Selective Predicate Prediction for Out-of-Order Processors

Eduardo Quiñones
Computer Architecture Department
Technical University of Catalonia
equinone@ac.upc.edu

Joan-Manuel Parcerisa
Computer Architecture Department
Technical University of Catalonia
jmanel@ac.upc.edu

Antonio Gonzalez
Intel Barcelona Research Center
Intel Labs Barcelona
antonio.gonzalez@intel.com

ABSTRACT

If-conversion transforms control dependencies to data dependencies by using a predication mechanism. It is useful to eliminate hard-to-predict branches and to reduce the severe performance impact of branch mispredictions. However, the use of predicated execution in out-of-order processors has to deal with two problems: there can be multiple definitions for a single destination register at rename time, and instructions with a false predicated consume unnecessary resources. Predicting predicates is an effective approach to address both problems. However, predicting predicates that come from hard-to-predict branches is not beneficial in general, because this approach reverses the if-conversion transformation, loosing its potential benefits. In this paper we propose a new scheme that dynamically selects which predicates are worthy to be predicted, and which one are more effective in its if-converted form. We show that our approach significantly outperforms previous proposed schemes. Moreover it performs within 5% of an ideal scheme with perfect predicate prediction.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*pipeline processors*

General Terms

hardware design

Keywords

if-conversion, predicate prediction, confidence prediction

1. INTRODUCTION

Branches are recognized as a major impediment to exploit instruction-level parallelism (ILP). The use of branch prediction in conjunction with speculative execution is typically used to remove control dependencies and expose ILP. However branch mispredictions can result in severe performance penalties that tend to grow with larger window sizes and deeper pipelines.

ICS06, June 28-30, Cairns, Queensland, Australia.


```
(a) mov r34 = 0;
    if(r32 == r33)
        mov r34 = 1;
    else
        mov r34 = 2;
    mov r35 = r34;

(b) mov r34 = 0;            mapped to ph1
    cmp.eq p6,p7 = r32,r33;
    (p6) mov r34 = 1;       mapped to ph2

    (p7) mov r34 = 2;       mapped to ph3
    mov r35 = r34; <-- which is the correct map?
```

Figure 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

If-conversion [2] is a technique that helps to eliminate hard-to-predict branches, converting a control dependence into a data dependence and potentially improving performance. Hence, if-conversion allows the compiler to collapse multiple control flow paths and schedule them based only on data dependencies. If-conversion takes full advantage of predicate execution. Predication is an architectural feature that allows an instruction to be guarded with a boolean operand whose value decides if the instruction is executed or converted into a no-operation.

Some studies have shown that predicated execution provides an opportunity to significantly improve hard-to-predict branch handling in out-of-order processors [4] [12]. This advantage tends to be even more important with larger window sizes and deeper pipelines. However, the use of predicate instructions has two performance issues on such processors:

1. When multiple control paths are collapsed, multiple register definitions are merged into a single control flow, and they are guarded with a predicate. At runtime, each definition is renamed to a different physical register, thus existing multiple possible names for the same logical register until the predicates are resolved. Since predicates are resolved at the execute 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 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 resour-

ces such as physical registers, issue entries and/or functional units, and can potentially degrade performance.

To avoid the problem of multiple register definitions and the unnecessary resource consumption, it would be desirable to know the predicate value of an instruction at renaming. A naive approach could stall the renaming until the predicate is resolved, but it would causes a serious performance degradation.

The transformation of a predicate instruction into a *cmov-like alpha instruction* and the generation of a special micro-operation are two proposed solutions to the multiple register definition problem [15]. However they present two important problems: 1) false predicated instructions still consume processor resources; 2) these techniques introduce new data dependencies that make the dependence graph deeper.

Predicting the predicates is another effective approach that addresses both problems [5]. Instructions with a predicate predicted to false are speculatively cancelled at the rename stage and removed from the pipeline, thus avoiding multiple definitions and avoiding also the resource pressure caused by cancelled instructions.

In some way, the prediction of predicates undoes if-conversion transformations done by the compiler. Therefore, since if-conversion appears to be more effective than branch prediction for hard-to-predict branches [4], applying prediction to all predicates misses the opportunities brought by if-conversion.

Hence in this paper we propose a new scheme that *selectively* predicts predicates based on a confidence predictor: a predicate is predicted only if the prediction has *enough* confidence. If not, the if-converted form remains but the predicated instruction is converted to a *cmov-like* instruction to avoid the multiple definition problem. Our approach tries to preserve if-conversion for hard-to-predict branches. We have found that up to 84% of if-conversion transformations are maintained. Our proposal outperforms a previous non-seletive prediction scheme by more than 11%. Compared to previous techniques without prediction, the speedups of our approach is 13%. Our technique performs within 5% of a scheme with perfect predicate prediction.

The paper is organized as follows. Section 2 discusses the state of the art on predication in an out-of-order execution model. Section 3 describes our proposed technique and discusses several design issues about the confidence mechanism. Section 4 presents the experimental results obtained. Section 5 discusses complexity issues. Finally, the conclusions are presented in section 6.

2. PREDICATION ON OUT-OF-ORDER PROCESSORS

Predication was proposed by Allen et al. [2]. Since then, many authors have focused their studies on the generation of efficient predicated code. In this section we will shortly review only those studies that focus on predication for out-of-order processors.

2.1 Software Approaches

Chang. et.al. [4] 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. Mahlke et al. [12], studied the benefits of partial and full predication code in an out-of-order execution model to achieve speedups in large control-intensive programs. The paper shows that, in comparison to a processor without predication support, partial predication improves performance by 33%, whereas full predication improves performance by 63%. August et.al. [3] showed that an effective compile strategy for predicated execution has to address *which* instructions are predicated and *where* are they scheduled. They argue that a detailed analysis of the dynamic program behavior and the availability of resources are required, since multiple control paths have to share processor resources.

Predication has already been implemented in real out-of-order processors. Many processors such as Alpha [10] or PowerPC [8] implement partial predication. This approach represents an attractive solution for designers since the required changes to existing instruction set architectures (ISA) and data paths are minimized. However, the use of full predicate execution provides more flexibility and a larger potential performance improvement [12].

Kim et.al. [11] have recently proposed a mechanism in which the compiler generates code including special *wish branch* instructions that can be executed either as predicated or non-predicated code based on a run-time confidence estimator. Unlike our proposal, this is a combined software/hardware technique that requires the compiler support.

As far as we know, just a few studies have proposed pure hardware techniques to use full predicate execution in out-of-order processors. These studies are explained in the following sections.

2.2 False Predicated Conditional Moves

A common way to solve the multiple register definitions problem is to change the semantic of predicated instructions. This new functionality can be expressed in a C-style operation: *register definition = (predicate)? normal execution : previous register definition.* If the predicate is evaluated to true, the destination register is updated with the normal instruction computation. However, if it is evaluated to false, the destination register is updated with the value of the previous register definition. In fact, the instruction copies the value from its previous physical register to the newly allocated physical register.

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. Moreover, instructions with a false predicated are not early-cancelled from pipeline, so they continue consuming physical registers, issue queue entries and functional units. The new

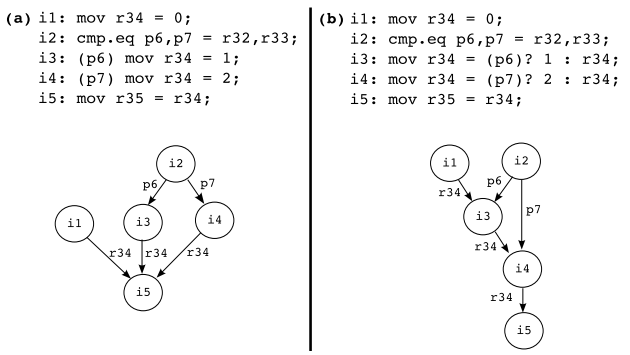

Figure 2: Data dependence graph changes when predicate instructions are converted to *false predicated conditional moves*. (a) Original code. (b) Converted Code.

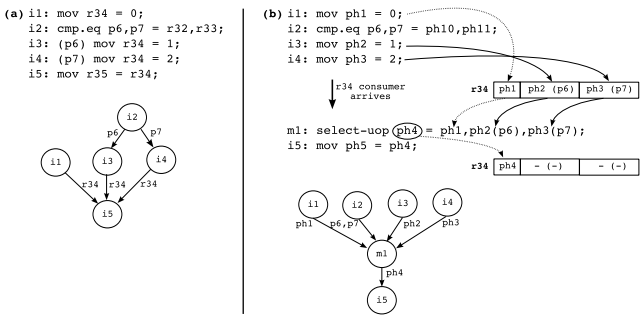

Figure 3: Generation of select-μops at the rename stage. (a) Original code. (b) The code has been renamed and the *r34* RAT entry modified. When instruction *i5* consumes *r34*, a select-μop is generated.

dependencies are shown on the right side of the Figure 2b.

2.3 Generation of Select-μops

Kling et al. [15] 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) [6].

They propose an augmented Register Alias Table (RAT). Each entry is expanded to record the physical register identifiers of all the 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 RAT. If multiple definitions are found in the RAT 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.

Figure 3 shows how the select-μop works. Figure 3a shows the original code before renaming. Figure 3b, shows the same code after renaming, as well as the modifications of the *r34* RAT entry. Predicated instructions *i3* and *i4* fill the extra RAT 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 complexity of the RAT is substantially increased to hold multiple register definitions and their predicate guards. In addition, 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 predicated are not early-cancelled from the pipeline so they continue consuming physical registers, issue queue entries and functional units.

2.4 Predicate Prediction with Selective Replay Mechanism

Predicting predicates is a good solution to overcome predication problems derived from out-of-order execution, because all predicates are known at rename stage. However, predicate mispredictions can produce a high performance degradation.

Chuang et al. [5] proposed a selective replay mechanism to recover the machine state from predicate mispredictions without flushing the pipeline. 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* (see section 2.2), 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. Figure 4 illustrates the execution of the predicted and the replay data dependence graphs.

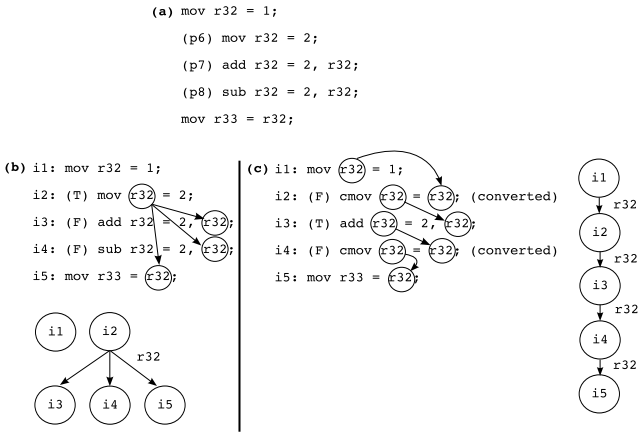

Figure 4: Selective replay. (a) Original code. (b) Data dependencies after predicate prediction if no misprediction occurs (assuming *p6=true, p7=false, p8=false*, *i3* and *i4* are inserted into the IQ but do not issue). (c) Data dependencies after predicate misprediction (assuming *p6=false, p7=true, p8=false*, *i2* and *i4* are converted to *false predicated conditional moves* so the correct *r32* value is propagated through the replay data dependence graph).

The proposal of Chuang et al. is based on the IA64 ISA. IA64 is a full predicate ISA, with a wide set of compare instructions that produce predicate values. These instructions are classified in 11 types. However, Chuang's proposal only predicts one type (the so called *unconditional* [9]). 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 can not 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 60% of compare instructions are unconditional.

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.

3. SELECTIVE PREDICATE PREDICTION

The use of predicate execution in out-of-order processors has to deal with two problems: the multiple definitions of a source register at rename time, and the unnecessary resource consumption caused by instructions with a false predicated. However, previous proposals have only focused on the first problem. Moreover, for proposals such as generation of select-μops [15] or *false predicated conditional moves* technique, instructions with false predicateds not only consume registers, issue queue entries, etc. but they also compete for execution resources. Still worse, the select-μops technique further aggravates the resource pressure with these extra pseudo-instructions injected in the pipeline.

The prediction of predicates appears to be 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 before renaming. Unfortunately, the selective replay mechanism [5] precludes this advantage because it needs to maintain all predicated instructions in the issue queue, regardless of the predicate being predicted true or false.

The prediction of predicates technique generates a prediction for every compare instruction that produces a predicate, and speculatively executes the instructions that are guarded with it. In fact, this technique produces an effect similar to branch prediction and speculation or, in other words, it reverses the if-conversion transformations. For predictable predicates, the code that results after prediction is more effective because it avoids unnecessary resource consumption. However, the if-converted form is more effective for hard-to-predict branches, because of the high misprediction penalties. Since selective replay predicts all predicates regardless of whether they are predictable or not, it loses the opportunities brought by if-conversion. On the positive side, selective replay avoids flushing the pipeline on predicate mispredictions, so it reduces its high associated penalties. However, note that this advantage is only important for hard-to-predict branches.

Here, we propose a selective predicate prediction scheme that addresses the multiple definitions problem. Unlike selective replay, the selective predicate prediction tries not to lose the potential benefits of if-conversion. It dynamically identifies easy-to-predict predicates with a confidence predictor: a predicated instruction is speculated only when its predicate is considered easy-to-predict; otherwise it is preserved in its if-converted form. In the latter case, the multiple definitions problem is avoided by converting the predicated instruction into a *false predicated conditional move* (see section 2.2).

With selective predicate prediction, instructions whose predicate is predicted false are cancelled and eliminated from the pipeline before renaming, so they do not consume resources unnecessarily. 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 produce higher penalties than with selective replay. 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 mechanism in detail.

3.1 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 IA64.

A predicate prediction is produced for every instruction that has a predicate outcome, such as compare instructions, and it is generated for every producer in early stages of

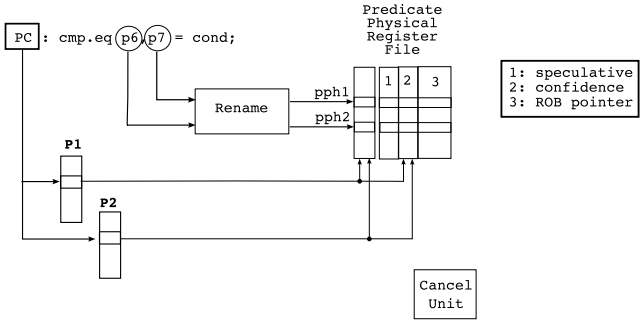

Figure 5: Generation of a predicate prediction

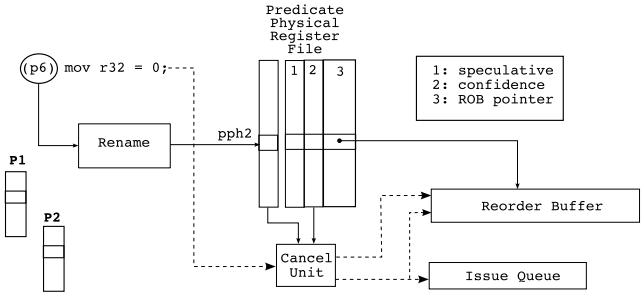

Figure 6: Predicate consumption at the rename stage

the pipeline from its *PC*. Figure 5 illustrates the prediction mechanism for producers. Since compare instructions have two predicate outputs, we have two predictors, 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 *pph1* and *pph2* 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 both predicted and 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 instructions that use it, such as predicated instructions.

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 points 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. 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 set to true, the Cancel Unit (CU), cancels the instruction and eliminates it from the pipeline. Figure 6 shows the predicate consumption scheme.

Notice that if the two outcomes of compare instructions were closely correlated, probably a single predictor would suffice. However, we found that two separate predictors are needed because these outcomes not only depend on the compare instruction type and its condition result, but they also depend on information that is not available in the front-end [9].

3.2 The Confidence Predictor Mechanism

This section introduces a confidence predictor [7] to the basic predicate prediction mechanism. The confidence predictions are used to select which if-conversion transformations are worthy 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 entry contains 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 the 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 to the *confidence* field of the PPRF. Figure 5 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; the confidence counter is updated according to the prediction success and the *speculative* bit is set to false.

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* (see section 2.2). The new semantic of the instruction can be represented in a C-style form: *result = (predicate)? normal execution : previous register definition.*

3.3 Confidence Threshold

The confidence threshold plays an important role in our proposal. On the one hand, high threshold values could result in lost opportunities of correct predictions. 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 4.1.

Figure 7 depicts the performance impact of different static confidence thresholds (values 2, 4, 8, 16 and 32). As shown in the chart, 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 in-

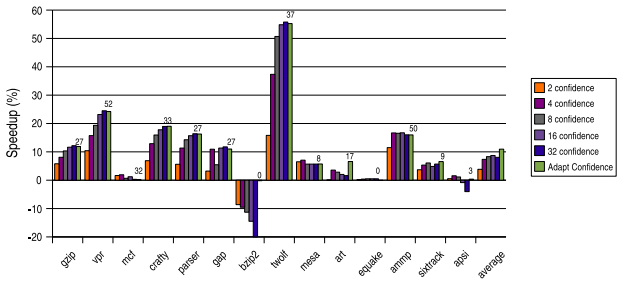

Figure 7: Performance impact of several confidence threshold values. A scheme without confidence has been chosen as baseline.

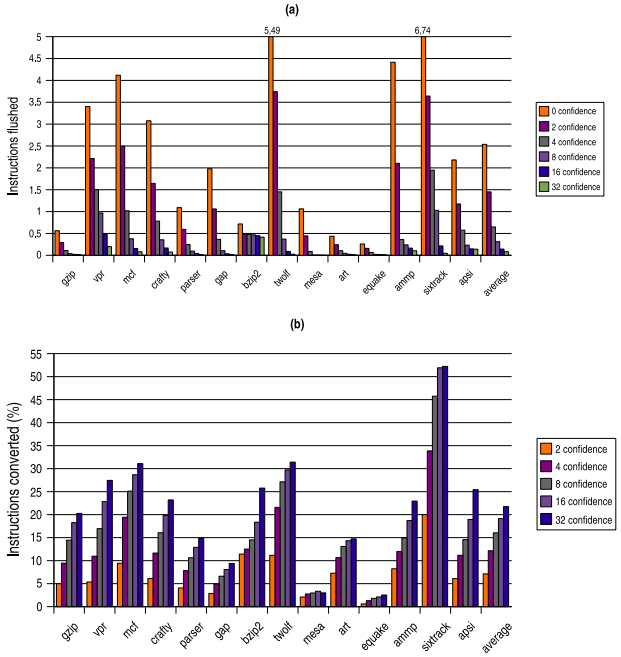

Figure 8: Variance of confidence threshold. (a). Number of flushed instructions per predicated committed instruction. (b) Percentage of predicated committed instructions that have been transformed to *false predicated conditional moves*.

creases. This observation suggests that adjusting dynamically the threshold could result in a performance improvement. Hence, an adaptive confidence threshold technique is presented in the next section.

3.4 Adaptive Confidence Threshold

Adjusting the confidence threshold changes the frequency of two important events with an important impact on performance: reducing the threshold augments the number of flushes caused by predicate mispredictions; increasing the threshold causes more predicated instructions to be converted into *false predicated conditional moves* because of a non-confident prediction.

Figure 8 shows the relationship between the two effects: 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 moves*. It is shown how threshold increases reduce the number of flushes but increase the number of converted instructions.

Comparing these results with those in Figure 7 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*, hence the different performance impact in each case. Notice also that for *bzip2* or *art* the number of flushes becomes stable at low thresholds, so the increase 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.

Both pipeline flushes and converted instructions have a negative performance impact, but not in the same way. Instructions converted to *false predicated conditional moves* increase resource consumption and may stretch the dependence graph. When the pipeline is flushed, the misspeculated instruction and all subsequent instructions must be re-fetched. An adaptive threshold mechanism must tradeoff 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 instructions that produce a predicate). 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 rules:

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 7 compares the performance of the adaptive threshold scheme (the bar labeled *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 static schemes by more than 2%.

4. PERFORMANCE EVALUATIONS

This section evaluates the performance of our selective predicate prediction scheme with adaptive confidence threshold. It is compared to the previous hardware approaches for predicated execution on out-of-order processors, described in section 2. The *false predicated conditional move* technique described in section 2.2 is taken as the baseline for all results. We have also evaluated a perfect predicate prediction scheme, for comparison purposes.

4.1 Experimental Setup

All the experiments presented in this paper use a cycle-accurate, execution-driven simulator that runs IA64 ISA binaries. This simulator has been built from scratch using the Liberty Simulation Environment (LSE) [14]. LSE is a simulator construction system based on module definitions and module communications. It also provides a complete IA64 functional emulator that maintains the correct machine state.

We have simulated fourteen benchmark programs from Spec2k [1] (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. For all benchmarks, 100 million committed instructions are simulated. To obtain representative portions of code to simulate, we have used Pinpoint tool [13].

The simulator models in detail an eight-stage out-of-order processor. It pays special attention to the implementation of the rename stage and models many IA64 peculiarities that are involved in the renaming, such as the register stack engine, the register rotation and the application registers. All instructions that produce predicates are take into account. Load-store queues, as well as the data and control speculation mechanisms defined in IA64, are also modeled and integrated in the memory disambiguation subsystem. The main architectural parameters are shown in Table 1.

To implement the predicate predictor we have evaluated the most common branch predictor designs and configurations. Our experiments show that, for a 2x16KB capacity, a *Gshare* predictor with 6 bits of global history is the best configuration. Surprisingly, a simple Last Value predictor achieved comparable performance, even better than more complex predictors such as two-level predictors. The adaptive confidence threshold mechanism monitors periods of 4096 predicate producer instructions. The confidence threshold is stored in a six bit counter that increments and decrements by one.

The simulator models also the generation of select-μops technique [15] in detail. Each RAT entry is augmented to record up to six register definitions with their respective guards, and there is an additional issue queue for select-μops.

Simulator Parameters	
Fetch Width	Up to 2 bundles (6 instructions)
Issue Queues	Integer Issue Queue: 80 entries Floating-point Issue Queue: 80 entries Branch Issue Queue: 32 entries Load - Store Queue: 2 separate 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
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 ITLB	512 entries. 10 cycles miss penalty 512 entries. 10 cycles miss penalty
Main Memory	120 cycles of latency
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

Table 1: Simulator parameter values used.

The predicate prediction with selective replay technique [5] is not 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.

4.2 Results

Figure 9 compares the performance of the different schemes as IPC speedups over the *false predicated conditional move* technique. For most benchmarks, our selective predicate prediction with adaptive threshold scheme significantly outperforms the previous schemes. On average, it achieves a 17% speedup over the baseline, it outperforms the other 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. However, for benchmarks where this fraction is very small, such

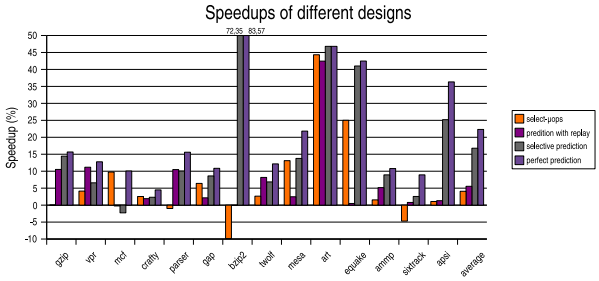

Figure 9: Performance comparison of selective predicate prediction with previous schemes. *False predicated conditional moves* has been taken as a baseline.

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, 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 textitbzip2 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.

5. IMPLEMENTATION REMARKS

This section considers several cost and complexity issues to compare the evaluated techniques.

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 RAT 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.

The selective replay mechanism needs to track several data flows to avoid re-fetch of wrong predicted instructions. Each issue queue entry is extended with one extra input tag for every source register, and another for the destination register. This 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.

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, the number of comparators needed is 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.

6. SUMMARY AND CONCLUSIONS

In this paper we investigate the execution of predicated code in out-of-order processors. There are two problems associated to predication 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. Previous approaches address only the first problem, without taking into account the second one. In fact, these techniques increase resource pressure, which reduces the potential benefits of predication.

We propose a new microarchitectural technique based on predicate prediction to avoid multiple register definitions, and minimize the consumption of unnecessary resources. Since predicting the predicates undoes if-conversion transformations, it may loose its potential benefits if not applied carefully. Therefore, our technique tries 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 predicated 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 results show that the selective predicate prediction scheme outperforms other previous schemes by more than 11% on average, and it performs within 5% of an ideal scheme with perfect predicate prediction. Moreover, the proposed technique adds less hardware complexity to the rename and issue logic than previous schemes.

7. ACKNOWLEDGEMENTS

This work is supported by the Spanish Ministry of Science and Technology and FEDER funds of the EU under contracts TIN 2004-03072, and TIN 2004-07739-C02-01, and Intel Corporation.

8. REFERENCES

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

[2] 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.

[3] D. I. August, W. mei W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 92–103. IEEE Computer Society Press, 1997.

[4] 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.

[5] 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, New York, NY, USA, 2003. ACM Press.

[6] 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.

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

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

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

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

[11] 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 IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, Washington, DC, USA, 2005. IEEE Computer Society.

[12] 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.

[13] 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 IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Washington, DC, USA, 2004. IEEE Computer Society.

[14] 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 ACM/IEEE international symposium on Microarchitecture*, pages 271 – 282, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[15] 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, Washington, DC, USA, 2001. IEEE Computer Society.