PREDICATED EXECUTION AND REGISTER WINDOWS FOR OUT-OF-ORDER PROCESSORS

Eduardo Quiñones

Barcelona, May 2008

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy / Doctor per la UPC

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Y que has aprendido en el camino?
Que importa el viaje, no el destino.

Patrice Chaplin

Acknowledgements

Son muchas las personas que han contribuido, en mayor o menor medida, a hacer posible este documento. Los principales responsables son mis directores de tesis Antonio González, quien me dio la oportunidad de sumergirme en el mundo de la investigación, y Joan Manel Parcerisa, con quien he pasado largas horas discutiendo. Ellos me han enseñado el significado de investigar, de plantear nuevas preguntas y ser capaz de dar respuestas.

También son muchas las personas que han contribuido a alargar esta tesis un poco más. Quiero hacer una mención especial a los distinguidos miembros que han pasado por la sala c6-e208. Con ellos he compartido largas tertulias de café. Eso sí, un tema siempre ha estado vetado, la arquitectura de computadores.

Sin embargo, estos años de doctorado no hubieran sido posible sin el apoyo incondicional de mis padres, que siempre me han animado a seguir adelante, a buscar mi futuro personal y profesional.

Finalmente, quiero agradecer muy especialmente a la persona con quien he vivido durante estos casi cinco años que ha durado mi tesis, y con la que espero vivir durante muchos años más, eso sí, como doctor. Ella ha hecho que cada comento díficil fuera fácil, que cada día malo fuera bueno, y que cada día bueno fuera increíble! Gracies Roser.

El objetivo de un doctorado, no debe ser únicamente el título de doctor, sino también el enriquezimiento personal. En un viaje, lo más importante no es el destino, sino disfrutar del camino. Me siento una persona afortunada, he tenido los mejores compañeros de viaje. Gracias a todos.

Abstract

ISA extensions are a very powerful approach to implement new hardware techniques that require or benefit from compiler support: decisions made at compile time can be complemented at runtime, achieveing a synergistic effect between the compiler and the processor. This thesis is focused on two ISA extensions: *predicate execution* and *register windows*. Predicate execution is exploited by the *if-conversion* compiler technique. If-conversion removes control dependences by transforming them to data dependences, which helps to exploit ILP beyond a single basic-block. Register windows help to reduce the amount of loads and stores required to save and restore registers across procedure calls by storing multiple contexts into a large architectural register file.

In-order processors specially benefit from using both ISA extensions to overcome the limitations that control dependences and memory hierarchy impose on static scheduling. Predicate execution allows to move control dependence instructions past branches. Register windows reduce the amount of memory operations across procedure calls. Although if-conversion and register windows techniques have not been exclusivelly developed for in-order processors, their use for out-of-order processors has been studied very little. In this thesis we show that the use of if-conversion and register windows introduce new performance opportunities and new challenges to face in out-of-order processors.

The use of if-conversion in out-of-order processors helps to eliminate hard-to-predict branches, alleviating the severe performance penalties caused by branch mispredictions. However, the removal of some conditional branches by if-conversion may adversely affect the predictability of the remaining branches, because it may reduce the amount of correlation information available to the branch predictor. Moreover, predicate execution in out-of-order processors has to deal with two performance issues. First, multiple definitions of the same logical register can be merged into a single control flow, where each definition is guarded with a different predicate. Second, instructions whose guarding predicate evaluates to false consume unnecesary resources. This thesis proposes a branch prediction scheme based on predicate prediction that solves the three problems mentioned above. This scheme, which is built on top of a predicated ISA that implement a compare-and-branch model such as the one considered in this thesis, has two advantatges: First, the branch accuracy is improved because the correlation information is not lost after if-conversion and the mechanism we propose permits using the computed value of the branch predicate when available, achieving 100% of accurary. Second it avoids the predicate out-of-order execution problems.

Regarding register windows, we propose a mechanism that reduces physical register requirements of an out-of-order processor to the bare minimum with almost no performance loss. The mechanism is based on identifying which architectural registers are in use by current in-flight in-

structions. The registers which are not in use, i.e. there is no in-flight instruction that references them, can be early released.

In this thesis we propose a very efficient and low-cost hardware implementation of predicate execution and register windows that provide important benefits to out-of-order processors.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Background and Motivation

In 1966 IBM released the *360/91* processor [2], the first pipelined processor with five execution units that allowed the simultaneous process of multiple instructions. With the introduction of instruction pipelining, i.e. multiple instructions in different stages can overlap their execution, *instruction-level parallelism* (ILP) became a crucial research topic to improve performance.

The maximum benefit of pipelining is obtained when all their stages are in use. This is achieved by finding sequences of independent instructions that can overlap their execution. However, if a dependence occurs, i.e. the execution of an instruction depends on the outcome of another instruction that has not been computed yet, the pipeline must stall. Stalls may occur also due to insufficient resources. Therefore, ILP is limited by dependences and available resources (data and control hazards and structural hazards). For the past decades, many of the computer architecture research has focused on mitigate such negative effects.

One of the most constraining type of dependences is *data dependence*. An instruction is data dependent if there is another instruction that generates a result that is required by its computation. Note that data dependences create a producer-consumer scheme that serialize the execution of a program.

Probably, the most well-known and studied technique to mitigate data dependence penalties is the reorganization of the program code [70,71], which takes into account dependence analysis and available resources. To expoit ILP, it is required to determine which instructions can be executed in parallel. If two instructions are independent and the pipeline has sufficient resources, they can be executed in any relative order between them or simultaneously without causing stalls. However, if two instructions are data dependent, they must be executed in program order. In order to avoid pipeline stalls, consumer instructions are separated from their producer instructions by a distance in clock cycles equal to the execution latency of the producer. Thereby, when a consumer instruction executes, their required source operands are already computed and the pipeline does not stall. The process of rearranging the code to minize the performance impact due to data dependences and resources consumption, without modifying the semantic of the program, is called *instruction scheduling*. Instruction scheduling can be performed statically at compile-time or dynamically at runtime. *In-order* processors exclusivelly rely on static schedulers [27,51], while *out-of-order* processors rely on both static and dynamic schedulers [26,38,74].

There are many factors that can limit the effectiveness of the ILP extracted by schedulers, but

two of them are especially important: *control dependences* and *memory hierarchy*.

A control dependence appears when the execution of an instruction depends on a branch outcome. Every instruction, except those that belong to the first basic-block (a straight-line code sequence with only one exit branch in it) of a program, is control dependent on a set of branches, that determine the correct program order execution. Hence, these control dependences must be enforced to preserve the program order, which imposes a very restrictive scheduling constraint: control dependent instructions cannot be moved past branches they depend on. This drastically limits the scope where ILP can be extracted, reducing the scope of work of a scheduler inside a basic block, which tends to be quite small. In fact, branches are very frequent. For instance, in a set of Itanium benchmarks from the SpecCPU2000 suite, the 14% of instructions in integer benchmarks are branches, and 11% of instructions for floating-point benchmarks are branches. This means that, on average, only between seven and nine instructions are executed between a pair of branches in case of integer and floating point benchmarks respectively. Moreover, since these instructions are probably dependent between one another, the amount of exposed ILP inside a basic-block is likely to be less than its average size.

The memory hierarchy provides the illusion of having a fast and huge memory. It is an economical solution to the desire program request of unlimited amounts of fast memory, and the fact that fast memory is expensive. It is formed by different levels of memory, with the goal to provide a memory system with the size of the largest level (disk) and a speed almost as fast as the fastest level. The register file is the first level of the hierarchy. It is the fastest memory level perfectly integrated into the pipeline data path with a fixed latency. Multilevel caches, main memory and disks form the other levels of the memory hierarchy. In theses levels, the memory latency is variable because it depends on which level the required datum actually resides. This constraints the scheduling of load memory instructions. On one hand, when a long-latency load, i.e. a load that requires access to main memory, is scheduled close to their dependent instructions, the pipeline may be stalled for hundred of cycles. On the other hand, when a short-latency load, i.e. a load whose load-value lays on cache memory, is scheduled far from its dependent instructions, its destination register increases unnecessarily its life-range.

Regarding control dependences, branch prediction [34, 50, 56, 68, 72, 73] is probably the most common hardware-based techniques to dynamically remove control dependences. Dynamic schedulers are specially benefited, because the use of branch prediction in conjuntion with speculative execution [29, 69] allow to expose instructions from multiple basic-blocks at the same time. Compile-based techniques are also very useful, especially when the behaviour of branches are known at compile-time, e.g. in scientific codes. Techniques such as loop unrolling [81] or software pipelining [42, 81] statically schedule instructions from different loop iterations; while trace scheduling [20] handles instructions from different basic-blocks, all of them belonging to the path with the highest execution frequency.

Focusing on memory hierarchy, many hardware cache designs have been developed to alleviate the increasing latency gap between CPU and main memory: non-blocking caches [41], multilevel caches [36, 79], victim caches [37], trace caches [63], etc. Other hardware techniques take advantatge of store-load memory dependence sequences, where the store value is directly forwarded to the load destination register. This technique is called *store-to-load value forwarding* [38, 53]. From the compile point of view, many loop-level transformations have been proposed [1, 43, 82]

to improve the cache locality of programs, i.e. to reduce the load latency.

As explained above, the negative effects of control dependences and memory hierarchy on schedulers can be mitigated by using either compile-based or hardware-based techniques. However, there is another set of techniques that combines the benefits by means of *Instruction set architecture* (ISA) extensions. *Predicate execution* and *register windows* are two well-known ISA extensions that mitigate the negative effects of control dependences and memory hierarchy: the former allows to remove control dependences by using *if-conversion*; the latter reduces the amount of memory operations across procedure calls by mapping procedure contexts to a large architected register file.

Although these two ISA extensions have not been developed exclusivelly for in-order processors, their use in out-of-order processors has been almost ignored. We claim that the application of these ISA extensions in out-of-order processors introduce new performance opportunities and new challenges to overcome: the former may improve branch prediction accuracy; the latter may reduce the physical register requirements, both being essencial factors in out-of-order execution efficiency.

This thesis focuses on the application of these two ISA extensions, if-conversion in conjuntion with predicate execution and register windows, to out-of-order processors. We propose novel hardware mechanisms to support predication and register windows that improve performance in out-of-order processors, bringing together the benefits of compiler and hardware technology. In the following sections, these two ISA extensions are explained in detail.

1.2 ISA Extensions: If-Conversion and Register Windows

During the compilation process, compilers extract plenty of information about the program code. Much of this information is lost during the code generation process, and it becomes not accessible to the processor at runtime. However, on the other side, there is some information that is only available at runtime, e.g. control dependences and memory latencies.

Predicate execution and register windows allow the compilers to pass valuable information to the microarchitecture to implement powerful techniques.

1.2.1 If-Conversion and Predicate Execution

Predicate execution or predication [6, 28, 31] allows an instruction to be guarded with a boolean operand called *predicate*. The guarding predicate value decides whether the instruction is executed or converted into a no-operation. If the predicate evaluates to *true* the processor state is modified with the outcome of the instruction, whereas if it evaluates to *false* the processor state is not modified.

If-Conversion [6, 46] is a compiler technique that takes advantatge of predicate execution. It removes branches by transforming control dependences into data dependences and exposing more ILP to the scheduler. In other words, multiple control flow paths are collapsed into a single bigger basic block with a potentially higher ILP to exploit, since removed control dependences are rescheduled based only on its transformed data dependence. If-conversion is especially beneficial

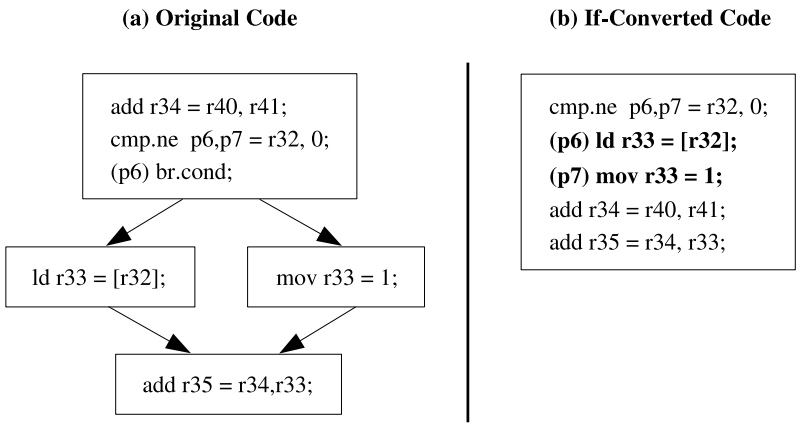

Figure 1.1: **(a) Original if-sentence with multiple control flow paths. (b) If-converted code with one single collapsed control flow path.**

when the behavior of the branch is difficult to predict, such as *if-then-else* statements. Although this thesis focuses on if-conversion, predication has a number of other possible uses [31].

Figure 1.1 illustrates an if-conversion transformation. In Figure 1.1(a) the *ld* instruction cannot be scheduled above its control dependent branch. In Figure 1.1(b) , after applying if-conversion, the branch has been removed and the *ld* can be re-scheduled based only on the *p6* data dependence. Notice that the resulting basic block in (b) is bigger than previous not collapsed basic blocks in (a).

By using predicate execution, the code has fewer branches, larger basic-blocks and fewer constraints caused by control dependences, which results in an important advantatge: the compiler has more instructions to extract parallelism from, which allows it to produce a better and more parallel schedule.

Predicate Execution on Out-of-Order Processors

Dynamic branch prediction in conjunction with speculative execution has proved an effective technique to overcome control dependences in out-of-order processors. It combines three key ideas: branch prediction for breaking the control dependence; speculative execution to allow the execution of control dependent instructions before the branch is solved; and dynamic scheduling to deal with instructions from different basic blocks. However, branch mispredictions may result in severe performance penalties that tend to grow with larger instruction windows and deeper pipelines.

If-conversion may alleviate the severe performance degradation caused by branch misprediction, by removing the hard-to-predict branches [10, 44, 45, 75]. However, when applying the if-conversion technique to an out-of-order processor, the use of predicate execution has to deal with two problems. First, multiple definitions of the same logical register can be merged into a single control flow, where each definition is guarded with a different predicate. At runtime, when the logical register is renamed, every guarded definition allocates a different physical register, and the correct register mappings remains ambigous until all the predicates are resolved, stalling the rename stage if an instruction wants to consume it. Second, instructions whose predicate evaluates

to false consume unnecesary processor resources such as physical registers, issue queue entries and/or functional units, and can potentially degrade performance.

Moreover, the removal of some conditional branches by if-conversion may adversely affect the predictability of other remaining branches [7], because it may reduce the amount of correlation information available to branch predictors. As a consequence, the remaining branches may become harder to predict, since they may have little or no correlation among them.

1.2.2 Register Windows

Register windows is an architectural technique [16,32,62] that helps to reduce the amount of loads and stores required to save and restore registers across procedure calls, by keeping multiple procedure contexts in the register file. This technique uses a dynamic renaming mechanism that allows to assign each procedure context to a different set of contiguous architected registers (called register window) independently of the static register assignment. Moreover, by overlapping register windows, parameter passing does not even require register copies. The register windows mechanism is capable of handling a large register file, so most implementations exploit this ability to get the extra benefit of a large pool of registers that enable compilers to extract more ILP.

When a procedure is called, its local context is mapped into a set of consecutive new architected registers, called register window. Through a simple runtime mechanism, the compile-defined local variables are renamed to its corresponding register name, based on the register window location inside the architectural register file. If there are not enough registers for allocating new procedure contexts, i.e. context overflow, local variables from previous caller procedures are automatically saved to memory and their associated registers are freed for new procedures. This operation is called *spill*. When these saved procedure contexts are needed, i.e. context underflow, they are restored to the register file without any program intervention. This operation is called *fill*.

Notice that, although register windows reduces the required memory references that take part into the procedure call-interface, there are still some memory transfers that automatically occur in case of context overflow and underflow. However, contrary to non-register window schemes, register window hardware support can take advantatge of unused memory bandwidth to dynamically issue the required memory transfer [51], overlapping it with useful program work.

Figure 1.2 shows an example of two procedure contexts mapped to the architectural register file. Registers from the caller procedure context are not required to be saved to memory, since they are kept in the register file. The *r3* compile-defined register in the callee context is dynamically translated to its corresponding architectural windowed register by simply adding the base pointer of its register window.

Register Windows in Out-of-Order Processors

The effectiveness of register windows technique depends on the size of the architectural register file because the more register it has, the less number of spills and fills are required [61]. In other words, there is more space to hold multiple procedure contexts at the same time without sending them to memory.

In an out-of-order processor, the size of the architectural register file determines the minimum size of the map table. Hence, the effectiveness of the register windows depends on the size of

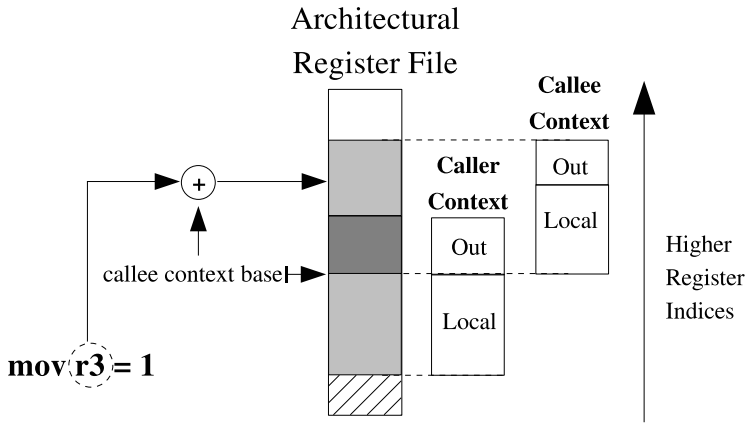

Figure 1.2: **Two procedure contexts are mapped at the same time into the architectural register file. Compile-defined *r3* register from the callee context is dynamically renamed to its corresponding windowed register.**

---

the map table, which in turn determines the minimum number of physical registers. Moreover, in order to extract high levels of parallelism, out-of-order processors require many more physical registers than architected ones, to store the uncomitted values of a large number of instructions in-flight [19]. Therefore, an out-of-order processor with register windows would require a large number of physical registers because of a twofold reason: to hold multiple contexts and to keep the uncommitted values. Unfortunately, the size of the physical register file has a strong impact on its access time [19], which may stay in the critical path that sets the cycle time. It has also an important cost in terms of area and power.

However, as it will be demonstrated in this thesis, register windows also supplies to the processor the required information to achieve the contrary effect. We will show how to drastically reduce the physical register requirements to the bare minimum with almost no performance loss in comparison to an unbounded physical register file. The register windows information allows the processor to identify which architectural registers are not being used by current in-flight instructions. As long as such registers have no in-flight references, they can be early released. Thereby, in this thesis we will show that contrary to common beliefs, register windows is not only beneficial for the reduction of memory references, but it also allows to manage more efficiently the first level of the memory hierarchy: the register file.

As stated above, the register window mechanism works by translating compiler-defined local variables to architected windowed registers. In an out-of-order processor this translation must be performed prior to physical register renaming. Hence, the information required for this translation is kept as a part of the processor state and must be recovered in case of branch misprediction or exception.

1.3 Related Work

The state of the art is explained in this section. Many of the works presented here are used in the next chapters to compare them with our proposals. This section focuses only on a brief explanation of them, leaving the comparison explanations for each corresponding chapter.

Predication was originally proposed in 1983 by Allen et al. [6] to remove the control dependences that constrained program analysis methods based on support for automatic vectorization. Those techniques, that converted sequential programs to a form more suitable for vector or parallel machines, were based only on data dependence analysis and not on control dependence. Thereby, all control dependences were transformed into data dependences by using a conditional expression as an extra input operand of instructions.

Predication has been implemented on real in-order and out-of-order processors. Many out-of-order processors such as Alpha [38] or PowerPC [21] implement partial predication, i.e. only a small number of instructions are provided with conditional execution. This approach represents an attractive solution for designers since the required changes to existing instruction set architectures (ISA) and data paths are minimized. Full predication has been implement only on in-order processors, such as Itanium [32].

Register windows was also early proposed in 1979 by Sites et.al. [62] as an organization mechanism for a huge register file: a context cache with fast multiport access and with short register names as addresses. Each location in the context cache was tagged with a context id and an offset. Instructions refered to registers by using the offset only. One of the earliest implementations of the register windows to speedup procedure calls was in [76].

SPARC [16] and Itanium [32] are two in-order commercial processors that implement register windows. In SPARC, where register windows are of fixed sized, overflows and underflows are handled by trapping to the operative system. In Itanium, where register windows are of variable size, overflows and underflows are handle by a hardware mechanism called *Register Stack Engine*.

The benefits brought by predicate execution and register windows ISA extensions has been widely exploited for in-order processors. However, the advantatges of using them on out-of-order processors have not been deeply studied. In this section we analyse those studies that mainly focus on if-conversion benefits and a more efficient management of the first level of the memory hierarchy: the physical register file.

1.3.1 Predicate Execution Benefits

Hsu et.al. [28] proposed the use of predication to reduce the penalty of *delayed conditional branches* [24] in deeply in-order pipelined processors, as alternative of out-of-order execution or branch prediction. Delayed branches are branches scheduled n instructions in advance, so the branch outcome is already computed at the time the fetch stage requires it. The n instructions following the delayed branch, called *delayed part*, are always executed regardless of whether the branch is taken or not. By using predicate execution, instructions that are control dependent on conditional delayed branches can be also scheduled inside the delayed part, by guarding all them with the same condition guard as its dependent conditional delayed branch.

Mahlke et.al. [44] eliminated branches that contribute to large numbers of mispredictions by using profile information. Compiler support for predicated execution is based on a structure called a *hyperblock* [46], which eliminates unbiased branches, while leaves highly biased branches. They show that the compiler can reduce the number of dynamic branches by 27% on average, which in turns reduces the branch misprediction misses by 20%. Moreover, the average distance between branches is increased from 3.7 to 6.1 instructions.

Tyson et.al. [75] arrived to similar conclusions. By using predication to remove branches with less prediction accuracy, they reduce the branch misprediction by 20%. Moreover, such a reduction can ease contention in branch predictors table access. They also study the benefits of instruction scheduling when applying predication. They show that an aggresive approach to predicating branches provides a basic block increment size by up to 45%, supplying the static scheduler with more instruction opportunities to fill the execution pipeline and significantly reducing pipeline stalls.

Pnevmatikatos et.al. [58] evaluated the effects of predicate execution on the performance of an out-of-order processor. They conclude that when using full predication, i.e. applied guarding predicates to all instructions, the effective block size, measured as the number of instructions between branches that actually contribute to useful computation, is increased by about 52%. This increment allows to also increase the average dynamic instruction window size (useful instructions between mispredicted branches) up to 258 instructions. However, the processor has to fetch and decode 33% more instructions that do not contribute to useful computation, i.e. those instructions whose predicate evaluates to false. This number is reduced when using partial predication, i.e. only basic blocks with no memory instructions inside are predicated, resulting in only 8% of false predicate instructions that fetch. However, the effective basic-block size increases by only 8%, and the average dynamic windows size is increased to 184 instructions. Finally, without any form of predication, the branch predictor could establish dynamic windows of only 156 instructions.

Chang. et.al. [10] studied the performance benefit of using speculative execution and predication to handle branch execution penalties in an out-of-order processor. They selectively applied if-conversion to hard-to-predict branches by using profile information to identify them; the rest of branches were handled using speculative execution. They found a significant reduction of branch misprediction penalties. The set of branches denoted by profiling as hard-to-predict accounted for an average of 73% of the total mispredictions for each of the SPECInt92 benchmarks.

Mahlke et al. [45], studied the benefits of partial and full predication code in an out-of-order execution model to achieve speedups in large control-intensive programs. With partial predicate support, only a small number of instructions are provided with conditional execution, such as conditional moves [13]. The paper concludes that, although using partial predicate support enables the compiler to perform full if-conversion transformations to eliminate branches and expose ILP, the use of full predicate support allow a highly efficient and parallel computation of predicate values. The paper shows that, for an eight issue processor that executes up to 1 branch per cycle, partial predication improves performance by 30% in comparison to a processor without predication sup-

port, whereas full predication improves performance by 63%.

August et.al. [8] proposed a framework for compiling applications for architectures that support predicated execution based on hyperblocks formation [46]. The framework consists of two major parts. First, aggressive if-conversion is applied early in the compilation process, which enables the compiler to take full advantatge of the predicate representation to apply aggresive ILP optimizations and control flow transformations. Second, partial reserve if-conversion [80] is applied at schedule time, which delays the final if-conversion decisions at the time the relevant information about the code content and the processor resource utilization are known. This allows to obtain the benefits of predication without being subject to the sometimes negative side effects of over-aggressive hyperblock formation, having an average speedup of almost 30% in comparison to superblock formation.

1.3.2 Predicated Execution and Branch Prediction

This section analyses proposals that use predication to improve branch prediction.

Mahlke et.al. [47] proposed a new approach for dynamic branch prediction, refered to as compiler synthesized prediction, that was not hardware based. Rather, the compiler is completely responsible for defining and realizing a prediction function for each branch by using profile feedback information. The result of this function, that is computed by extra instructions per branch inserted into the compiled code, is kept in a predicate register that later it will be consumed by conditional branches. The results shown that the performance of the compiler synthesized predictor is significantly better than that of the 2-bit counter branch predictor [68], and is comparable to that of the more sophisticated two-level hardware predictors [72].

August et.al [7] showed that the removal of some branches by if-conversion may adversely affect the predictability of other remaining branches, since it reduces the amount of available branch correlation information. Moreover, in some cases if-conversion may merge the characteristics of many branches into a single branch, making it harder to predict. They proposed the *Predicate Enhanced Prediction* (PEP-PA), that improves a local history based branch predictor by correlating with the previous definition of the branch guarding predicate. Depending on the pipeline depth and the scheduling advance of the predicate define, the predicate register value may not be available at the time the branch is fetched. Instead, the predicate register file contains the previous computed definition of that register. Assuming that its previous definition may be correlated with the current branch, whose predicate definition is not yet computed, the PEP-PA predictor uses this prior value to choose between one of two different local histories, both for using and for updating it. For branches whose predicate is available, the pattern history table (PHT) counters quickly saturate, and then prediction becomes equal to the computed predicate.

Simon et.al [66] incorporated predicate information into branch predictors to aid the prediction of region-based branches. The first presented optimization, called *Squash False Path*, stores the branch guarding predicate register number into its branch predictor entry, so future instances

can be early-resolved if the predicated value has been computed. The second presented optimization, called *Predicate Global Update Branch Predictor*, incorporates predicate information into the global history register (GHR) to improve the performance of region branches that benefit from correlation. Since the GHR is updated twice for every branch condition, once at the predicate define writeback, and another at the branch fetch, it stores redundant information. However, the main problem is that these updates are done at different places in the pipeline, so their scheme must include a complex *Deterministic Predicate Update Table* mechanism to guarantee that the GHR stores the conditions in program order. To overcome the existing delay between the branch prediction and the updating of the GHR by predicate computations, a new scheduling technique is also proposed. Their study was developed and evaluated for an in-order EPIC processor [25].

More recently, Kim et.al. [39] proposed a mechanism in which the compiler generates code that can be executed either as predicated code or non-predicated code, i.e. code that maintains their corresponding conditional branches. In fact, the code generated by the compiler is the same as the predicated code, except the conditional branch is not removed but it is transformed to a special branch type called *wish-branch*. At runtime, the hardware decides whether the predicate code or the non-predicated code is executed based on a run-time confidence estimation of the prediction of the wish-branch. The goal of wish branches is to use predicate execution for hard-to-predict dynamic branches and branch prediction for easy-to-predict dynamic branches. They claim that wish branches decrease the average execution time of a subset of SPECInt 2000 by 14% compared to tradicional conditional branches, and by 13% compared to the best-performing predicated code binary. Notice that, since if-converted branches are not removed but they are transformed into wish branches, this technique does not suffer from the loss of correlation information.

1.3.3 Predicate Execution on Out-of-Order Processors

As state above, the use of predicate execution on out-of-order processors has to deal with two performance issues: multiple register definitions at renaming, and unnecessary resources consumption. This section analyses proposals that try to eliminate such effects.

The problem of wasted computation resulting from if-conversion was first addressed by Warter et.al. [80]. They propose the use of if-conversion before the instructions scheduling phase of the compiler, to eliminate the control dependencies and expose parallelism to the optimizer. After the optimization phase, a *reverse if-conversion* transformation is proposed, in which guarded computation is transformed back into normal instructions covered by conditional branches.

Pnevmatikatos et.al. [58] proposed a predication execution model for out-of-order execution where each predicate instruction has three or four source operands: one or two source operands used for computing the value generated by the instruction, one predicate operand, and one implicit source operand specified by the destination register. If the predicate evaluates to true, the predicated instruction executes like a regular instruction, and the destination register is set with the value instruction computation. However, if the predicate is set to false, the predicated instruction writes the old value of the destination register (the implicit operand) back into the new destina-

tion register. This new functionality can be expressed in a C-style operation: *register definition = (predicate)? normal execution : previous register definition*. In other words, when the instruction evaluates to false, it copies the value from its previous physical register to the newly allocated physical register. Throughout this thesis we will use the term *false predicate conditional move* to refer to this predicate execution model.

Kling et al. [78] proposed to solve the ambiguity of multiple register definitions by automatically inserting into the instruction flow a micro-operation, called select-μop, that copies the correct register definition to a newly allocated physical register. The idea of the select-μop is derived from the ϕ-function used by compilers in static-single-assignment code (SSA) [15].

They propose an augmented map table. Instead of a single register mapping, each entry contains the physical register identifiers of all the predicated definitions as well as the guarding predicate of the instruction that defines each one. When an instruction renames a source operand, its physical register name is looked up in the map table. If multiple definitions are found in the map table entry, a select-μop is generated and injected into the instruction stream. The multiple register definitions and their guarding predicates are copied as source operands of the select-μop, and a new physical register is allocated for the result, so it becomes the unique mapping for that register. Later on, when the select-μop is executed, the value of one of its operands is copied to the destination register according to the outcomes of the various predicates. The goal of the select-μop is to postpone the resolution of the renaming ambiguity to latter stages of the pipeline.

Chuang et.al. [12] proposed a selective replay mechanism to recover the machine state from predicate mispredictions without flushing the pipeline in a predicate prediction scheme. When predicate instructions are renamed, they obtain the value of its guarding predicate from a predicate predictor. When a misprediction occurs, misspeculated instructions are re-executed with the correct predicate value, while other non-dependent instructions are not affected. With this mechanism all instructions, predicted true or false, are inserted into the issue queue. The issue queue entries are extended with extra tags to track two different graphs: the predicted and the replay data dependence graphs.

The predicted data flow tracks the dependencies as determined by the predicted values of predicates, as in conventional schedulers. When a misprediction occurs, the misspeculated instructions are re-executed according to the dependencies on the replay data flow graph. In this mode, predicated instructions are converted to *false predicated conditional moves* [58], which forces all the multiple definitions of the same register to be serialized. To maintain the replay data graph, one extra input tag is needed for each source operand. This tag contains the latest register definition, regardless of the predicate value of this definition. Recall that *false predicated conditional moves* also need an extra input tag containing the previous definition of the destination register.

1.3.4 Early Register Release Techniques

As explained above, the register file is the first memory hierarchy level perfectly integrated into the pipeline data path. Its size is a key factor to achieve high levels of ILP. However, there is a trade-off between size and access time: the bigger it is, the slower it is. There exist many proposals

that address this problem through different approaches. One approach consists of pipelining the register file access [26]. However, a multi-cycle register file requires a complex multiple-level bypassing, and increases the branch misprediction penalty. Other approaches improve the register file access time, area and power by modifying the internal organization, through register caching [83] or register banking [14]. Alternative approaches have focused on reducing the physical register file size by reducing the register requirements through more aggressive reservation policies: late allocation [23] and early release. This section focuses on early register release proposals.

Moudgrill et al [54] suggested releasing physical registers eagerly, as soon as the last instruction that uses a physical register commits. They identified three conditions: (1) the value has been writen to the physical register; (2) all issued instructions that need the value have read it; (3) the physical register has been unmapped from the map table. The last-use tracking is based on counters which record the number of pending reads for every physical register, and the *unmap flag* which is set when a subsequent instruction redefines the logical register, i.e. the old physical register is unmapped. Then, a physical register can be released once its usage counter is 0 and its unmapped flag is set.

This initial proposal did not support precise exceptions since counters were not correctly recovered when instructions were squashed. Akkary et al. [5] proposed to improve the Moudgrill scheme. For proper exception recovery of the reference counters, when a checkpoint is created the counters of all physical registers belonging to the checkpoint are incremented. Similarly, when a checkpoint is released, the counters of all physical registers belonging to the checkpoint are decremented.

Martin et.al. [48] introduced the *Dead Value Information* (DVI), which is calculated by the compiler and passed to the processor to help with early register releasing. DVI can be passed through explicit ISA extensions instructions, which contain a bit-mask indicating the registers to release, or implicitly on certain instructions such as procedure call and returns. They use the DVI such that when a dynamic call or return is committed the caller-saved registers are early-released because the calling conventions implicitly state that they will not be live.

Monreal et.al. [52] proposed a scheme where registers are released as soon as the processor knows that there will be no further use of them. Conventional renaming forces a physical register to be idle from the commit of its *Last-Use* (LU) instruction until the commit of the first *Next-Version* (NV) instruction. The idea is to shift the responsibility from NV instruction to LU instruction. Each time a NV instruction is renamed, its corresponding LU instruction pair is marked. Marked LU instructions reaching the commit stage will release registers instead of keeping them idle until the commit of the NV instruction. If the LU instruction is already committed when renaming NV, the releasing can proceed inmediately. However, if the NV instruction is speculative, which means there are branches between LU and NV, any release must be considered speculative and subject to squashing.

Martinez et.al. [49] presented *Cherry*: Checkpointer Early Resource Recycling, a hybrid mode of execution based on reorder buffer and checkpointing, that decouples resource recycling, i.e. re-

source releasing, and instruction retirement. In this scheme, physical registers are recycled if both the instruction that produces the value and all their consumers have been executed, which are identified by using the scheme described in [54], and are free of replay traps and are not subject to branch mispredictions To reconstruct the precise state in case of exceptions or interrupts, the scheme relies on periodic register file checkpointing.

Ergin et al. [17] introduced the checkpointed register file to implement early register release, by copying its value into a special shadow bit-cells implemented inside each single entry of the physical register file. A register is deallocated when its corresponding architectural register has been redefined, and all their uses have been executed, even when the redefining instruction is known to be speculative. This is done by copying its value into the shadow bit-cells of the register. If a misprediction occurs, the previous register value is recovered from the shadow bit-cell.

Jones et al [35] uses the compiler to identify registers that will only be read once and rename them to different logical registers. Upon issuing and instruction with one of these logical registers as a source, the processor can release the register. In order to maintain the constistent state in case of exceptions and interrupts, a checkpointer register is used [17].

Oehmke et.al. [55] have recently proposed the *virtual context architecture*(VCA), which maps logical registers holding local variables to a large memory address space and manages the physical register file as a cache that keeps the most recently used values. Logical register identifiers are converted to memory addresses and then mapped to physical registers by using a tagged set-associative rename map table. Unlike the conventional renaming approach, the VCA rename table look up may miss, i.e. there may be no physical register mapped to a given logical register. When the renaming of a source register causes a table miss, the value is restored from memory and a free physical register is allocated and mapped onto the table. If there are no free physical registers or table entries for a new mapping, then a replacement occurs: a valid entry is chosen by LRU, the value is saved to memory and its physical register is released. Although the VCA scheme does not properly define register windows as such, in practice it produces similar effects: multiple procedure contexts are maintained in registers, and the available register space is transparently managed without explicit saves and restores.

1.4 Thesis Overview and Contributions

The goal of this thesis is to propose novel mechanisms for predicated execution and register windows in out-of-order processors, with the objective of improving performance while keeping the complexity low. The following sections outline the challenges we face, the approaches we take to overcome these problems, and the novel contributions of our work.

1.4.1 Predicate Execution

Although the use of if-conversion is globally beneficial (see section 1.3.1), the use of predicate execution has to deal with three performance issues: multiple register definitions and unecessary

resource consumption, that affect to if-converted instructions; and a loss of the amount of correlation information that feeds the branch predictors, that affects to conditional branches. In ISAs that implement a predicate execution model such as the one considered in this thesis [32], the outcome of both if-converted instructions and conditional branches depends on the value of its guarding predicate, generated previously by a compare instruction. In such ISAs the knowlegde of the guarding predicate value at early stages of the pipeline may help to handle the three performance issues described above. In this thesis we propose a predicate prediction scheme that benefits from both if-converted instructions and conditional branches in four main aspects:

1. The knoweldge of the predicted predicate value at the rename stage allows those if-converted instructions whose predicate is predicted to *false* to be speculatively cancelled from the pipeline, while those if-converted instructions predicted to *true* are normally renamed. By doing this, we avoid the multiple register definitions and the consumption of unnecessary resources.

2. Predicting the predicates of an if-converted code is actually like reversing the if-conversion transformation. In order not to loose the benefits brought by if-conversion, the predicate predictor dynamically selects which if-conversions are worth to be reversed, and which ones should remain in its if-converted form because of a hard-to-predict branch. In the latter case, the if-converted instruction is transformed to a *false predicate conditional move* (see section 1.3.3) to avoid the problem of multiple register definitions.

3. Conditional branch outcomes depend on the value of their guarding predicates generated by compare instructions. By using a predicate prediction scheme instead of branch prediction, the predictor is not affected by the loss of correlation information caused by the removal of if-converted branches, because such information is still present in the compare instructions that generate branch predicates. In addition, since the same predictor is used for branches and other predicated instructions, our scheme handles if-conversion at almost no hardware cost, unlike all previous approaches, where substantial complexity was added to the renamer [78] and the issue queue [12, 78].

4. The branch prediction accuracy may be improved by using a predicate predictor, if the compare instruction is scheduled enough in advance, in such a way that conditional branches do not consume a predicted value but a computed value, achieveing 100% of accuracy in such a case.

In conclusion, our scheme replaces the branch predictor scheme by a predicate predictor scheme whose predictions are consumed by both if-converted instructions and conditional branches, allowing out-of-order processors to execute predicate code without loosing the benefits brought by if-conversion and without any branch accuracy degradation.

1.4.2 Register Windows

The effectiveness of register windows depends on the size of the architectural register file. In an out-of-order processor, the architectural register file determines the minimum size of the physical

register file to guarantee forward progress, i.e. the number of architected registers plus one. However such register requirements are increased to store the uncommited values produced by in-flight instructions. Hence, the use of register windows in out-of-order processors puts extra pressure on the physical register requirements, which may negativelly impact its access time.

The register windows hardware mechanism maps at runtime the compiler-defined local variables of a procedure context into a set of consecutive arhcitectural registers. By tracking all the uncommitted procedure contexts, the processor is able to identify which mappings are currently being used by in-flight instructions. This makes possible two early register release opportunities:

1. When a procedure finishes and its closing return instruction commits, all physical registers defined inside the closed procedure contain *dead values* because they will never be used again and so they can be early released.

2. When the rename stage runs out of physical registers, mappings defined by instructions belonging to caller procedures not currently in-flight can also be early released. However, unlike the previous case, these values may be used in the future when the context is restored, so they must be spilled to memory before being released.

By exploiting these two early release opportunities, our proposed scheme achieves a drastic reduction of the physical register requirements to the bare minimum, i.e. the number of architected registers plus one. Moreover, since the register windows mechanism is involved in the rename, it must be kept as a part of the processor state. By tracking all uncommitted procedure contexts, our scheme is also able to recover the register window state in case of branch mispredictions or exceptions.

1.4.3 Contributions

The main contributions of this thesis are:

1. We propose selective predicate predictor, a technique that enables predicated execution in out-of-order processors, thus enabling if-conversion optimization. This technique uses an adaptive runtime mechanism to select which if-conversion transformations are maintained in its original form, which ones are reversed. Some of its main features are:

   - It solves the multiple definitions problem.
   - It avoids unnecessary resource consumption of instructions whose predicate evaluates to false.
   - It requires very simple additional hardware to implement misprediction recovery.
   - It outperforms previous approaches.

2. We propose predicting branches with a second-level overriding predicate predictor instead of a branch predictor. The main features of this scheme are:

   - It uses the same hardware for branch prediction and for predicated execution, so the predicated execution is implemented for free with respect to a conventional processor, which supposes a huge cost reduction compared to previous approaches.

- It avoids loss of branch prediction accuracy caused by loss of correlation information when if-conversion removes some branches.
- It improves branch prediction accuracy because of early-resolved branches.

3. We propose a technique to implement register windows in out-of-order processors. Its main features are:

   - The proposed hardware requires a simple direct-mapped map table, more simple than previous approaches.
   - The mechanism to handle window information also permits simple recovery of the windows state in case of branch misprediction or exceptions.

4. We propose *Context Release*, *Early Register Release Spill* and *Alloc Release* that drastically reduce the physical register requirements, based on using the information associated with procedure contexts.

5. We propose a simple implementation of these early register release techniques by adding mininal changes to the hardware that manage the register windows, so that it may identify at any point in time which registers mappings are in use by current in-flight instructions.

6. We propose Delayed Spill/Fill, a technique to reduce the amount of spills and fills required by the register window mechanism.

1.5 Document Organization

The rest of this document is organized as follows:

Chapter 2 describes the assumed out-of-order microarchitectural model, as well as its default main parameters, and the experimental framework utilized throughout this thesis, including the simulation methodology and the benchmarks.

Chapter 3 proposes a predicate predictor scheme that overcomes the execution problems of predicate instructions in out-of-order processors. This scheme reverses if-conversion transformations. A confidence predictor is also introduced not to loose the benefits brought by if-conversion. Moreover, a new branch prediction scheme based on our predicate predictor that does not loose correlation information and improves the branch accuracy is studied. All these proposal are evaluated and compared with previous similar studies and with conventional branch prediction schemes.

Chapter 4 proposes a register windows scheme to reduce the physical register requirements of an out-of-order processor by identifying which mappings are not in used by current in-flight instruction. These mappings are released as well as their associative physical register. Moreover, a checkpoint mechanism is also introduced to recover the procedure context state in case of branch misprediction or exception. Our proposal is evaluated and compared with previous similar studies and with a scheme that does not use the information provided by register windows.

Chapter 5 summarizes the main conclusions of this thesis.

Chapter 2
Experimental Setup

Simulation is a well known established technique in both academic and industry research to evaluate new architectural ideas. In order to study the performance impact of predication and register windows on an out-of-order processor, we have developed a cycle-accurate, execution driven simulator that runs Itanium binaries. Itanium is a commercial architecture that incorporates predication and register windows, which makes it an excellent platform to develop our research. Our simulator models in detail most of the Itanium features that are involved in different pipeline stages, such as register rotation and memory speculation, executing into a nine-stage processor with an out-of-order execution model.

This chapter describes the most important contributions to the Itanium out-of-order microarchitecture design, its default main parameters and the experimental environment used along this thesis.

2.1 Itanium as a Research Platform

Explicitly Parallel Instruction Computing [64] (EPIC) is an evolution of VLIW architectures that has also absorbed many superscalar concepts. EPIC provides a way of building processors, along with a set of architectural features that support this philosophy. The term denotes a class of architectures that subscribe to a common philosophy. The first instance of a commercially available EPIC ISA is Itanium [31–33].

Itanium, also known as IA64 (*Intel Architecture 64-bits*) or IPF (*Itanium Processor Family*), is a commercial architecture based on EPIC that incorporates an array of architectural features that supplies to the compiler the required tools to extract high levels of ILP. There are two implementations of it: Itanium 1 [65] and Itanium 2 [51], both in-order processors.

The Itanium platform provides a competitive comercial compiler (Electron v.8.1) [9] that ensures a code generation quality, as well as developement tools such as *PinPoints*, that allows to obtain the most representative portions of code [57], or an open-source research compiler such as OpenCC [3]. All this makes Itanium an excellent platform to develop our research.

Two of the ISA extensions that Itanium incorporates are *predicate execution* and *register windows*. In the following subsections Itanium implementations of predication and register windows are brieftly explained. Other important Itanium ISA extensions, such as memory speculation and register rotation, also implemented in our Itanium out-of-order execution model, are also briefly described.

2.1.1 Register Rotation

Software pipelining allows compilers to execute multiple loop iterations in parallel. However, the concurrent execution of multiple iterations traditionally requires to perform loop unrolling several times, which entails the use of multiple registers to hold the values of multiple iterations.

Itanium architecture provides an ISA extension, *register rotation*, that allows to assign the same register name to values coming from multiple iterations, avoiding the need to unroll and copy these registers. At runtime, these compile-defined registers are renamed to a different architectural registers based on the *loop iteration number*. The loop iteration number distinguishes one iteration from another, making the value of register r[n] appear to move to register r[$n+1$]. It is held as a part of the procedure context state, and it is incremented by certain special loop branches (*br.ctop, br.cexit, br.wtop, br.wexit*) when executed at the end of each kernel iteration. When n is the highest numbered rotating register, its value wraps around to the lowest numbered rotating register.

Rotating predicate registers and rotating floating-point registers are defined as a fixed-size register subset, from *p16* to *p63* and from *f32* to *f127* respectively. Rotating general purpose registers are defined as a variable-size multiple of eight register subset, starting at register *r32* up to the maximum register window size defined at by the *alloc* instruction.

2.1.2 Memory Speculation

Memory latencies are recognized as one of the major factors that restricts performance, especially on integer programs. To mitigate such negative effects, *memory speculation* is used. The memory speculation is the scheduling in advance of load memory instructions regardless of possible data or control dependence violations [11, 22, 38]. The Itanium architecture implements compile-controlled explicit memory speculation with hardware support that allows to schedule loads ahead of branches (*control speculation*) and ahead of stores (*data speculation*):

- *Control speculation* allows loads and their dependent instructions to be safely scheduled ahead of a branch before it is known that the dynamic control flow of the program will actually execute them. Since these loads could cause a memory exception because of a wrong speculation, the exception is not immediately raised. Instead, a NaT bit (*Not a Thing*) is set on its destination register and it is propagated through all subsequent dependent instructions, until a non-speculative *check* instruction restores the state. Itanium defines a control speculative load instruction (*ld.s*) and a check instruction (*chk.s*) that verifies the correctness of the speculation.

- *Data speculation* allows loads to be scheduled ahead of stores even when it is not known whether the load and the store references overlap into the same memory location. An special check instruction is required to see if the advance load has violated any memory data dependence, using a special hardware mechanism, called *Advance Load Address Table* (ALAT). Each *advance load* allocates a new entry on the ALAT table, that can be indexed using either its destination register or its memory access address. Subsequent stores check if there is an ALAT entry with the same memory address. If it is, a memory dependence has been violated, and the entry is removed. Later, a check instruction searches the ALAT by using the same destination register of its corresponding advance load. In fact, both instructions

are data dependent, maintaining the order of advance loads and check instructions. If the entry is found, the speculation was successful; otherwise, the speculation was unsuccessful and the misspeculation must be fixed. Itanium defines a data speculative load instruction (*ld.a*) and two check instructions (*chk.a, ld.c*) that verify the correctness of the speculation.

2.1.3 Predication

Itanium is a fully predicated ISA that allows to predicate almost all instructions. In fact, there are only a few of them that can not be predicated, namely: allocate a new register window (*alloc*), clean the rotating register base (*clrrrb*) or counted branches (*br.cloop, br.ctop, br.cexit*). Predicates are one-bit values that affect the dynamic execution of instructions: if the predicate is *true*, the instruction executes normally; otherwise, the architectural state is not modified, except for unconditional compare type (*cmp.unc*), floating point approximation (*fprcpa, fprsqrta, frcpa, frsqrta*) and while-loop (*br.wtop, br.wexit*).

Predicates are generated by compare instructions (*cmp*), test a single bit or the NaT bit (*tbit, tnat*) and floating point approximation (*fprcpa, fprsqrta, frcpa, frsqrta*). Although our out-of-order execution model takes into account all these instructions, this section focuses only on compare instructions.

A compare instruction tests a single specified condition, formed by two integer registers or an integer register and an immediate operand, based on comparison relations such as equal, not-equal, greater-than, etc. and generates two boolean results that are written into a predicate register file. Itanium defines five types of compare instructions:*normal, unconditional, AND, OR* and *De-Morgan*. These types define how the compare instruction writes its destination predicate registers based on the result of its condition and on its own guarding predicate.

Table 2.1 shows the outcome of a compare instruction (p_1 and p_2) based on its type (*ctype*), its own guarding predicate (*qp*) and the result of its condition (*cond*). *NaTs* column is applied when one or two of the registers that form the condition has its NaT bit set to 1 (see section 2.1.2). An empty cell means that p_1 and p_2 remains with its previous value, i.e. the architectural predicate register is not modified.

A set of 64 predicate registers of 1 bit are used to hold the results of compare instructions. These registers are consumed by both if-converted instrucions and conditional branches.

Itanium also provides special compare instructions, called *parallel compares*, that compute efficiently complex compound conditions. These conditions normally require a tree-like computation to reduce several conditions into one. Parallel compares allow to compute conditions that have *and* and *or* boolean expressions in fewer tree levels. Unlike common compares, the destination predicate registers of parallel compares can be updated or not, depending on the previous value of its destination predicate register. In fact, the previous value acts as a source operand: in *or* parallel compare types, when the previous value of its destination register is 1, the register is not modified; while *and* parallel compare types do not modify its destination register when its previous value is 0.

Comparison Types								
ctype	qp = 0		qp = 1					
			cond = 0		cond = 1		NaTs	
	p_1	p_2	p_1	p_2	p_1	p_2	p_1	p_2
none			0	1	1	0	0	0
unc	0	0	0	1	1	0	0	0
or					1	1		
and			0	0			0	0
or.andcm					1	0		
orcm			1	1				
andcm					0	0	0	0
and.orcm			0	1				

Table 2.1: **The compare type *ctype* describes how the destination predicate registers p_1 and p_2 are updated based on the result of the condition *cond*, its guarding predicate *qp* and the NaT bits that form the condition (*NaTs*).**

2.1.4 Register Windows

Register windows or *stack frames* (Itanium terminology), provides Itanium with a large integer architectural register file to reduce the call conventions overhead. Itanium defines 128 architectural integer registers: 32 static registers that do not take part on the register windows mechanism, and 96 windowed or stacked registers that form the register windows mechanism.

For each register window or stack frame Itanium defines three regions: the *input region*, that contains the parameters passed by the caller procedures; the *local region*, that holds private procedure values; and the *output region*, that holds the parameters that will be passed to callee procedures. By overlapping the output and the input region of two register windows, parameters are passed through procedures without using memory references such as activation records [30] and without the need to copy registers. In fact, the time of a call, registers that belong to the output region of a caller procedure become automatically registers of the input region of the callee procedure.

Once the register window is created, its size can be modified using the *alloc* instruction. The alloc specifies the number of registers the procedure expects to use, i.e. the size of the input, local and output regions and the rotating register area (see section 2.1.1). A procedure can allocate a register window of up to 96 registers. When there are not enough available registers (*stack overflow*) the alloc stalls and registers of previous caller procedures are automatically sent to a special region of memory called *backing store*. This operation is called *spill*. At the return point, i.e. when a *br.ret* executes, the register window prior to the call is restored. If some of the caller's registers have been sent to memory, the return stalls until all required registers are brought back to the register file (*stack underflow*). This operation is called *fill*.

The *Register Stack Engine* (RSE) is the automatic mechanism that moves registers between the register file and the backing store without any program intervention. The RSE operates concurrently with the processor and can take advantatge of unused memory bandwidth to dynamically issue spill and fill operations. In this manner, the latency of spills and fills can be overlapped with

useful program work.

2.2 An Out-of-Order Execution Model for Itanium

Along this dissertation, we assume an out-of-order superscalar model with a nine-stage pipeline (fetch, decode, rename, dispatch, issue, register read, execute, writeback and commit), similar to that of the Alpha 21264 [38]. The backend is composed by three separate integer, FP and branch issue queues (IQ) for dynamically scheduling instructions. A merged physical register file stores both architectural and speculative values, with a renaming mechanism that maps logical to physical registers. Finally a reorder buffer (ROB) records the program order and holds instruction status, necessary to support speculation recovery and precise interrupts. Note that neither the ROB nor the IQ contain operand data but only physical register names (tags), so in this model source operands are read after instruction issue and before execution, either from the register file or from the bypass. We assume an aggressive instruction fetch mechanism to stress the instruction issue and execution subsystems, and a minimum branch misprediction penalty of 10 cycles.

Loads and stores are divided into two operations at the rename stage. One of them is dispatched to the integer IQ which computes the effective address. The other one is dispatched to a load/store queue and it accesses memory. When the first operation completes, it forwards the effective address to the corresponding entry in the load/store queue, to perform memory disambiguation and access to memory. A load access is issued when a memory port is available and all prior stores know their effective address. If the effective address of a load matches the address of a previous store, the store value is forwarded to the load. Store memory accesses are issued at commit time.

The common architectural parameters used along this thesis are shown in Table 2.2. Branch predictor type or register file size is detailed inside each Chapter.

The choice of Itanium as the base architecture to develop our researches requires an extra effort on the simulator design. Although Itanium ISA specification is not explicity defined as an in-order architecture, many of their architectural features such as predication, register windows, register rotation or compile-controlled memory speculation, require an extra hardware support to be executed on an out-of-order processor. Although these architectural features intervene in several pipeline stages, the main hardware additions reside in the rename stage and in the memory hierarchy, which are briefly explained in the following subsections. Predication and register windows are extensively explained in Chapters 3 and 4.

2.2.1 The Rename Stage

Probably, the rename stage concentrates most of the hardware mechanisms that support the ISA extensions implemented on Itanium, such as predication, register windows and register rotation.

Predication affects to the overall execution of instructions, although it has a strong impact on the rename stage due to the multiple register definitions problem (see section 1.2.1). In order to ensure a correct predicate execution model, our baseline scheme converts all predicate instructions into *false predicated conditional move* [58] (see section 1.3.3). This new functionality can be expressed in a C-style operation: *register definition = (predicate)? normal execution : previous*

Architectural Parameters	
Fetch & Decode width	6 instructions (2 bundles)
Issue Queues	Integer: 80 entries Floating-point: 80 entries Branch Issue: 32 entries
Load-Store Queue	2 separated queues of 64 entries each
Reorder Buffer	256 entries
L1D	64KB, 4way, 64B block, 2 cycle latency, Non-blocking 12 primary misses, 4 secondary misses, 16 write-buffer entries 2 load, 2 store ports
L1I	32KB, 4 way, 64B block, 1 cycle latency
L2 unified	1MB, 16 way, 128B block, 8 cycle latency Non-blocking, 12 primary misses 8 write-buffer entries
DTLB	512 entries. 10 cycles miss penalty
ITLB	512 entries. 10 cycles miss penalty
Main Memory	120 cycles of latency

Table 2.2: **Main architectural parameters used.**

register definition.

Although simple, this approach serializes the execution of predicated instructions due to the dependence on the previous definition, making deeper the dependence graph and reducing the effectiveness of the dynamic execution as shown in the graph of Figure 2.1b. Moreover, instructions with a false predicate are not early-cancelled from the pipeline, so they continue consuming physical registers, issue queue entries and functional units.

Register windows and register rotation intervene directly on the renaming process. Both work by translating compile-defined local variables to architected stacked registers. On an out-of-order processor this translation must be performed prior to access to the map table where they obtain the corresponding physical register.

Figure 2.2 shows the basic renaming scheme of our baseline architecure. Stacked compile-defined registers are translated by adding two base pointers, *Register Rotation Base* (RRBase) and *Register Window Base* (RWBase), both associated to each procedure context. At a call point, when a new procedure context is invoked, the RWBase is incremented up to the output region of the caller procedure, becoming the input region of the invoked procedure. The RRBase is a multiple of eight value defined at the time the register is allocated by the *alloc* instruction. Static compile-defined registers access directly to the map table, without any translation.

2.2.2 Memory Subsystem

Compile-controlled memory speculation requires the use of check instructions (*chk.s, chk.a, ld.c*) that verify at runtime the correctness of the speculation. In-order execution models ensure that, instructions younger than a check in program order, have not been executed at the time the misspeculation is detected. However, this is not the case in out-of-order execution models, where younger instructions may have been already executed, consuming the wrong data brought by the

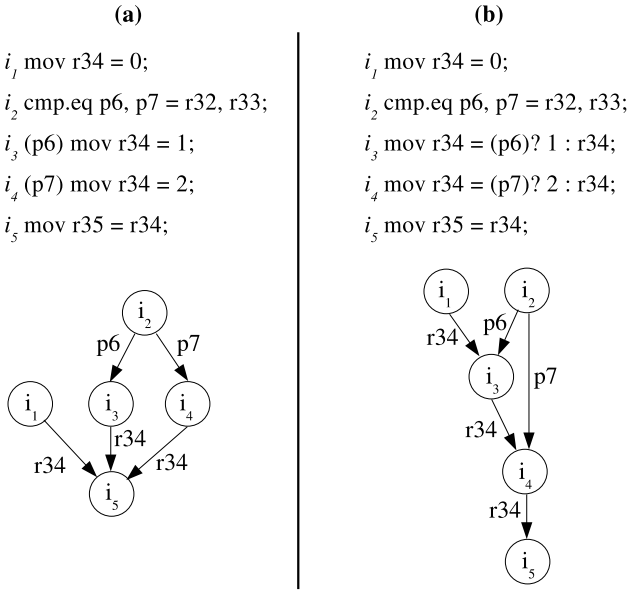

Figure 2.1: **Data dependence graph changes when all predicate instructions are converted to *false predicated conditional moves*. (a) Original code. (b) Converted code.**

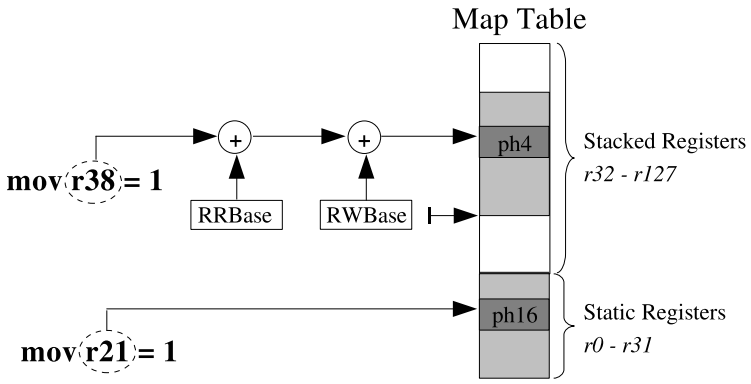

Figure 2.2: **Stacked register *r38* is dynamically translated to its corresponding architectural windowed register by using the *RRBase* and the *RWBase* pointers, prior to access to the map table. Static register *r21* access directly to the map table without any translation.**

mispeculated advance load. Hence, in out-of-order processors with compile-controlled memory speculation, when a memory misspeculation is detected all instructions younger than the check instruction must be flushed and re-fetched again.

Our out-of-order simulator model implements the NaT bit required to perform control speculation, which is associated to each integer and floating-point physical register. A NaT bit equal to 1 indicates that the register contains a deferred exception which value the software can not rely upon. The NaT bit is propagated through all subsequent dependent instructions until the corresponding

check is executed. Note that, when a check detects a control misspeculation, instructions that are older than check instructions and data-dependent of the misspeculated load are not flushed, although the NaT bit of their target physical registers is set to 1.

The ALAT has been also modeled in the memory disambiguation subsytem of the simulator. Check instructions index the ALAT not through the architectural register but through the physical register number. Physical register numbers are unique identifiers inside the processor, and they allow to chain a check with its correponding advance load ensuring that a load-check does not execute prior its corresponding advance-load. Moreover, to ensure that a load-check does not execute prior stores that lay between itself and its corresponding advance load in program order, the load-store queue requires to check that both instructions do not overlap into the same memory address.

2.3 Simulation Methodology

Our out-of-order simulator has been built from scratch using the Liberty Simulation Environment (LSE) [77]. LSE is a simulator construction system based on module definitions and module comunications, which also provides a complete IA64 functional emulator that maintains the correct state machine. The functional emulator required modifications, especially in the RSE and the ALAT structures, to allow execute IA64 code in our out-of-order simulator.

All the experiments conducted in this thesis have been performed with the SpecCPU 2000 [4] benchmark suite, using two different input sets: *MinneSpec* [40] and *Test*. All benchmarks have been compiled with IA64 Intel's compiler (Electron v.8.1) using maximum optimization levels and profile information. For each benchmark, 100 million of committed instructions have been simulated, starting at a representative portion of code, obtained using the Pinpoint tool [57]. For experimental reasons, benchmarks have also been compiled using the open source compiler Opencc [3], also using the maximum optimization levels.

Chapter 3
If-Conversion Technique on Out-of-Order Processors

If-conversion is a compiler technique that transforms control dependences into data dependences by using predicate execution. It is useful to eliminate hard-to-predict branches and reduce the severe performance penalties of branch mispredictions. Although it is globally beneficial, the execution of if-converted code on an out-of-order processor has to deal with two problems: (1) predicated code generates multiple definitions for a single destination register at rename time; (2) instructions whose predicate evaluates to false consume unnecessary resources. Besides, if-conversion has also a negative side-effect on branch prediction because the removal of some branches may eliminate correlation information useful for conventional branch predictors. As a result, the remaining branches may become harder to predict.

In predicated ISAs with a predicate execution model such as IA64, compare instructions compute the guarding predicate values that are consumed indistinctly for both if-converted and conditional branch instructions. Predicate prediction is an effective approach to address the problems stated above. First, it allows to know the predicate value of an if-converted instruction at rename stage, avoiding multiple register definitions and unnecessary resource consumption. Second, the use of a predicate prediction scheme as a branch predictor allows to recover the correlation lost by the removal of some branches, since the correlation information not only resides in branches, but also in compare instructions. In fact, the prediction of predicates reverses if-conversions to their original form. In this chapter we propose a hardware mechanism, in order not to loose the benefits brought by if-conversion, which dynamically selects which if-conversion is worth to be predicted, and which one is more effective in its if-converted form.

Our approach enables a very efficient implementation of if-conversion for an out-of-order processor, with almost no additional hardware cost, because the same hardware is used to predict the predicates of if-converted instructions and to predict conditional branches without any accuracy degradation.

3.1 Introduction

Branches are a major impediment to exploit instruction-level parallelism (ILP). The use of branch prediction in conjunction with speculative execution is typically used by out-of-order processors to remove control dependences and expose more ILP to the dynamic scheduler. However branch mispredictions result in severe performance penalties that tend to grow with larger instructions windows and deeper pipelines.

If-conversion [6] is a compiler technique that helps to eliminate hard-to-predict branches, by converting control dependences into data dependences and potentially improves performance.

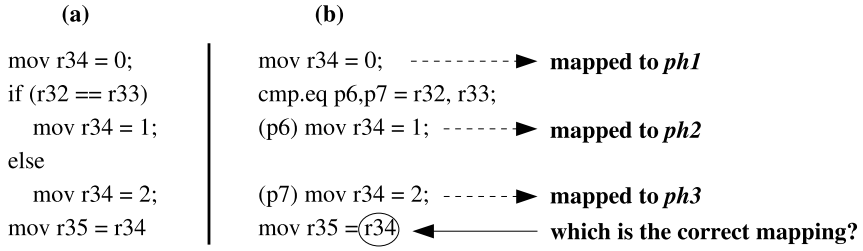

Figure 3.1: **If-conversion collapses multiple control flows. The correct map definition of *r34* depends on the value of *p6* and *p7* predicates. (a) Original if-sentence. (b) If-converted code**

Hence, if-conversion may alleviate the severe performance penalties caused by hard-to-predict branch mispredictions, by collapsing multiple control flow paths and scheduling them based only on data dependences.

If-conversion relies on predicate execution. Predication is an architectural feature that allows an instruction to be guarded with a boolean operand whose value decides whether the instruction is executed or converted into a no-operation. Although our study focuses on the effects of if-conversion, predication has other possible uses [31].

Many studies have shown that if-conversion provides an opportunity to significantly improve hard-to-predict branches in out-of-order processors [10, 44, 45, 58, 75] (see section 1.3.1). This advantage tends to be even more important with larger instruction windows and deeper pipelines. However, as stated in Chapter 1, if-converted instructions have to deal with two issues in such processors:

1. When multiple control paths are collapsed, multiple register definitions can be merged into a single control flow, guarding each one with a different predicate. At run-time, each definition is renamed to a different physical register, thus existing multiple possible physical names for the same logical register until the predicates are resolved. Since predicates are resolved at the execution stage of the pipeline, it may occur that the name of that register is still ambiguous when renaming the source of an instruction that uses it. Figure 3.1 illustrates the problem.

2. Instructions whose predicate evaluates to false have to be cancelled. If this is done in late stages of the pipeline, these instructions consume processor resources such as physical registers, issue queue entries and/or functional units, and can potentially degrade performance.

Moreover, the removal of some conditional branches by if-conversion may adversely affect the predictability of the remaining branches [7], because it reduces the amount of correlation information available for branch predictors. As a consequence, the remaining branches may become harder to predict, since they may have little or no correlation among themselves.

In conclusion, although if-conversion is globally beneficial, the use of if-conversion technique in out-of-order processors may affect negativelly both if-converted and conditional branch instructions.

In ISAs that implement a predicate execution model such as the one considered in this thesis [31], the execution of if-converted and branch instructions depends on the value of its guarding predicate which is produced in both cases by a previous compare instruction. On such a model, the predicate prediction alleviates the above mentioned performance issues.

In this chapter, we propose a novel microarchitecture scheme for out-of-order processors that improves the performance of if-conversion. Our approach predicts the guarding predicates of if-converted instructions and conditional branches at the time they are produced by compare instructions using the compare PC. Such a prediction affects in different ways the two types of predicate instructions: if-converted and conditional branch instructions.

- Regarding if-converted instructions, the knowledge of the guarding predicates at rename time allows that if-converted instructions whose predicates are predicted to false are speculatively cancelled and removed from the pipeline, thus avoiding multiple definitions and unnecessary resource consumption. Only those instructions with predicates predicted to true are normally renamed.

- Regarding conditional branch instructions, the branch predictor is, in fact, replaced by a predicate predictor. Branch predictions are actually generated by using the compare PC instead of the branch PC. Our predicate predictor scheme is able to keep all correlation information among branches, even for those removed by if-conversion, since such information is primarily associated with compare instructions. Unlike previous proposals, our scheme is able to fully correlate branch global history. In addition, it takes full advantage of the knowledge of *early-resolved branches* [7,66] to further improve branch prediction accuracy, since it uses the computed predicate value when it is available, instead of the predicate prediction.

A closer look to the predicate prediction scheme reveals that what it actually does is to undo if-conversion transformations done by the compiler. Therefore, since if-conversion appears to be more effective than branch prediction for hard-to-predict branches [10], applying prediction to all predicates may miss the opportunities brought by if-conversion. In order not to loose if-conversion benefits, we introduce a specific hardware mechanism for if-converted instructions, perfectly integrated in our predicate prediction scheme, that *selectively* predicts predicates based on a confidence predictor: a predicate prediction is used by an if-converted instruction only if the prediction has *enough* confidence. If not, the if-converted form remains. In this case, in-order to overcome the multiple register definition problem the predicated instruction is converted into a *false predicate conditional move* (see section 2.2.1). In fact, our approach tries to preserve if-conversion for those branches that are really hard-to-predict. We have found that only 16% of if-converted instructions are converted to false predicate conditional moves.

In summary, our proposal uses the same predicate predictor to predict conditional branches and if-converted instructions with no extra hardware cost. It allows to execute predicated code in out-of-order processors without loosing the benefits brought by if-conversion, and without any branch accuracy degradation. This chapter explains the techniques proposed in papers [59,60].

The rest of the chapter is organized as follows. Section 3.2 discusses our predicate predictor scheme for if-converted instructions. Similary, section 3.3 discusses our predicate predictor scheme, but for conditional branches. Finally, section 3.4 presents a brief summary and conclusions.

3.2 A Predicate Predictor for If-converted Instructions

As shown in previous section, the use of predicate execution in out-of-order processors has to deal with two problems: the multiple register definitions of a source register at rename time, and the unnecessary resource consumption caused by false predicated instructions.

As mentioned in section 1.3.3, previous proposals have focused only on the multiple register definitions problem. In proposals such as select-μops [78] or false predicated conditional move [58] techniques, instructions whose predicate evaluates to false consume physical registers, issue queue entries, and they compete for execution resources. Still worse, the select-μops technique further aggravates the resource pressure with extra pseudo-instructions injected in the pipeline.

The prediction of predicates is a good solution to avoid consuming unnecessary resources, because once the predicate of an instruction is predicted false, it can be cancelled and eliminated from the pipeline at rename time. Although the selective replay mechanism [12] also uses predicate prediction (which avoids multiple definition problems), it precludes this advantage because it needs to maintain all predicated instructions in the issue queue to reexecute them in case of predicate misprediction, regardless of the predicate being predicted true or false. As it will be shown in section 3.2.3, the reduced re-fetch penalty avoided by the replay mechanism is not worth the extra pressure on the issue queue and other key resources.

The predicate prediction technique generates a prediction for every predicate at the time the compare instruction is fetched. At rename stage, instructions with a predicate predicted to true are speculatively renamed and executed, while instructions whose predicate is predicted to false are not renamed and cancelled from the pipeline. Note that the prediction of predicates produces an effect similar to that of branch prediction and speculative execution in the sense that it reverses the if-conversion transformations.

This effect is illustrated in Figure 3.2. In Figure 3.2a, *i5* instruction will not be fetched if the conditional branch is predicted as not-taken, i.e. if *p7* is predicted to false. Similary, in Figure 3.2b, once if-conversion has been applied, *i5* instruction will not be dispatched to the issue queue if *p7* is predicted to false. Thereby, the effectiveness of the if-conversion form depends on the predictability of the removed conditional branch *i2* or similary, the predicatibiliy of the predicate *p7*. Hard-to-predict branches produce high misprediction penalties, being more effective its if-converted form.

Hence, we propose a selective predicate prediction scheme that addresses both the multiple definition and unnecessary resources consumption problems, without loosing the potential benefits of if-conversion. Our scheme dynamically identifies easy-to-predict predicates with a confidence predictor: a predicated instruction is speculated only when its prediction has *enough* confidence; otherwise it is preserved in its if-converted form. In the latter case, the multiple definitions problem is avoided by converting the instruction into a false predicated conditional move.

In case of a predicate misprediction, the misspeculated instruction and all subsequent instructions are flushed from the pipeline and re-fetched. Of course, handling mispredictions with flushes may produce high performance penalties. However, as far as the confidence predictor is able to correctly identify hard-to-predict predicates and avoid predicting them, these penalties are expected to have a minor impact. The next sections describe the selective predicate prediction mech-

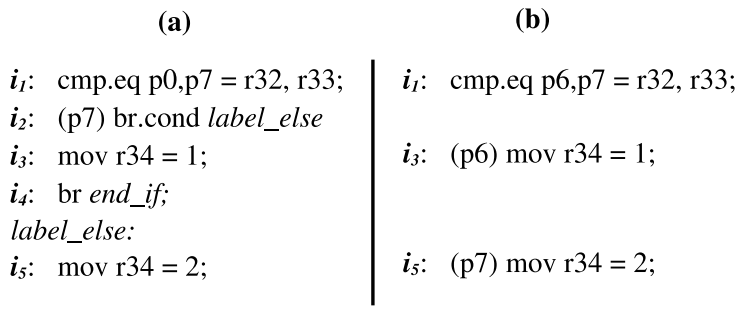

Figure 3.2: **Predicate prediction produces a similar effect to branch prediction. In both (a) and (b), *i5* is not executed if *p7* evaluates to false. (a) Original code. (b) If-converted code.**

anism in detail.

3.2.1 The Basic Predicate Prediction Scheme

This section describes the basic predicate predictor scheme (without the confidence predictor) for an out-of-order processor. This scheme assumes an ISA with full predicate support, such as the one considered in this thesis [33].

A predicate prediction is produced in early stages of the pipeline for every instruction that has a predicate outcome, such as compare instructions, using its *PC*. Figure 3.3 illustrates the prediction mechanism for producers. Since compare instructions have two predicate outputs, our predictor generates two predictions, one for each compare outcome. When the compare instruction is renamed, the two predictions are speculatively written to the Predicate Physical Register File (PPRF). Later on, when the compare instruction executes, the PPRF is updated with the two computed values. In the example, *p6* and *p7* are renamed to *pph2* and *pph1* respectively.

To support speculative execution of predicated instructions, each entry of the conventional PPRF is extended with three extra fields: *speculative* and *confidence* bits, and *ROB pointer*. Since the PPRF holds either predicted or computed predicate values, the *speculative* bit is used to distinguish both types: when the PPRF is first written with a prediction, the *speculative* bit is set to true; when it is updated with the computed value, the *speculative* bit is set to false.

A predicate prediction is consumed by one or more if-converted instructions that depend on it. When a predicated instruction reaches the rename stage, it always reads its predicate value from the PPRF. If the *speculative* bit is set to true the instruction will use the predicate speculatively, so the processor must be able to recover the previous machine state in case of misspeculation. The *ROB pointer* field is set to point to the ROB entry of the first speculative instruction that uses the predicted predicate. Thereby, if the prediction fails, the pointed instruction and all younger instructions are flushed from the pipeline. Thus, after the *speculative* bit is set to true, the *ROB pointer* field must be initialized by the first consumer that finds it empty. If the consumer predicate is predicted to false, the Cancel Unit (CU), cancels the instruction and eliminates it from the pipeline. Figure 3.4 shows the predicate consumption scheme.

As explained in Chapter 2, the IA64 ISA defines a rich set of instructions to produce predicate values. For some compare types, the two predicate results depend on the outcome of the condition.

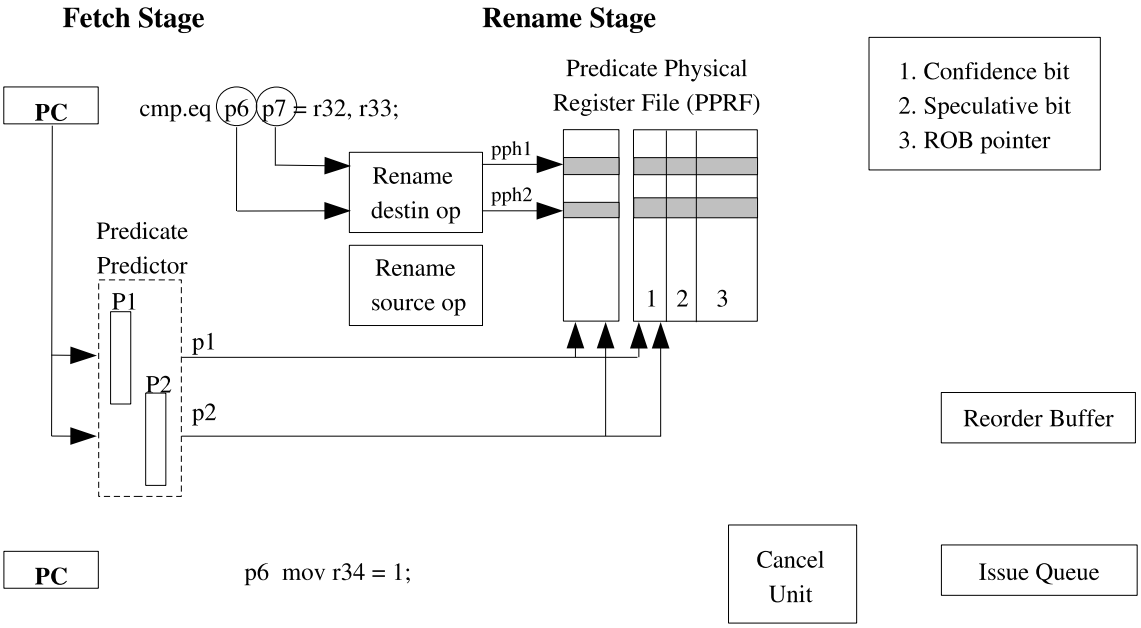

Figure 3.3: **Generation of a predicate prediction**

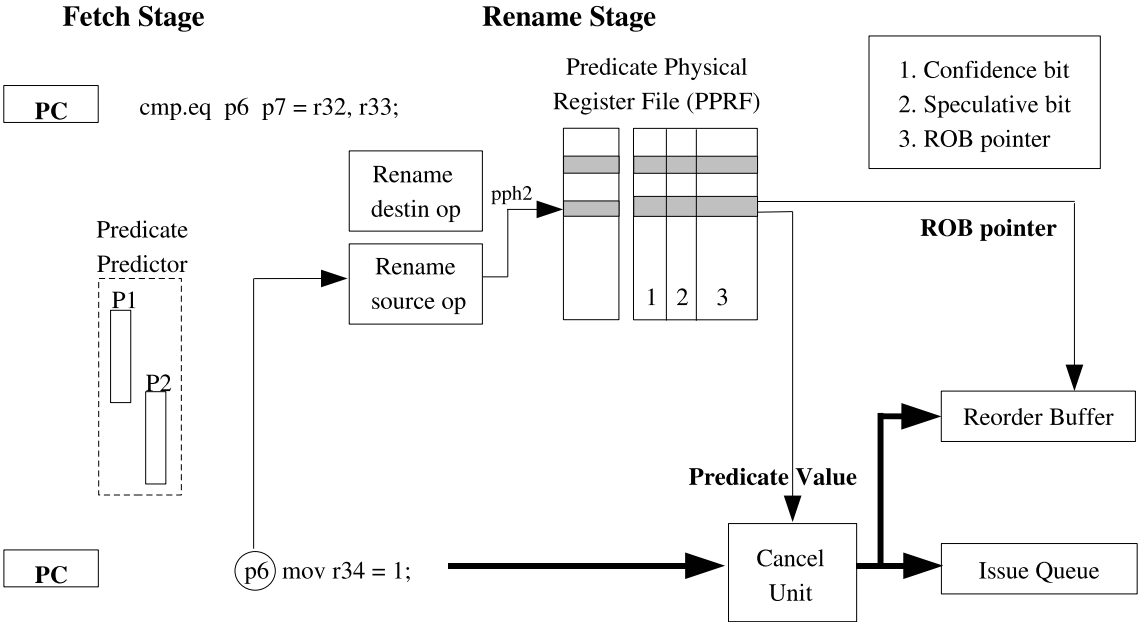

Figure 3.4: **Predicate consumption at the rename stage**

However, for some other types, these results also depend on state information that is not available in the front-end, such as the NaT bits. Thus, it is not possible to infer two predicate values from a single prediction and a comparison type, so two predictions must be generated.

3.2.2 The Confidence Predictor Mechanism

This section adds a confidence predictor [18] to the basic predicate prediction mechanism, which is used to select which if-conversion transformations are worth to be undone, and which are not; i.e., to select which predicates are predicted.

The confidence predictor is integrated with the predicate predictor: each predicate predictor entry is extended with a saturated counter that increments with every correct prediction and is zeroed with every misprediction. A prediction is considered confident if the counter value is saturated, i.e. it equals to a confidence threshold.

When a predicate prediction is generated for a compare instruction, its confidence counter is compared with the confidence threshold, producing a single *confidence bit*. When the compare instruction is renamed, this bit is written into the *confidence* field of the PPRF. Figure 3.3 shows the data paths for prediction and confidence information. Once the predicate is computed at the execution stage, the PPRF entry and the predicate predictor are updated with the computed value and the *speculative* bit is set to false. The confidence counter is updated according to the prediction success.

If a predicated instruction finds the *speculative* and *confidence* bits of its predicate set to true, it is speculated with the predicted value. However, if the confidence bit is set to false, the predicated instruction is converted into a false predicated conditional move. As state in Chapter 2, the new semantic of the *non-confident* instruction can be represented in a C-style form: *result = (predicate)? normal execution : previous register definition.*

Confidence Threshold

The confidence threshold plays an important role in our proposal, since a prediction is considered confident if its saturating counter is equal or higher than it. The confidence threshold value has two important performance effects: on the one hand, high threshold values could result in lost opportunities of correct predictions and increased resource pressure; on the other hand, low threshold values could result in severe performance penalties because of predicate mispredictions. In this section we present a study of different confidence threshold schemes. All the experiments presented in this section use the setup explained in section 3.2.3.

Figure 3.5 depicts the performance impact of different static confidence thresholds, i.e. thresholds determined by a microarchitecture implementation: values 2, 4, 8, 16 and 32. A scheme without confidence predictor has been taken as a baseline. As shown in the graph, the choice of a static confidence threshold does not affect equally to the various benchmarks. While *twolf* has an impressive performance gain with high threshold values, *bzip2* degrades drastically as the threshold increases. This observation suggests that adjusting dynamically the threshold could result in a performance improvement.

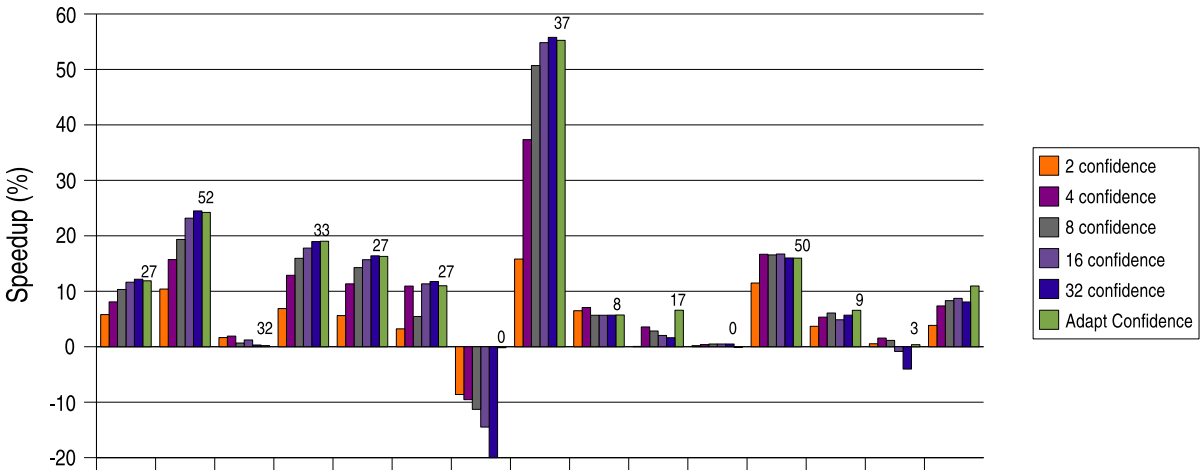

Figure 3.5: **Performance impact of several confidence threshold values. A scheme without confidence has been chosen as baseline.**

Adaptive Confidence Threshold

As explained before, the confidence threshold value strongly depends on the frequency of two events with an important impact on performance:

1. By reducing the threshold, the number of flushes caused by predicate mispredictions increases because of hard-to-predict predictions.

2. By increasing the threshold, the number of predicated instructions that are converted into *false predicated conditional moves* grows because of non-confident predictions.

Note that both events, predicate mispredictions and converted instructions, are closely related. Figure 3.6 shows this relationship: graph (a) depicts the amount of instructions flushed due to predicate mispredictions per committed predicated instruction; graph (b) depicts the amount of committed predicated instructions that have been converted to *false predicated conditional move*. As this figure shows, when the confidence threshold increases, the number of flushes are reduced and the performance may improve. However, such a performance increment may be precluded because of an increment of the number of converted instructions which may derive in an unnecessary resources consumption.

Comparing these results with those in Figure 3.5 shows that different benchmarks may have different performance sensitivity to threshold adjustments. For instance, while the amount of flushes is reduced by almost 100% for *twolf* (the number of flushed instructions decreases from 5,49 to 0,03 per commited predicated instruction), it is only reduced by 20% for *bzip2*, having different performance impact in each case. Note also that for *bzip2* or *art* the number of flushes

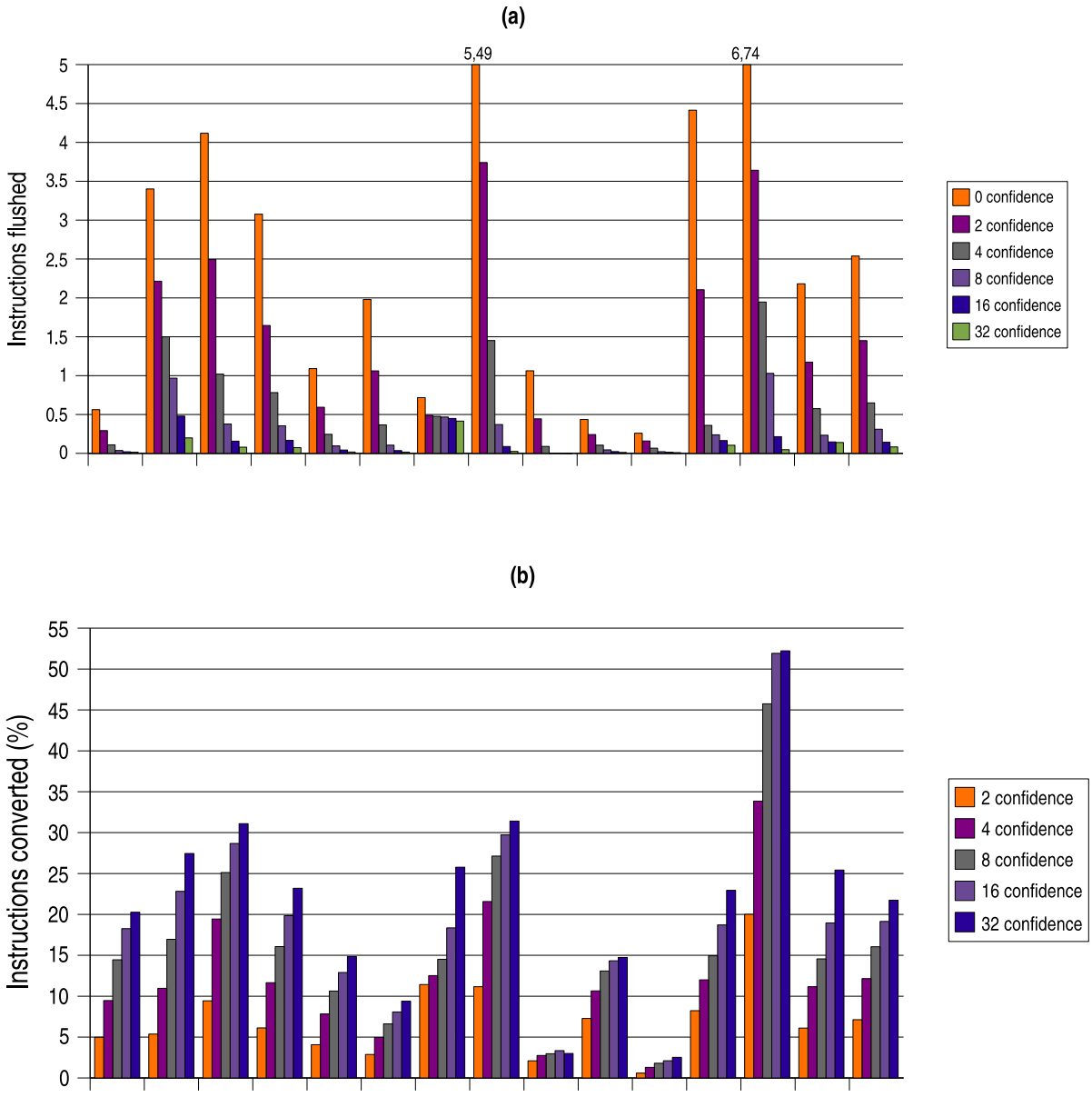

Figure 3.6: **Variance of confidence threshold. (a). Number of flushed instructions per predicated committed instruction. (b) Percentage of predicated committed instructions that have been transformed to *false predicated conditional moves*.**

becomes stable at low thresholds, so the increment of converted instructions beyond this point has an important impact on performance. On the other hand, for *twolf* or *vpr* the number of flushes stabilize at high confidence thresholds, so the impact of converted instructions is lower. For benchmarks such as *sixtrack* the number of flushes stabilize at high thresholds, but the number

converted instructions also increases considerably.

It is important to take into account that both effects, pipeline flushes and converted instructions, do not negatively affect the performance in the same way. Instructions converted to *false predicated conditional moves* increase resource consumption and may stretch the dependence graph (see section 2.2.1). However, when the pipeline is flushed, the processor must recover its architectural state, and the misspeculated instruction and all subsequent instructions must be re-fetched. An adaptive threshold mechanism must trade-off the two effects, but giving more weight to flush reduction because it causes a higher performance degradation.

Our adaptive threshold scheme counts the number of flushes and converted instructions during a fixed period (measured as a fixed number of compare instructions). After that period, the two counts are compared with those obtained in the previous period, and the confidence threshold is modified according to the following heuristic:

1. If the number of flushes decreases, then the number of converted instructions is checked. If it has not increased by more than 1%, then the threshold keeps increasing. Otherwise, the threshold begins to decrease; at this point, the number of flushes is considered a local minimum and is stored.

2. If the number of flushes increases, then it is compared to the last local minimum value. If the difference is less than 1%, then the threshold keeps decreasing. Otherwise, it begins to increase.

Figure 3.5 compares the performance of the adaptive threshold scheme (labeled as *Adapt Confidence*) and the static scheme with different thresholds. It shows that the adaptive scheme avoids the performance degradation of *bzip2* and *apsi*, while maintaining the performance improvement of *twolf* and *vpr*. The number that appears on top of the *adaptive confidence* bar represents the average confidence threshold value. Note that, in almost all cases, the adaptive scheme adjusts the confidence threshold around the value of the best static scheme. In case of *art*, the adaptive threshold scheme can even outperform the best static scheme by 3%, because it may dynamically adjust the threshold to an optimum value that varies along the execution. On average, the adaptive confidence threshold outperforms the best static scheme, i.e. a static scheme with a confidence threshold of 16, by more than 2%.

3.2.3 Evaluation

This section evaluates the performance of our selective prediction scheme with adaptative confidence threshold and compares it with the previous hardware approaches for predicate execution on out-of-order processors, described in section 1.3, and with a perfect predicate prediction scheme. The false predicated conditional move technique described in section 2.2.1 is assumed as the baseline. All the reported speedups are computed with respect to this baseline.

Experimental Setup

All the experiments presented in this section use the cycle-accurate, execution-driven simulator explained in Chapter 2. Specific microarchitectural parameters for the selective predicate prediction scheme are presented in Table 3.1.

Simulator Parameters	
Branch Predictor	Gshare, 18-bit BHR, 64K entries PHT 10 cycles for misprediction recovery
Predicate Predictor	Two predictors of 16KB each. Gshare, 6-bit Global History. 6-bit of Adaptive Confidence Threshold 10 cycles for misprediction recovery
Integer Physical Register File	256 physical registers

Table 3.1: **Simulator parameter values used not specified in Chapter 2.**

We have simulated fourteen benchmark programs from SpecCPU2000 [4] (eight integer and six floating point) using the test input set. All benchmarks have been compiled with IA64 Intel's compiler (Electron v.8.1) using maximum optimization levels.

The most common branch predictor designs and configurations have been evaluated to choose a reasonable baseline. Our experiments show that, for a 2x16KB capacity, a *Gshare* predictor with 6 bits of global history is the most reasonable configuration.

The simulator also models previous proposals presented in section 1.3.3. The select-μops technique [78] has been implemented in detail. Each map table entry has been augmented to record up to six register definitions with their respective guards, and there is an additional dedicated issue queue for select-μops.

The predicate prediction with selective replay technique [12] has not been modeled in detail. Instead, the simulator models a simpler scheme whose performance is an upper-bound for this technique. Instructions guarded with correctly predicted predicates execute as soon its source operands are available. On the contrary, an instruction guarded with an incorrectly predicted predicate is converted to a false predicated conditional move, and it issues exactly as it would do when re-executed in replay mode. In other words, instead of simulating all the re-executions, only the last execution is simulated. This model is more aggressive than the original because it puts less pressure on the execution units. In addition, in our simple model, an instruction leaves the issue queue when it has issued and all its sources and guarding predicates are computed. This makes it also more aggressive than the original because the replay mechanism requires also that all its sources be non-speculative.

Finally the simulator also models a perfect predicate prediction scheme, i.e. one in which the PPRF always contains a confident well-predicted predicate value, for comparison purposes.

Performance Results

Figure 3.7 compares the performance speedup of our selective predicate predictor (labeled as selective prediction) with previous proposals: generation of select-μops and selective replay mechanism (labeled as select-μops and prediction with replay respectively), and a perfect predicate predictor (labeled as perfect prediction). The false predicated conditional move technique has been taken as a baseline. For most benchmarks, our scheme significantly outperforms the previous schemes. On average, it achieves a 17% speedup over the baseline, it outperforms the other

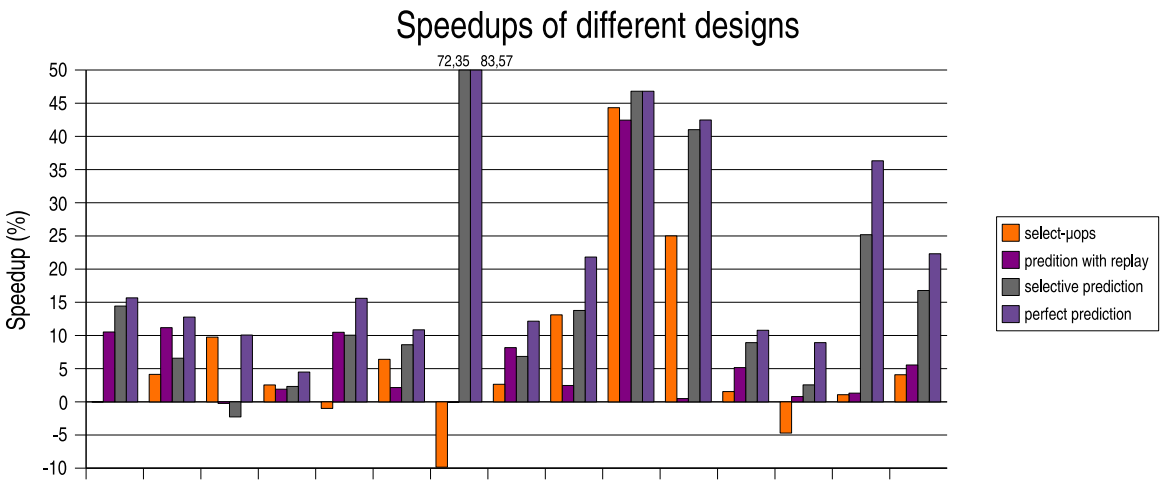

Figure 3.7: **Performance comparison of selective predicate prediction with previous schemes. *False predicated conditional moves* has been taken as a baseline.**

schemes by more than 11%, and it performs within 5% of the perfect prediction scheme.

There are two cases where the upper-bound of the selective replay performs significantly better than selective predicate prediction: *vpr* and *twolf*. For these benchmarks, more than 90% of compare instructions are *unconditional* compare type. However, for benchmarks where this fraction is very small, such as *apsi* (2%), *bzip2* (3%) or *mesa* (1%), the selective replay technique performs much worse. On average, this technique performs 6% better than the baseline. In fact, in order to propagate the correct value in case of predicate misprediction, it creates a new data dependence as the one created by false predicate conditional moves. Hence, this technique is actually converting all predicate instructions to false predicated conditional moves, which is the same as the baseline does. The 6% performance speedup over the baseline is achieved because the selective replay may execute sooner instructions that are guarded with correctly predicted predicates.

The select-μops technique has a bad performance for *bzip2* because of the large number of generated micro-operations per commited instruction (43%). On the opposite side, this technique achieves a good performance for *mcf* which generates only 8% of micro-operations per commited instruction. However, notice that *mcf* has a very low ipc, so this speedup actually translates to a very small gain.

3.2.4 Cost and Complexity Issues

The previous section evaluates our proposal only in terms of performance. However, it is also interesting to consider several cost and complexity issues to compare the evaluated techniques.

First, let us examine in detail the implications of the select-μop technique [78]. Figure 3.8

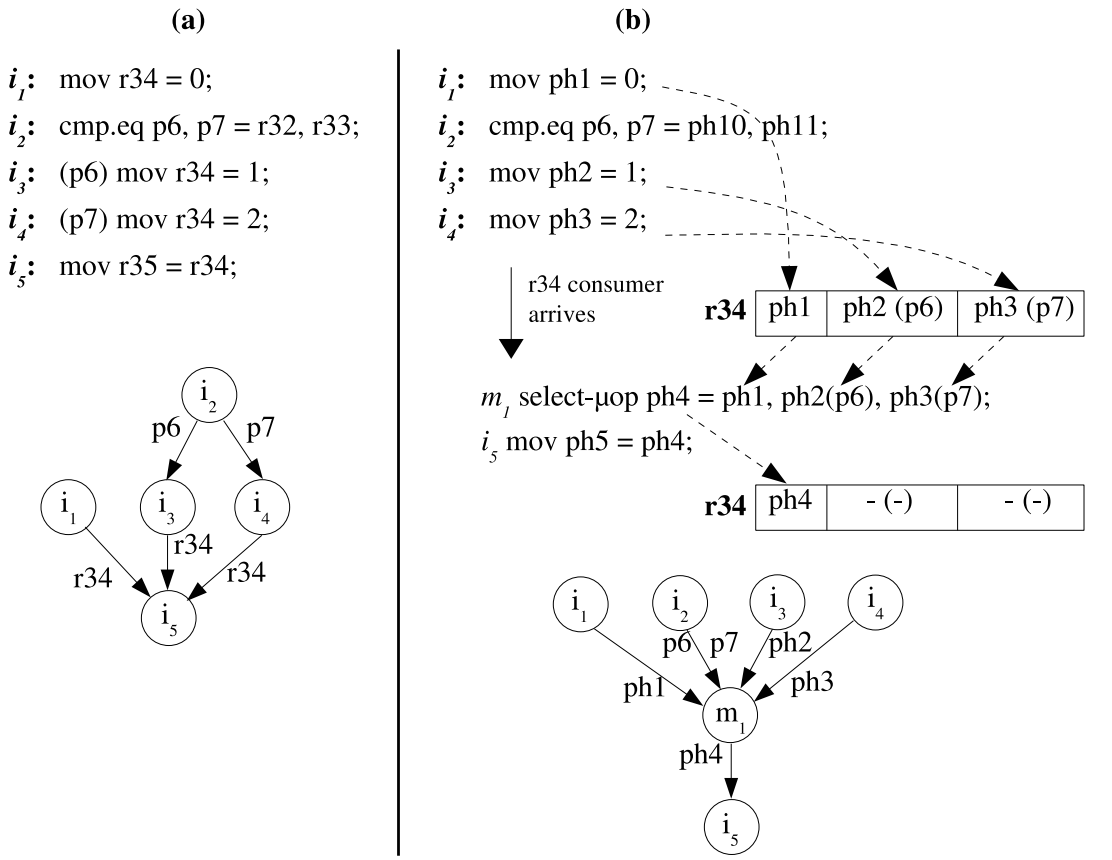

Figure 3.8: **Generation of select-μops at the rename stage. (a) Original code. (b) The code has been renamed and the *r34* map table entry modified. When the instruction *i5* consumes *r34*, a select-μop is generated**

shows how the select-μop works. Figure 3.8a shows the original code before renaming. Figure 3.8b, shows the same code after renaming, as well as the modifications of the *r34* map table entry. Predicated instructions *i3* and *i4* fill the extra map table fields, and are executed as non-predicated. Then, prior to renaming the dependent instruction *i5*, a select-μop is generated which leaves a unique definition for *r34*. Predicated instructions are not serialized between each other, but the select-μop acts like a barrier between predicate instructions and its consumers, as shown in the dependence graph.

The select-μop technique needs some special issue queue entries to hold the micro-operations, with more tag comparators than regular entries. In addition, the map table entries are also extended to track multiple register definitions and their guards. If the implementation supports many simultaneous register definitions, these extensions add a considerable complexity to the rename logic and also to the issue logic. On the other hand, if the implementation supports only a few register definitions, then the mechanism may generate a lot of additional select-μops and performance may degrade.

Moreover, this technique increases the register pressure for two reasons. First, because each

select-μop allocates an additional physical register. Second, because when a predicate instruction commits, it can not free the physical register previously mapped to its destination register as conventional processors do. In fact, this physical register can not be freed until the select-μop that uses it commits. Moreover, instructions guarded with a false predicate are not early-cancelled from the pipeline so they continue consuming physical registers, issue queue entries and functional units.

Next, let us analyse the implications with the selective replay mechanism [12]. The selective replay mechanism needs to track several data flows to avoid re-fetch of wrongly predicted instructions. Each issue queue entry is extended with one extra input tag for every source register, and another for the destination register. These extensions increase considerably the issue logic complexity. For instance, for instructions with two source registers, one destination register and one predicate, the number of comparators needed are six instead of three. Figure 3.9 illustrates the execution of the predicted and the replay data dependence graphs.

The selective replay is based on the IA64 ISA. IA64 is a full predicated ISA, with a wide set of compare instructions that produce predicate values (see Chapter 2). However, Chuang's proposal only predicts one type (the so called *unconditional* [33]). The unconditional comparison type differs from others because it always updates the architectural state, even if its predicate is evaluated to false. The proposed selective replay works fine with these kind of comparison, but cannot handle comparisons that do not update the state when the predicate evaluates to false, because an extra mechanism would be needed to obtain the previous predicate definition. Our experiments show that on average only 60% of compare instructions are unconditional.

Finally, another drawback of the selective replay is the increased resource pressure caused by instructions whose predicate is predicted false, because they remain in the pipeline. In addition, every instruction must remain in the issue queue until it is sure that no re-execution will occur, thus increasing the occupancy of the issue queue. Moreover, the extra input tags required for the replay mechanism significantly increase the complexity of the issue queue.

Our proposal also adds some complexity to the issue logic. Predicated instructions without confidence are converted to false predicated conditional moves, which require an extra input tag for the destination register. If we consider the previous example, instructions with two source registers, one destination register and one predicate, the number of comparators needed are four instead of three. However, since not all instructions need an extra tag, some hardware optimizations may be applied. Our experiments show that only 16% of the predicated instructions are converted. Hence, the complexity increment is lower than that of previous techniques.

3.3 A Predicate Predictor for Conditional Branches

This section extends our previous predicate predictor scheme also for conditional branches, when executing in an out-of-order processor. Like the previous section, this scheme assumes an ISA with full predicate support, such as the one considered in this thesis [33].

Many studies have shown that if-conversion transformations help to eliminate hard-to-predict branches [10, 44, 45, 58, 75]. However, it may also have negative effects in the predictability of the remaining branches [7]. First, the removal of branches may reduce the amount of correlation infor-

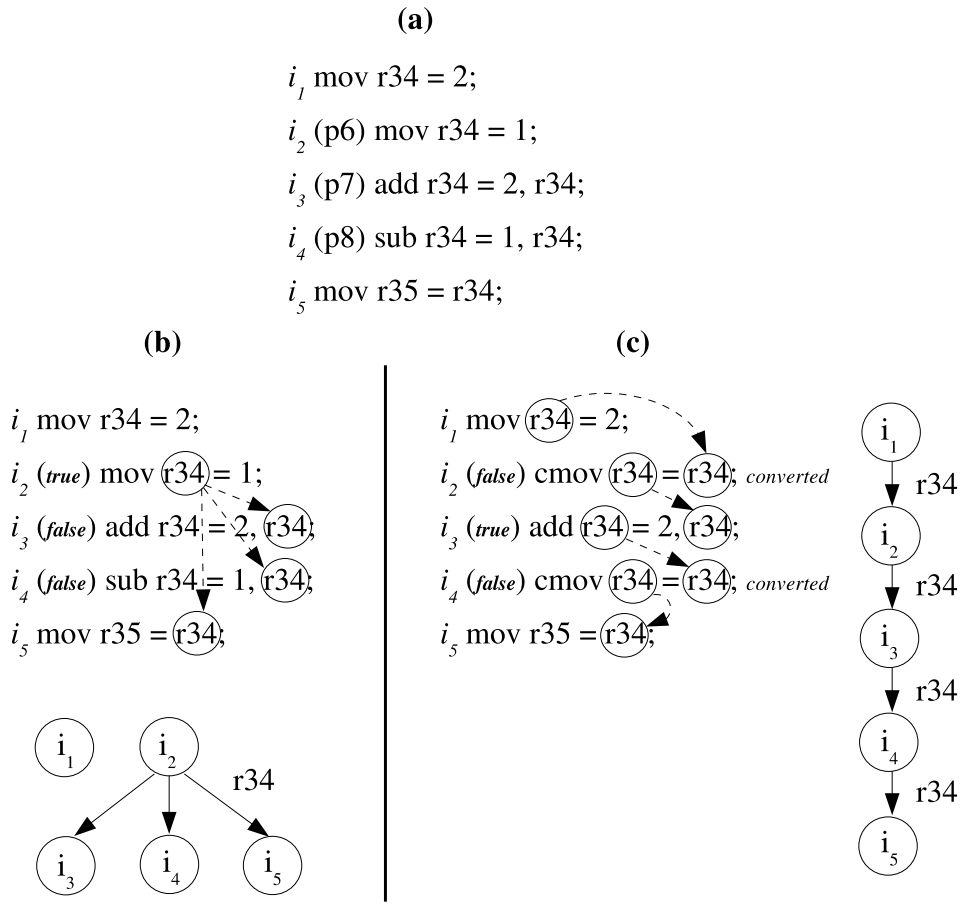

Figure 3.9: **Selective replay. (a) Original code. (b) Data dependences after predicate prediction if no misprediction occurs Assuming *p6 = true, p7 = false* and *p8 = false, i3* and *i4* are inserted into the issue queue but do not issue. (c) Data dependences after predicate misprediction. Assuming *p6 = false, p7 = true* and *p8 = false*, *i2* and *i4* are converted to false predicate conditional move, so the correct *r34* value is propagated through the replay data dependence graph.**

mation available inside branch predictors. This reduction may degrade prediction accuracy, since conventional branch predictors base their prediction on different levels of branch history to establish correlations between them. Second, if-conversion creates code regions where all instructions are guarded with a predicate. This includes unconditional branches that are thereby transformed to conditional branches, so they need to be predicted at fetch. Moreover, these *region-branches* are fetched more frequently than in their original form.

Figure 3.10 illustrates the above mentioned problems with an example. In Figure 3.10a, the unconditional branch *br.ret* executes only if conditions *cond1* evaluates to false and *cond2* evaluates to true. In Figure 3.10b the same code has been if-converted, so the unconditional branch becomes conditional. Since the two previous conditional branches have been removed, their correlation information is no longer available to a conventional branch predictor to predict the new

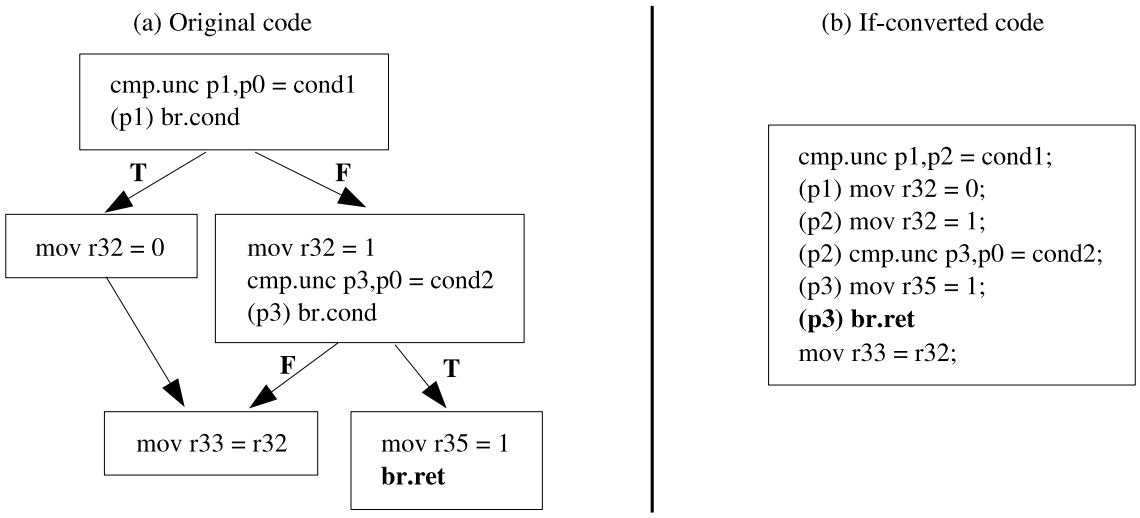

Figure 3.10: **(a) Original code with multiple control flow paths. (b) Multiple control flow paths have been collapsed in a single path. The unconditional branch *br.ret* has been transformed to a conditional branch and it now needs to be predicted. It is correlated with conditions *cond1* and *cond2***

conditional branch.

However, the correlation information associated with the removed branches has not been completely eliminated, since it is still present in the predicate registers that hold the conditions, and it may be associated to the compare instructions that define these predicates, as it will be shown in the following subsections. In our example, a branch predictor could incorporate these predicates to correlate with the prediction of the new conditional branch. Actually, this new conditional branch will be taken if *p1* and *p3* are true, and *p2* is false.

Of course, the use of full correlation information does not guarantee the easy predictability of a branch. The poor predictability of the removed branches may *migrate* to the remaining branches, thus making it useless the recovery of the lost correlation information. In Figure 3.10 the poor predictability of conditions *cond1* and *cond2* may *migrate* to branch *br.ret*.

In the following subsection, we describe in detail our proposed scheme to avoid losing any correlation information of if-converted branches, as well as to exploit early-resolved branches, both resulting in branch prediction accuracy improvements.

3.3.1 Replacing the Branch Predictor by a Predicate Predictor

As stated in section 1.3.2, previous presented studies incorporate predicate information into branch predictors in several ways. August et.al [7] propose to improve a local history branch predictor by correlating with the previous definition of the branch guarding predicate. However, correlation information is not fully recovered, since only the last predicate value definition is used to select and update one of two local histories. Simon et.al [66] propose to introduce recent computed predicates into the Global History Register GHR. Although a higher amount of correlation information is recovered, the effectiveness of the predictor may be reduced due to storing duplicate information,

and it requires a complex mechanism to keep the program order of the GHR. Finally, Kim et.al [39] do not remove branches when applying if-conversion transformations. Instead, those branches are transformed into *wish branches*, so the correlation information is not lost. However, this technique can not exploit early-resolved branches.

Here, we propose to replace the conventional branch predictor with a predicate predictor. In an ISA with a compare-to-branch model, the guarding predicate value of a conditional branch determines its outcome, i.e. if the branch is taken or not. In fact, in such a model, conventional branch predictors use the branch *PC* to predict the value of its guarding predicate, and the result feeds the history registers. In contrast, in our predicate predictor scheme, branches do not take part at all in the generation of predictions. Instead, the predicate predictor uses the *PC* of the compare instruction that produces the guarding predicate of the conditional branch. Note that, instead of predicting the branch *input*, we actually predict the compare *output*.

As explained in section 3.2.1, the predicate prediction is initiated at the fetch of the compare instruction and stored in a predicate physical register at rename stage. Later on, the depending consumer conditional branch will get its predicate prediction from that physical register, so it must be renamed first to find the corresponding location. That is, the physical register name is the unique identifier that binds a predicate producer with its consumers.

Since the prediction starts at the fetch stage with the compare *PC*, and is not stored until the destination predicate is renamed, a multicycle prediction can be performed, i.e., it may be designed as a pipelined large and highly accurate predictor. In addition, since the predictions are not accessed by branches until they reach the rename stage, our predicate predictor becomes the perfect candidate to be implemented in a two-level branch prediction scheme such as the one in the Alpha [38] or the PowerPC [74] processors. Such schemes have two different branch predictors that make two predictions for each branch: the first, fast though less accurate predictor, takes a single cycle and allows the processor to continuously fetch instructions without stalling; the second, slower but highly accurate, takes several cycles and overrides the first prediction. If the two predictions are different, the front-end is flushed and the fetch redirected according to the second prediction.

In addition, our scheme allows to exploit *early-resolved* branches. The predicted and the computed values of a predicate are written into the same physical register. If a compare instruction is scheduled enough in advance from its dependent branch, the branch will use the computed value as a prediction, thus effectively being 100% accurate.

Figure 3.11 illustrates the predicate prediction scheme for conditional branches. As explained in section 3.2.1, two predictions, one for each compare outcome, are generated in early stages of the pipeline, starting with the *PC* of the compare instruction. When the compare instruction is renamed, the two predictions are speculatively written to the PPRF. Later on, when the compare instruction executes, the PPRF is updated with the two computed values. In the example above, *p7* and *p6* are renamed to *pph1* and *pph2* respectively. When a dependent conditional branch reaches the rename stage, it renames its guarding predicate and obtains its input value from the PPRF. The obtained value overrides the first branch prediction performed at fetch stage. In the example above, the branch obtains its prediction from the predicate physical register *pph1*.

The extended PPRF fields, the *speculative* bit and the *ROB pointer*, used for if-converted instructions, are used for conditional branches in the same way. In fact, the *speculative* bit and

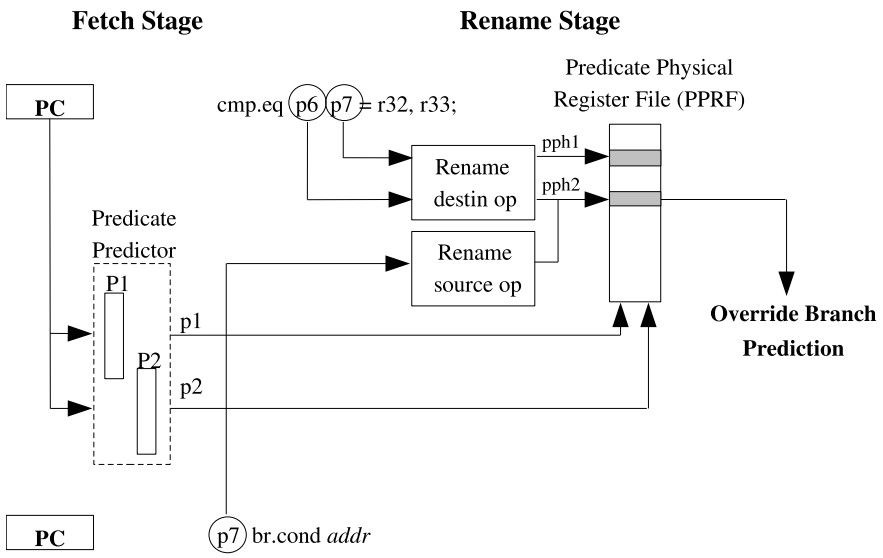

Figure 3.11: **Predicate Prediction scheme as a branch predictor.**

ROB pointer fields is used to advance the detection of a branch misprediction in the same way a predicate misprediction of a if-converted instruction does. Thereby, the branch misprediction detection is performed at compare execution instead of at branch execution. However, our experiments have shown that there is no performance gain at all. The *confidence* bit, which is used by the if-converted instruction to consume or not the predicate prediction, is never used by conditional branches, since the predicate prediction always override the fetch branch prediction.

3.3.2 The Predicate Predictor Implementation

The performance results for if-converted instructions presented in section 3.2.3 have been obtained using a simple Gshare predicate predictor with a 6-bit Global History Register. Although simple, our experiments showed that the Gshare predictor achieves a reasonable trade-off between accuracy and complexity. However, in this new scenario the branch predictor is completely replaced by a predicate predictor. Hence, in-order to compare our predicate predictor scheme with one that uses a conventional branch predictor, in both cases we have implemented a state of the art predictor.

This section gives a justification to the choice of a perceptron predictor for our predicate prediction scheme, and describes how it is adapted to predict predicates instead of branches. Of course, the prediction accuracy of if-converted guarding predicates is also benefited. Our experiments show that the amount of instructions that are converted into a false predicate conditional move are only 7% of the committed predicated instructions.

The Perceptron branch predictor, which is based on neural networks, obtains a very high accuracy for dynamic branch predictions [34]. However, the slow computation time of the prediction function may suppose an important drawback to use perceptrons as a single cycle branch predictor. As explained before, our scheme supports multicycle predicate predictions, so it makes the

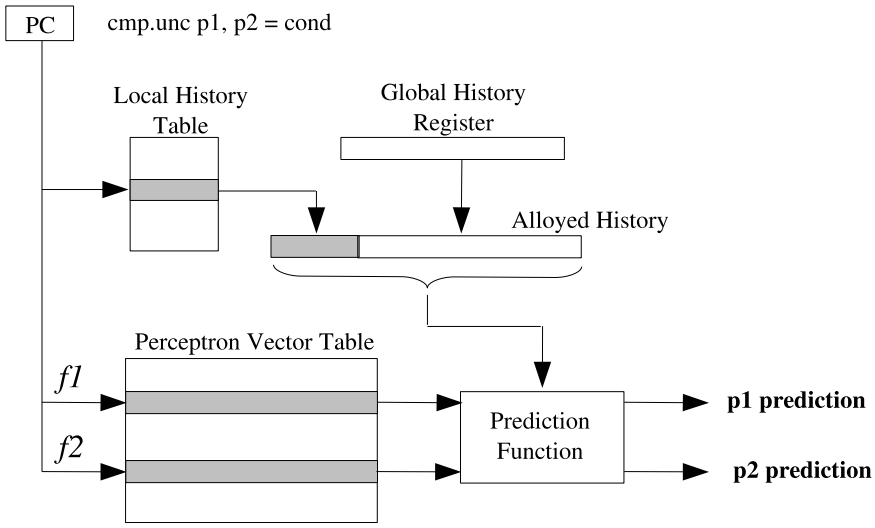

Figure 3.12: **Perceptron Predicate Predictor block diagram.**

perceptron a good candidate.

The original perceptron has been slightly modified to predict predicates more efficiently. As explained before, since compare instructions produce two results, the predicate predictor needs to perform two predictions for each compare instruction. The obvious solution might be to split the Perceptron Vector Table (PVT) to perform the two predictions. However, not all compare instructions produce two useful predicates. In fact, one of the destination predicate registers is often the read-only predicate register *p0*. In this case, only the non-zero predicate register is updated and only one prediction is needed. Having a split PVT table may result in a suboptimal utilization of the available space, producing an increase of aliasing conflicts. Instead, we use an unique PVT table that is accessed with two different hash functions, one for each predicate, so the prediction vectors are better given out.

The accuracy of a predicate predictor is also affected negatively by global history corruption. On a conventional branch predictor, processor state recovery is done by the same instruction that speculatively updates it [67]. Instead, on a predicate prediction based scheme, the global history is speculatively updated by a compare instruction while processor state recovery is done by its predicate consumer, i.e. a conditional branch or an if-converted instruction. In this case, a pipeline flush is triggered starting from the predicate consumer instruction. Although the correct global history bit may be corrected during the corresponding recovery actions, compare instructions that may come after the predicate producer and before the predicate consumer have already made their predicate predictions based on a corrupted global history.

Figure 3.12 shows a high level scheme of the predicate perceptron predictor. The PVT is indexed twice using two different hash functions, *f1* and *f2*. The first hash function, that is used when one prediction is needed, indexes the whole PVT using the corresponding *PC* bits. The second hash function, that is used when two predictions are needed, simply inverts the most significant bit of the first hash function.

Architectural Parameters	
Multilevel Branch Predictor	First level: Gshare 14-bit GHR. Total size: 4 KB. 1-cycle access. Second level: Perceptron. 30-bit GHR. 10-bit LHR. Total size :148 KB. 3-cycle access. 10 cycles for misprediction recovery
Predicate Predictor	Perceptron. 30-bit GHR. 10-bit LHR. Total size :148 KB. 3-cycle access. 10 cycles for misprediction recovery
Integer Physical Register File	256 physical registers

Table 3.2: **Simulator parameter values used not specified in Chapter 2.**

3.3.3 Evaluation

This section evaluates the effectiveness of our predicate predictor based branch prediction scheme in terms of branch prediction accuracy and performance.

Experimental Setup

All the experiments presented in this section use the cycle-accurate, execution-driven simulator described in Chapter 2. Specific microarchitectural parameters for our predicate prediction scheme are presented in Table 3.2.

We have simulated twenty-two benchmark programs from SpecCPU2000 [4] (eleven integer and eleven floating-point) using the MinneSpec [40] input set [1]. We have generated two set of binaries. The first set has been compiled without enabling predication techniques (if-conversion and software pipelining), and the second set has been compiled with only if-conversion transformations enabled. In both cases, all benchmarks have been compiled with IA64 Intel's compiler (Electron v.8.1) using maximum optimization levels.

The simulator also models in detail a 144 KB sized PEP-PA branch predictor with 14-bit local history, as described in [7]. This predictor was proposed for an in-order processor and it correlates consecutive predicate definitions with the same logical register name. Since we assume an out-of-order processor, in order to correctly model this predictor, the simulator maintains the state of a logical predicate register file. We assume that the local histories are updated speculatively and correctly recovered on a branch misprediction.

Branch Prediction Accuracy on Non-If-Converted Code

This section analyses the impact on branch prediction accuracy of three features that differ between our scheme and a conventional branch predictor. On the positive side, our scheme eliminates some branch mispredictions by exploiting early-resolved branches. On the negative side, predicate prediction introduces two factors that negatively affect prediction accuracy, as discussed in section 3.3.2. First, it may introduce additional alias conflicts in the prediction tables, because

[1] MinneSpec input set reduces considerabely the initialization period in terms of simulated instructions. This has allowed to increase the set of simulated benchmarks from 14 to 22 in comparison to the work presented in section 3.2.

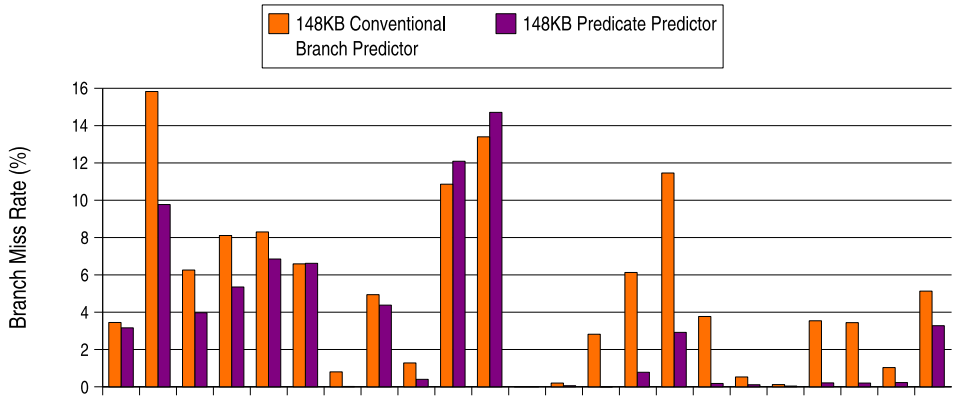

Figure 3.13: **Branch misprediction rates of a conventional branch predictor and our predicate predictor scheme, for non if-converted code.**

some compare instructions produce two predicates. Second, compare instructions that come after a wrong predicate prediction but before the first use of that predicate make predicate predictions based on a corrupted global-history.

In order to isolate these effects from those produced by the correlation improvement of our scheme, this experiment uses the binaries compiled without if-conversion. Figure 3.13 compares the branch misprediction rate when using a conventional branch predictor and our predicate predictor. Both predictors have the same size and latency and analogous configurations. With only three exceptions (*parser*, *bzip2* and *twolf*), the results show that the predicate predictor scheme achieves better accuracy than the conventional branch predictor. On average, it obtains an accuracy increase of 1.86%. This is a significant improvement, since we are using an already highly accurate predictor as a baseline.

The results show that the positive effect of early-resolved branches dominates over the negative effect of increased alias conflicts and global-history corruption, except for three benchmarks, where the net effect is the opposite. In order to evaluate the individual effect of early-resolved branches, isolated from the other two negative effects, we have also simulated idealized branch predictor and predicate predictor schemes, without alias conflicts and with perfect global-history update. As shown in Figure 3.14 an idealized predicate predictor scheme consistently achieves better accuracy for all benchmarks, and on average it increases branch accuracy by 2.24%. Overall, we conclude that the accuracy improvement contributed by early-resolved branches offsets the small negative effects (less than 0.40% on average) of predicate prediction for most benchmarks.

Branch Prediction Accuracy on If-Converted Code

This section evaluates the branch prediction accuracy of our predicate predictor scheme compared to a conventional branch predictor. Since part of the improvement is due to the ability to fully correlate branch global history, it is also compared to the PEP-PA branch predictor (see section 1.3.2),

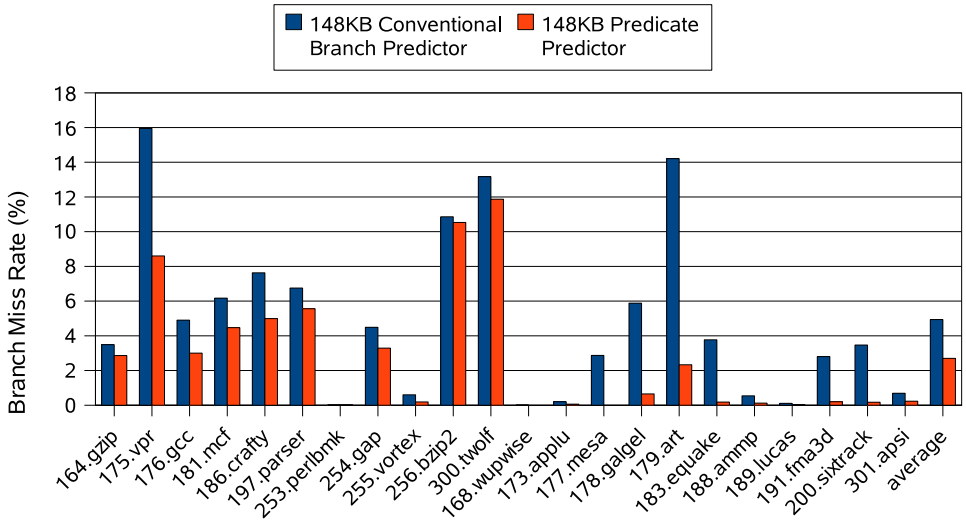

Figure 3.14: **Branch misprediction rates of an idealized conventional branch predictor and an idealized predicate predictor scheme, both without alias conflicts and with perfect global-history update, for non if-converted code.**

---

which addresses a similar goal by incorporating some predicate information. This section also analyses quantitatively the individual contributions to accuracy due to early-resolved branches and correlation improvement. These experiments use the binaries compiled with if-conversion enabled. Hence, their results can not be directly compared to those in the previous section, because different binaries have been used.

Figure 3.15a shows branch misprediction rates for three different branch prediction schemes. The first one is a 144 KB PEP-PA branch predictor. The second and the third schemes are a conventional branch predictor and our proposed predicate predictor respectively, both having a 148 KB size and analogous configurations. With only one exception (*twolf*), the results show that our predicate predictor scheme consistently has the lowest misprediction rate. On average, it obtains an accuracy increase of 1.5% with respect to the best scheme. Surprisingly, the PEP-PA scheme performs worse than the conventional predictor, but it may be produced by the out-of-order writing of the predicate registers, which causes it to choose the local history with a wrong predicate. Note that this scheme was conceived to work on an in-order processor.

Figure 3.15b breaks down the individual contributions of *early resolved branches* and *correlation improvement* to the accuracy difference observed between our scheme and the conventional branch predictor. In order to quantify the contribution of early-resolved branches, we have counted the number of times that the predicate was ready and the conventional branch predictor did a wrong prediction. The remaining accuracy difference is attributed to the correlation improvement. On average, the correlation factor has a higher contribution than early-resolved branches. The accuracy increases are 1% and 0.5% respectively.

Finally, Figure 3.15c shows the performance speedup of our 148KB predicate predictor scheme taking as a baseline the 148KB branch predictor scheme. On average, the branch accuracy increment of 1.5% showed in Figure 3.15a contributes a 4% speedup over the baseline.

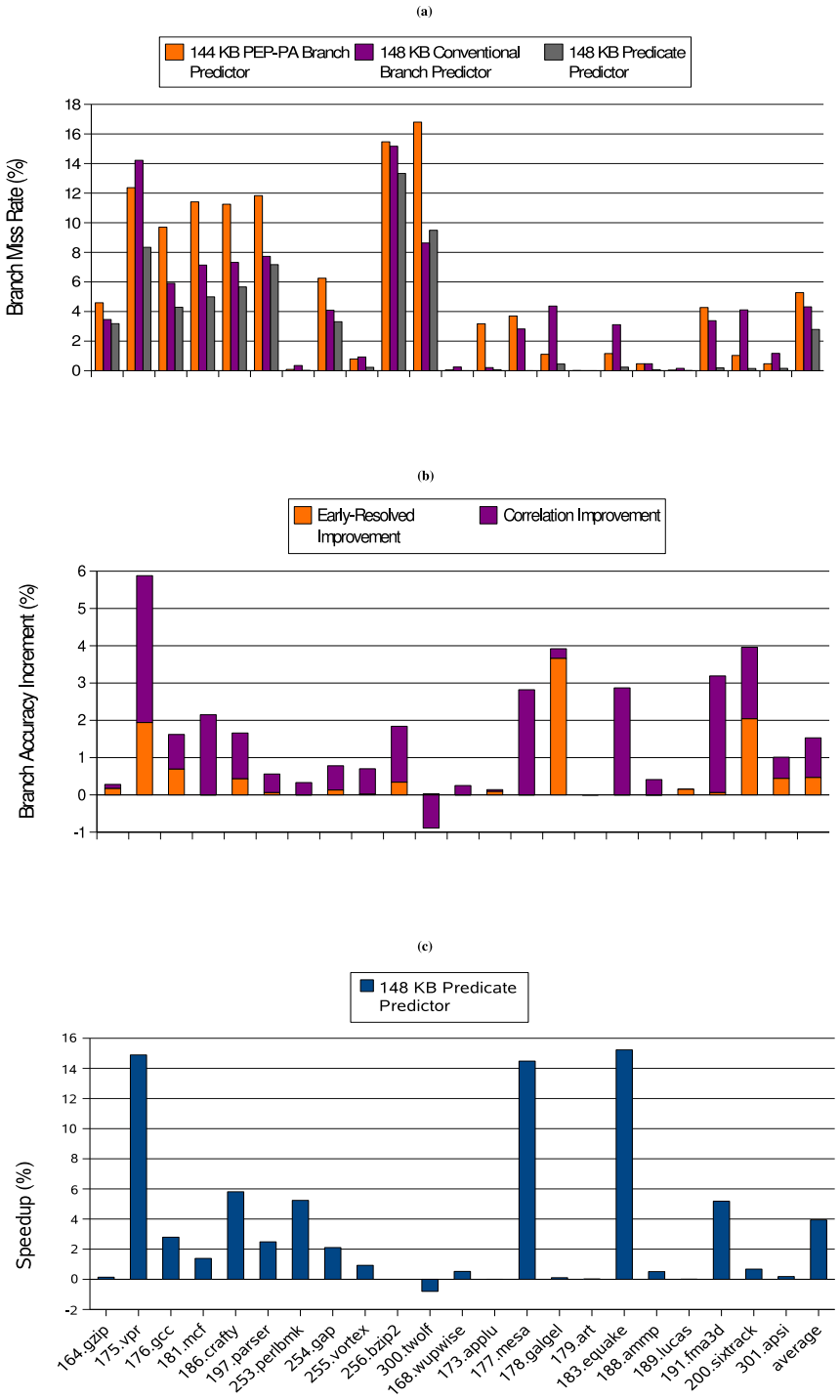

Figure 3.15: **(a) Comparison of branch misprediction rates for if-converted code. (b) Breakdown of the branch prediction accuracy differences between our predicate predictor scheme and a conventional branch predictor. (c) Performance comparison of the 148KB predicate predictor scheme taking as a baseline the 148KB conventional branch predictor scheme.**

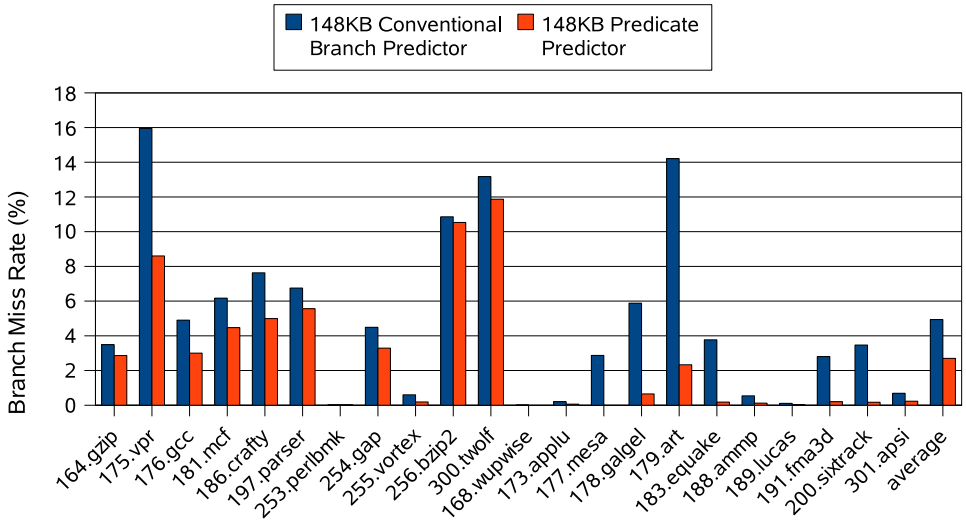

Figure 3.16: **Branch misprediction rates of an idealized conventional branch predictor and an idealized predicate predictor scheme, both without alias conflicts and with perfect global-history update, for if-converted code.**

However, note that the impact of the correlation improvement is actually underestimated in the Figure 3.15b, because these bars includes also the negative effects of the predicate predictor (see section 3.3.2). This explains why this contribution is negative for one benchmark in terms of accuracy and performance speedup(*twolf*). To evaluate separately the positive effects of our scheme over conventional branch prediction, we have repeated the experiment with idealized schemes assuming no alias conflicts and perfect global-history updates. Figure 3.16 shows a consistent accuracy improvement across all benchmarks and an average improvement of almost 2%. Overall, we conclude that the accuracy increases contributed by early-resolved branches and correlation improvement, offset the small negative effect (less than 0.5% on average, which does not have any significant speedup impact) of predicate prediction on all benchmarks but one.

3.4 Summary and Conclusions

If-conversion is a powerful compilation technique that may help to eliminate hard-to-predict branches. Reducing branch mispredictions is specially important for modern out-of-order processors because of their wide and deep pipelines. However, the use of predicate execution by the if-converted code on an out-of-order processor entails two performance problems: *1)* multiple register definitions at the rename stage, *2)* the consumption of unnecessary resources by predicated instructions with its guard evaluated to false. Moreover, as shown in previous works, if-conversion may also have a negative side-effect on branch prediction accuracy. Indeed, the removal of some branches also removes their correlation information from the branch predictor and may make other remaining branches harder to predict.

This thesis proposes a novel microarchitectural scheme that takes into account if-converted and conditional branch problems and provides an unified solution for both. Our proposal is based

on a predicate prediction scheme that replaces the conventional branch predictor with a predicate predictor. The predictions are made for every predicate definition, and stored until the predicate is used by a dependent if-converted or conditional branch instruction.

Our proposal allows a very efficient execution of if-converted code without no branch prediction accuracy degradation. Our scheme does not require adding any significant extra hardware because the second level branch predictor at rename stage is replaced with a predicate predictor. This makes our proposal an extremely low cost hardware solution. Compared with previous proposals that focus on if-converted instructions, our scheme outperforms them by more than 11% on average, and it performs within 5% of an ideal scheme with a perfect predicate predictor. Moreover, our scheme improves branch prediction accuracy of if-converted codes by 1.5% on average, which achieves an extra 4% of speedup.

Chapter 4

Register Windows on Out-of-Order Processors

Register windows is an architectural technique that reduces memory operations required to save and restore registers across procedure calls. Its effectiveness depends on the size of the register file. Such register requirements are normally increased for out-of-order execution because it requires registers for the in-flight instructions, in addition to the architectural ones. However, a large register file has an important cost in terms of area and power and may even affect the cycle time. In this chapter, we propose a software/hardware early register release technique that leverage register windows to drastically reduce the register requirements, and hence reduce the register file cost. Contrary to the common belief that out-of-order processors with register windows would need a large physical register file, this chapter shows that the physical register file size may be reduced to the bare minimum by using this novel microarchitecture. Moreover, our proposal has much lower hardware complexity than previous approaches, and requires minimal changes to a conventional register window scheme. Performance studies show that the proposed technique can reduce the number of physical registers to the number of logical registers plus one (minimum number to guarantee forward progress) and still achieve almost the same performance as an unbounded register file.

4.1 Introduction

Register windows is an architectural technique that reduces the amount of loads and stores required to save and restore registers across procedure calls by storing the local variables of multiple procedure contexts in a large architectural register file. When a procedure is called, it maps its context to a new set of architected registers, called a register window. Through a simple runtime mechanism, the compiler-defined local variables are then renamed to these windowed registers.

If there are not enough architectural registers to allocate all local variables, some local variables from caller procedures are saved to memory and their associated registers are freed for the new context. When the saved variables are needed, they are restored to the register file. These operations are typically referred to as *spill* and *fill*. SPARC [16] and Itanium [25] are two commercial architectures that use register windows.

The effectiveness of the register windows technique depends on the size of the architectural register file because the more registers it has, the less spills and fills are required [61]. Besides, for an out-of-order processor, the number of architectural registers determines the size of the rename map table, which in turn determines the minimum number of physical registers.

To extract high levels of parallelism, out-of-order processors use many more physical registers than architected ones, to store the uncommitted values of a large number of instructions in flight [19]. Therefore, an out-of-order processor with register windows requires a large amount of physical registers because of a twofold reason: to hold multiple contexts and to support a large instruction window. Unfortunately, the size of the register file has a strong impact on its access time [19], which may stay in the critical path that sets the cycle time. It has also an important cost in terms of area and power. There exist many proposals that address this problem through different approaches.

One approach consists of pipelining the register file access [26]. However, a multi-cycle register file requires a complex multiple-level bypassing, and increases the branch misprediction penalty. Other approaches improve the register file access time, area and power by modifying the internal organization, through register caching [83] or register banking [14]. Alternative approaches have focused on reducing the physical register file size by reducing the register requirements through more aggressive reservation policies: late allocation [23] and early release [52] [55] [5].

In this chapter we propose a new software/hardware early register release technique for out-of-order processors that builds upon register windows to achieve an impressive level of register savings. On conventional processors, a physical register remains allocated until the next instruction writing the same architectural register commits. However, procedure call and return semantics enables more aggressive conditions for register release:

1. When a procedure finishes and its closing return instruction commits, all physical registers defined by this procedure can be safely released. The values defined in the closed context are *dead values* that will never be used again.

2. When the rename stage runs out of physical registers, mappings defined by instructions that are not in-flight, which belong to caller procedures, can also be released. However, unlike the previous case, these values may be used in the future, after returning from the current procedure, so they must be saved to memory before they are released.

3. Architectural register requirements vary along a program execution. The compiler can compute the register requirements for each particular control flow path, and insert instructions to enlarge or shrink the context depending on which control flow path is being executed. When a context is shrunk, all defined physical registers that lay outside of the new context contain dead values and can be safely released.

By exploiting these software/hardware early release opportunities, the proposed scheme achieves a drastic reduction in physical register requirements. By applying them to a processor with an unbounded physical register file, the average register *lifetime* (number of cycles between the allocation and release of a physical register) drops by 30% (see Figure 4.1). This allows to reduce the number of physical registers to the minimum number that still guarantees forward progress, i.e. same number of architectural plus one (128 in our experiments for IPF binaries), and still achieve almost the same performance as an unbounded register file.

Contrary to the common belief that out-of-order processors with register windows would need a large physical register file, this chapter shows that register windows, together with the proposed

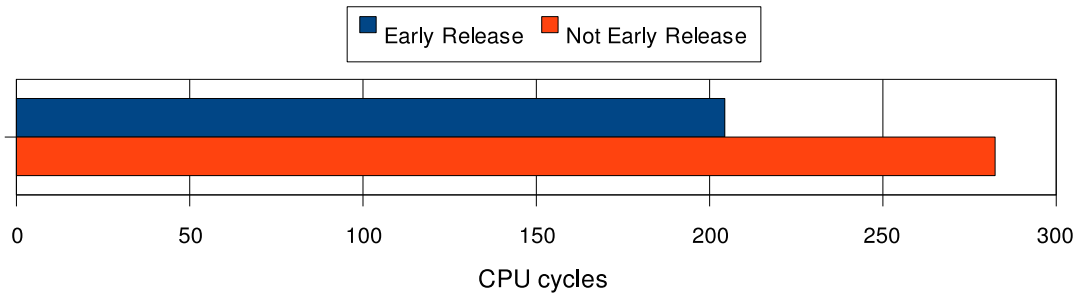

Figure 4.1: **Average lifetime of physical registers for the set of integer benchmarks executing in a processor with an unbounded register file, when applying our early release techniques and when not applying them.**

technique, can significantly reduce the physical register file pressure to the bare minimum. In other words, we show that the proposed scheme achieves a synergistic effect between register windows and out-of-order execution, resulting in an extremely cost-effective implementation.

Besides, our scheme requires much lower hardware complexity than previous related approaches [55], and it requires minimal changes to a conventional register windows scheme.

As stated above, a register windows mechanism works by translating compiler-defined local variables to architected registers prior to renaming them to physical registers. The information required for this translation is kept as a part of the processor state and must be recovered in case of branch mispredictions or exceptions. Our scheme also provides an effective recovery mechanism that is suitable for out-of-order execution.

The rest of the chapter is organized as follows. Section 4.2 describes our proposal in detail. Section 4.3 presents and discusses the experimental results. Section 4.4 analyzes several cost and complexity issues of the proposed solution and previous approaches. Finally, the main conclusions are summarized in section 4.5.

4.2 Early Register Release with Register Windows

This section describes our early register release techniques based on register windows. Our scheme assumes an ISA with full register window support, such as IA-64 [32]. Along this chapter we assume a conventional out-of-order processor with a typical register renaming mechanism that uses a map table to associate architected to physical registers. The IA-64 defines 32 static registers and 96 windowed registers. The static registers are available to all procedures while the windowed registers are allocated on demand to each procedure. Both, static and windowed architected registers map to a unified physical register file. Our technique applies to windowed registers.

In the following subsections, a basic register window mechanism for out-of-order processors is explained first. Then, we expose the early register release opportunities and present a mechanism to exploit them.

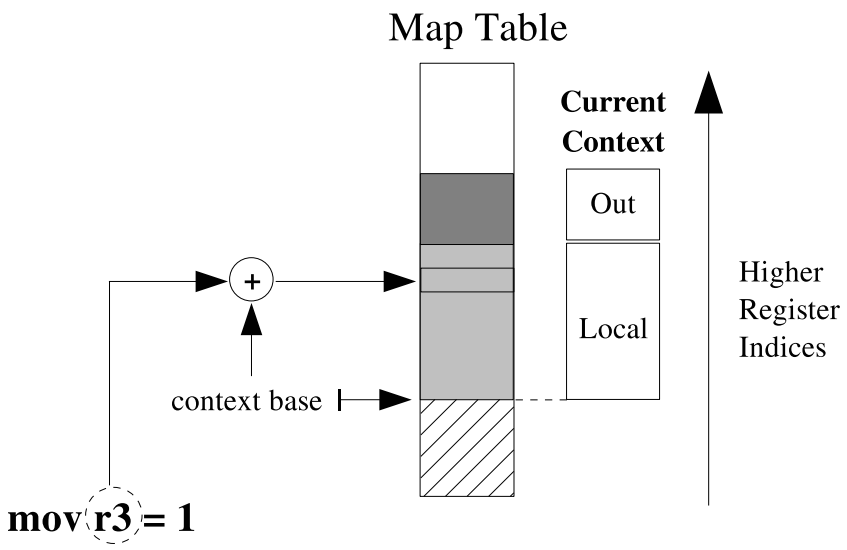

Figure 4.2: **Dynamic translation from virtual register *r3* to its corresponding architectural windowed register.**

4.2.1 Baseline Register Window

Register windows is a technique that helps to reduce the loads and stores required to save registers across procedure calls by storing the local variables of multiple procedure contexts in a large register file [61]. Throughout this chapter we will use also the term procedure context to refer to a register window.

From the ISA's perspective, all procedure contexts use the same 'virtual' register name space. However, when a procedure is called, it is responsible for dynamically allocating a separate set of consecutive architected registers, a register window, by specifying a *context base pointer* and a window *size*. Each virtual register name is then dynamically translated to an architected register by simply adding the base pointer to it (see Figure 4.2). Notice that register windows grow towards higher register indices.

Every register window is divided into two regions: the *local region*, which include both input parameters and local variables, and the *out region*, where it passes parameters to its callee procedures. By overlapping register windows, parameters are passed through procedures, so those registers holding the output parameters of the caller procedure become the local parameters of the callee. The overlap is illustrated in Figure 4.3.

Hence, every register window is fully defined by a *context descriptor* having three parameters: the *context base pointer*, which sets the beginning of the register window; the *context size*, which includes the local and out regions; and the *output parameters*, which defines the size of the out region.

Register windows are managed by software, with three specific instructions: *br.call, alloc* and *br.ret*. Br.call branches to a procedure and creates a new context, by setting a context descriptor with its base pointing to the first register in the out region of the caller context, and its context

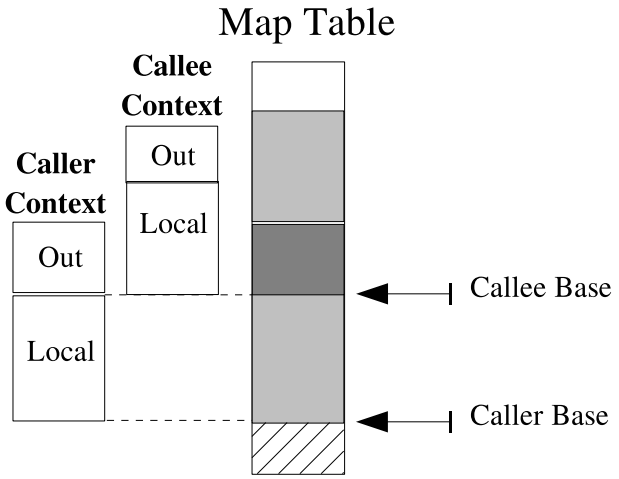

Figure 4.3: **Overlapping Register Windows.**

size equal to the out region's size. Alloc modifies the current context descriptor by setting a new context size and output parameters. Note that the new size may be larger or smaller than the previous size, so this instruction may be used either for enlarging or shrinking the context. Finally, br.ret returns to the caller procedure and makes its context the current context. The compiler is responsible for saving, on each procedure call, the caller context descriptor to a local register and restoring it later on return.

It may happen that an alloc instruction increases the window size but there is no room available to allocate the new context size in the architectural register file, i.e. at the top of the map table. In such case, the contents of some mappings and their associatived physical register at the bottom of the map table (which belong to caller procedure contexts) are sent to a special region of memory called *backing store*, and then are released. Such operation is called a *spill*. Note that spills produce both free physical registers and free entries in the map table. These entries can then be assigned to new contexts to create the illusion of an infinite-sized register stack [1].

When a procedure returns, it expects to find its context in the map table, and the corresponding values stored in physical registers. However, if some of these registers were previously spilled to memory, the context is not completely present. For each missing context register, a physical register is allocated and renamed in the map table. Then, the value it had before spilling is reloaded from the backing store. Such operation is called a *fill*. Subsequent instructions that use this value are made data dependent on the result of the fill so they are not stalled at renaming to wait for the fill completion.

The relationship between map table and backing store is shown in Figure 4.4. Notice that the map table is managed as a circular buffer. Spill and fill operations will be discussed in more detail in section 4.2.6.

[1] Each mapping has a unique associated backing store address.

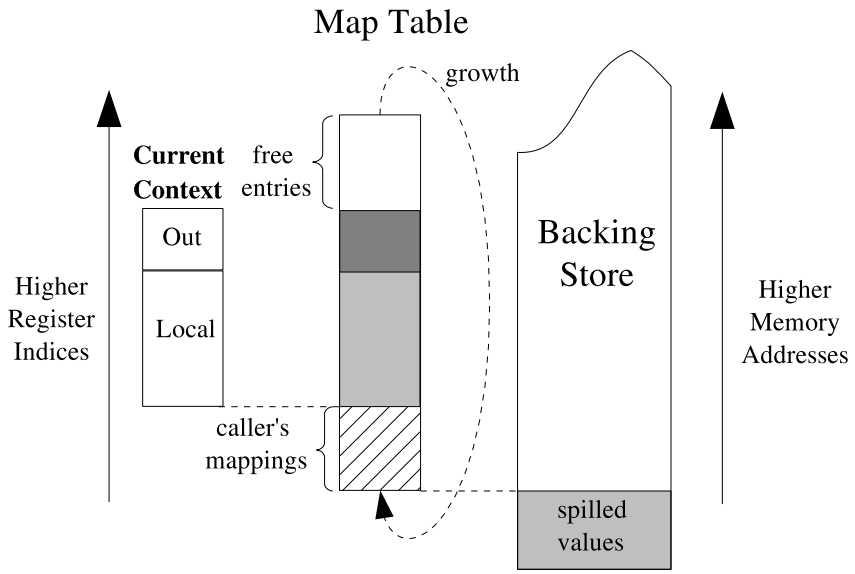

Figure 4.4: **Relationship between Map Table and Backing Store memory.**

4.2.2 Early Register Release Techniques with Compiler Support

On a conventional out-of-order processor, a physical register is released when the instruction that redefines it commits. To help reduce the physical register pressure, we have observed that by shrinking the context size, mappings that lay above the new shrunk context can be early released. We have identified an early release opportunity: the *Alloc Release*.

Alloc Release. Architectural register requirements may vary along the control flow graph. At compile-time, if a control flow path requires more registers, the context can be enlarged by defining a higher context size with an alloc instruction. At runtime, new architectural registers are reserved at rename stage, as explained in section 4.2.1. Analogously, if the architectural register requirements decreases, the compiler can shrunk the context by defining a lower context size with an alloc instruction. At runtime, when the alloc instruction commits, none of the architectural registers that lay above the shrunk context is referenced by any of the currently in-flight instructions. We refer to them as *not-active* mappings, and they can be early released as well as their associated physical registers without having to wait until the commit of a subsequent redefinition. In both cases (to enlarge or to shrink the context size) the compiler needs to introduce an alloc instruction. Figure 4.5 shows a control flow graph with unbalanced architectural register requirements. Alloc Release is illustrated in Figure 4.6a.

Compiler support may help to reduce the physical register pressure by shrinking contexts and releasing not-active mappings that lay above the shrunk context. In fact, what this technique actually does is to adapt better the architectural registers requirements. However, physical register requirements are not available at compile time. To overcome this limitation we propose two hardware techniques called *Context Release* and *Register Release Spill*.

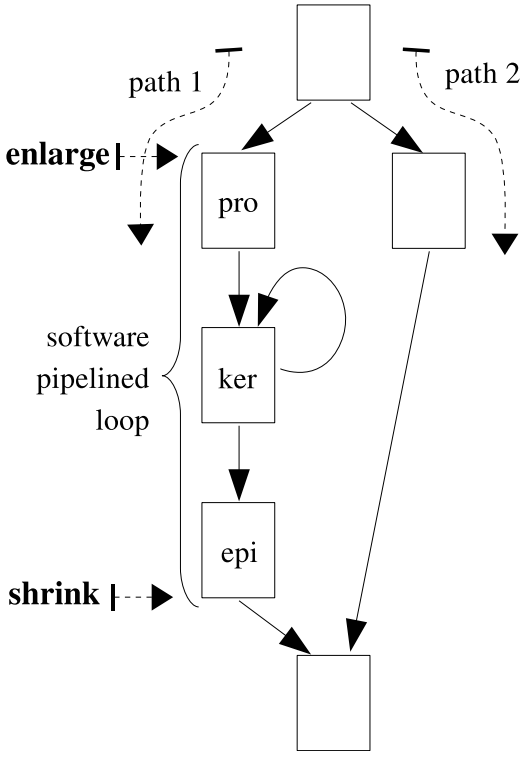

Figure 4.5: **Architectural register requirements are higher in *path 1* than *path 2*.**

4.2.3 Early Register Release Techniques without Compiler Support

Context Release takes full advantatge of call conventions. When a procedure finishes and its closing br.ret instruction commits, many of the physical registers defined inside the closed procedure become not-active and can be early released. Context Release is illustrated in Figure 4.6b. Note that this technique is very similar to Alloc Release, since br.ret also shrinks the current context size. However, although both techniques require the same hardware support, the compiler is not aware of applying Context Release.

Register Release Spill. This technique is based on an extension of the not-active mapping concept: if none of the instructions of a procedure are currently in-flight, not all mappings of the procedure context need to stay in the map table. Hence not-active mappings are not only those mappings that lay above a shrunk context, but also those mappings that belong to previous caller procedure contexts not currently present in the processor. The Register Release Spill technique is illustrated in Figure 4.7. As shown in Figure 4.7a, when *br.call* commits, context 1 mappings become not-active. This is not the case in Figure 4.7b, where the context 1 procedure has already returned before br.call commits, so their mappings have become active. Hence, only not-active mappings from Figure 4.7a can be early released. However, since these mappings belong to caller procedures, they contain live values and they must first spill the content of their associated physical registers before releasing them. We will refer to this operation as Register Release Spill.

Though beneficial for register pressure, this technique increases the amount of spill/fill operations. Our experiments have shown that a blind application of the Register Release Spill to the

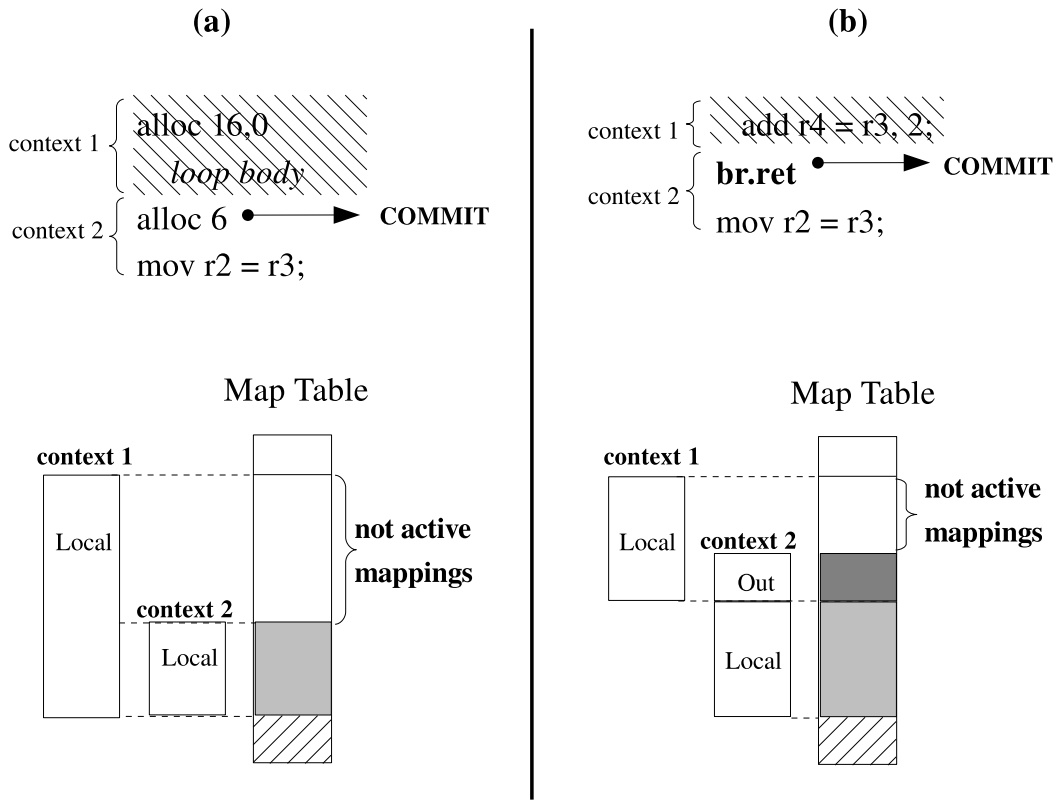

Figure 4.6: **In both figures, context 1 mappings that do not overlap with context 2 are *not-active* (none in-flight instructions refer them), so they can be released. (a) *Alloc-Release* technique, (b) *Context-Release* technique**

baseline register window scheme increases the amount of spill/fill operations from 1.1% to 7% of the total number of commited instructions. Since spilling registers has an associated cost, it could reduce the benefits brought by register windows. Hence, Register Release Spill is only triggered when the released registers are actually required for the execution of the current context, i.e. if the renaming runs out of physical registers. Our experiments have show that this policy increases only spill/fill operations to 1.4% of the total number of commited instructions.

Notice that there is a slight difference between the Register Release Spill and the conventional spill used in the baseline (see section 4.2.1). A conventional spill is triggered by a lack of mappings when the window size is enlarged. What Register Release spill actually does is ”to steal” physical registers from caller procedures at runtime to ensure an optimum execution of the current active zone.

4.2.4 Implementation

To implement register windows in an out-of-order processor, we propose the *Active Context Descriptor Table* (ACDT). The ACDT tracks all uncommitted context states to accomplish two main purposes: to identify the active mappings at any one point in time, which is required by our early register release techniques; and to allow precise state recovery of context descriptor information in

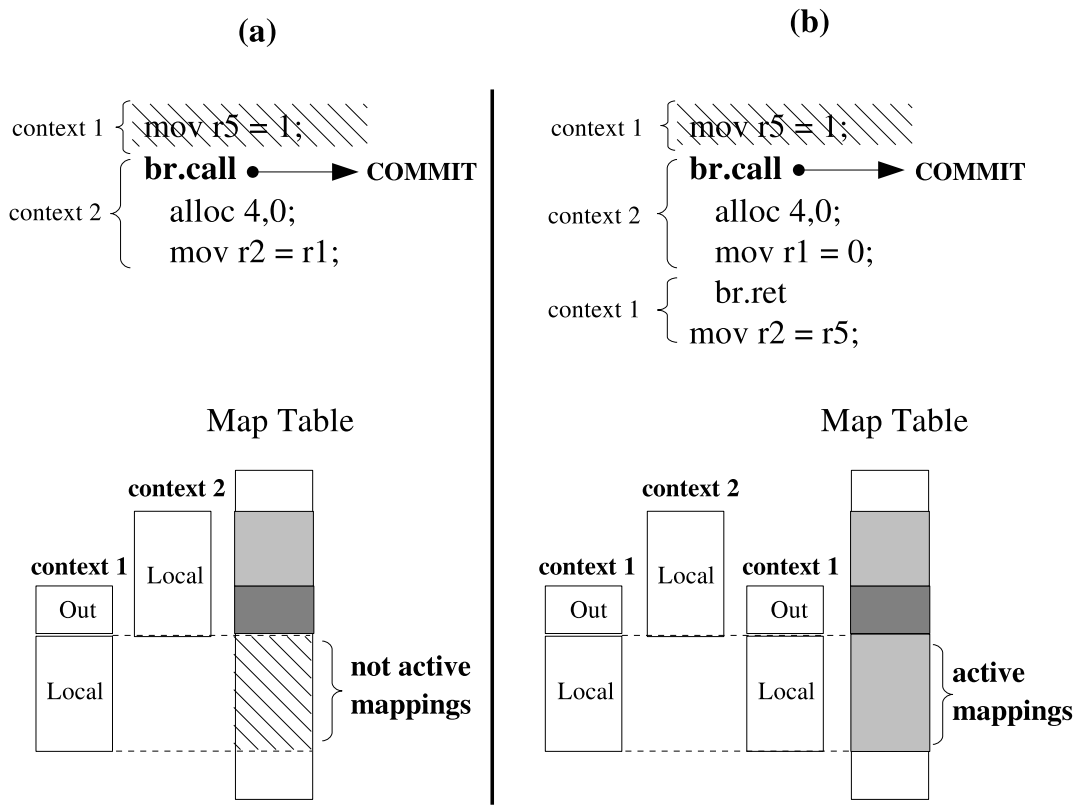

Figure 4.7: ***Register-Release Spill*** **technique (a) Context 1 mappings are *not-active* so they can be early released. (b) Context 1 mappings are *active* (after returning from the procedure) so they can not be released.**

case of branch mispredictions and other exceptions. Moreover, we introduce the *Retirement Map Table* that holds the rename map table at commit stage. It allows that Alloc Release and Context Release techniques identify at commit stage the not-active mappings that can be released, even when these mappings have been already redefined at rename stage by another procedure context.

Active Context Descriptor Table (ACDT). The ACDT acts as a checkpoint repository that buffers in a *fifo* way, the successive states of the current context descriptor. As such, a new entry is queued for each context modification (at rename stage in program order), and it is removed when the instruction that has inserted the checkpoint is committed. Hence, ACDT maintains only active context descriptors. With these information, the ACDT allows precise state recovery of context descriptor information: in case of branch mispredictions and other exceptions, when all younger instructions are squashed, the subsequent context states are removed from the ACDT, too.

As they are defined, active mappings stay in a continuous region on the map table, called *active region*. The active region is bounded by two pointers that are calculated using the ACDT information: the *lower active* and the *upper active*. Both pointers are updated at each context modification using the information present in the ACDT: the *lower active* equals to the minimum context base pointer present in the ACDT, and the *upper active* equals to the maximum sum of

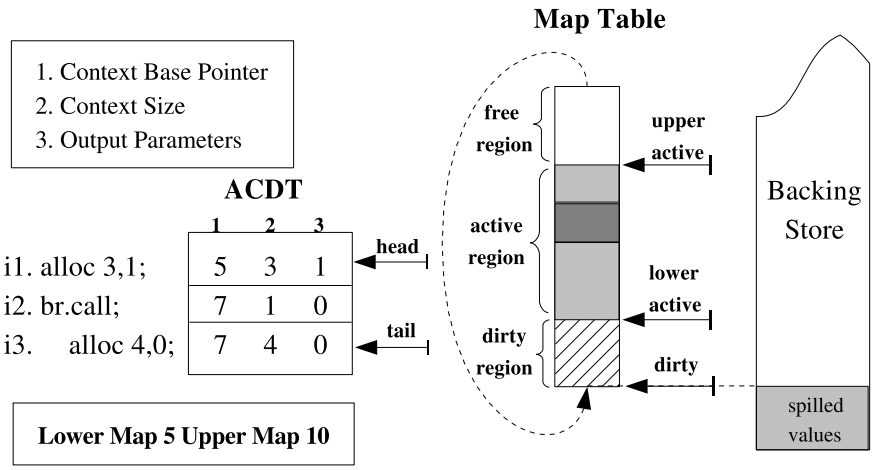

Figure 4.8: **The pointers *upper active, lower active* and *dirty* divide the map table in three regions: *free, active* and *dirty*. *Upper active* and *lower active* pointers are computed using ACDT information. *Dirty* pointer points to the first non-spilled mapping.**

base + size present in the ACDT.

The mappings staying below the *lower active* pointer belong to not-active contexts of callers procedures. These mappings may eventually become free if their values are spilled to memory, or may become active again if a br.ret restores that context. They form the so called *dirty region* of the map table, and it lays between the *lower active* pointer and a third pointer called *dirty*. The *dirty* pointer equals to the first mapping that will be spilled if the renaming engine requires it. Actually, the associated backing store address of the *dirty* pointer corresponds to the top of the backing store stack.

Finally, the *free-region* lays between the *upper active* pointer and the *dirty* pointer (note that the map table is managed as a circular buffer), and it contains invalid mappings. The three regions are shown in Figure 4.8.

Retirement Map Table. When a context is shrunk, mappings that become not-active are not always accessible. As shown in Figure 4.9, the alloc *i4* has increased the size of the context, creating a new context descriptor (*context 3*), just before the alloc *i2*, that has shrunk the the context and created the context descriptor *context 2*, has commited. However, mappings from *context 1* can not be release since their corresponding map entries have been already assigned to *context 3*, which is active. Notice that, although *context 1* mappings have become active, their associated physical registers contain dead values and can be early released.

In order to obtain *context 1* defined physical registers, we introduce the *Retirement Map Table* [26, 54], that holds the state of the *Rename Map Table* at commit stage. When an instruction commits, the Retirement Map Table is indexed using the same Rename Map Table register index, and updated with its corresponding destination physical register (which indicates that the physical register contains a commited value). When a branch misprediction or exception occurs, the offending instruction is flagged, and the recovery actions are delayed until this instruction is about to commit. At this point, the Retirement Map Table contains the architectural state just before the

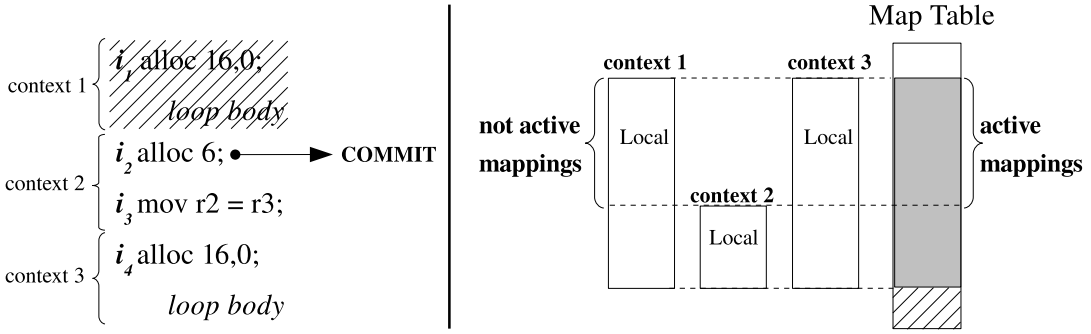

Figure 4.9: **Not-active mappings from *context 1* have become active because of *context 3*, so they can not be released if *context 2* commits**

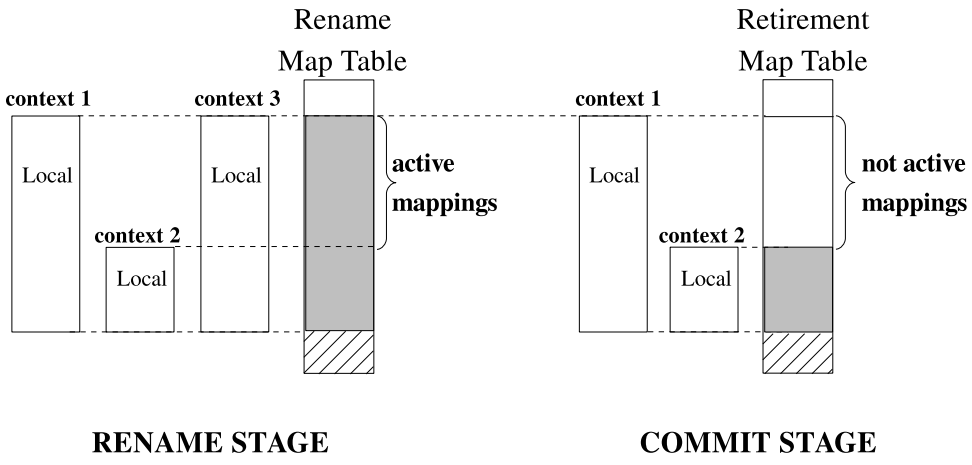

Figure 4.10: **The *Retirement Map Table* holds the commited state of the *Rename Map Table*. *Context 1* mappings will remain into the Retirement Map Table until *Context 3* commits.**

exception occurred. Then the processor state is restored by copying the Retirement Map Table to the Rename Map Table (to avoid significant penalty stalls caused by the use of Retirement Map Table, some optimizations have been proposed [5]).

Figure 4.10 shows the same previous example but introducing the Retirement Map Table. Note that when the alloc that has created the context descriptor *context 2* commits, not-active mappings from *context 1* are still present in the Retirement Map Table, so they can be safely early released.

In summary, when a shrunk context commits, Alloc Release and Context Release techniques are applied by releasing those not-active mappings from the Retirement Map Table that have layed above the commited shrunk context. Furthermore, when the rename stage runs out of physical registers, a Register Release Spill operation is triggered and it releases as many not-active mappings as needed from the *dirty region* starting at the *dirty* pointer. These spilled mappings become part of the *free region*.

4.2.5 Delayed Spill/Fill Operations

A closer look to the register window scheme for out-of-order processors described in the previous sections, reveals that there is still some room for physical register pressure reduction.

First, when a br.ret instruction is executed and the restored context is not present in the map table, fill operations are generated until all its mappings are restored. Assuming a realistic scenario, there may exist a limit on the number of fills generated per cycle. So a massive fill generation may stall the renaming for several cycles. Moreover, it may happen that some of these mappings will never be used. Actually, it occurs quite often that a br.ret is followed by another br.ret, so none of the restored mappings are used. Hence, unnecessary fill operations reserve unnecessary physical registers, which means a higher register file pressure. Hence, we propose to defer each fill and its corresponding physical register allocation until the mapping is actually used by a subsequent instruction. By delaying fill operations we achieve a lower memory traffic and we reduce the physical register pressure which results in a higher performance.

Second, when an alloc is executed and there is not enough space into the map table, spill operations are generated to free the required mappings. As it happens with the previous case, a massive spill generation may stall the renaming for several cycles. Hence, we propose to defer each spill until the map table entry is actually reassigned by a subsequent instruction.

Hence, when a br.ret is executed and the restored context is not present in the map table, the required mappings from the free region are appended to the active region, and marked with a *pending fill* bit, so the br.ret instruction is not stalled. When a subsequent dependent instruction wants to use a mapping with the pending fill bit set, a fill operation is generated and a new physical register is reserved. In a similar way, when an alloc is executed and there is not enough space for the new context, the required mappings from the dirty region are appended to the active region, and marked with a *pending spill* bit, so the alloc is not stalled. When an subsequent instruction redefines a mapping with its pending spill bit set, the current mapping is first spilled to the backing store.

It might happen that a mapping with the pending spill bit set is not redefined until a subsequent nested procedure, and the map table has wrapped around several times. In that case, it would be costly to determine its associated backing store address. Thus, it is less complex to have an engine that autonomously clear the pending spills that were left behind. This problem does not occur with pending fills because these are actually invalid mappings that may be safely reused.

4.2.6 Spill-Fill operations

This section discusses some implementation considerations about the spill and fill operations.

A naive implementation of spills and fills could insert them into the pipeline as standard store and load instructions. However, spills and fills have simpler requirements that enable more efficient implementations. First, their effective addresses are known at rename time, and the data that the spills store to memory are committed values. Since all their source operands are ready at renaming, they can be scheduled on a simpler hardware. Second, a given implementation could be further optimized by making that the system guarantees that spills and fills do not require memory disambiguation with respect to program stores and loads. In this case, spill and fill operations could be scheduled independently from all other program instructions by using a simple *fifo* buffer [55].

Architectural Parameters	
Fetch Branch Predictor	Gshare 14-bit GHR, 4KB, 1-cycle acces
Predicate Predictor	Perceptron. 30b GHR. 10b LHR Total size :148 KB. 3-cycle access 10 cycles for misprediction recovery
Integer Map Table	96 local entries, 32 global entries
Integer Physical Register File	129 physical registers

Table 4.1: **Simulator parameter values used not specified in Chapter 2.**

4.3 Evaluation

This section evaluates the proposed early register release techniques (Alloc Release, Context Release, Register Release Spill and Delayed Spill/Fill) presented in previous sections, and compares them to a configuration that uses the VCA register windows scheme (see section 1.3). For each configuration, performance speedups are normalized IPCs relative to a configuration that uses the baseline register windows scheme with 160 physical registers. Although this gives an advantage of 32 registers to the baseline, we believe that it is a more reasonable design point for such an out-of-order processor with 128 logical registers. We will show that, although our best scheme has much less registers, not only it outperforms the 160-registers baseline, but it still performs within a 192-registers baseline. At the end of this section we provide a more complete performance comparison for a wide range of physical register file sizes.

4.3.1 Experimental Setup

All the experiments presented in this section use the cycle-accurate, execution-driven simulator explained in Chapter 2. Specific microarchitectural parameters for our predicate prediction scheme are presented in Table 3.2.

We have simulated the integer benchmark SpecCPU2000 suite [4] (except two integer benchmarks (*eon,vortex*), which the LSE emulator environment could not execute) using the Minne Spec [40] input set. Floating-point benchmarks have not been evaluated because the IA-64 register window mechanism is only implemented on integer registers. All benchmarks have been compiled with IA-64 Opencc [3] compiler using maximum optimization levels. For each benchmark, 100 million committed instructions are simulated. To obtain representative portions of code to simulate, we have used the Pinpoint tool [57].

The simulator models in detail the baseline register windows, as well as our proposed techniques (Alloc Release, Context Release, Register Release Spill and Delayed Spill/Fill), the ACDT mechanism and the Retirement Map table, as described in the previous section. The simulator also models the VCA register windows technique, featuring a four-way set associative cache for logical-physical register mapping, with 11-bit tags, as described in [55], although it was adapted to the IA-64 ISA.

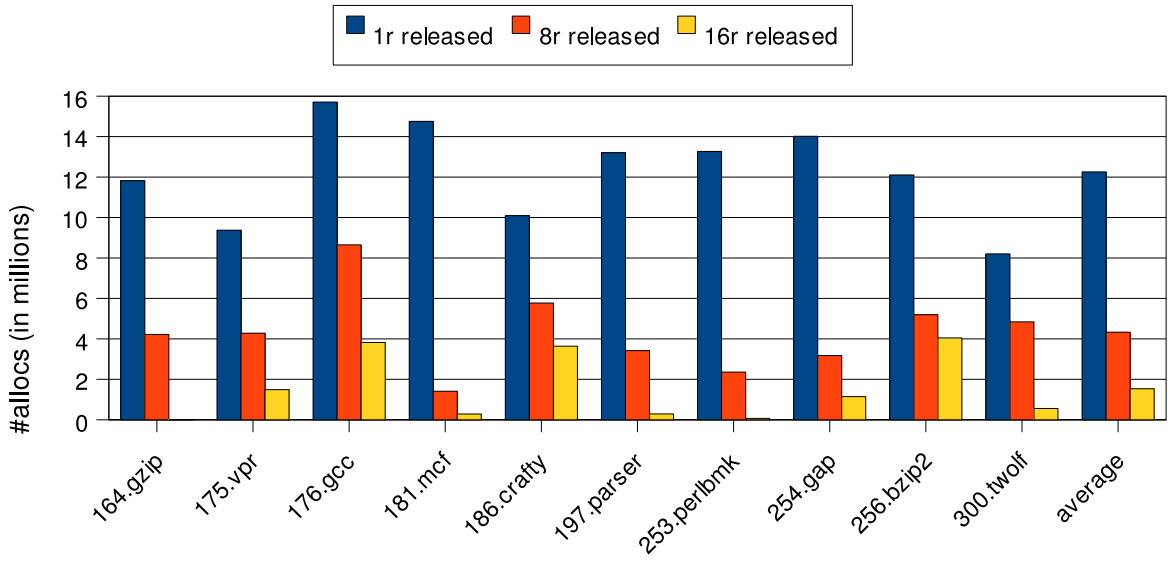

Figure 4.11: **Number of allocs that the compiler should introduce when executing 100 million committed instructions if a perfect shrunk of contexts is performed**

4.3.2 Estimating the Potential of the Alloc Release

The Alloc Release technique releases not required architectural registers (and their associated physical registers) by introducing alloc instructions that shrink the context size (see section 4.2.2). A context can be shrunk by deallocating the mappings that are above the highest architectural register holding a live value. Hence, the compiler should assign the shortest lifetime values to the highest architectural register indexes in order to release them as soon as possible by shrinking its context. However, this register assignment is not trivial, since the same value may have different lifetimes depending on the control flow path. Moreover, the introduction of shrunk allocs increases the program code size, having undesirable side effects on the instruction cache and the fetch bandwidth. Figure 4.11 shows the number of times there is one, eight or sixteen architectural registers that can be released at the beginning of each basic block if the context is perfectly shrunk (i.e. shortest lifetimes are allocated into the highest context indexes), when executing 100 million of committed instructions.

Evaluating the Alloc Release technique requires to compare optimized and unoptimized binaries. For a fair comparison, both binaries should be run until completion, not just running a portion of the execution, because code optimized with Alloc Release is different from a code that does not apply it. Because of the limited resources of our experimental infrastructure, we took an alternative workaround: we attempted to estimate the *potential* performance speedup of this technique, by evaluating an upper bound of the Alloc Release.

Instead of inserting alloc instructions, the compiler annotates the code at the beginning of each basic block with a list of *live* architectural registers at that point. A register is considered *live* in a given point if there is at least one consumer in any possible path downstream. At runtime, when the first instruction of a basic block commits, the simulator releases the physical registers associated

to current context mappings not included in the live register list, as would have occurred with a shrinking alloc instruction. The physical registers are easily identified from the Retirement Map Table.

However, note that by generating simple live register lists, there is no guarantee that the freed registers are located at the highest mappings of the context, to make the context shrink possible. Thus, the upper bound configuration does not attemp to shrink the context but only to free the physical registers that correspond to non live values. This is the effect that would have been seen on the Alloc Release if the non-live registers are always mapped by the compiler to the highest context positions, which is obviously an optimistic assumption.

4.3.3 Performance Evaluation of Our Early Register Release Techniques

This section compares in terms of performance our proposals: Alloc Release, Context Release, Register Release Spill and Delayed Spill/Fill. Each of them, either can be implemented independently, or as a combination of one with the rest. Although a thorough study have been realized, only the best four configurations are presented: (1) Context Release, (2) Alloc Release and Context Release, (3) Register Release Spill and Context Release and (4) Delayed Spill/Fill and Register Release Spill and Context Release. Notice that the four configurations use Context Release. This is because Context Release is a very powerful technique that does not require any binary code modification and it does not carry any side effect, unlike Alloc Release or Register Release Spill. Moreover, it is a low cost technique in terms of hardware.

Alloc Release and Context Release. Figure 4.12 compares configurations (1) and (2), presented in the previous paragraph, in terms of performance. The first column uses only Context Release technique. The remaining three columns are variants of Context Release and Alloc Release configurations, where Alloc Release is applied only if the number of released physical registers is bigger than sixteen, eight or one respectively. On average, applying a non restricted Alloc Release technique, i.e. release a physical register when possible, together with Context Release (fourth column) achieves a performance advantage of more that 3%. However, such a performance improvement does not make up for the amount of alloc instructions that the compile should insert, as shown in Figure 4.11. The restrictive use of Alloc Release (second and third columns) can drastically reduce the code size expansion (from 12 to 2 and 4 millions of generated allocs), but it carries a performance loss of 1.5% and 0.4% respectively in comparison to a non restictive use of Alloc Release.

One could expect a higher performance improvement, but the Alloc Release technique shrinks contexts because the architectural register requirements decreases, and not because of a lack of physical registers. However, there is one benchmark (*twolf*) that achieves a performance advantatge of 16% over Context Release configuration when applying the non restrictive Alloc Release. A detail explanation will be given below.

Register Release Spill and Delayed Spill/Fill. Figure 4.13 compares the performance of configuration (3) (Context Release and Register Release Spill) and (4) (Context Release, Register Release Spill and Delayed Spill/Fill) mentioned above. The use of Delayed Spill/Fill consistently outperforms the Register Release Spill configuration, with average speedups over the baseline of 6% and 4% respectively. Moreover, the number of spill/fill operations generated with Delayed

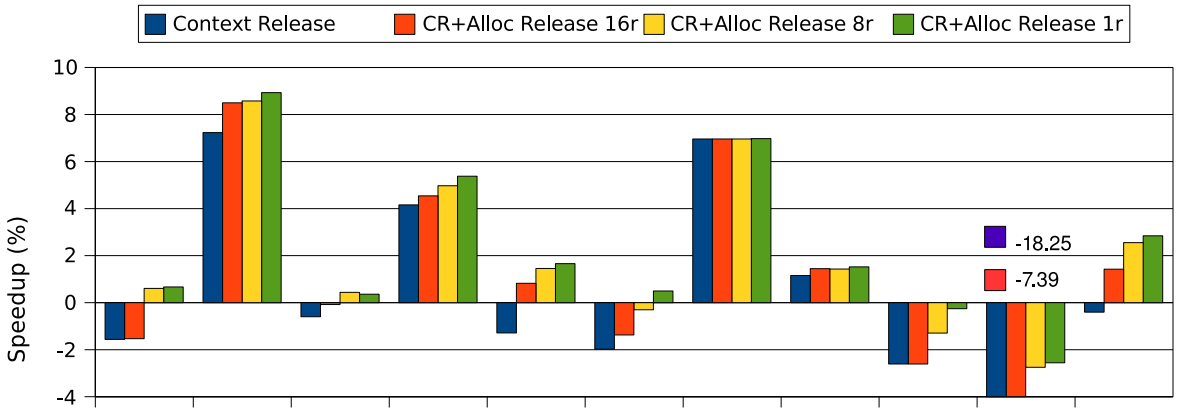

Figure 4.12: **Performance evaluation of Context Release and several configurations with Alloc Release and Context Release. Speedups are normalized to the baseline register window scheme with 160 physical registers.**

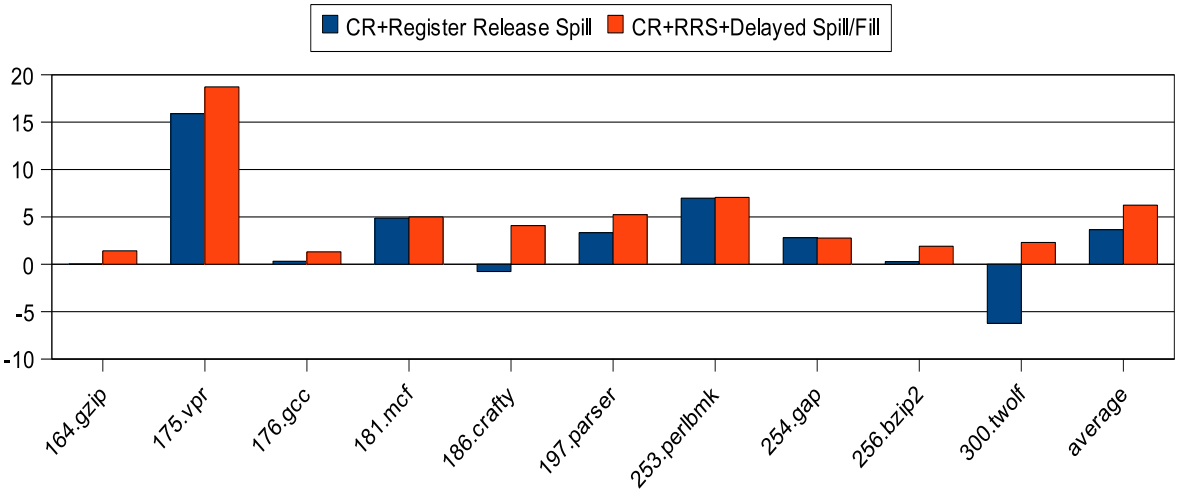

Figure 4.13: **Performance evaluation of Register Release Spill configuration and Delayed Spill/Fill configuration. Speedups are normalized to the baseline register window scheme with 160 physical registers.**

Spill/Fill drops from 7.3% to 2.6% of the total number of committed instructions, and it almost equals the number of spill/fill operations generated for the baseline scheme, that is 2.4% of the total number of committed instructions.

Such a reduction produces notable performance improvements because it not only reduces memory traffic but also the amount of physical registers allocated by fill operations. On average, by using the Delayed Spill/fill technique, fill operations drop from 4.3% to 1.3% of the total

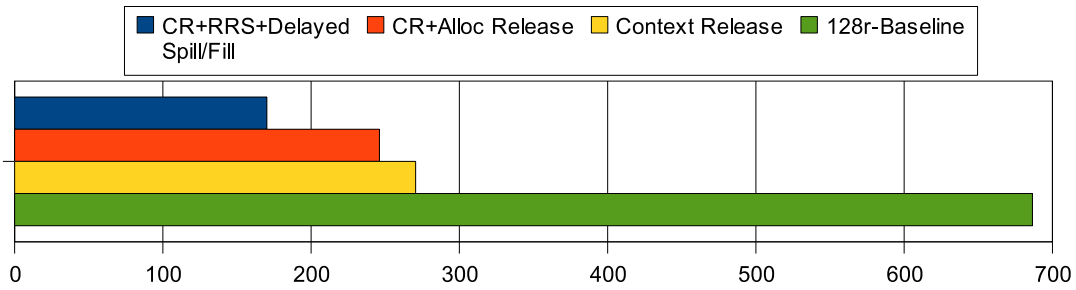

Figure 4.14: **Average lifetime of the set of integer benchmarks executing in a processor with a 128 register file size, when applying different configurations: Delayed Spill/Fill, Alloc Release, Context Release and register window baseline.**

number of committed instructions, in comparison to the Register Release Spill configuration (spill operations generated are reduced from 2.7% to 1.4%). It is especially beneficial for *twolf*, where the performance improvement of Delayed Spill/Fill configuration achieves an advantage of more than 8% over the Register Release Spill configuration. This is because, by using only Register Release Spill, physical registers obtained from dirty region are required again for fill operations when the procedure returns, even though they were not necessary. Actually, it is quite often that a br.ret is followed by another br.ret, so none of the required physical registers are used. Hence, by applying Delayed Spill/Fill technique, only required fill operations (and physical registers) are performed.

A similar effect explains the performance advantage of 16% of *twolf* when using a non restrictive Alloc Release together with Context Release in comparison to Context Release, shown in Figure 4.12. This is because physical registers freed by Alloc Release comes from dead values that do not need to be restored again to the map table. This also explains that a non restrictive Alloc Release configuration outperforms the Register Release Spill configuration by almost 4% for the case of *twolf*, when comparing Figures 4.12 and 4.13 (both use the same baseline). However, the Alloc Release configuration do not adapt as well as the Delayed Spill/Fill configuration to register requirements at runtime. On average, the Delayed Spill/Fill configuration outperforms the Alloc Release configuration by almost 4%.

Such a performance improvement is mainly due to a huge reduction of the average register lifetime. Figure 4.14 shows the average register lifetime of the set of the studied integer benchmarks, when using different configurations: Delayed Spill/Fill, Alloc Release, Context Release and the register window baseline scheme. All configurations execute in a processor with a 128 register file size. Our best proposal, i.e. the Delayed Spill/Fill configuration, reduces the lifetime by 75% over the register window baseline scheme, and by 31% and 37% over the Alloc Release and the Context Release respectively. Hence, by reducing the lifetime, the rename stage stalls caused by a lack of physical registers are also drastically reduced, increasing considerably the processor throughput.

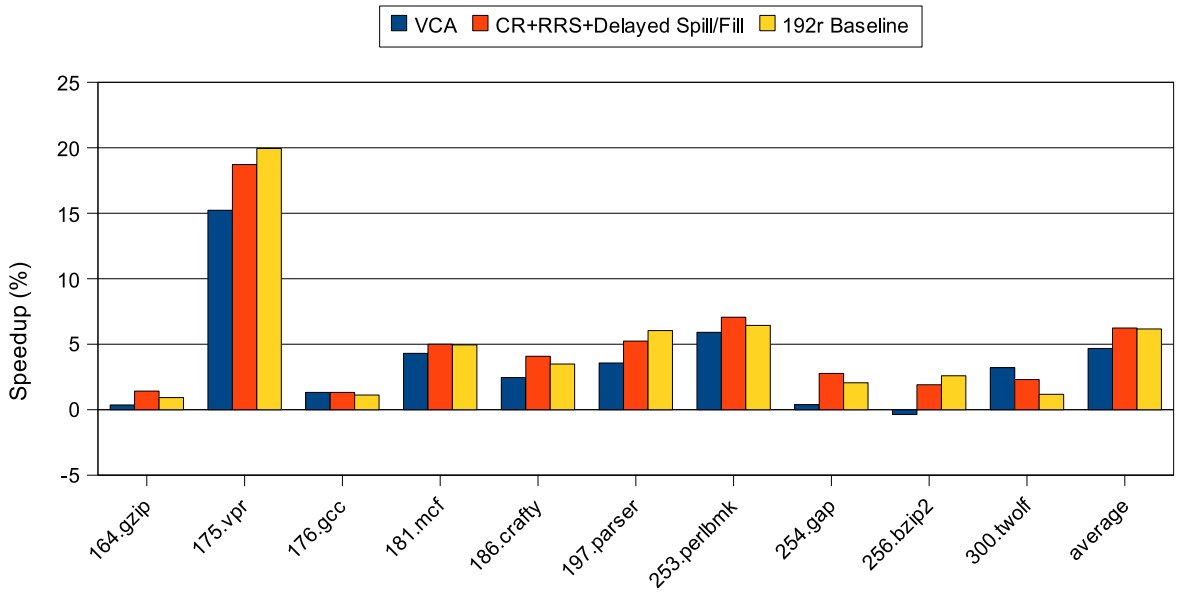

Figure 4.15: **Performance evaluation of VCA scheme, our Delayed Spill/Fill configuration and the baseline register window scheme with 192 physical registers. Speedups normalized to the baseline register window scheme with 160 physical registers.**

4.3.4 Performance Comparison with Other Schemes

Figure 4.15 compares the performance of our best proposal (4), that includes Context Release, Register Release Spill and Delayed Spill/Fill, to the VCA register window scheme. Our scheme outperforms the VCA on all benchmarks, except in *twolf*, where it loses by 1%. On average, our scheme achieves a performance advantage of 2% over the VCA. The graph also shows for comparison the performance of the baseline scheme but giving it the advantage of a large 192 physical register file (labeled as 192r Baseline), and assuming no increase on its access latency. On average, our 128-registers scheme achieves the same performance as the optimistic 192-registers baseline scheme.

4.3.5 Performance Impact of the Physical Register File Size

This section provides two performance comparisons of the Delayed Spill/Fill configuration (4) and the baseline scheme, for a wide range of 129 to 256 physical registers and 96 to 256 physical registers, respectively. Two different sets of binaries have been used. The first set has been compiled limiting the size of the register window up to 96 architectural registers (as it is defined in the IA-64 ISA). These binaries have been simulated for various physical register file sizes ranging from 128 to 256. The second set has been compiled with a maximum register window size of 64 architectural registers and simulated for physical register file sizes from 97 to 256. In both comparisons, performance speedups are normalized IPCs relative to the baseline configuration that uses the minimum required number of physical registers, i.e. 97 and 129 baseline register schemes for each set of binaries.

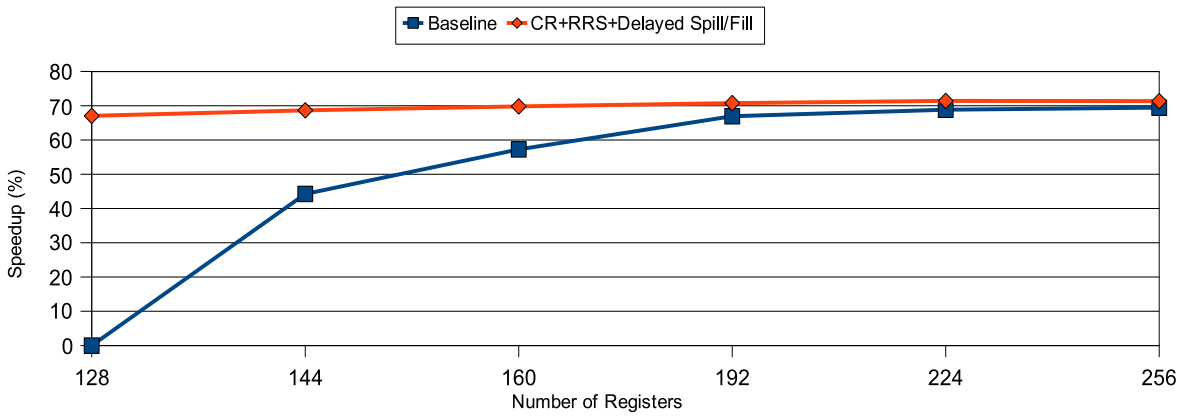

Figure 4.16: **Performance evaluation of our Delayed Spill/Fill configuration and the baseline register window scheme, when the number of physical registers varies from 128 to 256.**

Figure 4.16 compares the performance when the number of physical registers varies between 128 and 256. As expected (similar results have been published elsewhere), the baseline improves performance by increasing the number of registers, up to a saturation point around 192, beyond which it only gets marginal additional improvements. As shown in the graph, our scheme consistently outperforms the baseline. However, the most remarkable result is that the baseline curve drastically degrades as the number of registers decreases (up to a 80% speedup), while our proposal suffers just a very small performance loss (less than 4% speedup). On average, our scheme is capable to achieve the same performance as the 192-registers baseline despite having only the minimum number of physical registers (i.e. the number of architected registers plus one).

Finally, Figure 4.17 compares the performance when the number of physical registers varies between 96 and 256. Our scheme consistently outperforms the baseline. When reducing the number of physical registers from 256 to only 96, our scheme suffers only a small performance loss (7% speedup), while it achieves a enormous speedup (almost 70%) in comparison to the baseline scheme with 96 physical registers. On average, our scheme is capable to achieve the same performance as the 160-registers baseline with only the minimum number of physical registers.

In conclusion, the Delayed Spill/Fill configuration is able to reduce the number of required physical registers by up to 64 registers, with a minimal performance loss.

4.4 Cost and Complexity Issues

The previous section evaluates our proposal only in terms of performance. However, it is also interesting to analyze it in terms of cost and hardware complexity.

We have proposed a low cost implementation of the rename logic to support register windows on out-of-order processors. Compared to an in-order processor with register windows, our scheme only adds the ACDT and three map table pointers. We have experimentally found that a simple 8-entry ACDT table achieves near-optimal performance.

Compared to the VCA approach, our scheme is substantially less complex, since our proposal

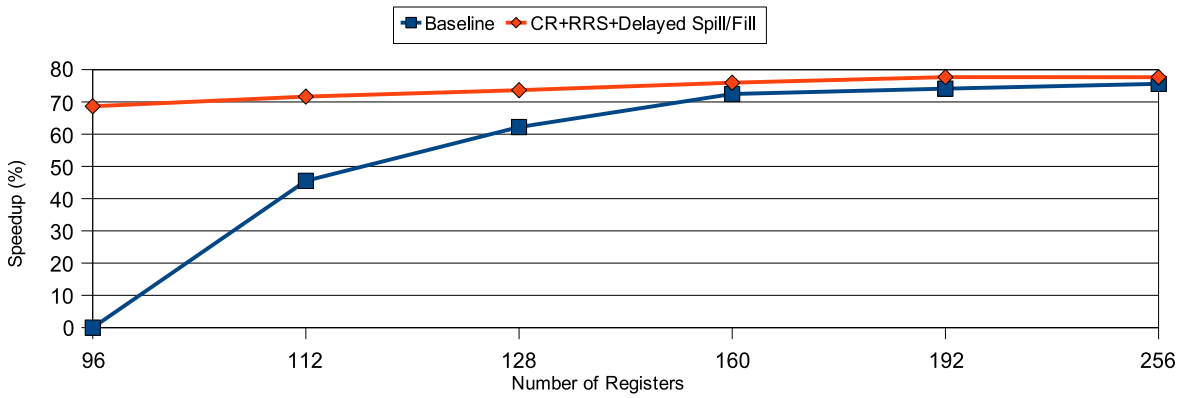

Figure 4.17: **Performance evaluation of our Delayed Spill/Fill configuration, and the baseline register window scheme, both with register window sizes up to 64 entries, when the number of physical registers varies from 96 to 256.**

uses a conventional, direct-mapped table whereas the VCA requires a larger set-associative map table to hold memory address tags. Fitting the rename delay into the processor cycle time is typically a challenging problem because of its inherent complexity. Hence, adding complexity to the rename stage increases its latency, which may have implications on cycle time and power.

Moreover, the effectiveness of the register windows technique depends on the size of the architectural register file [61] so future implementations may take advantage of increasing the map table size, which emphasizes the importance of a simple scheme for scalability.

Finally, our proposal produces a lower number of spill operations than the VCA, which not only affects to performance, but also reduces the power requirements. In comparison to the baseline, which generates a 1.1% of spill operations over the total number of commited instructions, the VCA generated up to 3%, whereas our scheme generates only up to 1.4%.

4.5 Summary and Conclusions

In this chapter we have proposed a software/hardware early register release mechanism for out-of-order processors that achieves a drastic reduction in the size of the physical register file leveraging register windows. This mechanism consists of two techniques: *Context Release* and *Register Release Spill*. They are based on the observation that if none of the instructions of a procedure are currently in-flight, not all mappings of the procedure context need to stay in the map table. These mappings, called *not-active*, can be released, as well as their associated physical registers.

Besides, we have developed another technique, called *Alloc Release*, which relies on the compiler to introduce alloc instructions in order to shrink the context and release all mappings that lay outside the new shrunk context and their associated physical registers. However, our experiments have shown that the speedup achieved by an upper-bound scheme does not make up for the code size increment.

We introduce the *Active Context Descriptor Table* (ACDT), that tracks all uncommitted context states to accomplish two main purposes: identify the active mappings at any point in time,

which is required by our early register techniques; and implement precise state recovery of context descriptor information in case of branch mispredictions and other exceptions.

Moreover, in order to avoid unnecessary rename stalls caused by the spills and fills generated by *alloc* and *br.ret* instructions respectively, we propose to defer spills and fills operations until the corresponding registers are used. Many fill operations are then eliminated, which results in a substantial reduction of the physical register file pressure.

By applying these techniques, the average register value lifetime is reduced by a 75%, so a processor fitted with only the minimum required number of physical registers, i.e. the number of architected registers plus one (which is 128 in our experiments with IPF binaries), achieves almost the same performance as a baseline scheme with an unbounded register file. Moreover, in comparison to previous techniques, our proposal has much lower hardware complexity and requires minimal changes to a conventional register window scheme.

Chapter 5
Summary and Conclusions

In this thesis we propose an efficient and low-cost hardware implementation of if-conversion and register windows to out-of-order processors. The main conclusions of the thesis are presented in this chapter, as well as future directions.

5.1 If-Conversion Technique on Out-of-Order Processors

If-conversion is a powerful compilation technique that may help to eliminate hard-to-predict branches. Reducing branch mispredictions is specially important for modern out-of-order processors because of their wide and deep pipelines. However, the use of predicate execution by the if-converted code entails two performance problems in out-of-order processors: *1)* multiple register definitions at the rename stage, *2)* the consumption of unnecessary resources by predicated instructions with its guard evaluated to false. Moreover, as shown in previous works, if-conversion may also have a negative side-effect on branch prediction accuracy. Indeed, the removal of some branches also removes their correlation information from the branch predictor and may make other remaining branches harder to predict.

There have been many proposals, focused on a predicated ISA such as IA64, to address separately both if-converted and conditional branch instructions problems. From the if-converted point of view, previous approaches address only the multiple definition problem, without taking into account the resource consumption. In fact, these techniques increase resource pressure, which reduces the potential benefits of predication. From the conditional branch point of view, the recovery of the correlation information proposed by previous approaches is done either by incorporating the previous value of the branch guarding predicate into the prediction scheme, or by adding all the predicate defines to the global history. However, in the first case, only a single predicate is correlated; in the second case the global history is fed with redundant results, since all predicates are inserted twice: once when they are produced by a compare instruction, and then when they are consumed by a branch.

This thesis proposes a novel microarchitectural scheme that takes into account if-conversion and conditional branch problems and provides an unified solution for both. This proposal is based on a predicate prediction scheme that replaces the conventional branch predictor with a predicate predictor. The predictions are made for every predicate definition, and stored until the predicate is used by a dependent if-converted or conditional branch instruction. We have shown that this approach has a number of advantages.

First, if-converted instructions know its predicted guarding predicate value at renaming, avoiding the multiple definition problem and not wasting resources. Instructions whose predicate is predicted to be true are normally renamed, while those predicted to be false are cancelled from the pipeline. However, predicting predicates may loose its potential benefits if not applied carefully, since it undoes if-conversion transformations. Therefore, our scheme is extended for if-converted instructions to dynamically identify and predict only those predicates that come from easy-to-predict branches. Instructions whose predicate is not predicted are converted to *false predicate conditional moves*. The selective mechanism is based on a confidence predictor with an adaptive threshold that searches the best trade-off between pipeline flushes and lost prediction opportunities. Our experiments show that only the 16% of the if-converted instructions are transformed to a *false predicate conditional move*.

Second, branch prediction accuracy is not negatively affected by if-conversion because compare instructions keep all the correlation information in the predictor. Our approach can be adapted easily to powerful predictors that exploit global and local correlation. On average, the branch correlation contributed by the predicate predictor adds at least 1% accuracy over a conventional branch predictor with the same configuration. Moreover, branch accuracy is further improved by exploiting early-resolved branches. Each compare instruction stores the predicate predictions in the same physical registers where the corresponding computed values will be later written. Hence, if the compare instruction is scheduled enough in advance, the prediction read by the branch is actually the computed value and is always correct. We have shown that, on average, exploiting such early-resolved branches adds an extra 0.5% to the branch prediction accuracy of our scheme.

In conclusion, we have proposed a predicate prediction scheme that allows for a very efficient execution of if-converted code without no branch prediction accuracy degradation. Moreover, our scheme does not require to add any significant extra hardware because the second level branch predictor at rename stage is replaced with a predicate predictor. This makes our proposal an extremely low cost hardware solution. Compared with previous proposals that focus on if-converted instructions, our scheme outperforms them by more than 11% on average, and it performs within 5% of an ideal scheme with perfect predicate predictor. Moreover, our scheme improves branch prediction accuracy of if-converted codes by 1.5% on average, which achieves an extra 4% of speedup.

5.2 Register Windows on Out-of-Order Processors

Register windows is a hardware technique that allows to reduce the amount of loads and stores required to save and restore registers across procedure calls by storing the procedure contexts on a large register file. We propose software/hardware early register release techniques for out-of-order processors that, by using the information provided by register windows, achieve a drastic reduction in the size of the physical register file. These techniques, called *Context Release*, *Alloc Release* and *Register Release Spill*, are based on the observation that if none of the instructions of a procedure are currently in-flight, not all mappings of the procedure context need to stay in the map table. These mappings, called *not-active*, can be released, as well as their associated physical registers.

The *Context Release* technique releases mappings defined by a procedure whose closing return

instruction has committed. The values of these mappings are *dead values* that will never be used again, so their associated physical registers can be safely released.

The *Register Release Spill* technique is automatically triggered when the rename stage runs out of physical registers and it releases not-active mappings that belong to caller procedures. However, unlike the previous technique, the values contained in these mappings are *live values* that may be used in the future, after returning from the procedure, so they must be spilled before releasing them.

The *Alloc Release* technique is a compiler technique that, by introducing alloc instructions, shrinks the context size and releases all mappings that lay outside the new shrunk context and their associated physical registers. In order to explore the potential of the Alloc Release technique we have simulated an upper-bound scheme based on the knowledge of the register live list at the entry of each basic block. However, our experiments have shown that the speedup achieved by the upper-bound scheme does not make up for the code size increment.

We introduce the *Active Context Descriptor Table* (ACDT), that tracks all uncommitted context states to accomplish two main purposes: identify the active mappings at any point in time, which is required by our early register release techniques; and implement precise state recovery of context descriptor information in case of branch mispredictions and other exceptions. We also introduce the *Retirement Map Table* that holds the state of the *Rename Map Table* at commit stage, and it allows to release not-active mappings from contexts that have already been redefined at the rename stage by new contexts.

Moreover, in order to avoid unnecessary rename stalls caused by the spills and fills generated by *alloc* and *br.ret* instructions respectively, we propose to defer spill and fill operations until the corresponding registers are used. Hence, a fill is generated and its corresponding physical register is allocated when the mapping is actually used by a subsequent instruction. Many fill operations are then eliminated, which results in a substantial reduction of the physical register file pressure. In the same way, a spill is generated when the mapping is actually reassigned by a subsequent instruction.

By applying these techniques, the average register lifetime is reduced by a 75%, so a processor fitted with only the minimum required number of physical registers, i.e. the number of architected registers plus one (which is 128 in our experiments with IPF binaries), achieves almost the same performance as a baseline scheme with an unbounded register file size. Moreover, in comparison to previous techniques, our proposal has much lower hardware complexity and requires minimal changes to a conventional register window scheme.

5.3 Future Work

This thesis opens up several topics from which we emphasize the following:

- Although predication has been widely proved to be a good solution to deal with branches that are considered hard-to-predict (see section 1.3), it also has some drawbacks that should be considered. Predicate execution may increase the size of the instruction word because of the extra bits require to codify the predicate register name. This word size increment might have some effects over memory bandwidth, fetch bandwidth and cache sizes.

- The register windows mechanism considers that all the architectural registers that form a procedure context contains *live values*. Hence, if new space is required to allocate a new procedure context, all the required architectural registers are spilled into memory, although some of them will never be used again when the procedure context be restored. Other schemes, such as Alpha [38], have a pool of scratch registers whose values are not saved in the call interface. Although, the Alloc Release technique may help to reduce the number spill operations, it could be beneficial to assign a set of architectural registers as *scratch*, avoiding unnecessaries spill operations.

Publications

Conferences

- Early Register Release for Out-of-Order Processors with Register Windows. Eduardo Quiñones, Joan-Manuel Parcerisa, Antonio González. PACT '07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques, Brasov, Romania. September 2007.
- Improving Branch Prediction and Predicate Execution in Out-of-Order Processors. Eduardo Quiñones, Joan-Manuel Parcerisa, Antonio González. HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture. Phoenix, Arizona, USA. February 2007.
- Selective Predicate Prediction for Out-of-Order Processors. Eduardo Quiñones, Joan-Manuel Parcerisa, Antonio González. ICS '06: Proceedings of the 20th annual international conference on Supercomputing. Cairns, Queensland, Australia. June 2006.

Submited

- Leveraging Register Windows to Reduce Physical Registers to the Bare Minimum. Eduardo Quiñones, Joan-Manuel Parcerisa, Antonio González. IEEE Transactions on Computers.

Bibliography

[1]

[2] Ibm360/91, http://www-03.ibm.com/ibm/history/reference/glossary_9.html.

[3] Opencc, the open research compiler, http://www.open64.net/home.html.

[4] Standard performance evaluation corporation. spec. *Newsletter, Fairfax, VA*, September 2000.

[5] H. Akkary, R. Rajwar, and S. t. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO 36: Proceedings of the 36th annual international symposium on Microarchitecture*, 2003.

[6] J. R. Allen, K. Kennedy, and C. P. an Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, New York, NY, USA, 1983. ACM Press.

[7] D. August, D. Connors, J. Gyllenhaal, and W. M. Hwu. Architectural support for compiler-synthesized dynamic branchprediction strategies: Rationale and initial results. In *HPCA '97: Proceedings of the 3th International Symposium on High-Performance Computer Architecture*, pages 84–93, 1997.

[8] D. I. August, W. mei W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *MICRO 30: Proceedings of the 30th annual international symposium on Microarchitecture*, pages 92–103, 1997.

[9] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The intel ia64 compiler code generator. *Micro IEEE*, pages 44–53, September-October 2000.

[10] P.-Y. Chang, E. Hao, Y. N. Patt, and P. P. Chang. Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution. In *PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pages 99–108, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[11] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *ISCA '98: Proceedings of the 25nd annual international symposium on Computer architecture*, June, 1998.

[12] W. Chuang and B. Calder. Predicate prediction for efficient out-of-order execution. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 183–192, 2003.

[13] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, 1999.

[14] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 316–325, 2000.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[16] D.L.Weaver and T.Germond. *SPARC Architecture Manual (version 9)*.

[17] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing processor performance through early register release. In *ICCD: Proceedings of International Conference on Computer Design*, 2004.

[18] E. R. Erik Jacobsen and J. Smith. Assigning confidence to conditional branch predictions. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 142–152, 1996.

[19] K. Farkas, N. Jouppi, and P. Chow. Register file considerations in dynamically scheduled processors. In *HPCA '96: Proceedings of the 2th International Symposium on High-Performance Computer Architecture*, pages 40–51, 1996.

[20] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, C-30,7:478–490, July 1981.

[21] Freescale Semiconductor Inc. *AltiVec Technology Programming Environments Manual*, 2002.

[22] D. Gallagher, W. Chen, S. Mahlke, and J. G. ans W.W Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the 6th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.

[23] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-physical registers. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, 1998.

[24] T. R. Gross and J. L. Hennessy. Optimizing delayed branches. In *MICRO 15: Proceedings of the 15th annual workshop on Microprogramming*, pages 114 –120, Piscataway, NJ, USA, 1982. IEEE Press.

[25] L. Gwennap. Intel, hp make epic disclosure. *Microprocessor report*, 11:1 – 9, October 2001.

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

[27] T. Horel and G. Lauterbach. Ultrasparc-iii: Designing thrid-generation 64-bit performance. *Micro IEEE*, pages 73–85, May-June 1999.

[28] P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 386–395, 1986.

[29] W. W. Hwu and Y. N. Patt. Checkpoint repair for high-performance out-of-order execution machines. *IEEE Transactions on Computers*, C-36, December 1987.

[30] Intel Corporation. *Intel(R) Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*, 1999.

[31] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual. Volume 1: Application Architechiture*, 2002.

[32] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual. Volume 2: System Architechiture*, 2002.

[33] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual. Volume 3: Instruction Set Reference*, 2002.

[34] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[35] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin. Compiler directed early register release. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 110–122, Washington, DC, USA, 2005. IEEE Computer Society.

[36] N. Jouppi and S. Wilton. Trade-offs in two-level on-chip caching. April 1994.

[37] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer architecture*, pages 364–373, 1990.

[38] R. Kessler. The alpha 21264 microprocessor. *Micro IEEE*, 19:24–36, March-April 1999.

[39] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *MICRO 38: Proceedings of the 38th annual International Symposium on Microarchitecture*, pages 43–54, 2005.

[40] A. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. 2002.

[41] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '81: Proceedings of the 8th annual international symposium on Computer architecture*, pages 81–87, May, 1981.

[42] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the SIGPLAN'88 Conference on Programming Languages Design and Implementation*, pages 318–328. ACM Press, June 1988.

[43] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[44] S. A. Mahlke, , R. H. anf R.A. Bringmann, J. Gyllenhaal, D. Gallagher, and W.-M. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 217–227, 1994.

[45] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 138–150, New York, NY, USA, 1995. ACM Press.

[46] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, 1992.

[47] S. A. Mahlke and B. Natarajan. Compiler synthesized dynamic branch prediction. In *MICRO 29: Proceedings of the 29th annual international symposium on Microarchitecture*, pages 153–164, 1996.

[48] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *MICRO 30: Proceedings of the 30th annual international symposium on Microarchitecture*, 1997.

[49] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resources recycling in out-of-order microprocessors. In *MICRO 35: Proceedings of the 35th annual international symposium on Microarchitecture*, 2002.

[50] S. McFarling. Combining branch predictors. *WRL Technical Note TN36*, June, 1993.

[51] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *Micro IEEE*, pages 44–55, March-April 2003.

[52] T. Monreal, V. Vials, A. Gonzalez, and M. Valero. Hardware schemes for early register release. In *ICPP-02: Proceedings of International Conference on Parallel Processing*.

[53] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *MICRO 30: Proceedings of the 30th annual international symposium on Microarchitecture*, pages 92–103, 1997.

[54] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 202–213, Nov. 1993.

[55] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt. How to fake 1000 registers. In *MICRO 38: Proceedings of the 38th annual International Symposium on Microarchitecture*, pages 7 – 18, 2005.

[56] S. Pan, K. So, and J. Rameh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proceedings of the 5th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, October 1992.

[57] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *MICRO 37: Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 81–92, 2004.

[58] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ilp processors. April 1994.

[59] E. Quinones, J.-M. Parcerisa, and A. Gonzalez. Selective predicate prediction for out-of-order processors. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, 2006.

[60] E. Quinones, J.-M. Parcerisa, and A. Gonzalez. Improving branch prediction and predicate execution in out-of-order processors. In *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, Feb. 2007.

[61] R. Rakvic, E. Grochowski, B. Black, M. Annavaram, T. Diep, and J. P. Shen. Performance advantage of the register stack in intel itanium processors. In *Workshop on Explicit Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques*, 2002.

[62] R.L.Sites. How to use 1000 registers. In *Caltech Conference on VLSI*, pages 527 – 532. Caltech Computer Science Dept., 1979.

[63] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *MICRO 29: Proceedings of the 29th annual international symposium on Microarchitecture*, 1996.

[64] M. S. Schlansker and B. R. Rau. Epic: Explicitly parallel instruction computing. *Computing Practices IEEE*, pages 37–45, February 2000.

[65] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *Micro IEEE*, pages 24–43, September-October 2000.

[66] B. Simon, B. Calder, and J. Ferrante. Incorporating predicate information into branch predictors. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[67] K. Skadron, M. Martonosi, and D. W. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction Level Parallelism*, January 2000.

[68] J. E. Smith. A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th annual international symposium on Computer architecture*, pages 135–148, May, 1981.

[69] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37:562–573, May 1988.

[70] M. Smith, M. Johnson, and M. Horowitz. Limits on multiple instruction issue. In *Proceedings of the 3th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302. ACM Press, April 1989.

[71] G. S. Sohi and S. Vajapeyam. Tradeoffs in instruction format design for horizontal architectures. In *Proceedings of the 3th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–25. ACM Press, April 1989.

[72] T. Teh and Y. N. Patt. Alternative implementations of two-level adaptative branch prediction. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 124–134, May 1992.

[73] T. Teh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 124–134, May 1993.

[74] J. M. Tendle, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[75] G. S. Tyson. The effects of predicated execution on branch prediction. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 196–206, 1994.

[76] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of soar: Smalltalk on a risc. *IEEE MICRO*, 1984.

[77] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with liberty. In *MICRO 35: Proceedings of the 35th annual international symposium on Microarchitecture*, pages 271 – 282, 2002.

[78] P. H. Wang, H. Wang, R. M. Klin, K. Ramakrishnan, and J. P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 15, 2001.

[79] W. Wang, J. Baer, and H. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, pages 140–148, May 1989.

[80] N. Warter and S. A. Mahlke. Reverse if-conversion. In *Proceedings of the SIGPLAN'93 Conference on Programming Languages Design and Implementation*, pages 290–299, June 1993.

[81] S. Weiss and J. Smith. A study of scalar compilation techniques for pipelined supercomputers. In *Proceedings of the 2th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–109. ACM Press, October 1989.

[82] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN'91 Conference on Programming Languages Design and Implementation*, pages 30–44, June 1991.

[83] R. Yung and N. Wilhelm. Caching processor general design. In *ICCD: Proceedings of International Conference on Computer Design*, pages 307–312, Oct. 1995.