Improving Branch Prediction and Predicated Execution in Out-of-Order Processors

Eduardo Quiñones
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
equinone@ac.upc.edu

Joan-Manuel Parcerisa
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
jmanel@ac.upc.edu

Antonio González
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Intel Barcelona Research Center
Intel Labs
antonio.gonzalez@intel.com

Abstract

If-conversion is a compiler technique that reduces the misprediction penalties caused by hard-to-predict branches, transforming control dependencies into data dependencies. Although it is globally beneficial, it has a negative side-effect because the removal of branches eliminates useful correlation information necessary for conventional branch predictors. The remaining branches may become harder to predict. However, in predicated ISAs with a compare-branch model, the correlation information not only resides in branches, but also in compare instructions that compute their guarding predicates. When a branch is removed, its correlation information is still available in its compare instruction.

We propose a branch prediction scheme based on predicate prediction. It has three advantages: First, since the prediction is not done on a branch basis but on a predicate define basis, branch removal after if-conversion does not lose any correlation information, so accuracy is not degraded. Second, the mechanism we propose permits using the computed value of the branch predicate when available, instead of the predicted value, thus effectively achieving 100% accuracy on such early-resolved branches. Third, as shown in previous work, the selective predicate prediction is a very effective technique to implement if-conversion on out-of-order processors, since it avoids the problem of multiple register definitions and reduces the unnecessary resource consumption of nullified instructions. Hence, our approach enables a very efficient implementation of if-conversion for an out-of-order processor, with almost no additional hardware cost, because the same hardware is used to predict the predicates of if-converted code and to predict branches without accuracy degradation.

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 result in severe performance penalties that tend to grow with larger window sizes and deeper pipelines.

If-conversion [2] is a compiler technique that helps to eliminate hard-to-predict branches, by converting a control dependence into a data dependence and potentially improving performance. Hence, if-conversion may alleviate the severe performance penalties caused by hard-to-predict branch mispredictions, by collapsing multiple control flow paths and scheduling them based only on data 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 whether the instruction is executed or converted into a no-operation. Although our study focuses on the effects of if-conversion, predication has a number of other possible uses.

Many studies have shown the benefits of if-conversion [4] [13]. However, the removal of some

1-4244-0805-9/07/$25.00 ©2007 IEEE

branches by if-conversion may adversely affect the predictability of other remaining branches [3], because it may reduce the amount of correlation information available on branch predictors. As a consequence, the remaining branches may become harder to predict, since they may have little or no correlation among themselves.

In ISAs that implement a compare-and-branch model such as the one considered in this paper [7], the branch outcome depends on the value of its guarding predicate, that is produced by a previous compare instruction. On such a model there are two identified opportunities to alleviate the above mentioned accuracy loss. First, although a branch is removed by if-conversion, its correlation information is still present within the predicates that guard the if-converted code and are produced by previous compare instructions. Some studies have proposed strategies to incorporate part of such predicate information to improve the accuracy of branch predictors [17] [3]. Second, the branch prediction may become unnecessary if the compare instruction is scheduled enough in advance so the predicate is already computed when the branch is fetched. Such *early-resolved branches* may be exploited to increase branch prediction accuracy [3] [17].

In this paper, we propose a new branch prediction scheme that is not negatively affected by if-conversion. Our approach predicts the guarding predicates of conditional branches at the time they are produced by compare instructions, so this predictor uses the compare PC instead of the branch PC. In other words, we propose to replace the branch predictor by a predicate predictor. Our predictor is able to keep all correlation information among branches, even for those removed by if-conversion, since such information is primarily associated with compare instructions. Unlike previous proposals, our scheme is able to fully correlate branch global history. In addition, it takes full advantage of the knowledge of early-resolved branches to further improve branch accuracy, since it is able to use the computed predicate value when it is available, instead of the prediction.

Moreover, our proposal has another important advantage. Some studies have shown that predicated execution provides an opportunity to significantly improve hard-to-predict branch handling for out-of-order processors [4] [13]. However, predicate execution in out-of-order processors has to deal with two problems: *1)* multiple register definitions at the rename stage, *2)* the consumption of unnecessary resources by predicated instructions whose guard is evaluated to false. Predicting predicates is an effective technique that addresses both problems [5] [16]. Instructions with a predicate predicted to false are speculatively cancelled at the rename stage and removed from the pipeline, thus avoiding multiple register definitions and avoiding also the resource pressure caused by cancelled instructions. However, the main drawback of this approach is the huge hardware cost of the predicate predictor. Since our proposal uses the same predictor to predict branches and the predicates of if-converted code, it gets all the reported performance benefits of the selective predicate predictor [16] with minimal extra hardware cost.

In summary, our proposal has three main advantages: First, it improves branch prediction accuracy on if-converted codes by integrating all branch correlation information into the prediction of branches. Second, it exploits early resolved branches to further improve branch prediction. Third, it is an effective technique to enable if-conversion on an out-of-order processor, as well as a low-cost solution that does not require extra hardware, because the same is used for branch prediction and for predicate prediction.

The rest of this paper is organized as follows. Section 2 discusses the state of the art on branch prediction incorporating predicate information. Section 3 describes our proposed technique. Section 4 presents the experimental results obtained. Finally, the conclusions are presented in section 5.

2 Related Work

Predication was proposed by Allen et al. [2]. In this section we will shortly review studies that focus on the impact of predication on branch predictability.

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. [13], studied the benefits of partial and full predication code in an out-of-order execution model to achieve speedups in large control-intensive programs. They showed that, in comparison to a processor without predication support, partial predication improves performance by 33%, whereas full predication improves performance by 63%.

Mahlke et.al. [14] proposed several compiler synthesized techniques with hardware support to improve dynamic branch prediction. They proposed new compile techniques that define a prediction function for each branch by using profile feedback. The result of this function, that is computed by extra instructions per branch inserted into the compiled code, is kept in a predicate register.

August et.al [3] exposed that the removal of some branches by if-conversion may adversely affect the predictability of other remaining branches, since it reduces the amount of available branch correlation information. Moreover, in some cases if-conversion may merge the charac-

teristics of many branches into a single branch, making it harder to predict. They proposed the *Predicate Enhanced Prediction* (PEP-PA), that improves a local history based branch predictor by correlating with the previous definition of the branch guarding predicate. Depending on the pipeline depth and the scheduling advance of the predicate define, the predicate register value may not be available at the time the branch is fetched. Instead, the predicate register file contains the previous computed definition of that register. Assuming that its previous definition may be correlated with the current branch, whose predicate definition is not yet computed, the PEP-PA predictor uses this prior value to choose between one of two different local histories, both for using and for updating it. For branches whose predicate is available, the pattern history table (PHT) counters quickly saturate, and then prediction becomes equal to the computed predicate. Our approach aims at exploiting similar opportunities, but is able to use a wider range of correlated information from previous branches and predicates, not only a single value.

Simon et.al [17] incorporate predicate information into branch predictors to aid the prediction of region-based branches. The first presented optimization, called *Squash False Path*, stores the branch guarding predicate register number into its branch predictor entry, so future instances can be early-resolved if the predicated value has been computed. The second presented optimization, called *Predicate Global Update Branch Predictor*, incorporates predicate information into the global history register (GHR) to improve the performance of region branches that benefit from correlation. Since the GHR is updated twice for every branch condition, once at the predicate define writeback, and another at the branch fetch, it stores redundant information. However, the main problem is that these updates are done at different places in the pipeline, so their scheme must include a complex *Deterministic Predicate Update Table* mechanism to guarantee that the GHR stores the conditions in program order. To overcome the existing delay between the branch prediction and the updating of the GHR by predicate computations, a new scheduling technique is also proposed. Their study was developed and evaluated for an in-order EPIC processor [6]. In contrast, our approach updates the GHR only once, for every compare instruction fetched, so it does not store redundant bits neither it requires a complex ordering mechanism.

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. Since if-converted branches are not removed but they are transformed into wish branches, this technique does not suffer from the loss of correlation information, but it can not exploit early-resolved branches.

3 The Predicate Prediction Scheme

This section describes our predicate predictor scheme for an out-of-order processor. This scheme assumes an ISA with full predicate support, such as IA64 [8].

Many studies have shown that if-conversion transformations help to eliminate hard-to-predict branches [4] [13]. However, it may also have some negative effects in the predictability of the remaining branches [3]. First, the removal of branches may reduce the amount of correlation information in the predictor. This reduction may degrade prediction accuracy, since conventional branch predictors base their prediction on different levels of branch history to establish correlations between them. Second, if-conversion creates code regions where all instructions are guarded with a predicate. This includes unconditional branches that are thereby transformed to conditional branches, and need to be predicted during fetch. Moreover, these *region-branches* are fetched more frequently than in their original form.

Figure 1 illustrates the above problems with an example. In Figure 1a, the instruction *br.ret* executes only if condition *cond1* evaluates to false and *cond2* evaluates to true. In Figure 1b the same code has been if-converted, so the unconditional branch becomes conditional and the two previous conditional branches are removed, so their correlation information is not yet available to a conventional branch predictor.

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

Of course, the use of full correlation information does not guarantee the easy predictability of a branch. The poor predictability of the removed branches may *migrate* to the remaining branches, thus making it useless the recovery of the lost correlation information. In Figure 1 the poor predictability of conditions *cond1* and *cond2* may *migrate* to branch *br.ret*.

In the following subsection it is described in detail the mechanism proposed to avoid losing any correlation information of if-converted branches, as well as to exploit early-resolved branches, both resulting in branch prediction accuracy improvements. In the subsection after that, it is shown how the same scheme, with only minor hardware extensions, is used to enable executing efficiently predicated code on an out-of-order processor.

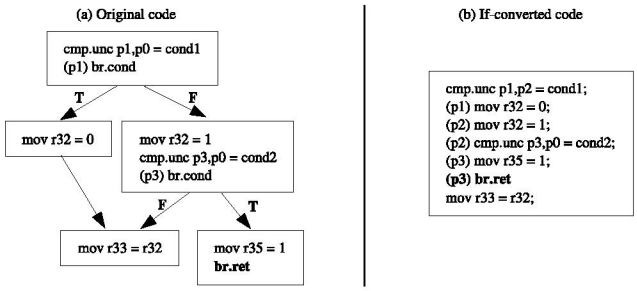

Figure 1. (a) Original code with multiple control flow paths. (b) Multiple control flow paths have been collapsed in a single path. The unconditional branch *br.ret* has been transformed to a conditional branch and it now needs to be predicted. It is correlated with conditions *cond1* and *cond2*

3.1 A Predicate Predictor for Conditional Branches

Previous presented studies incorporate predicate information into branch predictors in several ways [17] [3]. August et.al [3] proposed to improve a local history branch predictor by correlating with the previous definition of the branch guarding predicate. However, correlation information is not fully recovered, since only the last predicate value definition is used to select and update one of two local histories. Simon et.al [17] proposed to introduce recent computed predicates into the GHR. Although a higher amount of correlation information is recovered, the effectiveness of the predictor may be reduced due to storing duplicate information, and it requires a complex mechanism to keep the program order of the GHR.

Here, we propose to replace the conventional branch predictor by a predicate predictor. In a compare-to-branch model, the value of the branch guarding predicate determines the direction of the branch. In such model, conventional branch predictors use the branch *PC* to predict the value of its guarding predicate, and the result feeds the history registers. In contrast, in our predicate predictor scheme, branches do not take part at all in the generation of predictions. The predicate predictor uses the *PC* of the compare instruction that produces the branch guarding predicate. So, instead of predicting the branch *input*, we actually predict the compare *output*.

The predicate prediction is initiated at the fetch of the compare instruction and stored for latter use by a consumer branch. As mentioned before, our scheme assumes an out-of-order processor, so all registers, including predicates, are renamed to physical locations in the Predicate Physical Register File (PPRF). In our scheme, each produced predicate prediction is stored in the predicate physical register allocated to it at rename. Later on, the consumer conditional branch will get its predicate prediction from that physical register, but it must be renamed first to find the corresponding location. That is, the physical register name is the unique identifier that binds a predicate producer with it(s) consumer(s).

Since the prediction starts at the fetch stage with the compare *PC*, and is not stored until the destination predicate is renamed, a multicycle prediction can be performed, i.e., it may be designed as a pipelined large and highly accurate predictor. In addition, since the predictions are not accessed by branches until they reach the rename stage, our predicate predictor becomes the perfect candidate to be implemented in a two-level branch prediction scheme such as the one in the Alpha [10] or the PowerPC [19] processors. Such schemes have two different branch predictors that make two predictions for each branch: the first, fast though less accurate predictor, takes a single cycle and allows the processor to continuously fetch instructions without stalling; the second, slower but highly accurate, takes several cycles and overrides the first prediction. If the two predictions are different, the front-end is flushed and the fetch redirected according to the second prediction.

In our scheme, since the predicted and the computed values of the predicate are written to the same physical register, should the compare instruction be scheduled enough in advance of the branch (an early-resolved branch), the branch will use the computed value as a prediction, thus effectively being 100% accurate.

Figure 2 illustrates the prediction mechanism for producers. Predicate predictions are generated for instructions that produce predicates, such as compare instructions. The predictions are generated in early stages of the pipeline, starting with the *PC* of the compare instruction. Since these instructions have two predicate outputs in the IA64 ISA, the predictor generates two predictions for every compare instruction. When the compare instruction is renamed, the two predictions are speculatively written to the PPRF. Later on, when the compare instruction executes, the PPRF is updated with the two computed values. In the example above, *p1* and *p2* are renamed to *pph1* and *pph2* respectively. When a conditional branch reaches the rename stage, it renames its guarding predicate and obtains its input value from the PPRF. The obtained value overrides the first branch prediction performed at fetch stage. In the example above, the branch obtains its prediction from the predicate physical register *pph1*.

The IA64 ISA defines a rich set of instructions to produce predicate values. For some compare types, the two predicate results depend on the outcome of the condition. However, for some other types, these results also depend on state information that is not available in the front-end [8].

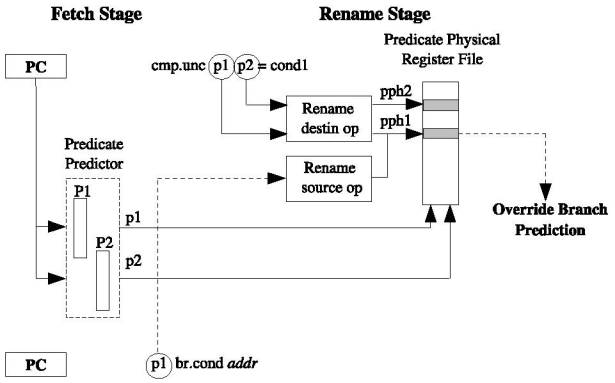

Figure 2. Operation of the Predicate Predictor to predict branches: two predictions are generated by a compare instruction and one is consumed by a branch.

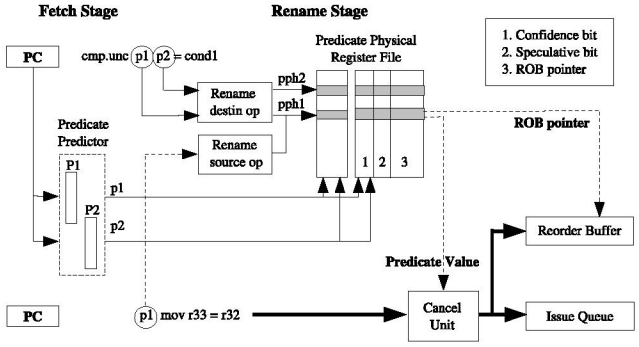

Figure 3. Operation of the Selective Predicate Predictor on if-converted code: two predictions are generated by a compare instruction and one is consumed by a predicated instruction.

Thus, it is not possible to infer the two predicate values based only on the condition and the comparison type, so two independent predictions must be generated. The predictor implementation is described in detail in section 3.3.

3.2 A Predicate Predictor for If-converted Instructions

Predicated execution on out-of-order processors originates two problems. First, if-conversion collapses multiple control paths and may produce multiple definitions of the same register. If these assignments are guarded with different predicates, each register must be renamed to a different physical register. Since predicates are resolved at the execution stage of the pipeline, it may occur that the name of the register is still ambiguous when renaming the source of an instruction that uses it. Second, instructions whose predicates are evaluated to false are cancelled. Since this is usually done late in the pipeline, these instructions needlessly consume resources such as physical registers, issue queue entries, etc.

Chuang et al. [5] proposed to predict predicates and a selective replay mechanism to recover the machine from predicate mispredictions without flushing the pipeline. Although this scheme avoids the multiple definitions problem, instructions whose predicates are predicted to false are still kept in the issue queue needlessly consuming resources. More recently [16], we proposed a more effective predicate prediction scheme that addresses both problems: instructions with a predicate predicted to false are speculatively cancelled at the rename stage and removed from the pipeline, thus avoiding multiple register definitions and avoiding also the resource pressure caused by cancelled instructions. Therefore, the predicate prediction scheme proposed here appears to fit perfectly in that context: predicate predictions are generated when renaming compare instructions, and can be consumed indistinctly either by conditional branches or by predicated instructions generated after if-conversion.

In the latter case, after predicting the predicates, the processor speculates on one of the control paths, so it actually reverses the if-conversion transformation. Since if-conversion appears to be more effective than branch prediction for hard-to-predict branches [4], blindly applying prediction to all predicates misses the opportunities brought by if-conversion. The use of a confidence predictor to select which predicates are worthy to be predicted and which are not, permits not to lose if-conversion benefits [16]. This confidence predictor requires adding some extra hardware to our predicate predictor, which is described below.

Figure 3 shows the extra hardware needed to implement a selective predicate predictor for if-converted instructions. Each entry of the PPRF is extended with three fields: *confidence* and *speculative* bits and a *ROB pointer*. The *confidence* bit is used to determine if the prediction has enough confidence to be used. The *speculative* bit is set to true when a prediction is generated, and set to false once the predicate value has been computed. When the *speculative* bit is set to true, the *ROB pointer* field points to the first speculative instruction that has used the prediction. Thereby, if the prediction fails, the pointed instruction and all younger instructions are flushed from the pipeline. In order to implement the confidence predictor, each predicate predictor entry is extended with a saturated counter, that is incremented with every correct prediction and zeroed if a misprediction occurs. The prediction is considered confident if its associated counter is saturated.

3.3 The Predicate Predictor

This section gives some justification to the choice of a perceptron predictor, and describes how it is adapted to predict predicates instead of branches.

The Perceptron branch predictor, which is based on neural methods, obtains a very high accuracy for dynamic branch predictions [9]. However, the slow computation time of the prediction function may suppose an important drawback to use perceptrons as a single cycle branch predictor. As explained before, our scheme supports multicycle predicate predictions, so it makes the perceptron a good candidate.

The original perceptron predictor [9] has been slightly modified to predict predicates more efficiently. As explained before, since compare instructions produce two results, the predicate predictor needs to perform two predictions for each compare instruction. The obvious solution might be to split the perceptron vector table (PVT) to perform the two predictions. However, not all compare instructions produce two useful predicates. In fact, one of the destination predicate registers is often the read-only predicate register *p0*. In this case, only the non-zero predicate register is updated and only one prediction is needed. Having a split PVT table may result in a suboptimal utilization of the available space, producing an increase of aliasing conflicts. Instead, we use an unique PVT table that is accessed with two different hash functions, one for each predicate, so the prediction vectors are better given out.

The accuracy of a predicate predictor is also affected negatively by global history corruption. On a conventional branch predictor, processor state recovery is done by the same instruction that speculatively updates it [18]. Instead, on a predicate prediction based scheme, the global history is speculatively updated by a compare instruction while processor state recovery is done by its predicate consumer. In this case, a pipeline flush is triggered starting from the predicate consumer instruction. Although the correct global history bit may be corrected during the corresponding recovery actions, compare instructions that may come after the predicate producer and before the predicate consumer have already made their predicate predictions based on a corrupted global history.

Figure 4 shows a high level scheme of the predicate perceptron predictor. The PVT is indexed twice using two different hash functions, *f1* and *f2*. The first hash function, that is used when one prediction is needed, indexes the whole PVT using the corresponding *PC* bits. The second hash function, that is used when two predictions are needed, simply inverts the most significant bit of the first hash function.

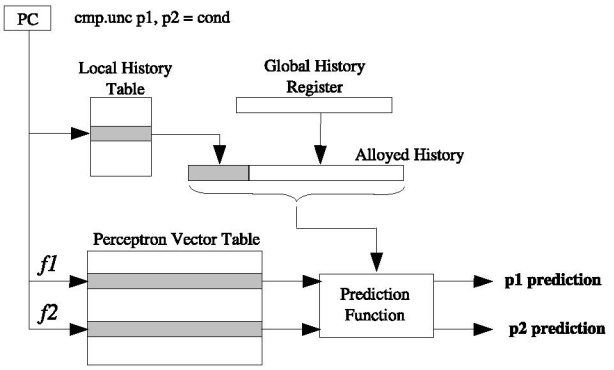

Figure 4. Perceptron Predicate Predictor block diagram.

4 Evaluation

This section evaluates the effectiveness of our Predicate Predictor based branch prediction scheme in terms of accuracy.

4.1 Experimental Setup

All the experiments presented in this paper use a cycle-accurate, execution-driven simulator that runs IA64 ISA binaries. It has been built from scratch using the Liberty Simulation Environment (LSE) [20]. LSE is a simulator construction system, based on module definitions and module communications, that also provides a complete IA64 functional emulator that maintains the correct machine state.

We have simulated twenty-two benchmark programs from Spec2000 [1] (eleven integer and eleven floating-point) using the MinneSpec [12] input set. We have generated two set of binaries. The first set is compiled without enabling predication techniques (if-conversion and software pipelining), and the second set is compiled with only if-conversion transformations enabled. In both cases, all benchmarks have been compiled with IA64 Intel's compiler (Electron v.8.1) using maximum optimization levels and profile information. For all benchmarks, 100 million committed instructions are simulated. To obtain representative portions of code to simulate, we have used the Pinpoint tool [15].

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 taken into account. Load-store queues, as well as the data and control speculation mechanisms defined in IA64, are also modeled

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

Table 1. Main architectural parameters used.

and integrated in the memory disambiguation subsystem. The main architectural parameters are shown in Table 1.

The simulator also models in detail a 144 KB sized PEP-PA branch predictor with 14-bit local history, as described in [21]. This predictor was proposed for an in-order processor and it correlates consecutive predicate definitions with the same logical register name. Since we assume an out-of-order processor, in order to correctly model this predictor, the simulator maintains the state of a logical predicate register file. We assume that the local histories are updated speculatively and correctly recovered on a branch misprediction.

4.2 Branch Prediction Accuracy on Non-If-Converted Code

This section analyses the impact on branch prediction accuracy of three features that differ between our scheme and a conventional branch predictor. On the positive side, our scheme eliminates some branch mispredictions by exploiting early-resolved branches. On the negative side, predicate prediction introduces two factors that affect negatively to prediction accuracy, as discussed in section 3.3. First, it may introduce additional alias conflicts in the prediction tables, because some compare instructions produce two predicates. Second, compare instructions that come after a wrong predicate prediction but before the first use of that predicate make predicate predictions based on a corrupted global-history.

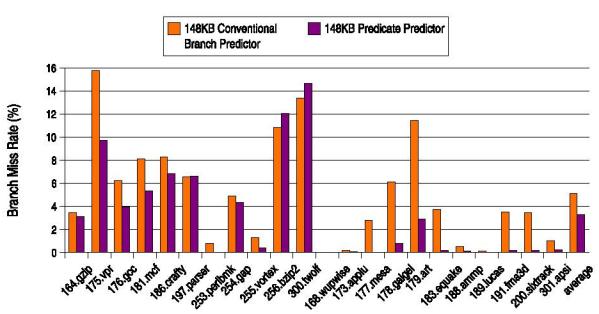

Figure 5. Branch misprediction rates of a conventional branch predictor and our predicate predictor scheme, for non if-converted code.

In order to isolate these effects from those produced by the correlation improvement of our scheme, this experiment uses the binaries compiled without if-conversion. Figure 5 compares the branch misprediction rate when using a conventional branch predictor and our predicate predictor. Both predictors have the same size and latency and analogous configurations. With only three exceptions, the results show that the predicate predictor scheme achieves better accuracy than the conventional branch predictor. On average, it obtains an accuracy increase of 1.86%. This is a significant improvement, since we are using an already highly accurate predictor as a baseline.

The results show that the positive effect of early-resolved branches dominates over the negative effect of increased alias conflicts and global-history corruption, except for three benchmarks, where the net effect is the opposite. In order to evaluate the individual effect of early-resolved branches, isolated from the other two negative effects, we have also simulated idealized branch predictor and predicate predictor schemes, without alias conflicts and with perfect global-history update (results not shown in the graph). We have found that the predicate predictor scheme consistently achieves better accuracy for all benchmarks, and on average it increases branch accuracy by 2.24%. Overall, we conclude that the accuracy improvement contributed by early-resolved branches offsets the small negative effects (less than 0.40% on average) of predicate prediction for most benchmarks.

4.3 Branch Prediction Accuracy on If-Converted Code

This section evaluates the branch prediction accuracy of our scheme, compared to a conventional branch predictor. Since part of the improvement is due to the ability to fully correlate branch global history, it is also compared to the PEP-PA branch predictor [3], which addresses a similar goal by incorporating some predicate information. Next, this section also analyses quantitatively the individual contributions to accuracy due to early-resolved branches and correlation improvement. Obviously, this experiment uses the binaries compiled with if-conversion enabled. Hence, of course, its results can not be directly compared to those in the previous section.

Figure 6a shows branch misprediction rates for three different branch prediction schemes. The first one is a 144 KB PEP-PA branch predictor. The second and the third schemes are a conventional branch predictor and our proposed predicate predictor respectively, both having a 148 KB size and analogous configurations. With only one exception (*twolf*), the results show that our predicate predictor scheme consistently has the lowest misprediction rate. On average, it obtains an accuracy increase of 1.5% with respect to the best scheme. Surprisingly, the PEP-PA scheme performs worse than the conventional predictor, but it may be produced by the out-of-order writing of the predicate registers, which causes it to choose the local history with a wrong predicate. Note that this scheme was conceived to work on an in-order processor.

Figure 6b breaks down the individual contributions of *early resolved branches* and *correlation improvement* to the accuracy difference observed between our scheme and the conventional branch predictor. In order to quantify the contribution of early-resolved branches, we have counted the number of times that the predicate was ready and the conventional branch predictor did a wrong prediction. The remaining accuracy difference is attributed to the correlation improvement. On average, the correlation factor has a higher contribution than early-resolved branches. The accuracy increases are 1% and 0.5% respectively.

However, note that the impact of the correlation improvement is actually underestimated in this graph because this bar includes also the negative effects of the predicate predictor (see section 3.3), which is not measured separately. This explains why this contribution is negative for one benchmark (*twolf*). To evaluate separately the positive effects of our scheme over conventional branch prediction, we have repeated the experiment with idealized schemes assuming no alias conflicts and perfect global-history updates. The results (not depicted in the graph) show a consistent accuracy improvement across all benchmarks and an average improvement of almost 2%. Overall, we conclude

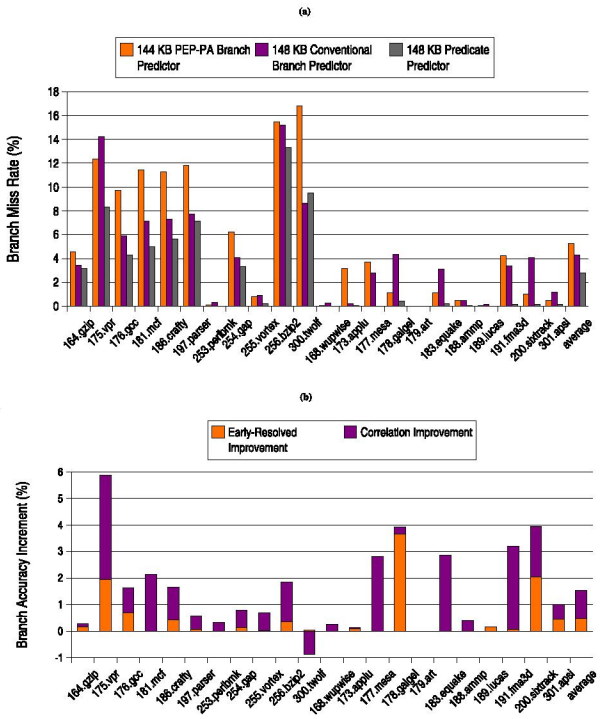

Figure 6. (a) Comparison of branch misprediction rates for if-converted code. (b) Breakdown of the branch prediction accuracy differences between our predicate predictor scheme and a conventional branch predictor.

that the accuracy increases contributed by early-resolved branches and correlation improvement more than offset the small negative effect (less than 0.5% on average) of predicate prediction on all benchmarks but one.

5 Summary and Conclusions

If-conversion is a powerful compilation technique that may help to eliminate hard-to-predict branches. Reducing branch mispredictions is important for modern processors because of their wide and deep pipelines. However, previous works have shown that if-conversion may have also a negative side-effect on branch prediction accuracy. Indeed, the removal of some branches also removes their correlation information from the predictor and may make other remaining branches harder to predict.

There have been some proposals - focused on a predicated ISA with a compare-branch model - to recover the correlation information, either by incorporating the previous value of the branch guarding predicate into the prediction scheme, or by adding all the predicate defines to the

global history. In the first case, only a single predicate is correlated; in the second case the global history is fed with redundant results, since all predicates are inserted twice: once when they are produced by a compare instruction, and then when they are consumed by a branch.

In this paper we have proposed a different approach, which predicts the outcome of branches by predicting their guarding predicates. The predictions are made for every predicate definition, and stored until the predicate is used by some branch. Unlike previous approaches, it is not the branch itself but the compare instruction that is involved in the generation of the prediction. We have shown that this approach has a number of advantages.

First, branch prediction accuracy is not affected negatively by if-conversion because compare instructions keep all the correlation information in the predictor. Unlike the PEP-PA scheme, our approach may use the full global history instead of a single bit, and the mechanism adapts easily to powerful branch predictors that exploit global and local correlation. On average, the branch correlation contributed by the predicate predictor adds at least 1% accuracy over a conventional branch predictor with the same configuration.

Second, branch accuracy is further improved by exploiting early-resolved branches. Each compare instruction stores the predicate predictions in the same physical registers where the corresponding computed values will be later written. Hence, if the compare instruction is scheduled enough in advance, the prediction read by the branch is actually the computed value and is always correct. We have shown that, on average, exploiting such early-resolved branches adds an extra 0.5% to the branch prediction accuracy of our scheme.

Third, our predicate predictor may be extended with minimal hardware cost to implement a selective predicate predictor that enables efficient predicated execution on out-of-order processors. This technique solves the problem of multiple register definitions and the unnecessary resource consumption of instructions with a false predicate, and it has been reported to outperform previous techniques by 11% IPC. Since our proposal uses the same hardware for branch prediction and for predicated execution, this performance improvement is achieved with almost no additional cost.

In summary, we have proposed a scheme that improves branch prediction accuracy of if-converted codes by 1.5% on average for the Spec2000 benchmark suite and enables an efficient implementation of predicate execution on out-of-order processors without adding any significant extra hardware.

6 Acknowledgements

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

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. August, D. Connors, J. Gyllenhaal, and W. M. Hwu. Architectural support for compiler-synthesized dynamic branchprediction strategies: Rationale and initial results. In *HPCA '97: Proceedings of the 3th International Symposium on High-Performance Computer Architecture*, pages 84–93. IEEE Computer Society, 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] L. Gwennap. Intel, hp make epic disclosure. *Microprocessor report*, 11:1 – 9, October 2001.

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

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

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

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

[13] 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.

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

[15] 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.

[16] E. Quiñones, J.-M. Parcerisa, and A. Gonzalez. Selective predicate prediction for out-of-order processors. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, New York, NY, USA, 2006. ACM Press.

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

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

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

[20] 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.

[21] 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.