *extended instruction window* effects provided by a single VB-based processing node. Notice that the main difference between this study and the study of potential presented in Section 3.3.1 lies in the fact that the architecture now under evaluation includes the HB implementation shown in Figure 5.2, that is, it is a processor model susceptible of being integrated into a multiprocessor environment. However, single-threaded benchmarks from the SPEC2000 suite [30] are used for the moment to avoid processor stalls due to synchronization and communication delays. To avoid mispeculation in the control flow stream, a perfect branch predictor is utilized, and, unless explicitly stated, the remaining machine parameters match the baseline configuration shown in Table 5.2.

5.3.1 Enhanced Register Usage

The VB architecture uses an aggressive register renaming strategy. As in traditional renaming mechanisms, instructions reclaim registers at the decode/rename stage. However, register release is decoupled from the withdrawal of instructions from the VB, which entails a more efficient management of the register file. The following experiment illustrates this advantage.

When an instruction is dispatched (i.e., enters the VB), it allocates a physical register p as its destination mapping; when the instruction exits the VB, it may be ready to release the physical register p' corresponding to the previous mapping of its destination logical register (this is a necessary, but not sufficient condition). We have measured the average register allocation time as the number of cycles between the time an instruction allocates physical register p and the time that physical register p' is released. Notice that in the ROB architecture this time is equals to the time spent by the mapping instruction in the ROB, whereas these times differ in the VB architecture.

Figure 5.4 presents differences in register allocation time between the ROB and VB architectures. The bars represent the ratio between the allocation time of physical registers in the ROB and the VB architecture. As observed, this ratio varies considerably for different benchmarks, with an average value of 6.5%. This means that, for the modeled machine with an ideal branch predictor, instructions take 6.5% more time (the absolute value is roughly 2.5 cycles) in the ROB architecture to release a physical register since they were dispatched, increasing the probability of decode stalls due to a full register file.

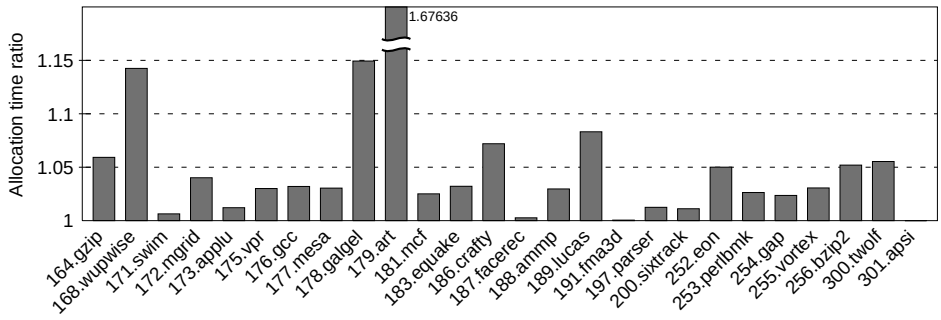


Figure 5.4: Register allocation time ratio between the ROB and the VB architecture, measured with an ideal branch predictor.

5.3.2 Extended Instruction Window

Sequentially consistent multiprocessors need to maintain very wide instruction windows to support long-latency loads and stores. By *instruction window size*, we refer to the decode distance between the youngest and the oldest instruction in the pipeline (assuming no squashed mispeculated instructions). A ROB-based processor usually limits this width to a fixed number of instructions, which is equal to the sum of the entries in the ROB and the HB for our baseline ROB-based multiprocessor architecture.

The second source of potential speedup in the proposed VB-based multiprocessor is that, unlike ROB-based architectures, the number of entries in the VB and HB does not constrain (in general) the instruction window size. Specifically, very wide instruction windows can occur when long chains of instructions other than memory accesses and branches enter the pipeline. To explore the potential benefits of this effect, a machine with an unbounded instruction queue, load-store queue, and register file is modeled, using the baseline sizes for the ROB/VB and HB. A perfect branch predictor is used again to avoid gaps of squashed instructions that affect the true occupancy of the instruction window.

The instruction window sizes are plotted in Figure 5.5, where a group of two bars represents a single SPEC2000 benchmark run on a ROB-based architecture (left bar) and VB-based architecture (right bar). The bar height represents the number of in-flight instructions. In the case of the ROB architecture, instructions in flight are located either in the ROB or in the HB. However, notice that instructions in the VB architecture may not be allocated in either of them, since they might leave the HB without being completed. Each bar in the figure has two parts. The lower part shows the average number of in-flight instructions, while the upper part shows the maximum width of the instruction window.

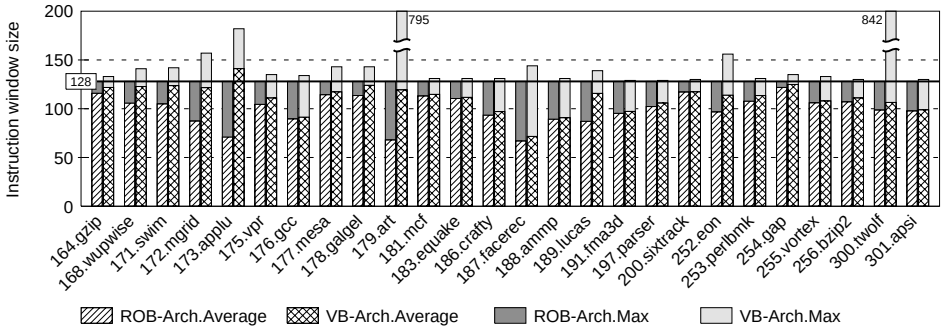


Figure 5.5: Instruction window size, measured with an ideal branch predictor and unbounded IQ, LSQ, and RF.

Results show that, while the maximum window size in the ROB architecture is limited to 128 (ROB size + HB size), the window size in the VB architecture exceeds this value for most applications. The average window size also increases, so there are more chances during program execution to further exploit ILP. This potential can be especially observed, for instance, in benchmarks *applu* or *art*, which contain portions of code with long chains of arithmetic instructions that allow instructions to leave the HB without being completed.

Processor Cores	
Machine width (decode, dispatch, issue, commit/ validate)	4 uops/cycle
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 type	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.

Table 5.2: Baseline multicore processor parameters.

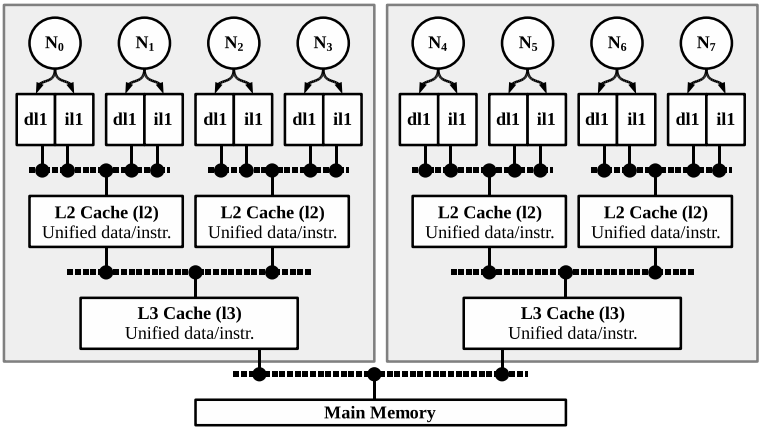


Figure 5.6: Block diagram of the modeled multicore system.

5.4 Performance Evaluation

A realistic multicore processor with 8 cores has been modeled to evaluate the proposed architecture. Figure 5.6 shows a block diagram of the modeled system, whose characteristics are summarized in Table 5.2. 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 inspired on the recent commercial quad-core AMD Opteron 8350 processor [48]. In the figure, the gray boxes represent replicas of this same chip. Parallel workloads with shared data, i.e., the SPLASH2 benchmark suite [31], have been used for our performance evaluations to stress shared components involved by coherence actions and long-latency memory operations. Programs have been run until completion.

5.4.1 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 Figures 5.7a, 5.7b and 5.7c, 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

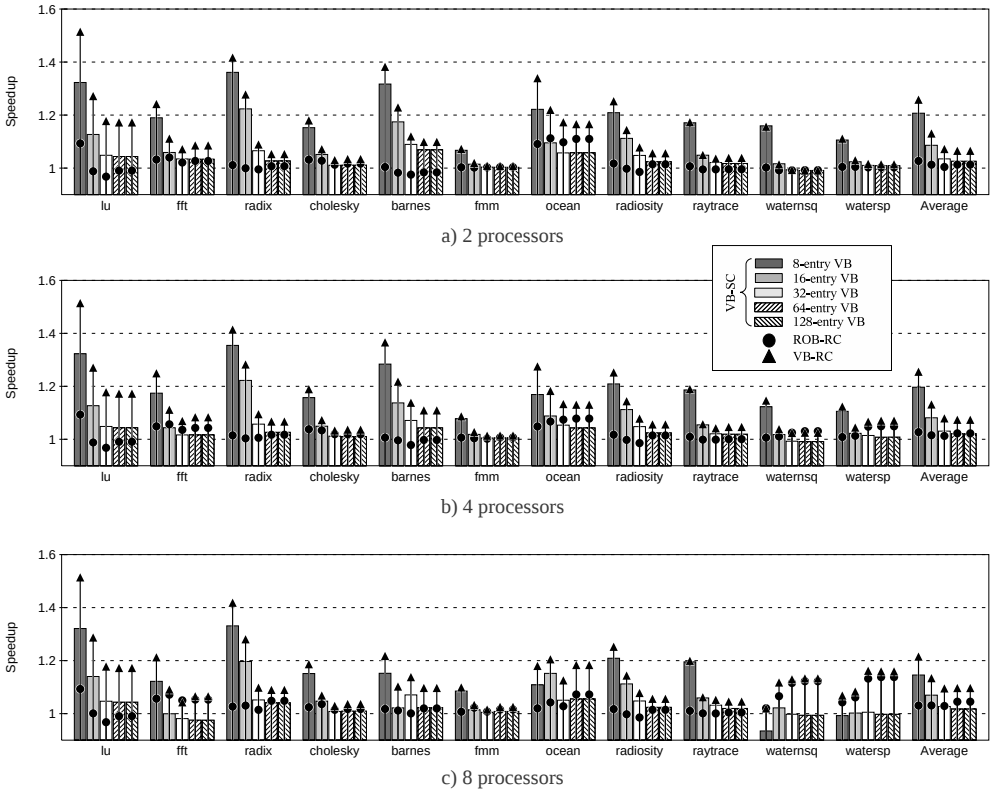


Figure 5.7: Performance speedups relative to ROB-SC.

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 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.

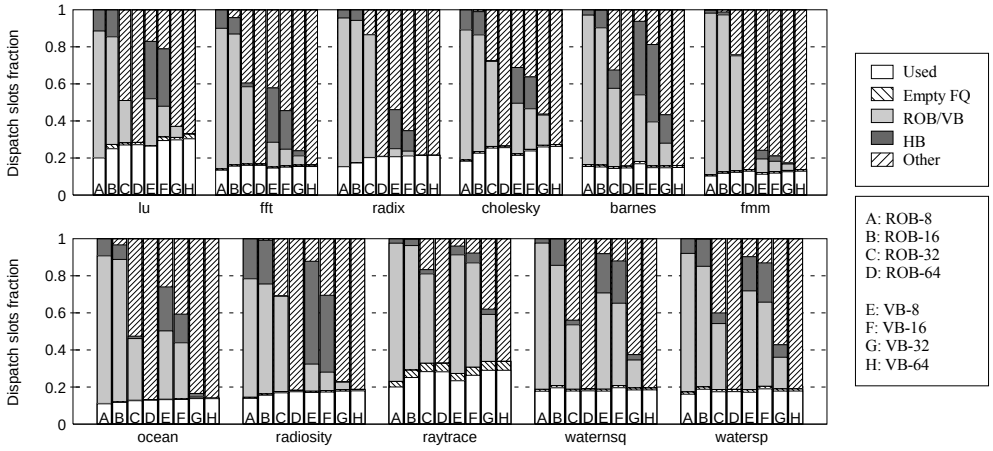


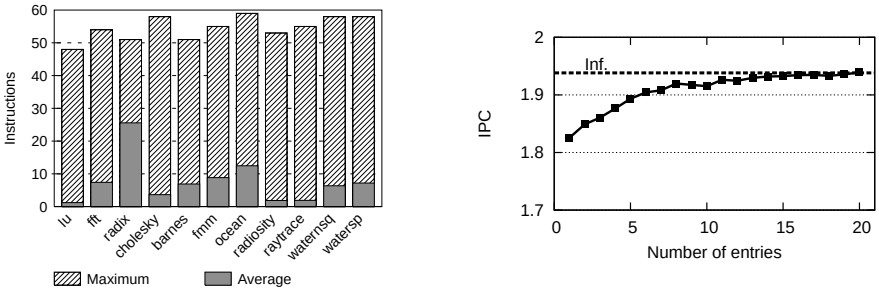
Figure 5.8: Processor bottlenecks at the dispatch stage.

Finally, given the simulation results, the VB-RC model is the best approach, reaching 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.

5.4.2 Performance Bottlenecks

To observe how bottlenecks shift when the ROB/VB size varies in both architectures, the dispatch stage has been instrumented in the following experiment. In a 4-way processor, four instructions are potentially dispatched in each cycle, so each of the four available dispatch slots can be either used or wasted, depending on the availability of processor resources. Figure 5.8 plots the amount of dispatch slots for the execution of the SPLASH2 benchmarks in the baseline 8-core processor, normalized with respect to the total number of dispatch slots, which is equal to the number of execution cycles multiplied by the dispatch width and the number of cores. A dispatch slot can be classified in the following categories:

- *Used*: a microinstruction located at the head of the fetch queue has been dispatched. The instruction can either belong to the correct execution path or be mispeculated.
- *Empty FQ*: the fetch queue is empty and the dispatch slot is wasted. As dispatch slots are tracked for all cores, this case is only considered while there is a running task in the core.
- *ROB/VB*: the ROB or VB is full and the dispatch slot cannot be used.



a) CoW table occupancy.

b) CoW table size.

Figure 5.9: Delayed writebacks and the CoW table.

- *HB*: the HB is full, and has caused the ROB/VB to become full as well, preventing the dispatch slot from being used.

- *Other*: the register file, instruction queue or load-store queue is full.

Let us focus on benchmark *watersp*, which shows a representative trend for the rest of the benchmarks. The bars labeled from A to D represent the ROB architecture with different ROB sizes. For an 8-entry ROB, a lack of space in this structure is the cause for most stalls at the dispatch stage. When its size is doubled, a bottleneck shift into the HB can be observed. Then, a 32-entry ROB makes the dispatch stalls be balanced among all processor structures, while a 64-entry ROB suffices to totally prevent this structure from causing pipeline stalls.

In the VB architecture, represented by the bars labeled from E to H, the same trend is observed. However, a balance of reasons for pipeline stalls is achieved with a smaller VB. As observed, an 8-entry VB is large enough to make other queues (or the register file) become the processor bottleneck in some cases. For 16- and 32-entry VBs, the fraction of wasted dispatch slots due to a full VB decreases with respect to the ROB architecture, while a 64-entry VB completely shifts the bottleneck away.

5.4.3 Impact of Delayed Writebacks

Two experiments have been carried out on top of the baseline 8-core processor to evaluate how delayed writebacks to the HB affect performance of the VB architecture. On one hand, we have measured the occupancy of the CoW table cycle by cycle, and average and maximum values are shown in Figure 5.9a. The height of the bars represents the number of instructions in the HB that have been issued but have not written back their result yet, and thus, occupy an entry in the CoW. Results show that the number of busy entries is variable depending on the benchmark, but the average is low with respect to the number of entries in the HB.

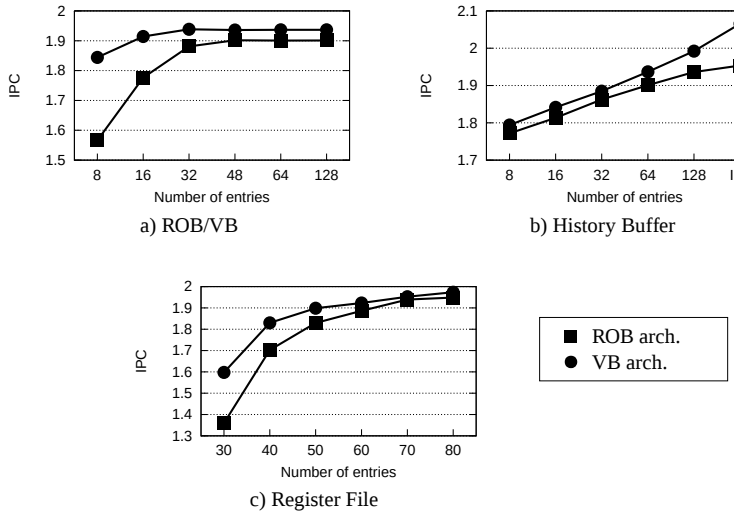


Figure 5.10: Performance scaling for different resource sizes. Each point represents average values for all SPLASH2 benchmarks.

On the other hand, we have tuned the CoW table size by looking for the point where a reduction of the number of entries prevents incomplete instructions from entering the HB and causes pipeline stalls. Figure 5.9b shows average performance for the whole benchmark suite and different table sizes. A 5-entry table makes performance drop by only 2.3% relative to an unbounded CoW table (represented with the dashed line labeled *Inf*), whereas a 15-entry table provides a negligible impact.

5.4.4 Impact of the Resources Size

This section explores the impact on performance of varying the size of the three main processor structures affected by our proposal —ROB/VB, HB, and register file (RF)— focusing on both ROB- and VB-based, sequentially consistent, 8-core processors. Performance values are shown in Figure 5.10, computed as the average of single IPC values obtained from the execution of the SPLASH2 benchmarks.

- **ROB/VB.** Figure 5.10a shows the results obtained when changing the ROB/VB size from 8 to 128 entries, with the remaining parameters matching the baseline configuration. The VB efficacy can be analyzed either from a cost or a performance point of view. On one hand, a 16-entry VB provides similar performance to a 64-entry ROB. This makes the VB a cost-effective solution that sustains performance, while reducing its size by a factor of 4. A smaller VB not only saves hardware devoted to storage, but also reduces the amount of logic (decoders/encoders) required to implement access to a larger structure.

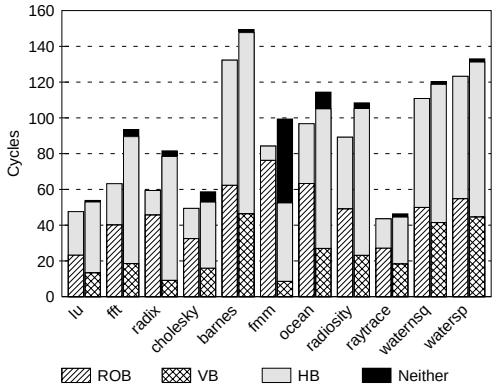


Figure 5.11: Lifetime of instructions classified as per the structure they are placed in (ROB/VB, HB, or neither of them).

On the other hand, VB sizes of 8, 16, and 32 entries outperform their similarly sized ROB-based designs by 17.7%, 7.8%, and 3%, respectively.

- **History Buffer.** We evaluated HB sizes from 8 to 128, including an infinite HB (Figure 5.10b). A ROB-based architecture provides a modest performance boost when the HB size is increased beyond 64 entries. The reason is that the ROB is a restrictive bottleneck, where instructions spend the bulk of their lifetime. The only instruction that can advance to the HB without being completed is a *store*, so the HB only causes a machine stall when a long-latency *store* blocks the head and propagates the stall until the ROB tail. On the other hand, a VB-based system alleviates the ROB bottleneck, and makes the HB a more critical structure. In this case, an increase of the HB size results in nice performance gains. The VB-based processor with a 64-entry HB outperforms its homologous ROB-based design by 7.1%, and behaves similarly to the in-order retirement processor with an infinite HB.

  The effects of addressing the ROB bottleneck can be seen in Figure 5.11. This figure plots the number of cycles that instructions spend on average in the ROB/VB, in the HB, or in neither of them. The case where instructions are still in the pipeline without being either in the ROB/VB or in the HB can only occur in the VB architecture. As observed, the lifetime portion of instructions in the VB decreases with respect to the ROB. In contrast, instructions remain longer in the HB in the VB architecture, causing a higher occupancy of this structure, and thus explaining the observed performance improvements due to increased HB sizes.

- **Physical Register File.** The register file (RF) is a critical resource in a w-way superscalar processor, since it is potentially accessed $2w$ times for reading

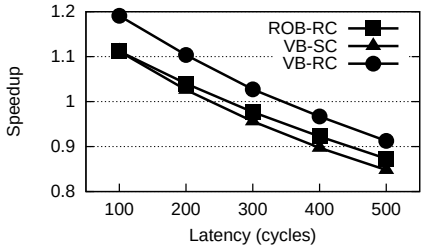


Figure 5.12: Impact of main memory latency.

and w times for writing (assuming instructions with 2 source and 1 destination operand). The register renaming strategy used in the VB architecture provides for the management of the register mapping tables and is able to release unused registers even if younger instructions have not finished execution. A shorter allocation time of physical registers provides two main benefits: on one hand, a higher number of in-flight instructions can be supported; on the other hand, a lower occupancy of the RF enables hardware complexity to be reduced without adversely impacting performance.

Experiments model RF sizes of 40 to 80 physical registers. As shown in Figure 5.10c, a 40-entry RF on a VB-based multiprocessor behaves similarly to a 60-entry RF on a ROB-based design. A difference of 20 physical registers can entail notable hardware and energy overhead due to the high number of read and write ports implemented in this structure. From a performance point of view, the VB architecture improves the ROB-based approach by about 15.3% and 7.7% for RF sizes of 40 and 80, respectively.

5.4.5 Main Memory Latency

Different latencies for main memory have been simulated to test their impact on the performance of the multicore VB architecture. Figure 5.12 shows the performance speedup with respect to the baseline sequentially consistent, 8-core, ROB-based multiprocessor with a memory access time of 200 cycles. The plot includes memory latencies ranging from 100 to 500 cycles both for ROB- and VB-based, relaxed and sequentially consistent machines.

A longer access time makes speedups decrease for the different architectures and consistency models. However, the ROB-based system implementing release consistency is less affected than the VB-based systems. Specifically, the ROB-RC curve touches the VB-SC plot at a short latency, and tends to shorten the gap with the VB-RC curve for higher latencies. When the main memory latency increases, the amount of information created by speculative overlapped computations grows, and it

becomes harder to enforce sequential consistency. This makes any SC implementation perform worse than RC for very large memory access times. However, the VB is especially appealing for systems with low memory access times, where an implementation of sequential consistency performs similarly to relaxed consistency for the ROB.

When observing the interpolated curve for VB-SC between the memory latency values 200 and 300, it can be appreciated that a speedup of 1 is achieved with a memory latency of about 230 cycles (this interpolated value has been confirmed by further experiments using 230 cycles as memory latency). In this case, one can conclude that the VB architecture tolerates higher memory access delays (caused, for example, by the reduction of the cache size), while still sustaining performance of the baseline processor.

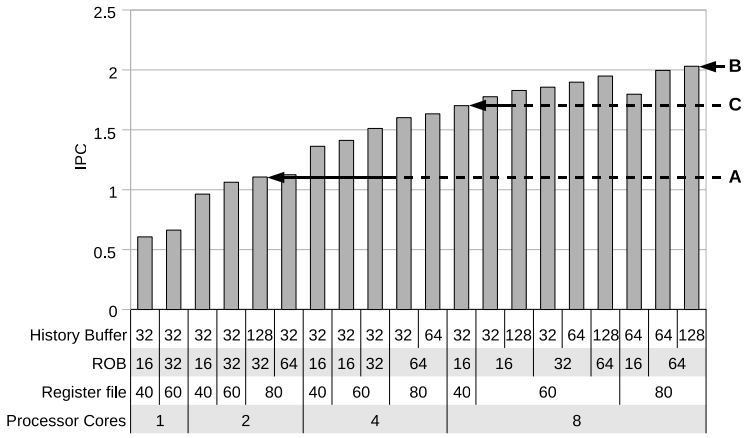
5.5 Hardware Complexity

5.5.1 Size of the Major Processor Components

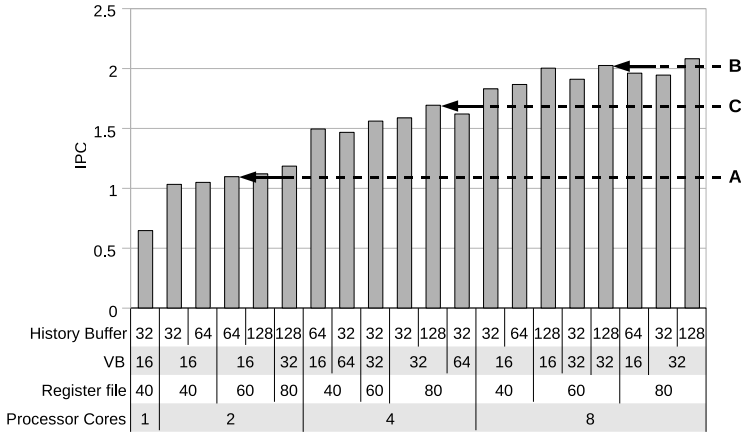
Several combinations of structure sizes have been modeled to compare hardware complexity of the ROB and VB architectures with similar performance levels. To this aim, the HB, the ROB (or VB), and the RF size have been varied between 32 and 128, between 16 and 64, and between 40 and 80, respectively. All these combinations have been evaluated for 1-, 2-, 4-, and 8-core, sequentially consistent processors. Likewise, all considered benchmarks have been run for each configuration. Due to the high number of simulations, only average performance results for the whole benchmark suite are shown, and some of them are filtered with the following criterion.

Twenty performance results are shown in Figure 5.13 both for the ROB (5.13a) and VB (5.13b) architectures. They have been picked out from the totality of experiments by grouping similar performance values and discarding those with higher complexity. Each bar represents the average performance for a specific combination of simulation parameters specified at the bottom of each plot.

The performance levels of interest are labeled as A, B, and C. Each of them is plotted for both architectures for comparison purposes. Performance level A corresponds to an IPC of 1.11. It is achieved by a 2-core, ROB-based processor with 80 physical registers, a 32-entry ROB, and a 128-entry HB, but also by a VB-based processor with 20 registers less, a VB 2 times smaller than the ROB, and a 2 times less complex HB. For an 8-core processor, performance level B shows that the VB architecture provides an average IPC of 2.02 with 20 registers less, and a VB twice as small as for the analogous ROB-based processor. Finally, performance level C (IPC of 1.7) is provided by a VB-based processor that doubles the RF and VB size of a



a) ROB architecture



b) VB architecture

Figure 5.13: Performance for different hardware complexity levels.

ROB-based machine, and also uses an HB 4 times as large, but halves the number of processor cores.

5.5.2 Impact of the Pipeline Width

Different values between 1 and 8 have been tested for the pipeline width (decode, dispatch, issue, and commit/validate bandwidth) for the baseline ROB- and VB-based, 8-core, sequentially consistent architectures. Figure 5.14 shows performance results for the ROB (light gray) and VB (dark gray) systems. Notice that performance is computed globally for all cores, which is the reason why the IPC value can exceed the number of ways.

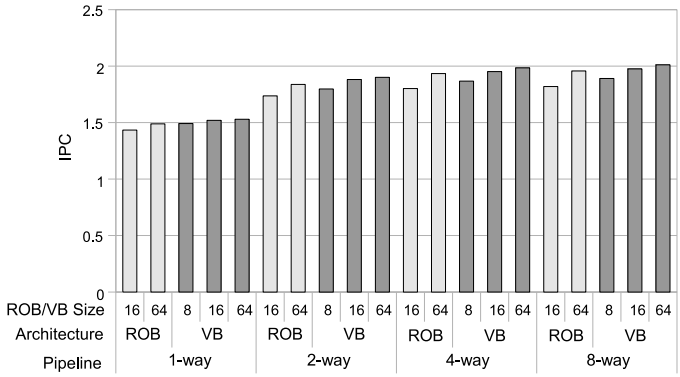


Figure 5.14: Pipeline width.

Results show that the pipeline width especially constrains performance of 1-way processors. In this case, the transition from the ROB into the VB architecture is less beneficial (6% for 16 entries) than for wider pipelines, which sustain a speedup of at least 8.3%. Though 2-way cores clearly outperform 1-way cores, wider pipelines provide a moderate performance boost. The following comparisons are highlighted.

A 2-way, VB-based processor with 16-entry VBs outperforms an 8-way, ROB-based processor with the same number of ROB entries. Likewise, a 4-way, VB-based processor with 64-entry VBs performs better than an 8-way, ROB-based processor with similar ROBs. An increase of the pipeline width involves more instruction cache ports (fetch stage), register file ports (rename stage), and associative search ports in the IQ (issue stage), as well as an increase of complexity of other hardware structures. Thus, it is preferable to transition to the VB architecture rather than introducing more pipeline ways in the above cited cases.

5.6 Summary

In this chapter, the VB multicore architecture has been presented as a sequentially consistent, out-of-order retirement, multiprocessor approach. Sequential consistency is based on speculatively retiring load instructions using a history buffer (HB), while maintaining a non-speculative approach for retiring the rest of instructions. This approach relies on the introduction of a small table (CoW) to support delayed destination operands to be written into the HB.

An extensive evaluation has been carried out dealing with single-thread potential, performance on multicore processors with different number of cores, impact of the resources size, and memory consistency models. Experimental results show that the multicore VB architecture can speed up both relaxed and sequentially consis-

tent in-order retirement in future multiprocessor systems by between 3% and 20%, depending on the ROB size.

Chapter 6

Related Work

In this chapter, some work related with this thesis is discussed. The cited approaches are classified in two sections, depending on whether they address uniprocessor or multiprocessor environments. For both of them, previously proposed out-of-order retirement architectures are briefly described. The summary of the previous work dealing with multiprocessors includes memory consistency implementations.

6.1 Proposals Based on Uniprocessors

Long-latency operations constrain the output rate of the ROB, and thus, microprocessor performance. Many microprocessor mechanisms have been recently proposed dealing with this problem. Some of them try to alleviate this bottleneck by aggressively and carelessly releasing some resources out of program order, using checkpoints to return to a previous valid state on mispeculation. Others release pipeline resources non-speculatively and also out of program order, avoiding checkpoints by carefully taking retirement decisions. And others reduce the impact of long-latency operations by enlarging some processor structures that constrain the instruction window size. In what follows, some proposals related with this thesis and based on uniprocessors are classified into these three categories and summarized.

6.1.1 Speculative Out-of-Order Retirement with Checkpoints

Proposals in this category permit the retirement of instructions in a speculative mode when a long-latency operation blocks the ROB head. These solutions introduce specific hardware to periodically checkpoint the architectural state and guarantee correct execution. When a misprediction occurs, the processor rolls back to the last checkpoint, discarding all intermediate computations.

In [49], Mutlu et al. propose the run-ahead architecture. 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 among dependent instructions to enable them 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 normal mode, discarding previous results. The run-ahead execution provides useful prefetching requests (both for instructions and data), as well as effective train for branch predictors.

In [6], 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 misprediction, the processor rolls back to the checkpoint.

In [8], Cristal et al. propose to replace the ROB with a set of checkpoints at specific instructions. This mechanism uses a CAM structure for register mapping purposes, which is also in charge of freeing physical registers. Stores 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., instructions

dependent on a load) are moved from the instruction queue to a secondary buffer, freeing resources that can be used by other instructions. These instructions are re-inserted into the instruction queue when the instruction they are dependent on has completed (e.g., the load data has been fetched). This problem has also been tackled by Akkary et al. in [7].

In [50], Martínez et al. propose an in-order retirement mechanism which identifies irreversible instructions for early release of resources. Unlike the VB microarchitecture, this proposal retires instructions in order from the processor pipeline, and, like the works discussed above, checkpoints are required to eventually roll back to a previous machine state.

6.1.2 Non-Speculative Out-of-Order Retirement Without Checkpoints

In [9], Bell and Lipasti present a checkpoint-free approach. A ROB is still used, into which instructions are inserted in order, but from which they are extracted out of program order. To this end, a specific number of entries is scanned at the ROB head in the commit stage, and the empty slots caused by instructions withdrawn out of order are removed by collapsing the remaining entries. Unfortunately, this proposal is unsuitable for large ROB sizes, and resources are handled like in a typical ROB-based processor, without any focus on improving resources usage.

6.1.3 Enlargement of the Major Processor Structures

Some proposals alleviate the performance degradation caused by ROB stalls by enlarging the major microprocessor structures or managing them more efficiently. In the case of non-scalable structures, such as the load-store queue, the register file, or the instruction queue, alternative designs allow an increase in number of entries compensated by a limited functionality, which still provides a global performance gain.

In [51], Raasch et al. propose a segmented instruction queue (IQ) for dynamically scheduled processors. Though a larger IQ increases the exposed instruction level parallelism (ILP), it also slows down its access time, and might impact on the clock speed. The authors of this work propose to divide the IQ into small segments among which instructions are promoted until they reach a small issue buffer, from which they can be actually scheduled into the corresponding functional unit. This allows the global IQ complexity to be reduced for a given total number of entries, or performance to be increased for a specific complexity and higher number of entries.

In [52], Balasubramonian et al. deal with the complexity of the register file (RF). While a large multiported RF improves ILP, the same side effects occur as in the IQ, namely that the access latency rises, and the system clock might be forced to slow down. First, the authors reduce the global number of RF entries by designing a two-

level RF. Physical registers are assigned to each level by being hierarchically divided into those with active consumers and those waiting for specific conditions. Second, a banked organization of the RF is proposed to reduce the port requirements. Similarly to banked data caches, the RF is able to provide high read/write bandwidth while still being minimally ported.

Finally, in [53], Park et al. overcome the scarce scalability of the load-store queue (LSQ), motivated by an increasing pressure on this structure in terms of capacity and search bandwidth. As an associative structure, the LSQ presents similar problems to the IQ when its size tries to be straightforwardly increased. First, the authors of this work present two techniques to reduce the required search bandwidth. On one hand, they introduce a store-load aliasing predictor. On the other, speculatively issued loads are stored in a separate load buffer, so the store-load order violation detection is moved off the LSQ. Second, the authors propose a segmentation of the LSQ to scale its size. In the resulting design, the LSQ is divided into segments connected in a chain.

Since the previous proposals focus on structures other than the ROB and the renaming tables, they are orthogonal with the VB architecture proposed in this thesis. With the joint implementation of these techniques, a very aggressive processor could support a very large instruction window by alleviating the bottlenecks imposed by the traditional management of the main pipeline structures in conventional superscalar processors.

6.2 Proposals Based on Multiprocessors

Regarding multiprocessor environments, two mainstream research topics have been merged in this thesis. On one hand, sequential consistency (SC) implementations have been studied to investigate their compatibility with out-of-order retirement in multiprocessors. On the other hand, some proposals dealing with out-of-order retirement of instructions in multiprocessors have been examined.

6.2.1 Sequential Consistency Implementations

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 [47], and *speculative retirement of loads* is discussed in [29]; both features have been detailed in Section 5.1 and are implemented in the baseline multicore architecture used for simulations due to their low complexity and effectiveness.

More sophisticated SC implementations can be found in the literature. In [54], 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 mispeculation, SC++ performs a burst of cache writes to roll back to a previous valid state. In contrast, this work avoids to impose these complexities in the cache hierarchy.

SC++lite [55] 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 mispeculation, SC++lite recovers at a slower rate than SC++, but consistency mispeculations are rare enough to afford it.

Finally, BulkSC [56] 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.

6.2.2 Out-of-Order Retirement in Multiprocessors

A number of key papers have addressed the subject of out-of-order retirement in multiprocessors. In the Cherry-MP architecture [57], 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 mispeculations. 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 Multiprocessor (or KIMP) [58] was also proposed as an 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.

6.3 Summary

In this chapter, some previous work pursuing similar aims to the VB architecture has been presented. They range from alternative ways to retire instructions out of order from the processor pipeline, to memory consistency implementation in multi-core environments. The novelties of the work proposed in this thesis with respect to the cited works reside in i) the design and evaluation of a low-cost, non-speculative, out-of-order retirement architecture, ii) an evaluation and integration of out-of-order retirement in multithreaded environments, and iii) the design and evaluation of a checkpoint-free, out-of-order retirement multiprocessor design.

Chapter 7

Conclusions

The Validation Buffer (VB) architecture has been proposed in this dissertation as an alternative design where instructions are retired out of program order from the processor pipeline. This technique has been implemented and evaluated on top of the three major processing models available in the microprocessor market roughly for the last two decades, namely, superscalar, multithreaded, and multicore processors. In this chapter, the main contributions on each of these fields are summarized, followed by a discussion about future working directions and an enumeration of the scientific publications related with this thesis.

7.1 Contributions

In Chapter 3, the baseline VB architecture has been proposed for superscalar processor. First, the potential of this proposal has been evaluated by assuming the major microprocessor structures unbounded, namely the register file, the instruction queue, and the load-store queue, showing that current applications, and especially floating-point benchmarks, can be considerably benefited from out-of-order retirement. Then, a realistic VB-based machine has been modeled and compared against two ROB-based proposals, one retiring instructions in order and the other out of order. Results show that a 32-entry VB provides performance similar to a 256-entry ROB. These performance benefits are accompanied by a reduction in the occupancy of the remaining structures, which has further implications in terms of hardware cost or power consumption.

In a study trading off complexity and performance, results show that a given performance level is achieved in the VB architecture with simpler hardware cost for the major microprocessor structures than for in-order retirement processor. Moreover, the pipeline width can be narrowed in a VB-based processor, drastically reducing the cost of a superscalar pipeline, while sustaining performance of a ROB-based design with the same structure sizes. This might be of special interest in those architectures where complexity is a crucial issue, such as power-aware or embedded systems.

In Chapter 4, the VB architecture has been extended for multithreaded architectures, including coarse-grain, fine-grain and simultaneous multithreading. The behavior of different thread selection policies at the fetch stage has been explored, showing that out-of-order retirement and both multithreading and fetch policies are techniques that orthogonally contribute to increase the issue bandwidth utilization and processor performance.

Performance 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 of the hardware threads than a simultaneous multithreaded ROB-based processor; iii) benefits of fetch policies are orthogonal to the ones provided by the VB. These contributions justify the viability and cost-effectiveness of an out-of-order retirement multithreaded processor microarchitecture.

Finally, Chapter 5 has shown the extension of the VB architecture for multicore/multiprocessor environments, where instructions are retired out of order in different processing nodes, while sequential consistency is still enforced. This strict memory model is enforced by using the *speculative retirement of loads* technique with an additional hardware structure, called History Buffer (HB). Out-of-order retirement in multiprocessors is implemented by focusing on three centralized 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 register. 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 memory hierarchy or interconnects, the VB multiprocessor architecture remains highly scalable with the number of processors.

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 HB 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.

An additional contribution of this thesis is the development of the Multi2Sim simulation framework, presented in Chapter 2. This tool integrates important features of existing simulators and extends them to provide additional functionality. The main characteristics of Multi2Sim are a model of superscalar, multithreaded and multicore processors, a cache coherence protocol, interconnection networks, and the extensions to support the VB architecture with different memory consistency models. The Multi2Sim simulation framework has been made publicly available and researchers are encouraged to contribute to its further development through SVN ($subversion$) access and mailing lists.

7.2 Future Work

More research is planned as for future work on the topic of out-of-order retirement of instructions. Specifically, we intend to design a superscalar processor pipeline where neither a reorder buffer nor an alternative FIFO structure (like the validation buffer) is present. In the VB architecture proposed in this thesis, the characteristic head-of-line blocking effect caused by the ROB is largely alleviated. However, instructions are still extracted in program order from the VB (albeit with less restrictive conditions), and this may still be a cause for pipeline stalls, especially for very small VBs. The key observation that allows the removal of this bottleneck is that the validation of correctly speculated instructions, as well as the discard of mispeculated ones, is performed in form of instruction bursts embraced by epoch initiators. Thus, the only information about sequentiality that needs to be kept track of is the program order of epoch initiators.

In this proposal, an alternative register renaming strategy needs to be designed again to handle register reclamation by itself, without the information about register

mappings that was stored both in the ROB and the VB. Specifically, a hybrid renaming scheme has been already proposed and published [59], which combines a content addressable memory (CAM) and a random access memory (RAM) in order to perform efficient checkpoints of the aliasing table at specific execution points, recover the machine state in constant time after mispeculation, and provide the pipeline with register mappings at full decode bandwidth, while even reducing power consumption and increasing performance of traditional register renaming.

The removal of a FIFO structure holding instructions in flight opens new opportunities when pipeline resources are shared by instructions among which no sequential order needs to be enforced, such as those that belong to different tasks. This is the case of multithreaded processors, which decode instructions from different threads that share some processor structures. As discussed in [60], sharing the ROB among threads is preferable when a multithreaded processor is loaded with a low number of tasks, because a single task can compete for all its entries. However, a shared FIFO structure has other pernicious effects when highly loaded, such as inter-thread blocking, thread starvation, and holes at recovery. All these effects automatically vanish in an architecture without ROB or VB, and the benefits of a shared resource are kept when the proposed non-FIFO (i.e., associative) structure storing epoch initiators is shared among threads.

New challenges are also posed by this proposal when introduced in a multiprocessor environment. Existing techniques to enforce sequential consistency, as cited in this thesis, may not be straightforwardly applicable to architectures that alter the order in which instructions leave the processor pipeline. Even speculative retirement of loads using a history buffer (HB), as adapted in this thesis for the VB architecture, is not suitable in this case, since the income of instructions into the HB needs to be provided in sequential order by some other structure in the rest of the pipeline. Thus, it becomes a research opportunity to design an efficient implementation of SC that is again transparent to the memory hierarchy and interconnects in a system where processor pipelines lack a ROB or VB.

In summary, out-of-order retirement of instructions is still a hot topic that can increase processor performance in state-of-the-art systems, by removing head-of-line blocking effects on FIFO structures and increasing the number of instructions in flight.

7.3 Publications Related with This Work

The following list enumerates the papers related with this dissertation that have been published, or are under review process, in specialized conferences or journals.

- R. Ubal, J. Sahuquillo, S. Petit, P. López, and J. Duato, "The Validation Buffer Microarchitecture for Multithreaded Processors", *ACACES Summer School, L'Aquila (Italy)*, Jul. 2007.

- 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, Brasov (Romania)*, Sept. 2007.

- R. Ubal, J. Sahuquillo, S. Petit, and P. López, "A Simulation Framework to Evaluate Multicore-Multithreaded Processors", in *Proc. of the 19th International Symposium on Computer Architecture and High Performance Computing, Gramado (Brazil)*, Oct. 2007.

- R. Ubal, S. Petit, J. Sahuquillo, P. López, and J. Duato, "A First Approach to Non-Speculative Out-of-Order Instructions Retirement", *XVIII Jornadas de Paralelismo, Zaragoza (Spain)*, Sept. 2007.

- R. Ubal, J. Sahuquillo, S. Petit, and P. López, "The Impact of Out-of-Order Commit in Coarse-Grain, Fine-Grain and Simultaneous Multithreaded Architectures", in *Proc. of the 22nd International Parallel and Distributed Processing Symposium, Miami (Florida, USA)*, Apr. 2008.

- R. Ubal, J. Sahuquillo, S. Petit, and P. López, "An Experimental Framework to Simulate Sequential and Parallel Workloads in Multicore-Multithreaded Processors", *XIX Jornadas de Paralelismo, Castellón (Spain)*, Sept. 2008.

- S. Petit, R. Ubal, J. Sahuquillo, P. López, and J. Duato, "An Efficient Low-Complexity Alternative to the ROB for Out-of-Order Retirement of Instructions", in *Proc. of the 12th Euromicro Conference on Digital System Design, Patras (Greece)*, Aug. 2009.

- S. Petit, J. Sahuquillo, P. López, R. Ubal, and J. Duato, "A Complexity-Effective Out-of-Order Retirement Microarchitecture", *IEEE Transactions on Computers, Vol. 58 No. 12*, Dec. 2009.

- R. Ubal, J. Sahuquillo, S. Petit, P. López, and D. Kaeli, "A Sequentially Consistent Multiprocessor Architecture for Out-of-Order Retirement of Instructions", *IEEE Transactions on Parallel and Distributed Systems*, submitted.

All published works listed above are exclusively related with this thesis, and none of them are or will be used as supporting material for other theses. The specific contributions of the PhD candidate reside mostly in the implementation of the proposed techniques (including the complete code of the Multi2Sim simulation framework, as

well as the necessary modifications for the VB architecture), the setup and execution of most simulation experiments, and the writing of the paper drafts and technical reports describing the work. Along these processes, the coauthors have repeatedly provided useful hints and advices, which the PhD candidate has then applied to make the work evolve into its final version.

References

[1] J. E. Smith and G. Sohi. The Microarchitecture of Superscalar Processors. *Proc. of the IEE*, 83(2), Dec. 1995.

[2] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, Apr. 1996.

[3] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyler, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal. Q1*, Feb. 2001.

[4] J. E. Smith and A. R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proc. of the 12th Int'l Symposium on Computer Architecture*, June 1985.

[5] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proc. of the 24th Int'l Symposium on Computer Architecture*, June 1997.

[6] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martínez. Checkpointed Early Load Retirement. In *Proc. of the Int'l Symposium on High Performance Architecture*, Feb. 2005.

[7] 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.

[8] 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.

[9] 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.

[10] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: a Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2), 2004.

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

[12] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium

Processor. *IEEE Micro*, 25(2), 2005.

[13] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), Mar. 1999.

[14] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. Power4 System Microarchitecture, Technical white paper. *IBM Server Group*, Oct. 2001.

[15] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, 1997.

[16] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. *Univ. of Virginia Dept. of Computer Science Technical Report CS-2003-05*, 2003.

[17] D. Madon, E. Sanchez, and S. Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *European Conference on Parallel Processing*, 1999.

[18] J. Sharkey. M-Sim: A Flexible, Multithreaded Architectural Simulation Environment. *Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton*, 2005.

[19] D. M. Tullsen. Simulation and Modeling of a Simultaneous Multithreading Processor. *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[20] M. Moudgill, P. Bose, and J. Moreno. Validation of Turandot, a Fast Processor Model for Microarchitecture Exploration. *IEEE Int'l Performance, Computing, and Communications Conference*, 1999.

[21] M. Moudgill, J. Wellman, and J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, pages 15–25, 1999.

[22] D. Brooks, P. Bose, V. Srinivasan, M. Gschwind, P. Emma, and M. Rosenfield. Microarchitecture-Level Power-Performance Analysis: The PowerTimer Approach. *IBM J. Research and Development*, 47(5/6), 2003.

[23] B. Lee and D. Brooks. Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency. *Workshop on Complexity Effective Design*, 2005.

[24] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2), 2002.

[25] M. R. Marty, B. Beckmann, L. Yen, A. R. Alameldeen, M. Xu, and K. Moore. GEMS: Multifacet's General Execution-driven Multiprocessor Simulator. *Int'l Symposium on Computer Architecture*, 2006.

[26] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-Oriented Full-System Simulation Using M5. *6th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Feb. 2003.

[27] The Multi2Sim Simulation Framework. *http://www.multi2sim.org*.

[28] T. Y. Yeh and Y. N. Patt. A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History. In *Proc. of the 20th Int'l Symposium on Computer Architecture*, 1993.

[29] 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.

[30] Standard Performance Evaluation Corporation. *http://www.spec.org/cpu2000/*.

[31] 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.

[32] 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.

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

[34] J. H. Edmondson, P. Rubinfeld, and R. Preston. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, 15(2), 1995.

[35] Free Software Foundation, *http://www.gnu.org/software/gcc/onlinedocs/*, GCC Online Documentation. 2006.

[36] S. E. Raasch and S. K. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proc. of the 12th Int'l Conference on Parallel Architectures and Compilation Techniques*, 2003.

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

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

[39] F. J. Cazorla, A. Ramírez, M. Valero, and E. Fernández. Dynamically Controlled Resource Allocation in SMT Processors. In *Proc. of the 37th Int'l Symposium on Microarchitecture*, 2004.

[40] J. Sharkey, D. Balkan, and D. Ponomarev. Adaptive Reorder Buffers for SMT Processors. In *Proc. of the 15 Int'l Conference on Parallel Architectures and Compilation Techniques*, 2006.

[41] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Paired ROBs: A Cost-Effective

Reorder Buffer Sharing Strategy for SMT Processors. In *Proc. of the Euro-Par Conference*, Aug. 2009.

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

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

[44] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Madison, WI, USA, 1993.

[45] 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. of the 17th Int'l Symposium on Computer Architecture*, May 1990.

[46] 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.

[47] 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.

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

[49] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Window for Out-of-Order Processors. In *Proc. of the 9th Int'l Symposium on High Performance Architecture*, Feb. 2003.

[50] J. F. Martínez, J. Renau, MC. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Processors. In *Proc. of the 35th Int'l Symposium on Microarchitecture*, Nov. 2002.

[51] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A Scalable Instruction Queue Design Using Dependence Chains. In *Proc. of the 29th Int'l Symposium on Computer Architecture*, May 2002.

[52] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the Complexity of the Register File in Dynamic Superscalar Processors. In *Proc. of the 34th Int'l Symposium on Microarchitecture*, Dec. 2001.

[53] I. Park, C. L. Ooi, and T. N. Vijaykumar. Reducing Design Complexity of the Load-Store Queue. In *Proc. of the 36th Int'l Symposium on Microarchitecture*, Dec. 2003.

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] S. Petit, R. Ubal, J. Sahuquillo, and P. López. A Power-Aware Hybrid RAM-CAM Renaming Mechanism for Fast Recovery. In *Proc. of the 27th Int'l Conference on Computer Design*, Oct. 2009.

[60] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Paired ROBs: a Cost-Effective Reorder Buffer Sharing Strategy for SMT Processors. In *Proc. of the 2009 Euro-Par Conference*, Aug. 2009.