Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes

Alexandre Farcy [*], Olivier Temam [*]
PR*i*SM, Université de Versailles
Versailles, France
{farcy,temam}@prism.uvsq.fr

Roger Espasa [†], Toni Juan [†]
DAC, Universitat Politècnica de Catalunya
Barcelona, Spain
{roger,antonioj}@ac.upc.es

Abstract

The goal of this study is twofold: to analyze in detail the nature of conditional branch mispredictions in correlation-based branch predictors, and, based on this analysis, to reduce the impact of branch mispredictions on processor performance by decreasing the branch resolution delay instead of improving the branch prediction accuracy.

We classify conditional branches with the highest number of mispredictions according to the nature of their branch condition analytical expression. Based on these expressions, we can analyze and even precisely explain the origin of mispredictions in many cases. Moreover, we find that many such branches belong to small sets of blocks inside loops, and within such sets we find that some of the branch expressions have regularity properties. We show how to exploit this regularity property by anticipating *the branch outcome, where* anticipation *is a combination of* value prediction *and normal dataflow execution.*

We investigate a hardware mechanism to implement the concept of branch outcome anticipation*. This mechanism relies on the separate execution of the normal program flow and a* branch flow*, which is a subset of the program flow corresponding to copies of the instructions needed to compute branch outcomes. The* branch flow *uses the regularity properties of branch condition expressions to get ahead of the normal program flow whenever possible. Currently, the mechanism can only target a subset of the conditional branches, but with these branches we experimentally show that the anticipation mechanism successfully reduces the average branch misprediction latency by 60%.*

[*]This work was supported by the Esprit Agency DG XIII under grant Esprit LTR MHAOTEU No 24942.

[†]This work was supported by the Ministry of Education of Spain under contract 0429/95.

1. Introduction

The impact of conditional branches on processor performance has been almost exclusively addressed with branch prediction. The best performing branch predictors rely on correlation with past behavior of the branch itself or previous branches [2]. However, the concept of correlation-based branch prediction has limits. Lately, branch prediction improvements essentially consisted in reducing conflicts in branch prediction tables [6, 5, 7]. Recently, [4] provided hints at correlation-based branch prediction upper bounds.

In this study, we focus our attention on branches that perform poorly with current correlation-based branch prediction techniques. We still use branch prediction but we add a mechanism to compute the branch condition before the branch is resolved. The smaller this time interval, the fewer useless instructions are fetched and the smaller the impact of a misprediction on processor performance. For that purpose, we attempt to *anticipate* the branch outcome, i.e., to compute it sooner than in the normal program flow. The scheme we propose dynamically duplicates the instructions in the dataflow tree of the conditional branch, we call this second set of instructions the *branch flow*. Both flows execute simultaneously in the processor but we show how to get the *branch flow* ahead of the normal program flow to compute the branch outcome early enough, i.e., to *anticipate* the branch outcome.

The central issue of this scheme is to get the *branch flow* ahead of the normal program flow using *value prediction*. For that purpose we analyze in detail the behavior of conditional branches with the highest number of mispredictions. We analyze the dataflow path of these branches and build the mathematical function corresponding to their condition expression. We then explain why branch prediction fails, and for about 60% of the static branches studied we show that their behavior exhibits some form of regularity. Moreover, we experimentally show that these branches are the less likely to improve as correlation-based prediction mech-

0-8186-8609-X/98 $10.00 (c) 1998 IEEE

anisms scale up. We then show that for about 20% of all branches studied it is possible to exploit this regularity property to *anticipate* the computation of the branch condition. Based on these results, we present and evaluate a hardware implementation of the *anticipation* method.

In Section 2, we analyze branch condition expressions and highlight their regularity properties when applicable. In Section 3 we outline the hardware design used for implementing the early branch resolution scheme, and we evaluate its performances in Section 4.

2. Analyzing Branches with High Misprediction Rates

In this section, we categorize the branches depending on their condition expression. We rewrite branch condition expressions as functions, using instructions in the dataflow tree of the branch condition, then we show why some of them are predictable.

2.1. General Principles

Consider the code in Figure 1. The meaning of each instruction is indicated on the right. Here, we study the `bne r5, label` condition expression inside this piece of code.

```
label:
  ...Other Instructions...
  lda    r1, 4(r1) ; R1 = R1 + 4
  lda    r0, 1(r0) ; R0 = R0 + 1
  cmpult r1, r0, r5; R5 = (R1 < R0)
  bne    r5, label ; if (R5 != 0) goto label

  cmpult r0, r2, r6; R6 = (R0 < R2)
  bne    r6, label ; if (R6 != 0) goto label
label2:
  sll    r0,1,r0   ; R0 = R0 << 1
  ...
  bne    r7, label ; if (R7 != 0) goto label
label3:
  lda    r0,2(r0)  ; R0 = R0 + 2
  ...
  bne    r8, label ; if (R8 != 0) goto label
  ...
```

Figure 1. *Alpha assembly code example.*

Functions Let us denote instructions of Figure 1 as functions, i.e., $r1 = lda(r1, 4)$ stands for instruction `lda r1, 4(r1)`, and other instructions become $r0 = lda(r0, 1)$, $r5 = cmpult(r1, r0)$. Then we can rewrite $r5$ as:

$$r5 = cmpult(lda(r1, 4), lda(r0, 1))$$

Assume now the conditional branch on $r5$ is taken several times in a row. The expression of the condition $r5$ on the i^{th} iteration is:

$$r5 = cmpult(lda^i(r1, 4), lda^i(r0, 1))$$

where $lda^i(x, a) = lda^{i-1}(x, a) + a = x + i \times a$

This expression is the *function* defining the branch condition $r5$ inside this piece of code.

Instruction neighborhood We built the expression of $r5$ using the instructions of a single basic block. Now, we identify all possible control paths to the studied branch and the corresponding basic blocks in these paths but we restrict the control path length to N blocks from the studied branch. We call the corresponding set of blocks the *instruction neighborhood*. The branch condition expression is built using this neighborhood only. We call the operands computed outside the neighborhood the *leaf operands*. In the $r5$ expression, $r0$ and $r1$ are initially computed outside the neighborhood considered; they are the *leaf operands* of this function.

In Figure 2, we vary the control path length, N, to build different instruction neighborhoods around the `bne r5, label` instruction. Each node is a basic block ending with a conditional branch, and for each node, only the instructions in the dataflow tree of the conditional branch studied are shown. The dataflow tree of the branch condition inside the neighborhood is called the *branch flow*. When increasing N, new instructions are added to the dataflow tree of $r5$. Thus, the function corresponding to the expression of $r5$ is also a function of N. We provide the expressions of $r5$ at the end of this section.

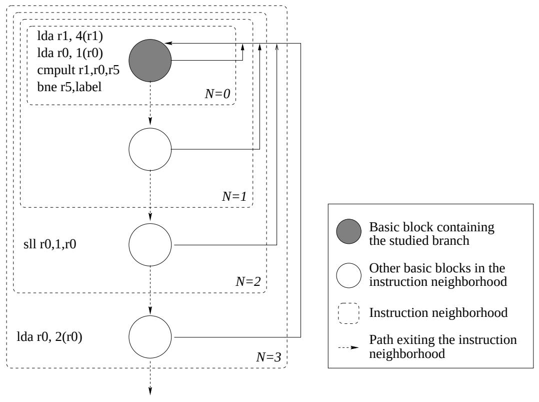

lda r1, 4(r1)
lda r0, 1(r0)
cmpult r1,r0,r5
bne r5,label
N=0
N=1
sll r0,1,r0
N=2
lda r0, 2(r0)
N=3
Basic block containing the studied branch
Other basic blocks in the instruction neighborhood
Instruction neighborhood
Path exiting the instruction neighborhood

Figure 2. *Varying the Instruction Neighborhood Size.*

Local loops If the neighborhood of a branch, or any subset of basic blocks including the branch, is repeated at least once, it determines a *local loop*. One iteration of this *local loop* is defined as one execution of the conditional branch considered, whether it is taken or not taken.

Note that unlike a normal loop, a *local loop* does not end when the conditional branch outcome changes from *taken* to *not taken* or vice-versa. In our example, assume condition $r6$ is almost always true. As the local loop lasts until $r6$ is false, the *local loop* can contain iterations where $r5$ is true and iterations where $r5$ is false.

Regularity A branch condition expression C is said to be regular and predictable once the following conditions are met:

- The conditional branch is inside a *local loop*.
- Let i the number of iterations of the *local loop*, we can write C on iteration i as:
  $C = F(f_1^i(\textit{leaf operands}), f_2^i(\textit{leaf operands}), \ldots)$
  where F is itself a function or a composition of functions and f_j is a function or a composition of functions such that $f_j^i = f_j^{i-1} + \textit{constant}$. We restrict to this type of functions because it is the most frequent case among branches with regularity properties. Also, it is easily predictable with a stride-based value predictor.

In our example, the innermost functions of the expression of $r5$ are *regular* and easily *predictable*:

$$r5 = F(f_1^i(r1, 4), f_2^i(r0, 1))$$

where $F = cmpult$ and the innermost functions are $f_1 = lda$ and $f_2 = lda$.

In [10], the concept of *dependence collapsing* was presented to combine multiple dependent instructions into a single one, and execute it on a special-purpose multiple-operand functional unit. The f_j^i functions were collapsed provided the f_j were a composition of arithmetic operations. Here, we also transform the branch condition expression into a function, but we execute it on conventional units.

Exploiting regularity properties We exploit the regularity property of the branch condition expressions to *skip several iterations* of a *local loop* and get ahead of the normal program flow. For that purpose, we duplicate the instructions belonging to the dataflow tree of the branch condition. We call this set of instructions the *branch flow*. In our example, we would jump ahead in the *local loop* by k_{jump} iterations at iteration i and compute $lda^{i+k_{jump}}$ while the normal program flow computes lda^i.

Here, we would predict, at iteration i, the value of the *leaf operands* at iteration $i + k_{jump}$, then we would compute the function $f^{i+k_{jump}}$ itself. Once we are k_{jump} iterations ahead, we can execute all the instructions without needing any further prediction and compute in advance the branch conditions for the next iterations. We can apply this technique to any branch whose condition expression C is *regular* and *predictable* as defined above.

To some extent, we recognize the *generation* and *propagation nodes* described in [9]. The *leaf operands* of the *local loop* generate the predictability of the f_j functions. Once these operands are computed, the f_j functions can propagate this predictability property from one iteration to the next thanks to their regularity.

Impact of the instruction neighborhood on the branch expression In figure 2 we show the different neighborhoods of the `bne r5, label` instruction. We now build the function corresponding to the expression of $r5$ as a function of the neighborhood size.

When $N = 0$, we have a *local loop* with, as shown before, $r5 = cmpult(lda^i(r1, 4), lda^i(r0, 1))$.

When $N = 1$, no new instruction enters the dataflow path of the branch condition, so the expression of $r5$ is unchanged.

When $N = 2$, the program can loop n times in the $N = 1$ neighborhood, then perform $r0 = sll(r0, 1)$, then loop again in the $N = 1$ neighborhood and so on. The general expression for $N = 2$ is:

$$r5 = cmpult(lda^n(r1, 4)^m, sll(lda^n(r0, 1), 1)^m)$$

Also, when $N = 3$, the program can loop in any smaller neighborhood, then perform $r0 = lda(r0, 2)$, then loop again... The general expression of $r5$ for $N = 3$ is:

$$r5 = cmpult((lda^n(r1, 4)^m)^p, lda(sll(lda^n(r0, 1), 1))^m, 2)^p)$$

The expressions of $r5$ for $N > 1$ are less "regular" (predictable) than for $N = 0$ or $N = 1$. In general, increasing N provides more information on the branch behavior. However, in the present study we only intend to anticipate branches whose internal functions are strided so large values of N would be useless. Even though we systematically used $N = 10$ to build our functions, we found that, in most cases, $N = 5$ is large enough to detect most branch condition expressions with regularity.

2.2. Branch Condition Functions

We study conditional branches responsible for 50% of the dynamic mispredictions of each SpecInt95 code. We simulated a *gshare* [6] predictor with an 18-bit history and an infinite table size in order to select the branches that least benefit from correlation. Here, we analyze the main different types of branch condition expressions of "regular" branches, i.e., branches that belong to a *local loop*. Such branches account for 60% of all studied static branches. For each branch, we build *by hand* its condition expression.

The different f_j functions can be classified into several categories and we highlight which f_j functions are predictable.

```
label:
  ...Other Instructions...
  lda    r0, 1(r0) ; R0 = R0 + 1
  cmpult r0, r1, r5; R5 = (R0 < R1)
  bne    r5, label ; if (R5 != 0) goto label
  ...Other Instructions...

do {
  ...Other Instructions...
  r0 = r0 + 1;
} while (r0 < r1);
```

Figure 3. *Arithmetic Expression.*

Benchmark	Number of Conditional Branches Studied	In *Local Loops*					Not In *Local Loops*
		Predictable Functions				Non	
		Arithmetic	Table	List	Tree	Linear	
`compress`	5	0	0	1	1	1	2
`gcc`	7	1	1	3	0	0	2
`go`	12	0	0	7	3	0	2
`ijpeg`	6	0	2	0	0	1	3
`li`	6	0	1	1	3	0	1
`m88ksim`	7	2	2	0	0	0	3
`perl`	10	2	0	0	0	0	8
`vortex`	0	0	0	0	0	0	0
Total	53	5	6	12	7	2	21
	100%	9.43%	11.32%	22.64%	13.20%	3.77%	39.62%

Table 1. *Nature of Static Branches Studied*

Arithmetic Expressions Consider the example of Figure 3. The Alpha assembly code and an equivalent C code are presented. The instructions not in the dataflow path of the branch condition have been removed. This is the most simple case where the branch condition is a test on an arithmetic expression. Let i be the iteration number, then the condition expression is $f(i) < value$ where $f(i) = a_0 + i$ and a_0 is the first value of a. Such branches are poorly predicted with history-based predictors when the loop does a small number of iterations.

Non-Linear Arithmetic Expressions In *arithmetic expressions*, f_j is linear. However, there are other cases where f_j is arithmetic but not linear. Consider the example of Figure 4, the condition is $C = (r7 <= r1)$ with

$$r1 = ldq(addq(r21, addq(r0, r0)), 4)$$

After i iterations, it becomes:

$$r1 = ldq(addq(r21, addq^i(r0, r0)), 4)$$

Let us write @[A] the value loaded from address A. We can then write $r1$ as: $r1 = @[r21 + r0 \times 2^i + 4]$.

Even though we do not consider such functions in our hardware design, they are clearly predictable.

```
label:
  ...Other Instructions...
  addq  r0, r0, r0  ; R0 = R0 + R0
  addq  r21, r0, r18; R18 = R21 + R0
  ldq   r1, 4(r18)  ; R1 = @[R18 +4]
  cmple r7, r1, r5  ; R5 = (R7 <= R1)
  bne   r5, label   ; if (R1 != 0) goto label
  ...Other Instructions...

do {
  ...Other Instructions...
  a = a + a;
} while (tree[a] < value);
```

Figure 4. *Non-Linear Arithmetic Expression.*

Table Traversals Consider the example of Figure 5. This is a table traversal. The branch misprediction is high because the branch condition depends on table values, even though for some tables, table values themselves are such that the branch is well predicted. The branch condition function is:

$$r5 = cmpult(ldq(lda(r0, 8), 0), r2)$$

Even though the function depends on a load instruction (`ldq`), the anticipation mechanism can apply because the innermost function does not depend on memory accesses and has a linear arithmetic expression. After i iterations, the expression of $r5$ is:

$$r5 = cmpult(ldq(lda^i(r0, 8), 0), r2)$$

which we can write $r5 = (@[r0 + 8 \times i] < r2)$

```
label:
  ...Other Instructions...
  lda    r0, 8(r0) ; R0 = R0 + 8
  ldq    r1, 0(r0) ; R1 = @[R0]
  cmpult r1, r2, r5; R5 = (R1 < R2)
  bne    r5, label ; if (R5 != 0) goto label
  ...Other Instructions...

do {
  ...Other Instructions...
  a = a + 1;
} while (table[a] < value);
```

Figure 5. *Table Traversal.*

List Traversals The fourth type of expressions is list traversals. In Figure 6, the expression of $r5$ is:

$$r5 = cmpult(ldq(ldq(r0, 4), 8), r2)$$

After i iterations, it becomes:

$$r5 = cmpult(ldq(ldq^i(r0, 4), 8), r2)$$

where

$$ldq^i(r0, 4) = @[@[\ldots @[R0 + 4] \ldots + 4] + 4]$$

There is no easy way to speculate on a list traversal with a value predictor because of the nested memory accesses, i.e., ldq^i. The misprediction rate is high if list element data values are irregular or if there are few iterations each time.

We still have to investigate a method to anticipate accesses to linked elements as in [8], where successive elements of linked structures are prefetched, but this is beyond the scope of this paper.

```
label:
  ...Other Instructions...
  ldq    r0, 4(r0) ; R0 = @[R0 + 4]
  ldq    r1, 8(r0) ; R1 = @[R0 + 8]
  cmpult r1, r2, r5; R5 = (R1 < R2)
  bne    r5, label ; if (R5 != 0) goto label
  ...Other Instructions...

do {
  ...Other Instructions...
  a = a->next;
} while (a->data < value);
```

Figure 6. *List Traversal.*

Tree Traversals The fifth type of expressions is even more complex and corresponds to tree traversals (see Figure 7). When the data structure is arranged as a tree and the likelihood of taking one path or another depends only on irregular data values, the misprediction rate can be very high. We found one such case of tree traversal with very high misprediction rate (almost 50%) in `compress`; this code is part of a random number generator routine implementing a dichotomic traversal. In this case, even using the $@^i$ operator, it would not be possible to write the branch condition $r5$ as a function without using conditionals.

```
label:
  ...Other Instructions...
  ldq    r1, 16(r0); R1 = @[R0 + 16]
  cmpult r1, r2, r5; R5 = (R1 < R2)
  bne    r5, label2; if (R5 != 0) goto label2
label1:
  ...Other Instructions...
  ldq    r0, 0(r0) ; R0 = @[R0]
  br     label3                  ; goto label3
label2:
  ...Other Instructions...
  ldq    r0, 8(r0) ; R0 = @[R0 + 8]
label3:
  ...Other Instructions...
  bne    r0, label ; if (R0 != 0) goto label

do {
  v = a->data;
  if (v < value) a = a->left;
  else a = a->right;
} while (a != NULL);
```

Figure 7. *Tree Traversal.*

In Table 1, we indicate for each function category, to what fraction of all static branches studied it corresponds for each SpecInt95 benchmark.

2.3. Branches that Can Benefit From the Anticipation Mechanism

In this study, we focus on branches that belong to *local loops* and whose branch condition expressions are predictable as defined above. Consequently, we focus on *linear arithmetic expressions* and *table traversals* that account for 20% of all static branches studied.

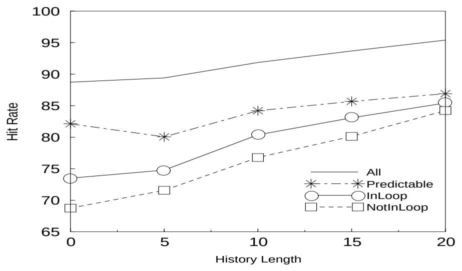


Figure 8. *Prediction Rates Depending on Branch Nature.*

We now want to examine how the different types of branches behave with correlation-based branch predictors. In Figure 8 we present the misprediction rate of a *gshare* branch predictor averaged across the whole execution of the 8 SpecInt95 codes for the studied conditional branches. We separate branches in 3 categories: branches not in *local loops* (`NotInLoop`), branches in *local loops* (`InLoop`) and both *arithmetic expressions* and *table traversals* (`Predictable`), which are subsets of `InLoop` branches (see Table 1). Finally, we also provide the average misprediction rates for all branches (`All`). With this figure we want to see the evolution of branch prediction performance as branch prediction mechanisms scale up to decide which types of branches less benefit from correlation. Because we want to isolate the effect of correlation and to avoid as much as possible the effect of table conflicts, we use a very large table size (20-bit index) and only vary history length.

Even though their initial misprediction rate is higher, we can observe that `NotInLoop` branches improve faster with correlation-based prediction than `All` branches in average and especially faster than `InLoop` and `Predictable` branches. Now, branches are said not to belong to *local loops* if no other instance of the basic block of the branch studied is in a 10-block long history path of this branch. For such `NotInLoop` branches, we have observed that the number of distinct control paths leading to the basic block is much higher than for `InLoop` branches. `NotInLoop` branches belong to procedures called from many locations in the program and increasing the history size of the branch predictor reduces the risk of confusion among paths. Therefore, the causes of mispredictions for branches not in *local loops* might be different than for the branches studied in Section 2.2, and moreover, their performance improve quickly as correlation-based mechanisms scale up, i.e., history length grows. Although `Predictable` branches have a better prediction rate than other studied branches, they prove to be the least likely to improve as correlation-based branch prediction table grows. This latter observation suggests to focus optimizations on these types of branches.

3. Implementing the Anticipation Mechanism

In this section, we propose a hardware mechanism to exploit the branch condition regularity properties observed in Section 2. The principle is to run a *branch flow* that computes the branch conditions ahead of the *normal program flow*. These anticipated outcomes are sent to the corresponding branch instructions of the program flow.

In theory, the *branch flow* could get ahead of the normal program flow simply by executing less instructions, i.e., only the instructions of the *branch flow*. However, we found that the dataflow tree of the branch condition often corresponds to a large share if not all instructions in the instruction neighborhood so this type of gain is often negligible.

The highly-decoupled architecture presented in [1] introduces *control decoupling*: a loop body can be sent to an address or data processor to allow the control processor to execute further in advance. Similarly here, we isolate and duplicate the *branch flow* that will compute the branch condition in advance, then forward the results to the *normal flow*.

We use a stride-based value predictor to predict the value of the regular part of the branch condition expression. The rest of the function can be complex and its result hard to predict. So, instead of predicting the outcome of the whole function, we simply compute the complex part of it. For example, predicting the next *value* to be loaded in a table traversal is hard, as there may not be any regularity within the table values, but the *address* of the next value is highly predictable. In this case, we would predict the address then execute the load.

Once we have computed the stride of the regular part of the condition expression, the *branch flow* can run ahead of the normal flow and compute the branch condition for the next branch instances. Then, thanks to value prediction, the *branch flow* executes instances of instructions that will be fetched and decoded later on in the normal program flow.

In this section we describe the hardware proposed to evaluate this anticipation concept. Figure 9 provides an overview of the main building blocks of the hardware design.

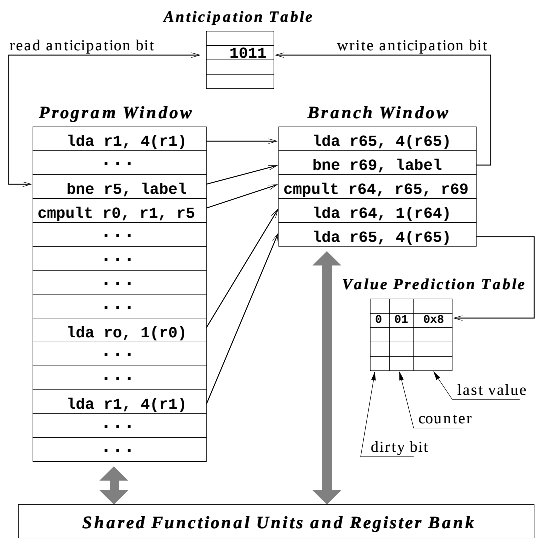


Figure 9. *Hardware Implementation (overview).*

3.1. Instruction Tagging

To extract the *branch flow,* we tag instructions on the dataflow tree of the branch condition. Tagging can be done either statically or dynamically.

Static tagging consists in tagging the whole branch neighborhood using profiling information. This is the approach used in this paper.

Another approach is to tag the instructions at run-time. Initially, we only need to tag target conditional branch instructions. We can use a misprediction counter in the branch condition table to determine target branches. Then, each time a tagged instruction is decoded, the instructions producing its register operands, i.e., the parent instructions in its dataflow tree, are tagged as well, and so on. Progressively, all instructions in the dataflow tree can be tagged.

Consider the example of Figure 1. When `bne` reads $r5$ from `cmpult`, it tags `cmpult`. The next time `cmpult` is decoded and reads its operands, it in turn tags the instructions producing its operands (here, both `lda`s) and so on.

We now want to jump ahead in the *branch flow* to compute the branch condition earlier than in the normal program flow. Let k_{jump} be the size of the jump. We need to predict at iteration i the value of the operands of the function at iteration $i + k_{jump}$ (see Section 2.1).

To limit the size of the instruction neighborhood, we add a distance counter to the tag. Each time an instruction reads an operand that is n branches away from the instruction in the program window, it adds n to the counter and passes the counter with the tag. We stop propagating the tagging when the counter reaches a threshold value.

The *leaf operands* are the registers whose producers are not tagged because they are outside the instruction neighborhood. Instructions having such source registers are marked with a special tag in order to tell the anticipation mechanism that they will need to be predicted.

We ignore load/store dataflow dependences but we found they occur only in a few irregular functions within *local loops*.

In this study, we have only tagged branches with most mispredictions, but tagging can be extended to all conditional branches if done dynamically. Regardless of the tagging technique used, the result is that instructions flowing into the decode stage carry a tag information that indicates whether or not they belong to the dataflow tree of a branch.

3.2. Duplicating Branch Condition Flow

We use a separate instruction window called the *branch window*, in opposition to the normal *program window* to process copies of tagged instructions. The duplication is done during the decode stage. Both flows remain tightly synchronized as the *branch window* is only fed with a copy of each tagged instruction from the program flow.

The instruction windows are separate, but both instruction flows share the same physical register bank and functional units. In our implementation, the *branch window* instructions have priority over normal instructions to access the functional units.

When instructions are duplicated, we also assume their register operands are distinct from those of the original instruction. Recall we need to predict the *leaf registers* of the *branch flow* to allow anticipation. However, duplicated instructions need to use the same operands as original instructions on the first iteration of the *local loop* (*leaf operands*) to initiate the anticipation process.

Each time the program leaves the *local loop*, registers can get *corrupted* by other non-tagged instructions. Therefore, we maintain one information bit (*dirty bit*) with each register to indicate whether it was last written by a *branch flow* (tagged) instruction or a normal program flow instruction. In the first case we are inside the *local loop*, in the other case we assume the program has left the *local loop*.

In our simulations, we use 64 logical registers for the program flow, and 64 logical registers for duplicated instructions operands. However, both flows share the same physical registers. For example, consider the tagged instruction `lda r3,8(r0)` is duplicated and read its operand. If the *dirty bit* for $r0$ is set then $r0$ is returned. If it is not set, then $r64$ is returned. The result of the duplicated instruction will be stored in $r67$ anyway, and its *dirty bit* will be reset.

3.3. Reading/Writing Anticipation Bits

Tagged branch instructions do not behave like branches once they are duplicated. Duplicated conditional branches do not affect the PC, they only send their outcome bit to a table called the *anticipation table*, indexed with the branch PC.

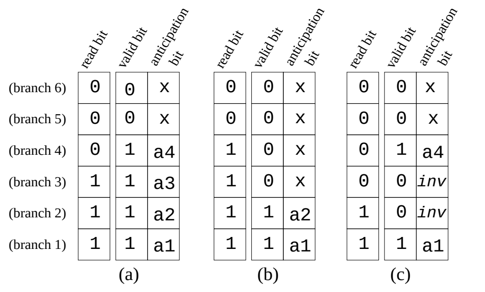
Each entry of the anticipation table contains three fields. The *anticipation bit* indicates the outcome of the branch. It is written by the duplicated branch of the *branch flow*.

When the duplicated branch writes its *anticipation bit*, it also sets a *valid bit* to indicate that the outcome has been produced and can be consumed. If the *valid bit* is not set, then the *anticipation bit* has to be ignored by the corresponding branch of the normal program flow.

The *read bit* is set when the *anticipation bit* of the entry is read by a branch of the *program flow*, even if the *valid bit* is not set. It indicates that the consumer of this *anticipation bit* is in the pipeline, i.e., it already read the table. Each tagged branch reads the table once, at fetch time. A duplicated branch that writes its outcome in an entry which has been already read (the *read bit* of the entry is set) has to forward its *anticipation bit* to the corresponding branch.

When a tagged branch is fetched from the instruction cache, it speculates its outcome using the branch predictor as usual, and it simultaneously accesses the anticipation table to read the anticipation bit. The branch reads the *anticipation bit* of the oldest unread entry of the table (an entry is unread if its *read bit* is not set). If the *valid bit* is not set, the branch simply uses the prediction bit, otherwise it uses the anticipation bit and ignores the prediction bit.

Therefore, there are two ways to improve processor performance with the anticipation mechanism: by providing an anticipation bit instead of a speculation bit to newly fetched tagged conditional branches, or by resolving such branches earlier than in the normal pipeline stage: if no anticipation bit is available for the branch at fetch time, it can still be sent as soon as it is computed, for an *early resolution*.



	(a) read bit	(a) valid bit	(a) anticipation bit	(b) read bit	(b) valid bit	(b) anticipation bit	(c) read bit	(c) valid bit	(c) anticipation bit
(branch 6)	0	0	x	0	0	x	0	0	x
(branch 5)	0	0	x	0	0	x	0	0	x
(branch 4)	0	1	a4	1	0	x	0	1	a4
(branch 3)	1	1	a3	1	0	x	0	0	*inv*
(branch 2)	1	1	a2	1	1	a2	1	0	*inv*
(branch 1)	1	1	a1	1	1	a1	1	1	a1

Figure 10. *Anticipation Table.*

In Figure 10(a), the anticipation flow is ahead of the normal program flow as we see that 4 *anticipation bits* are available while only 3 branches have read the table yet. The next branch fetched will read anticipation bit $a4$ and set the corresponding *read bit*. In Figure 10(b), the anticipation process is late: $branch3$ and $branch4$ have read the table while their *anticipation bits* were not available. The next anticipation bit will still be stored in the table and sent to branch 3 for its *early resolution*. If branch 3 is already resolved, then all entries up to this one are pulled out of the table.

In case of branch misprediction, the program flow is restarted and the *read bits* are cleared. But the anticipation bits are assumed to be correct until a *leaf operand* gets corrupted. In this case, the whole anticipation table is flushed. So we potentially need to have as many entries as there are branches in the program window (in our experiments, we use 32 entries in our table).

The *anticipation bit* can be incorrect for two reasons: PC conflicts in the anticipation table and wrong value predictions. Consequently, a branch in the normal program flow can induce two squashes: if a branch has no anticipation bit ready when it is fetched, it is speculated. Then, if it gets the anticipation bit before it is resolved, and the anticipation differs from the speculation, the branch induces a first squash. If the anticipation later proves to be false at the resolution, it induces a second squash. In this last case, we also have to restart the anticipation process by flushing the *anticipation table*.

3.4. Selective Triggering of Value Prediction

To effectively get ahead of the normal program flow, we need to jump forward by a few iterations. For that purpose, we use a stride-based value predictor [9] for *leaf operands*. As mentioned in Section 3.2, we can detect when the program flow enters a *local loop* by checking register corruption. Also, we maintain an iteration counter for each *leaf operand* register. Each time this register is written again during the *local loop*, the counter is incremented, and it is reset when the program enters the *local loop*. After $k_{threshold}$ iterations, the value of the register is predicted k_{jump} iterations ahead using a stride-based value predictor and the register value is replaced with the predicted value. A single iteration is enough ($k_{threshold} = 1$) to know the stride value of our linear arithmetic expression, and we found $k_{jump} = 3$ to be a reasonable tradeoff.

Since intermediate anticipation bits between iterations $k_{threshold}$ and $k_{threshold} + k_{jump}$ are not generated by the branch flow, we have to fill the corresponding entries of the *anticipation table* with an invalid value that will be ignored by the branch when it reads the table. As shown in the example of Figure 10(c), the anticipation process writes the *anticipation bit* for the 1^{st} iteration, then jumps to the 4^{th} and fills the 2^{nd} and 3^{rd} entries with an invalid value (*inv*). Branch 2 reads the anticipation bit but ignores it as it is *invalid*. Also, when branch 3 is fetched, it ignores the invalid anticipation bit. But when branch 4 is fetched, it reads a *valid anticipation bit* ($a4$).

We need not check for wrong value predictions because the anticipation bit is checked when the branch is resolved (see Section 3.3). But since the functions we consider are linear, the prediction is always correct with a stride-based value predictor. Also, in our experiments, we used an infinite anticipation table so no PC conflict can occur.

4. Performance Evaluation

4.1. Methodology

We first ran the 8 SpecInt95 to completion with the `ref` datasets and simulated a *gshare* predictor with an 18-bit history, maximum history size being near optimal beyond 14-bits [5]. We used a predictor table large enough to avoid PC conflicts between branches. We collected misprediction statistics for each static conditional branch, and using this data, we selected the branches responsible for 50% of the mispredictions in each code except for `gcc` and `vortex`. In `gcc` mispredictions are almost evenly distributed over many branches so we would have had to consider 173 branches to reach the 50% threshold[1]. Instead, we only selected the 7 branches with most mispredictions which account in total for 7% of all mispredictions. `vortex` has a very low misprediction rate (less than 0.4%) so we did not include this code in the experiments. Also `compress` and `go` do not contain *arithmetic expressions* or *table traversals* as mispredictions are concentrated in a few non-linear arithmetic expressions, tree/list traversals and branches not belonging to *local loops* (see Table 1). As a result, we did not apply the anticipation mechanism to these two codes.

We have statically tagged the branches in Table 1. As a result, we only address a small subset of all branch instances and IPC measurements would be meaningless in our case, so we opt for metrics focused on tagged branches like the number of cycles saved per branch or the fraction of the misprediction latency saved.

For our experiments, we extracted 100-million instruction traces, skipping the first 100-million, using ATOM [3] on an Alpha 21164. We used our own superscalar out-of-order processor simulator to evaluate the anticipation mechanism. The processor has a 5-stage pipeline (i.e., a minimum of 4-cycle branch latency), 4-issue slots, a 64-instruction window, 32KB instruction and data caches and fully pipelined functional units. In most experiments, we consider the number of functional units is unlimited, then we precisely evaluate the impact of resource conflicts in Section 4.4. We used a perfect branch target buffer and a *gshare* algorithm. The condition table is voluntarily large to reduce branch conflicts and focus on correlation issues. To simulate branch mispredictions using our trace-driven simulator, we load the correct instruction path in case of misprediction pretending it is the wrong path, then squash it when the branch is resolved and reload it again.

4.2. Dynamic Analysis of Tagged Branches

In this section, we provide several dynamic information on the tagged branches.

Number of iterations in local loops In Figure 11, for each code we provide the cumulative distribution of the number of iterations in *local loops* averaged over all tagged branches. An iteration lasts until one of the registers used

[1]Recall we built all the branch condition expressions by hand

by an instruction in the dataflow tree of the branch condition gets dirty, i.e, it is modified by an instruction that does not belong to the instruction neighborhood. When a register gets corrupted we consider the program has left the instruction neighborhood, i.e., the *local loop*.

Even though the optimal size of the jump, k_{jump}, depends on the size of the *local loop*, we generally consider that a great number of iterations potentially leads to a better performance of our anticipation mechanism.

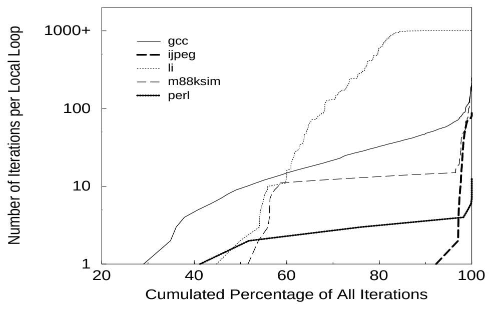


Figure 11. *Number of Iterations in Local Loops.*

Fraction of dynamic branches tagged In Figure 12 we show the fraction of tagged branch instances over all conditional branch instances. We also show the fraction of mispredictions in tagged branches over all mispredictions. Overall, the fraction of dynamic tagged branches is rather small varying from 0.3% for `gcc` to 23% for `m88ksim`. However, the fraction of mispredictions due to a tagged branch is significantly higher with 41% and 41.5% respectively for `m88ksim` and `ijpeg` and a minimum of 1% for `gcc`. This is confirmed by the misprediction rate of tagged conditional branches shown in Figure 8 (`Predictable`), on average 10% more than the average misprediction rate (`All`).

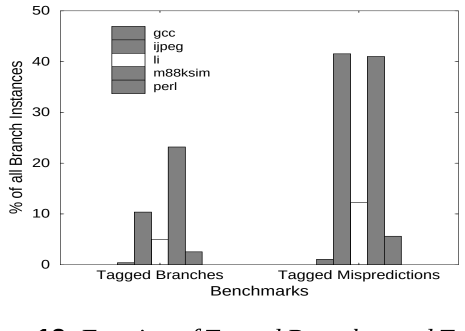


Figure 12. *Fraction of Tagged Branches and Tagged Mispredictions.*

4.3. Impact of Anticipation on Early Branch Resolution

Number of cycles saved The purpose of the anticipation mechanism is to decrease the branch resolution time, so we measure the number of cycles saved for each branch instance. For each tagged and mispredicted conditional branch instance, we measure the time interval between the moment the branch is anticipated and the moment it is resolved. 0 means the branch did not get an anticipation bit before it was resolved. In Figure 13, we show the cumulative distribution of the number of cycles saved: the y-axis is the number of cycles saved per branch instance, and the x-axis is the cumulated fraction of tagged mispredicted conditional branches. The results are very contrasted: for `li` and `ijpeg` more than 90% of branch instances save several cycles, while 70% of `perl` branches win nothing because of the small number of iterations per *local loop*. The number of cycles saved also varies a lot: `gcc` mostly wins less than 10 cycles per branch, while more than 70% of `li` branches win more than 50 cycles.

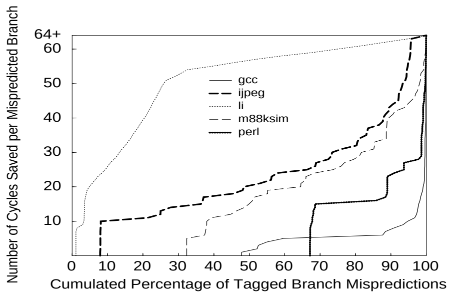


Figure 13. *Number of Cycles Saved.*

Fraction of branch misprediction latency saved A branch can benefit from the anticipation mechanism in two ways: either the branch gets an anticipation bit when it is fetched or the branch is predicted and gets an anticipation bit before it is resolved. For that purpose, we first measure the branch misprediction latency, i.e., the time interval between the moment the branch is fetched (i.e, its outcome is predicted) and the moment it is resolved, then we get the fraction of the misprediction latency saved with the anticipation mechanism, i.e., the ratio of the number of cycles saved over the misprediction latency. Figure 14 shows the cumulative distribution of the fraction of branch latency saved: the y-axis is the fraction of branch latency saved per branch instance and the x-axis is the cumulated fraction of tagged mispredicted conditional branch instances. Within each code, the results are very contrasted among branches. Some branches get no improvement and other branches can hide almost 100% of the branch misprediction latency.

4.4. Impact of Duplicate Flows on Resource Conflicts

Until now, the processor configuration assumes unlimited number of functional units. However the anticipation mechanism adds a second instruction flow which can induce

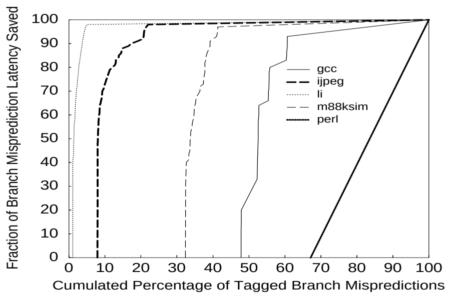


Figure 14. *Fraction of Latency Saved.*

additional resource conflicts. For that purpose, we have run simulations with the full anticipation mechanism active except no anticipation bit is provided to the normal program flow. Consequently, this processor configuration only has the flaws of the mechanism, i.e., additional resource conflicts, without the advantages. The resulting difference in number of cycles is the overhead of the anticipation mechanism. We used two configurations: 2 functional units of each type and 4 functional units of each type. Because few branches are tagged, for the sake of fairness we have divided the number of additional cycles by the number of tagged branches to get an average "overhead per branch". In Table 2, we have averaged the ratio over all codes and we observe that the average overhead per branch is equal to 1.5 cycles for the 2-FU configuration and it becomes almost negligible with the 4-FU configuration at 0.2 cycle per branch. Though the number of resource conflicts would increase if we would tag more branches, we only need to tag the branches with most mispredictions.

	2-FU	4-FU
$\frac{\textit{Number of Cycles Saved}}{\textit{Number of Tagged Branches}}$	1.51	0.2

Table 2. *Impact of Anticipation on Resource Conflicts.*

5. Conclusions and Future Work

In this study, we have extracted the branch condition expression of branches with most mispredictions. We have classified these expressions in several categories depending on their regularity (in *local loops*) and predictability. Using these expressions we could explain why these branches resist branch correlation. Moreover, we have shown branches not targeted in this study are likely to improve well as classic correlation-based predictors scale up. We have shown that for 20% of the studied branches, we can anticipate the branch outcome using a combination of value prediction and normal dataflow execution. We have proposed a hardware implementation of an anticipation mechanism that exploits these different observations. Currently, the mechanism only applies to a subset of all mispredictions, but in these cases, we have demonstrated that it successfully hides a large share of the branch misprediction latency.

To extend the scope of this mechanism we will follow two paths. We need to apply dynamic tagging so as to target as many branches as possible, and we need to increase the number of branches with predictable functions. For the latter purpose, we intend to use more complex value predictors to address non-linear arithmetic expressions, and linked data structures like lists and trees.

References

[1] Peter L. Bird, Alasdair Rawsthorne, and Nigel P. Topham. The effectiveness of decoupling. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[2] Po-Yung Chang, Eric Hao, , and Yale N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th IEEE International Symposium on Microarchitecture*, Ann Arbor, Michigan, November 1995.

[3] Digital Equipment Corporation. Atom, user manual. Technical report, Digital Equipment Corporation, Maynard, Massachusetts, June 1995.

[4] Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[5] Toni Juan, Sanji Sanjeevan, and Juan J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proceedings of the 25th International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[6] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital WRL, June 1993.

[7] Pierre Michaud, Andre Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, Denver, Colorado, June 1997.

[8] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, SanJose, California, October 1998.

[9] Yiannakis Sazeides and James E. Smith. Modeling program predictability. In *Proceedings of the 25th International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[10] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th IEEE International Symposium on Microarchitecture*, pages 238–247, Paris, France, December 1996.