SPREPI: Selective Prediction and REplay for predicated Instructions

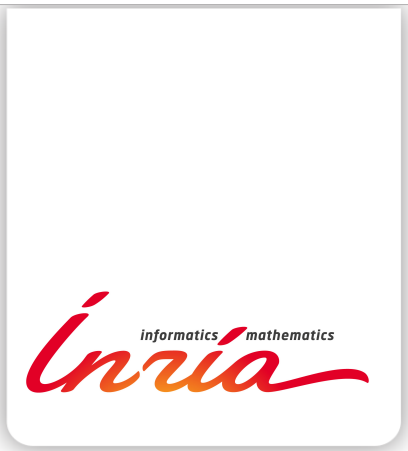
Nathanaël Prémillieu, André Seznec

SPREPI: Selective Prediction and REplay for Predicated Instructions

Nathanaël Prémillieu, André Seznec

RESEARCH REPORT
N° 8351
Aout 2013
Project-Team ALF

ISSN 0249-6399 ISRN INRIA/RR--8351--FR+ENG



SPREPI: Selective Prediction and REplay for Predicated Instructions

Nathanaël Prémillieu*, André Seznec†

Project-Team ALF

Research Report n° 8351 — Aout 2013 — 22 pages

Abstract: ARM ISA-based processors are no longer low-cost low-power processors. Nowadays ARM ISA based processor manufacturers are struggling to implement medium-end to high-end processor cores, and this implies implementing a state-of-the-art out-of-order execution engine. Unfortunately providing efficient out-of-order execution on legacy ARM codes may be quite challenging due to predicated instructions.

In this paper, we propose a new hardware solution, Selective Prediction and REplay for Predicated Instructions (SPREPI), to provide efficient out-of-order execution of codes featuring predicated instructions. Predicting the predicated instructions addresses the so-called multiple definition problem. Predicated instructions are predicted using either a global branch-and-predicate history predictor or a global history predictor. But systematic usage of predicate prediction sometimes impairs the performance dramatically. Efficient filters are proposed to disable predicate prediction uses when they are likely to be counter-productive. Moreover predicate misprediction penalty can be as high as the branch mispenalty. To reduce this penalty we introduce a specific selective replay hardware component targeting mispredicted predicated instructions.

SPREPI is shown to allow high out-order execution performance on ARM codes generated even with a compiler applying if-conversion only to very short branches. Moreover since SPREPI predicts most of the predicated instructions, a relatively inefficient hardware solution is sufficient for executing the few predicated instructions on which prediction is not used.

Key-words: prediction, predicated instructions, predication, seletive, replay, out-of-order execution, ARM

* IRISA/Université de Rennes 1
† IRISA/INRIA

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

SPREPI : prédiction et rejeu sélectif pour les instructions prédiquées

Résumé : Les processeurs basés sur le jeu d'instructions ARM ne sont plus seulement des processeurs à petit budget et à faible consommation. De nos jours, les concepteurs de processeurs ARM cherchent à créer des coeurs d'exécution à performances moyennes voire hautes, impliquant l'implémentation de l'état de l'art d'un moteur d'exécution dans le désordre. Cependant, permettre une exécution dans le désordre efficace peut se révéler difficle sur les codes ARM anciens à cause des instructions prédiquées.

Dans cet article, nous proposons une nouvelle solution matérielle, la prédiction et le rejeu sélectif pour les instructions prédiquées (SPREPI) pour permettre une exécution efficace de codes avec des instructions prédiquées. Prédire les instructions prédiquées permet de résoudre le problème appelé le problème des définitions multiples. Elles sont prédites en utilisant soit un prédicteur à historique global des branchements et prédicats soit un prédicteur à historique des branchements seul. Cependant, l'utilisation systématique de la prédiction entraîne parfois des pertes de performances dramatiques. Des filtres efficaces sont proposés pour désactiver l'utilisation de la prédiction quand il y a des chances qu'elle soit contre-productive. De plus, une mauvaise prédiction de prédicat peut être aussi coûteuse qu'une mauvaise prédiction de branchement. Pour réduire ce coût, nous proposons un mécanisme matériel spécifique de rejeu sélectif ciblant les instructions prédiquées mal prédites.

Nous montrons que SPREPI permet une exécution dans le désordre performante sur du code ARM, même lorsque le compilateur n'applique la transformation de if-conversion qu'aux branchements très courts. De plus, comme SPREPI prédit la plupart des instructions prédiquées, une solution matérielle relativement inefficace suffit à exécuter les quelques instructions prédiquées pour lesquelles la prédiction n'est pas utilisée.

Mots-clés : prédiction, instructions prédiquées, prédication, sélectif, rejeu, exécution danns le désordre, ARM

1 Introduction

On the low-end and low power (general purpose) processor segment, in-order pipelined execution has been privileged. The ARM-v7 ISA is dominating this market segment. With the rise of mobile devices (smartphones, tablets), the demand for higher performance on these processors is very high. Manufacturers are now adapting all the concepts that were used in high end microprocessors to the ARM ISA. And new high-end ARM based processors feature out-of-order execution. However, providing efficient out-of-order execution on legacy ARM codes may be quite challenging due to predicated instructions[1].

The ARM ISA features predicated instructions, i.e. instructions whose execution is conditional. For the compiler, predicated instructions allow to limit the number of branches in the binary code through guarding the execution of some instructions by a condition computed at execution time. At compile time, if-conversion [1] can be used to transform control dependencies into data dependencies. Instructions that were on the taken path are controlled through a predicate which is true if the branch was to be taken; the not-taken path instructions are controlled through the opposite predicate. Instructions from both paths can be intermingled in a single basic block. At execution only the correct instructions will be effectively validated. On pipelined processors or on in-order execution superscalar processors, the use of if-converted branches may bring some performance benefit 1) it allows to launch at the same time the sequencing (fetch, decode, operand reads) of instructions from both paths 2) it removes the (maybe hard-to-predict) branch from the execution path. However, it is not possible nor always desirable to if-convert all branches.

Most instruction sets offer a limited form of predicated instructions, generally the conditional move, e.g. X86 ISA, Alpha, MIPS, SPARC V9, ... For these instruction sets, the compiler has limited facility to generate if-converted branches, and in practice, the number of predicated instructions in codes is quite limited. The impact of predicated instructions on the effective performance of the processor is also limited. On the other hand, other instruction sets such as ARM or IA64 have taken a much more radical approach: (nearly) all instructions can be predicated. Therefore the compiler has much more opportunity to generate predicated instructions. On out-of-order execution superscalar processors implementing such a fully predicated instruction set, the effective performance on legacy code may significantly depend on the performance of predicated instructions.

Handling predicated instructions efficiently on a superscalar out-of-order processor is a challenge. The main issue is known as the multiple definition problem [24]. This issue arises when the last instruction that may have written an architectural register R_i was a predicated instruction. In that case, when renaming the registers for a subsequent instruction I which uses R_i as an operand, the difficulty is to determine the effective physical register which will provide the value of R_i to instruction I. A working yet not efficient solution is to insert an extra non-architectural instruction after the predicated instruction [11]. This non-architectural instruction affects either the result of the operation of the predicated instruction or the old value depending on the dynamic predicate (see Figure 3 in Section 2.3). However, this solution hurts performance as it serializes the execution of possibly independent instructions e.g. when the same register is written on both paths of a branch that has been if-converted, thus reducing the available instruction level parallelism. More aggressive solutions [14, 5] have been proposed to handle the multiple definition problem, but induce a significant hardware overhead.

In this paper, we address out-of-order execution of predicated instructions through a two-component micro-architectural solution, Selective Prediction and Replay for Predicated Instructions (SPREPI). As in [7, 16], SPREPI builds upon state-of-the-art branch prediction (e.g. TAGE

[1] The handling of predicated instructions in out-of-order execution processors is largely undocumented.

[18]) to predict predicates and to avoid execution of predicated false instructions. The predictor uses either a global branch-and-predicate history or a global branch history. The predicate prediction is used to solve the multiple definition problem. However, while this predicate prediction turns out to be quite performance effective on some applications, systematic predicate prediction use may turn into performance losses on other applications. Therefore, SPREPI requires filters on the predicate prediction use to decrease its misprediction rate. Misprediction penalties on predicated instructions might also impaired performance. To address this issue, we introduce a cost effective selective replay mechanism derived from the SYRANT proposal [15] that allows to repair the execution by just re-executing the necessary instructions when resuming execution after a mispredicted predicated instruction.

The ARM ISA has been chosen to illustrate our proposal since it is the most widespread fully-predicated ISA, and, in addition of the Cortex A-15 ARM, several manufacturers are working on delivering high performance out-of-order implementation for this ISA. However SPREPI could be adapted to any predicated instruction set.

Simulations results show that SPREPI allows performance benefits on ARM code generated by a compiler applying if-conversion only to very short branches. Our simulations also show that since SPREPI transforms most of the predicated instructions in conventional instructions through prediction, aggressive implementation of predicated instruction execution is not worth the extra hardware complexity and power consumption.

The remainder of this paper is organized as follows. Section 2 provides background on the problems that arise when dealing with predicated instructions and how they are currently tackled in a standard ARM core. Section 3 presents several related works on predicated instructions and the multiple definition problem. Section 4 details the SPREPI proposition and describes the selective replay mechanism. Section 5 presents simulation results on ARM codes generated with a standard gcc compiler. Finally, Section 6 concludes this study.

2 Executing Predicated Instructions on an Out-of-order Engine

2.1 Registers Renaming (no predication)

In an out-of-order execution engine, the mapping table is used to store the links between architectural registers and their associated physical registers value. This mapping table is used to avoid false dependencies between instructions that write to the same architectural register. Hence, for each instruction, at the rename stage, the architectural destination registers are assigned a new physical register and the architectural source registers are renamed in order to read the physical register corresponding to their correct occurrences. A physical register P associated with architectural register R is considered as dead when the next write on R has been committed; at this time it can be inserted in the free list and may be used again for renaming.

Figure 1 illustrates an example of the register renaming process. Instruction I reads from architectural registers $R1$ and $R2$, and writes in architectural register $R3$. To obtain the renamed form of instruction I, one has to read the mapping table. In this example, $R1$ is mapped to $P12$ and $R2$ to $P15$. The result register $R3$ is assigned to the first physical register available of the free list of physical register, $P22$ in this case. Then, the renamed form of I is $I : P22 \leftarrow P12$, $P15$.

All these steps are performed at the renaming stage, before executing the instructions. Though renaming is applied to multiple instructions in parallel, the process preserves the in-order semantic and conserves the correct dependencies in the program.

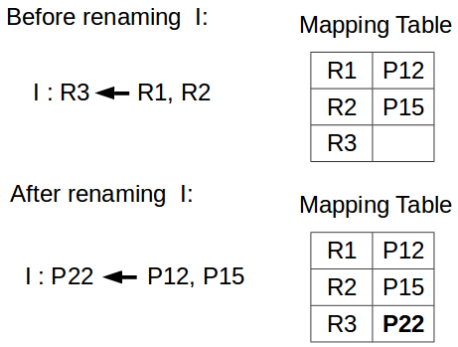


Figure 1: Illustration of register renaming process in an out-of-order processor.

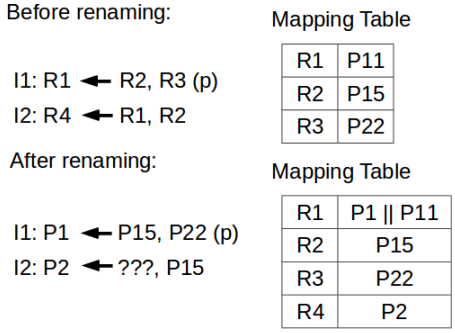


Figure 2: The multiple definition problem on an out-of-order execution engine.

2.2 The multiple definition problem

When considering a predicated instruction, one can not determine at the rename stage which instruction will be the last dynamic writer of its (conditional) architectural register target.

Figure 2 illustrates this, known as the multiple definition problem [24]. I_1 conditionally writes to architectural register $R1$, I_1 being predicated with the predicate p. After renaming, I_1 conditionally writes to $P1$. I_2 reads from $R1$, but it is not possible to know whether the correct physical register associated with $R1$ is $P1$ or $P11$ before the predicate associated with I_1 is computed.

2.3 Dealing with the Multiple Definition Issue

2.3.1 A working solution

A working yet not efficient solution [11] consists in splitting the predicated instruction in two consecutive micro-operations: the first micro-operation implementing the computation and the second micro-operation implementation the selection between the previous target register value and the result of the first micro-operation. The operation executed by this instruction is

$$P_{after} = \textbf{(predicate) ? } P_{new} \textbf{ : } P_{before}$$

This is illustrated in Figure 3. Figure 3 also illustrates the serialization of the sequence of accesses on the registers, as well as the artificial creation of long dependency chains. Even if

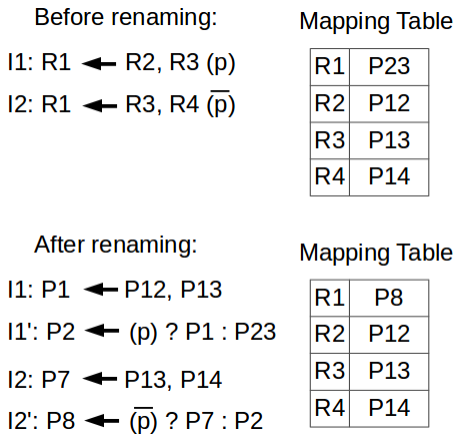


Figure 3: Dealing with the multiple definition problem: for each predicated instruction, a second instruction chooses between the new value created by the instruction or the old value.

instructions I_1 and I_2 were executed on the same cycle T, the operand for I_3 is only available after the extra instruction for I_1 is executed (cycle $T+1$) and after the extra instruction for I_2 is executed (cycle $T+2$).

Such an implementation is acceptable when predicated instructions are quite infrequent (e.g. when the instruction set only allows conditional moves), but may impair performance when a significant amount of predicated instructions are executed.

2.3.2 Aggressive multiple definition handling

A more aggressive solution [14, 5] to the multiple definition problem consists in packing the two micro-operations considered above in a single micro-operation. That is at rename time, the instruction is translated in:

$$P_{after} = \textbf{(predicate) ? } Operation(Op1, Op2) \textbf{ : } P_{before}$$

This solution also serializes the execution of the two instructions in Figure 3, but saves two cycles (Figure 4).

However, we would like to point out that this implementation requires an extra physical register read for each predicated instruction. Therefore implementing such an aggressive multiple definition handling will lead to quite high complexity in the physical register design (an extra register port per way), on operand tracking in the issue logic and on the bypass network.

Apart in Section 5.4.5, our simulations in Section 5 will assume this aggressive multiple definition handling.

2.4 Predication in the ARM ISA

Most of the previous related works on out-of-order execution of predicated instructions assume the Intel Itanium ISA [10] For this ISA, the predicate of a predicated instruction is the boolean value contained in a predicate registers.

As our study is illustrated with the ARM ISA, we present here the specificities of the predicate model of the ARM ISA. Instructions are predicated through a boolean value computed from the value of four flags: the Negative flag (N), the Zero flag (Z), the Carry flag (C) and the Overflow

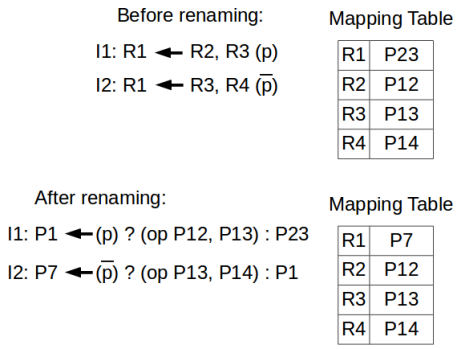


Figure 4: Aggressive dealing with the multiple definition problem: each predicated instruction is added an extra register operand: the old value of the target operand

flag (V). These flags are written by specific instructions, like compare instructions and specific arithmetic instructions [2]. Table 1 summarizes the possible predicates in the ARM ISA. Thus, the value of these predicates is not directly known through reading a register, but also requires to evaluate the logical formula associated to the predicate.

Predicate Mnemonic	**Logical Formula**
EQ	Z == 1
NE	Z == 0
CS	C == 1
CC	C == 0
MI	N == 1
PL	N == 0
VS	V == 1
VC	V == 0
HI	(C == 1) && (Z == 0)
LS	(C == 0) || (Z == 1)
GE	N == V
LT	N != V
GT	(Z == 0) && (N == V)
LE	(Z == 1) || (N != V)
AL	(always true)

Table 1: ARM ISA possible predicates for predicated instructions

3 Related Works

Efficiently dealing with control instructions in a processor has always been a challenge. Two directions have been proposed, using prediction to know in advance the direction and the target of the branch [20], and using predicated instructions.

If-conversion was proposed by Allen et al. [1]. They define an algorithm to convert control dependencies into data dependencies by replacing branches and their dependent instructions by predicated instructions.The predicated instructions are only executed if their predicate is eval-

uated to true. This conversion algorithm is often called if-conversion. If-conversion allows to merge the taken and not-taken paths in the binary, It removes a branch, and allow to sequence both paths at the same time. However, it is not possible to if-convert all conditional branches. It is not always performance effective either, since both paths are fetched thus increasing consumption of the processor resources. Thus, one should only if-convert a subset of the convertible branches. Often compilers only if-convert short branches.

Combining branch prediction and predication has been proposed in several studies [13] [12] [5]. The main idea is to use prediction for easy-to-predict branches and if-convertion for hard-to-predict branches. This reduces the number of mispredicted branches and should increase performance. Chang et al. [5] use profiling to identify the hard-to-predict branches to convert. They show that profiling is efficient at identifying hard-to-predict branches.

Kim et al. [13] propose to reduce the potential dynamic instruction overhead induced by predication by storing two versions of the code, the predicated and the non-predicated code in the binary. Then, the runtime chooses which version is executed, based on a confidence estimation. So, contrary to [5], using if-converted code is decided at runtime, with more precise information.

Several studies [14] [22] points out that removing branches by if-conversion can have an influence on the accuracy of branch predictors. One of the main effects is the reduction in correlation as there are less branches. Simon et al. [19] show that the outcome of some branches can be directly related to the value of some predicates. They also propose to add the predicate predictions in the global branch history to try to capture the correlation lost by the if-conversion.

As we already pointed out, a major issue for out-of-order execution of predicated code is the multiple definition problem.

Wang et al. [24] propose to introduce a new instruction, the select-μ op instruction to solve this problem. This instruction is conceptually similar to the ϕ-function used in the Single Static Assignment analysis [8]. It is used to merge the multiple predicated definition of an architectural register into one, removing the renaming ambiguity for this register. The select-μ op source operands are the multiple predicated definitions of the architectural register and its destination register will contain the correct value based on the predicate values of the multiple definitions. Compared with the solution presented in the previous section, this allows to postpone the effective selection of the register to be used by an instruction to the effective first consumption of the register.

Predicate prediction is an other way to solve the multiple definition problem. Chuang and Calder [7] propose a predicate predictor, derived from a branch predictor, with a replay mechanism. Indeed, contrary to branch prediction, on predicate misprediction, there is no need to squash the entire pipeline. It is only necessary to re-rename the instruction stream, starting from the mispredicted instruction. The authors also propose a selective replay mechanism, where only the instructions that depend on the mispredicted instruction are re-executed. Selective replay is handled by keeping tags that are used to serialize the predicated instructions during replay.

Quiñones et al. [16] proposed to selectively predict predicates instead of always using the predicate prediction. This selection is based on a confidence estimator. If the confidence is high enough, the predicate is predicted. If not, the predicated instruction is handled as suggested as described in Section 2.3.2.

Quiñones et al. [16] is the closest related work on predicate prediction to our study. Compared with previous work, our SPREPI proposition addresses the ARM ISA. It also presents the advantage of very high accuracy enabled by adapting state-of-the-art TAGE predictor to predicate prediction. We also introduce the use of global branch-and-predicate history instead of simple global branch history. Selective predication is enabled either by switching the prediction use on and off (global branch-and-predicate history) or through predicate prediction use on high

confidence only (global branch history).

Moreover, our study proposes an efficient selective replay mechanism specifically targeting predicated instructions.

4 Selective Prediction and Replay for Predicated Instructions

Our proposal, Selective Prediction and Replay for Predicated Instructions (SPREPI), aims at handling efficiently predicated instructions in an out-of-order execution engine. SPREPI is based on two major components: a predicate predictor and a specific selective replay engine adapted to handle predicate mispredictions.

4.1 Selective Predicate Prediction

Predicate prediction solves the multiple definition problem. Only predicated instructions with predicate predicted true are renamed while the instructions with predicate predicted false are transformed as noop instructions at rename. Therefore there is a single valid definition for any particular architectural register.

Predicting predicate can be implemented similarly as predicting conditional branches. In our study, we are using a predicate predictor derived from the state-of-the-art global history branch predictor TAGE [18].

The quality and accuracy of the prediction ensured by a predictor highly depends on the quality of the vector information it uses as an input. We experimented two major alternatives the global branch history and the global branch-and-predicate history, i.e. inserting both branches and predicates in the history.

4.1.1 Predicated group

If-conversion tends to generate several predicated instructions using the same predicate or the opposite predicate — the original taken and not-taken paths. This leads us to the concept of a predicated group of instructions: the group of predicated instructions that use the same predicate value or its opposite. A predicated group is associated with the use of the same occurrence of a specific predicate. These instructions are not necessarily contiguous in the code since if-conversion leads to encapsulate and schedule instructions from the taken path, instructions from the not-taken path and instructions common to both paths in the same basic block. Our study targets the ARM ISA, therefore, for our study, a predicated group starts at the first use of a predicate and ends when a flag-defining instruction is encountered.

Figure 5 illustrates an example of two predicated groups. A single predicate prediction is performed for each predicated group in the front-end pipeline before rename. In the following, a predicated group will be represented by its first instruction.

The predicate has to be predicted for the first instruction in the predicate group and some logic is needed to propagate the predicate value to the whole predicated group at rename time. In the remainder of the paper, when referring to the global branch-and-predicate history vector, we assume that the predicate is appended only once to the history on its first encountering in the instruction flow, even when the same predicate occurs multiple times.

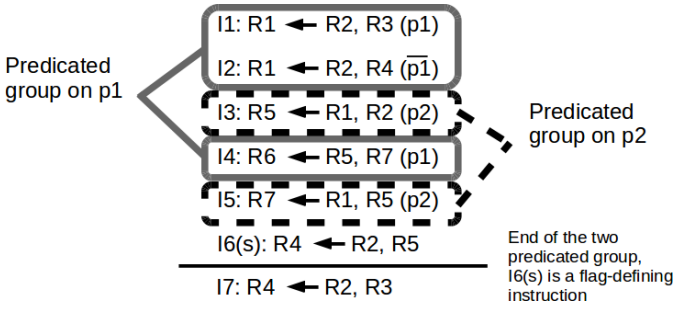


Figure 5: Predicated groups are groups of instructions that use the same predicate value (or its contrary). They end when a flag-defining instruction is encountered.

4.1.2 Selective Predicate Prediction

Branch vs Branch-and-Predicate history The accuracy of a branch or a predicate predictor depends on the prediction scheme, on the predictor size and also on the quality of the vector of information that the predictor is exploiting. Global branch history was shown to be a very high quality information vector to predict branches. TAGE [18] is generally considered as state-of-the-art global history branch predictor. Therefore we use TAGE-based predictors to predict predicates.

There is a correlation of predicate values with previous branch outcomes but also with previous predicate values. Therefore global branch-and-predicate history is a natural candidate to predict predicates instead of global branch history. In the remainder of the paper, we will refer to the predictor using global branch-and-predicate history as *BrPred*.

An alternative approach is to use the conventional global branch history to predict the predicates. In the remainder of the paper, we will refer to the predictor using global branch history as *BrO*.

In our experiments (see Section 5, Figure 8), assuming systematic usage of predicate prediction we observe that global branch-and-predicate history slightly outperforms global branch history. But we also observe that some applications exhibit very high predicate misprediction rates. This induces performance losses compared with our baseline architecture where predicated instructions are handled by the serialization described in Section 2.3.2. This tends to indicate that in case of hard-to-predict predicates, it is more efficient to execute the predicated instruction through this serialization instead of predicting it.

Selective Predicate Prediction To discriminate hard-to-predict predicate occurrences from easy-to-predict, confidence estimators are the natural vehicle. Such a cost-effective confidence estimator was proposed for the TAGE predictor [17]. Using predicate predication only on high confidence and the serialization process described in Section 2.3.2 for other confidence levels is quite effective as illustrated by our experiments using *BrO* (see Section 5, Fig 9). In the remainder of the paper, we will refer to using *BrO* with the high confidence prediction estimation as *BrO-HighConf*.

Unfortunately, the high confidence filter cannot be applied on *BrPred* using the global branch-and-predicate history due to corruptions on the speculative global branch-and-predicate history. Prediction is effective and accurate if the predictor is accessed with the same information vector (history + PC) at read time, i.e. prediction time, and update time, i.e. commit time. If the prediction of a predicate instruction is not used at run-time and the execution of the instruction

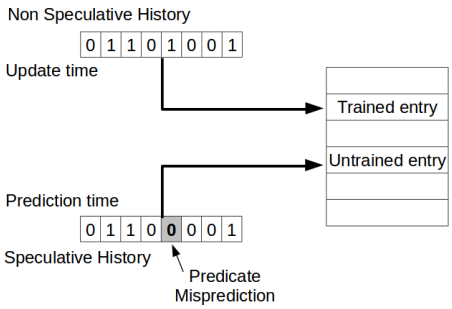


Figure 6: Corrupted branch-and-predicate history leads to read wrong predictor entry

handled through serialization then the speculative global branch-and-predicate history becomes possibly corrupted for subsequent instructions (see Figure 6). The high confidence filter would result in such a corrupted filter.

Therefore for *BrPred*, we use a coarse grain mechanism to decide whether or not predicate prediction will be used, i.e. we use a dual-mode with *predicate-prediction use* on and *predicate-prediction use* off. Switching from *off* mode to *on* mode necessitates to drain the pipeline and to restart with a correct speculative global branch-and-predicate branch history. Therefore one can not switch mode too often. We found that the simple coarse grain mode switching heuristic described below is quite effective.

BrPred is continuously updated and monitored at commit time, and the *on* or *off* mode for predicate prediction use is selected periodically. The chosen interval is 10,000 committed predicated instructions. *Predicate-prediction use* is enabled if the number of *BrPred* mispredictions in the interval is smaller than 500. In the remainder of the paper, we will refer to using *BrPred* using this *OnOff* filter as *BrPred-OnOff*.

4.1.3 Prediction usage

As already mentioned, predicate prediction is systematically used to neutralize the predicated false instructions in *BrPred* or in *On* mode for *BrPred-OnOff* while in *BrO-HighConf* only high confidence predictions are used. Moreover, if the predicate is already known at rename, the effective predicate is used.

On our simulator model, even a predicted false predicated instruction is stored in the issue queue. This predicted false instruction stays in the issue queue till the predicate prediction is checked. On a misprediction, execution has to be repaired.

4.2 Selective Replay for Predicate Mispredictions

In case of a predicate misprediction, some instructions following the mispredicted predicated instruction may have already been executed. A selective replay mechanism can be used to correct the execution instead of just squashing the pipeline and re-executing the whole sequence of instructions. The selective replay identifies and re-executes only the instructions which are in the dependency chain of the mispredicted instructions. On current high-end (x86) processors, selective replay is implemented to deal with events like L1 cache hit/miss misprediction or L1 cache bank conflicts.

However in case of a mispredicted predicate, a major difficulty arise: the register renaming is now incorrect. In order to deal with this issue without completely flushing the pipeline on

each mispredicted predicate, we propose to adapt SYRANT [15], a hardware mechanism that was initially proposed for exploiting control independence on a non-predicated instruction set.

4.2.1 Symmetric Resource Use

In order to allow the exploitation of control independence, SYRANT [15] enforces the same out-of-order engine resource consumption on both the taken and not-taken paths of a branch, the resource being the physical registers, the ROB entries and the load-store queue entries. To enable this, SYRANT allocates the same number of entries for both paths. Forcing this equal allocation is quite complex for conditional branches.

However, in this study, we only consider exploiting control independence from predicated instructions. In this case, the equal allocation of resource to both paths (predicate true and predicate false) is immediate: registers, ROB entries, LSQ entries are allocated even if the predicate is predicted false. That is renaming of instruction results does not depend on the predicate prediction and a ROB entry is always allocated. The (predicted) dependency chain at rename time still depends on the predicate prediction. Therefore the renaming of the register operands has to be replayed.

4.2.2 Initiation of a replay on a predicate misprediction

In case of a repair on a predicate misprediction, the sequence of fetched instructions with the correct prediction is exactly the same as the predicted sequence. Therefore our model assumes a buffer where all fetched instructions are stored. This buffer size is equal to the maximum number of inflight instructions in the processor.

The renaming is re-instantiated from this buffer at full speed and with the correct value of the predicate. At the same time, the prediction of the predicates is re-initiated with the correct history (when executing in *BrPred* mode).

4.2.3 Identifying valid instruction results

Some instructions following the mispredicted predicated instruction have already been executed. In order to save their results, our mechanism preserve the register renaming allocation and the LSQ allocation. However, one has also to assert the validity of the preserved results: this is illustrated on Figure 7.

On this example, I_3 must be re-executed since one of its operand registers has changed, its result is therefore invalid. Despite using the same operand registers, instruction I_4 must also be re-executed since one of its operand register (produced by I_3) is invalid.

To resolve this result validity issue, we implement the exact same rename-sequence tag solution that was described for SYRANT [15]. This mechanism preserves the results of data independent instructions that follow the mispredicted instruction.

5 Experimental study

The experimental study for validating our propositions was built upon the Gem5 simulator [4].

5.1 Simulator parameters

Unless otherwise mentioned, the simulator models an aggressive 4-way superscalar processor with a 128-entry ROB, a 64-entry Load Queue, 64-entry Store Queue and 256 physical integer and

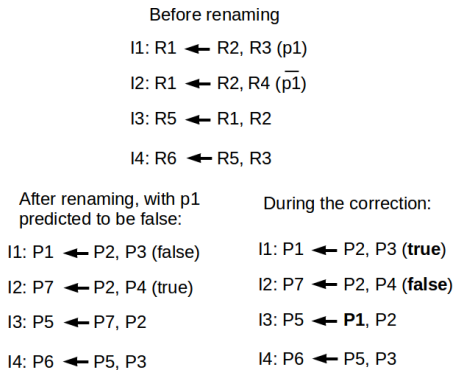


Figure 7: Instructions that do not have the same rename form before and after the correction need to be re-executed. Instructions depending on these re-executed instructions also need to be re-executed.

floating point registers. The processor also features a state-of-the-art conditional branch predictor, the TAGE predictor described in [18]. The store sets predictor [6] is used to predict memory dependencies. A mechanism is used to improve the management of the Return Address Stack (RAS) [23]. When not predicted, predicated instructions are executed through the aggressive mechanism described in Section 2.3.2. The other characteristics are summarized in Table 2. We will refer to this configuration as the base configuration (BASE).

5.2 Benchmarks

The simulated benchmarks constitute a subset of the Spec 2006 benchmarks set [21] listed in Table 3. To reduce the amount of simulation time, we use the Simpoint methodology [9] to summarize each benchmark in a set of 100 millions instructions slices. Each slice is representative of a part of the benchmark execution and is affected a weight representing the portion that it represents in the execution. For each benchmark, the illustrated results are the weighted mean of simulations on the set of slices. Table 3 display the weighted mean of the Instruction Per Cycle (IPC) count for each benchmark, for the 4-way and 8-way BASE configurations.

As we have been targeting the ARM instruction set, some of the benchmarks or some of their input sets are missing. There are three reasons why some benchmarks are missing: 1) the binary produced by our cross-compiler is not executable on a native ARM architecture, 2) the binary is not executable on *qemu-arm* [3] which was used to compute the basic block vector (BBV) needed to compute the simpoints 3) the simulator is not able to run them. In the end, we were able to run 12 integer benchmarks (the complete integer benchmarks) and 7 floating point benchmarks. Some benchmarks are used with several inputs (all the inputs that are working are used). In total, we were able to simulate 38 different workloads.

The binaries were generated with the gcc compiler using the O3 optimization level. Gcc decision to if-convert a branch mainly depends on the number of instructions that are controlled by the branch. By default, for the ARM target, this number is set to 4.

5.3 Benchmarks ratio of predicated instructions

Table 3 also list the ratio of predicated instructions per benchmark. The first column presents the total percentage of predicated instructions. This includes the conditional branches which are not predicted by the predicate predictor. The second column excludes conditional branches.

OoO@1GHz	4-way	8-way
Memory	100 cycles 12.8GBps, Across a 128B bus	
Caches	L1D 4-way 64KB, 64B, 1 cycle L1I 4-way 64KB, 64B, 1 cycle L2 8-way 4MB, 64B, 8 cycles Stride Prefetcher for the L2	
TLBs	Perfect, 4K pages	
ROB	128 entries	256 entries
IQ	128 entries	256 entries
LSQ	128 entries (64L/64S)	256 entries (196L/64S)
Width (F/D/R/I/E/W/C)	4	8
Pipeline	12 stages	
FU(latency) IntAlu(1) IntMultDiv(3/12*) FpAlu(5) FpMultDiv(4/9*) Ld/Str(2)	 3 1 2 2 2	 6 2 4 4 4
Branch Predictor	BTB 4-way, 1K entries RAS, 16 entries, WP corruption detection TAGE 1+12 components, 15K entries total	
Misprediction penalty	15 cycles	15 cycles

Table 2: Simulator configuration overview. *not pipelined.

For all benchmarks, conditional branches instructions represent a large part of the predicated instructions. However, some benchmarks like *401.bzip2*, *403.gcc*, *445.gobmk* and *456.hmmer* contain a significant portion of effective predicated instructions. Some other benchmarks, like *436.cactusADM*, *459.GemsFDTD* and *470.lbm* feature nearly no effective predicated instructions.

A simple optimization to save energy would be to monitor at run-time the ratio of effective predicated instructions and to turn-off the predicate prediction when this ratio is under a predefined threshold.

5.4 Simulation Results

We report simulation results (speed-ups over the base configuration) assuming a 4-way superscalar processor, except in Section 5.4.4 which shows that trends are amplified for a 8-way superscalar processor.

5.4.1 Branch history vs Branch-and-Predicate

First experiments assuming systematic use of predicate prediction (Figure 8) shows that using branch-and-predicate history allows in general slightly higher performance than using branch history only. In some cases, the difference is quite significative, e.g. on *462.libquantum* or *401.bzip.liberty*.

Benchmarks	**Input**	**IPC (BASE)**		**% predicated**	
		4-way	**8-way**	**with branches**	**without branches**
400.perlbench	checkspam	1.46	1.82	17.17	4.57
	diffmail	1.31	1.57	16	4.68
401.bzip2	chicken	2.21	3.01	15.17	3.49
	combined	1.73	2.16	15.77	5.20
	liberty	2.33	3.38	17.98	5.72
	program	1.85	2.23	14.82	3.92
	source	1.66	2.02	17.75	5.45
	text	2,11	3.15	17.28	4.29
403.gcc	166	1,7	2.37	35.91	24.23
	200	1.58	2.1	31.07	18.69
	c-typeck	1.65	2.41	44.44	34.27
	cp-decl	1.72	2.46	37.1	25.9
	expr	1.84	2.71	38.16	26.12
	scilab	1.47	1.92	29.96	17.6
416.gamess	cytosine	2.62	4.6	7.01	2.85
	h2ocu2+	2.76	5.09	6.39	2.93
429.mcf	ref	0.7	0.81	24.5	6.10
435.gromacs	ref	2.81	4.74	4.62	1.02
436.cactusADM	ref	2.26	4.3	0.09	0.02
444.namd	ref	2.49	3.95	8.54	5.18
445.gobmk	13x13	1.75	2.27	18.92	7.39
	nngs	1.73	2.24	18.02	6.79
	trevorc	1.71	2.22	18.39	7.13
	trevord	1.87	2.52	16.86	6.11
453.povray	ref	1.48	2	8.09	1.9
456.hmmer	nph3	2.62	5.21	17.18	14.55
	retro	2.48	4.61	17.64	14.78
458.sjeng	ref	1.82	2.38	18.76	7.05
459.GemsFDTD	ref	2.17	3.89	1.06	0.001
462.libquantum	ref	1.82	2.31	23.56	13.42
464.h264ref	baseline	2.04	3.59	6.45	2.68
	main	1.7	2.99	6.75	2.42
	sss	1.68	3.05	5.99	2.13
470.lbm	ref	1.88	2.36	0.6	0.02
471.omnetpp	ref	0.8	0.92	17.1	4.33
473.astar	BigLakes	1.29	1.57	15.53	3.88
	rivers	1.42	1.69	15.7	3.29
483.xalancbmk	ref	1.81	2.49	21.09	3.4

Table 3: Benchmarks, their inputs, their IPC for the BASE 4-way and 8-way configurations and the ratio of predicated instructions over the total number of instructions.

Note that for some applications, the ratio of non-branches predicate instructions is rather low, but the performance improvement is significant e.g. *429.mcf* or *471.omnetpp.ref*

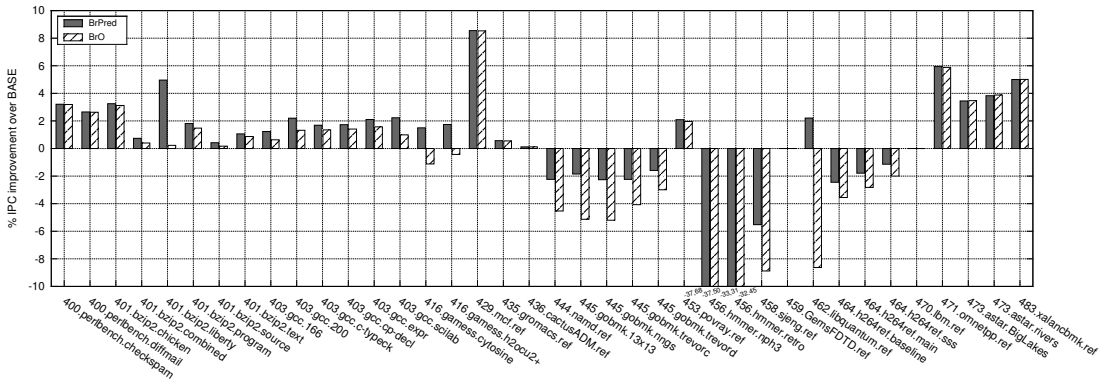


Figure 8: Speedups using *BrPred* (branch-and-predicate history) and *BrO* (branch history)

However in many cases (e.g. *444.namd*, *445.gobmk*, *464.h264ref*, ...), systematic use of predicate prediction induces performance losses compared with the base configuration. This performance loss can be quite high e.g. 37% on *456.hmmer* and therefore systematic predicate prediction use should not be considered for implementation in real hardware.

5.4.2 Filtering predicate predictions

We have proposed two different ways for selectively using predicate predictions, the main goal being to avoid the dramatic performance losses described above.

When using global branch history, *BrO-HighConf* selects the predicate prediction use at very fine grain based on the confidence of the predicate prediction. When using global branch-and-predicate, *BrPred-OnOff* enables the predicate prediction use at a coarse grain granularity.

Figures 9 and 10 illustrate the associated experiments. As expected, these filters essentially remove the performance losses induced by high predicate misprediction rates. However they also maintain and sometimes enhance the performance benefits enabled by predicate prediction when encountered.

When using *BrO* (i.e global branch history only), the encountered performance losses compared with baseline are now becoming marginal (maximum 2.2% on *456.hmmer*). But also, through removing many of the mispredictions, but preserving most of the correct predictions, the high confidence filter allows significant improvement on most of the benchmarks (e.g. *401.bzip2* or *403.gcc*).

When using *BrPred* (i.e. global branch-and-predicate history), the *OnOff* filter removes all the performance loss at the exception of *444.namd* (1.9%). The *OnOff* filter is also very effective at removing the predicate mispredictions and enables some performance improvements.

5.4.3 Benefits of Selective Replay

Our selective replay mechanism intends to reduce the performance loss associated with predication misprediction. This is illustrated in Figures 11 and 12.

For both *BrO* and *BrPred*, some marginal performance improvement is obtained (generally less than 1%). In practice, the high confidence filter and the *OnOff* filters are quite radical at eliminating the use of prediction of hard-to-predict predicates, thus the general benefit that can be obtained on reducing predicate misprediction penalty is low. However (not illustrated) preliminary experiments on systematic use of predicate prediction and our selective replay were showing

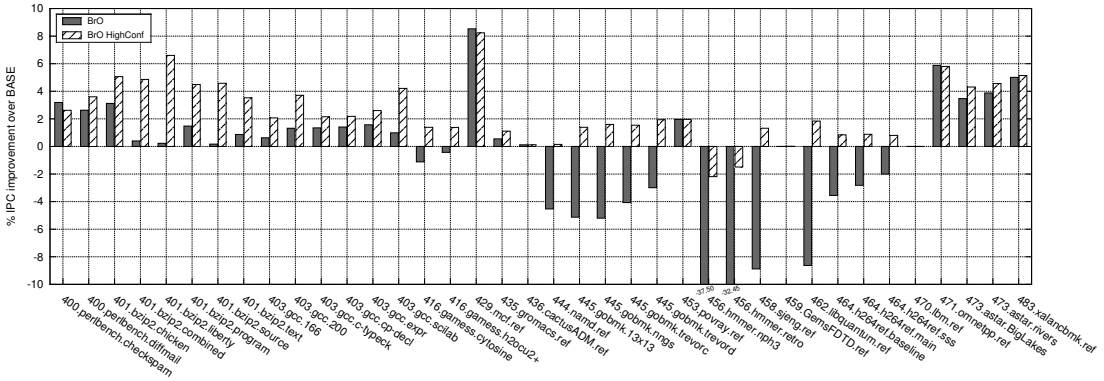


Figure 9: Speedups using *BrO-HighConf* (selective prediction and branch history)

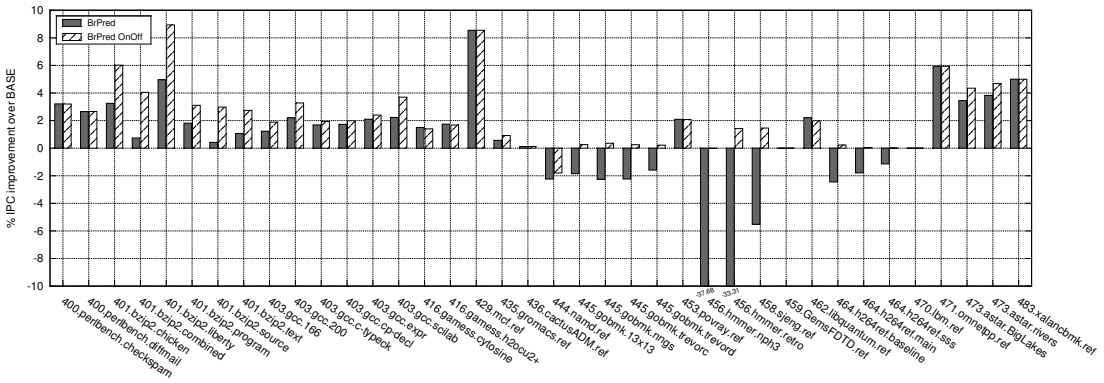


Figure 10: Speedups using *BrPred-OnOff* (selective prediction and branch-and-predicate history)

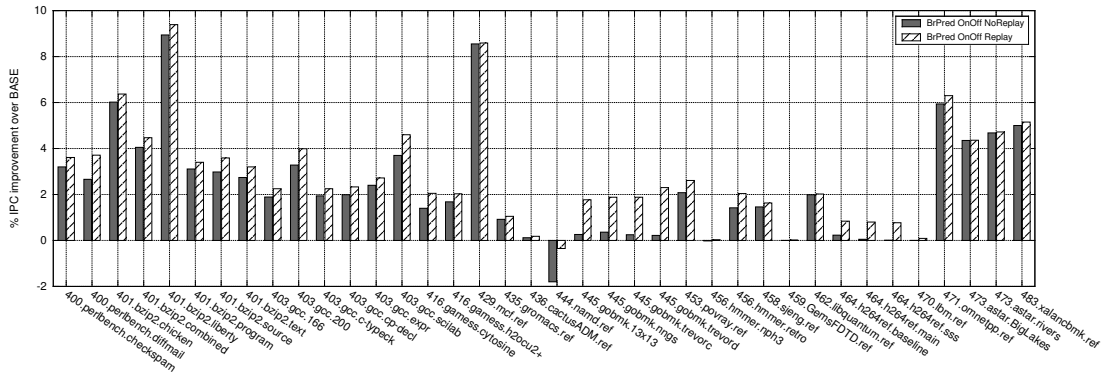


Figure 11: SPREPI: Selective replay and *BrPred-OnOff*

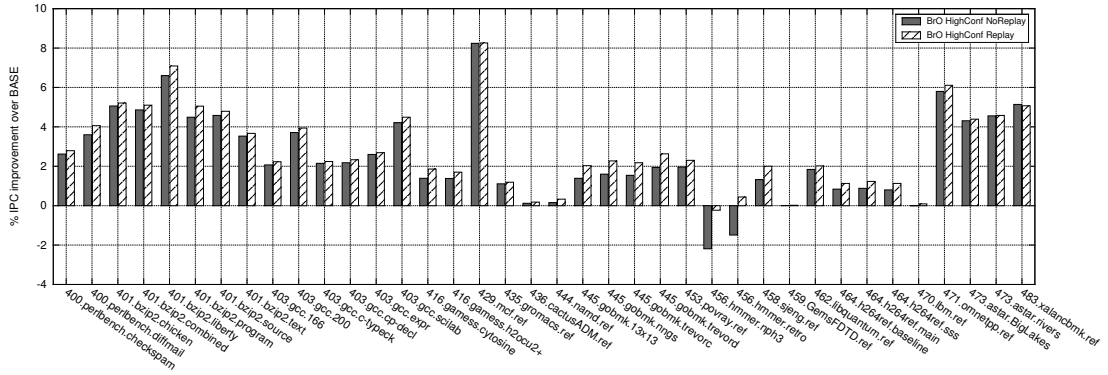


Figure 12: SPREPI: Selective replay and *BrO-HighConf*

that it absorbs quite well the misprediction penalty: for instance, it reduces the performance loss on *456.hmmer* from 37% to 9%.

5.4.4 Benefits of SPREPI on a wide issue superscalar processor

SPREPI speedups on a 4-way superscalar processor is limited to a few percent (up to 9% on two of our workloads). However, this advantage is growing when one considers a more aggressive implementation featuring a wide issue out-of-order engine. Figure 13 illustrates this on a 8-way superscalar processor for *BrPred*. The speedup over an 8-way base superscalar grows to up to 17% and the relative speedup is systematically higher for 8-way issue than for 4-way issue for nearly every benchmark.

5.4.5 Aggressive vs non-aggressive predicated instruction execution

Performance of out-of-order execution of predicated code depends on how multiple definition issue is handled. So far we have assumed the quite aggressive predicated instruction execution described in Section 2.3.2. Simulations were also run assuming the much less aggressive working solution described in Section 2.3.1.

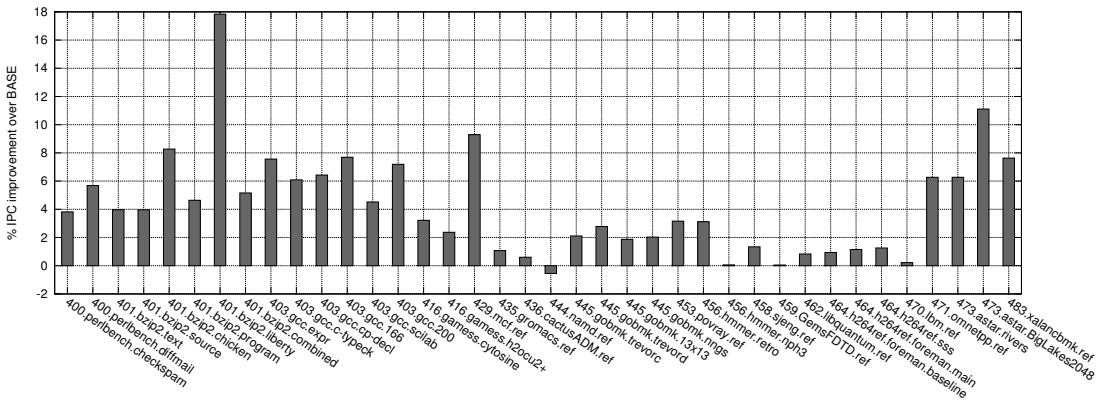


Figure 13: Benefits of SPREPI on a 8-way issue superscalar processor

Figure 14 illustrates the relative performance of a non aggressive predicated instruction execution labeled *BASE Non Aggressive*, SPREPI on top of the non aggressive implementation and SPREPI on top of the aggressive implementation compared with the BASE implementation (aggressive predicated instruction execution).

At a first point, the performance difference between *BASE aggressive* and *BASE non-aggressive* implementation is already quite limited. A few percent ($< 4\%$) performance loss is sometimes encountered, but many of our benchmarks only suffer from a very marginal performance loss. *470.lbm* even encounters a performance benefit; however since *470.lbm* exhibits only 0.02% non-branch predicated, this result is probably associated with some scheduling artifact.

Since most of the predicated instructions are predicted, the performance difference between *SPREPI Aggressive* and *SPREPI non-aggressive* is becoming even less significant: less than 1 % except for *458.sjeng.ref* in our experiments.

The performance difference between *BASE aggressive* and *BASE non-aggressive* would not probably justify the extra hardware complexity and power consumption for *BASE aggressive* (extra register ports, wider bypass network, more complex issue logic, ..). When considering SPREPI on top of these two possible implementations, an aggressive implementation is obviously not worth this extra hardware complexity and power consumption.

6 Conclusion and Perspective

ARM based processors are becoming ubiquitous in many modern appliance including smartphones and tablets. The demand for high performance pushes manufacturers of ARM processors to use the same techniques that have been used for the two last decades on PCs and servers processors including wide-issue superscalar processors. 32-bit ARM ISA features predicated instructions. Providing an efficient solution to efficiently execute predicated instructions out-of-order is challenging due to the multiple definition problem.

In this paper, we have shown that state-of-the art branch prediction solutions can be adapted to predicate prediction. Predicate prediction resolves the multiple definition problem and per se brings some extra performance on some applications with a reasonable predicate prediction accuracy; using branch-and-predicate history instead of branch history further improves performance. However, systematic predicate prediction use is not always effective and applications/section of codes with high predicate misprediction rates perform generally worse than without predicate

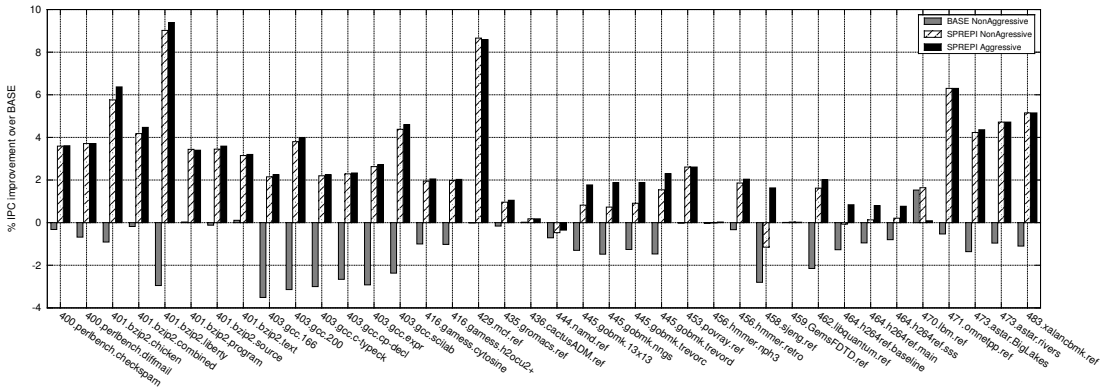


Figure 14: Aggressive vs Non-aggressive predicated instruction execution

prediction. We have shown that efficient filtering of predicate prediction use can be implemented for both predicate prediction using branch history and predicate prediction using branch-and-predicate history.

As the second contribution of the paper, we have adapted the SYRANT selective replay mechanism [15] to selective replay mispredicted predicate instructions. SYRANT was initially designed for exploiting general control independence in out-of-order processors. Adapting it to handle only predicated instructions is straightforward and greatly simplified the design. This selective predicated instruction replay reduces the performance penalty associated with predicate misprediction.

The combination of these two mechanisms, Selective Prediction and Replay for Predicated Instructions (SPREPI) achieves speed-ups on codes generated with the gcc standard compiler. This performance gain is within a few percent for a 4-way superscalar processor, but grows with issue width (up to 17% on some applications with a 8-way issue processor).

Through transforming the execution of most of the predicated instructions in execution of non-predicated instructions (at the risk of misprediction), SPREPI reduces the number of serializations due to predicated instructions. SPREPI almost annihilates the performance benefit of hardware optimizing these serializations, thus enabling a more cost and power effective implementation.

Standard solutions to implement out-of-order execution of predicated ISA tend to push compiler writers to avoid using predicated instructions since those instructions are likely to induce some serialization on the execution. On the other hand, our SPREPI proposal could allow the compiler to use if-conversion more aggressively. Our initial study (out-of-the-scope of this paper) on increasing the size of the potential if-converted branches in the gcc compiler tends to indicate that this is a direction that might be explored if a SPREPI-like mechanism was implemented in effective hardware.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1983, pp. 177–189.

[2] ARM, "Arm architecture reference manual. arm v7-a and arm v7-r edition."

[3] F. Bellard, "QEMU," *http://wiki.qemu.org/Main_Page*.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[5] 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," *International Journal of Parallel Programming*, vol. 24, no. 3, pp. 209–234, 1996.

[6] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, 1998, pp. 142–153.

[7] W. Chuang and B. Calder, "Predicate prediction for efficient out-of-order execution," in *Proceedings of the 17th annual international conference on Supercomputing*, 2003, pp. 183–192.

[8] 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.*, vol. 13, no. 4, pp. 451–490, 1991.

[9] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0 : Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. vol. 7, September 2005.

[10] Intel Corp, "Intel itanium architecture software developer's manual. volume 3: Instruction set reference," 2002.

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

[12] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt, "Diverge-merge processor (dmp): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 53–64.

[13] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt, "Wish branches: Combining conditional branching and predication for adaptive predicated execution," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005, pp. 43–54.

[14] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ilp processors," in *Proceedings of the 21st annual international symposium on Computer architecture*, ser. ISCA '94, 1994, pp. 120–129.

[15] N. Premillieu and A. Seznec, "Syrant: Symmetric resource allocation on not-taken and taken paths," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 43:1–43:20, Jan. 2012.

[16] E. Quiñones, J.-M. Parcerisa, and A. Gonzalez, "Selective predicate prediction for out-of-order processors," in *Proceedings of the 20th annual international conference on Supercomputing*, 2006, pp. 46–54.

[17] A. Seznec, "Storage free confidence estimation for the tage branch predictor," in *Proceedings of the 17th International Symposium on High-Performance Computer Architecture*, 2011, pp. 443–454.

[18] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction Level Parallelism*, February 2006.

[19] B. Simon, B. Calder, and J. Ferrante, "Incorporating predicate information into branch predictors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003, pp. 53–64.

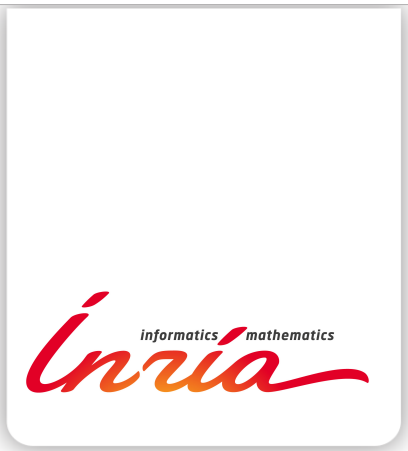
[20] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*, 1981, pp. 135–148.

[21] SPEC, "SPEC CPU2006," *http://www.spec.org/cpu2006/*, 2006.

[22] G. S. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 196–206.

[23] H. Vandierendonck and A. Seznec, "Speculative return address stack management revisited," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 3, pp. 15:1–15:20, Dec. 2008.

[24] P. H. Wang, H. Wang, R.-M. Kling, K. Ramakrishnan, and J. P. Shen, "Register renaming and scheduling for dynamic execution of predicated code," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001, pp. 15–25.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399