The Effects of Mispredicted-Path Execution on Branch Prediction Structures

The Effects of Mispredicted-Path Execution on Branch Prediction Structures

Stéphan Jourdan† Tse-Hao Hsing Jared Stark Yale N. Patt

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

Institut de Recherche en Informatique de Toulouse †
Université Paul Sabatier, 31062 Toulouse, France

Abstract

Branch prediction accuracies determined using trace-driven simulation do not include the effects of executing branches along a mispredicted path. However, branches along a mispredicted path will pollute the branch prediction structures if no recovery mechanisms are provided. Without recovery mechanisms, prediction rates will suffer. In this paper, we determine the appropriateness of recovery mechanisms for the four structures of the Two-Level Adaptive Branch Predictor: the Branch Target Buffer (BTB), the Branch History Register (BHR), the Pattern History Tables (PHTs), and the Return Address Stack (RAS). We then propose cost-effective recovery mechanisms for these branch prediction structures. For five benchmarks from the SPECint92 suite, we show that performance is not affected if recovery mechanisms are not provided for the BTB and the PHTs. On the other hand, without any recovery mechanisms for the BHR and RAS, performance drops by an average of 29%.

1 Introduction

Trace-driven simulation has been widely used to evaluate the performance of superscalar machines employing branch prediction. However, traces contain only instructions executed along the correctly predicted path. Speculative execution inevitably results in executing some instructions along a mispredicted path. To model the execution of instructions along mispredicted paths, full simulation is needed. Throughout this paper, we refer to instructions executed along a mispredicted path as wrong-path instructions, and to instructions executed along a correct path as correct-path instructions.

The wrong-path instructions can have various effects on the machine, particularly, on the following structures:

- **caches** - including the instruction cache and the data cache
- **functional units**
- **branch prediction structures** - including the Branch Target Buffer (BTB), Pattern History Tables (PHTs), Branch History Registers (BHRs), and Return Address Stack (RAS).

These effects can be either beneficial or detrimental. For the caches, the effect of memory read operations along mispredicted paths is the allocation and the replacement of cache lines which would not have occurred had we executed only correct-path instructions. Hence, mispredicted path execution results in instruction and data cache prefetching and pollution. The overall effect is reported to be beneficial as stated in [2] and [12]. In addition, [12] presents an instruction cache prefetching technique based on mispredicted path execution that performs better than other common techniques (next-line prefetching ...). Other cache-like structures such as the BTB may behave in ways similar to the instruction and data caches. One should note that memory write operations to the data cache are performed non-speculatively in order to preserve memory coherency should an exception occur.

Wrong-path instructions may compete for access to the functional units, thus delaying the execution of correct-path instructions. When non-pipelined functional units are used, this problem becomes especially severe; a wrong-path instruction blocks access to its functional unit for the entire time it is executing. However, [2] reports that the impact on the processor performance is insignificant. The execution model used in our study features an oldest-first scheduling technique [2] and pipelined functional units. Because of this, correct-path instructions are always given priority over wrong-path instructions during scheduling, and the execution of a wrong-path instruction will never delay the scheduling of a correct-path instruction.

Wrong-path instructions also affect the structures in Two-Level Branch Predictors [17], such as the BTB, the BHRs, the PHTs, and the RAS. Information related to the wrong-path instructions may be inserted

into these structures, possibly resulting in a degradation in branch prediction accuracy.

Example 1 illustrates the effect of wrong-path instructions on the RAS when a recovery mechanism is not provided. Assume the conditional branch for *if (<cdtn>)* is incorrectly predicted taken. As a result, a subroutine call is made to *subroutine()*. Before calling *subroutine()*, its return address **RA** is pushed onto the RAS. The conditional branch for *if (<cdtn>)* is resolved before the fetch encounters the subroutine return at the end of *subroutine()*. As a result of resolving the conditional branch, the misprediction is discovered and instruction fetch is redirected to *return(value)* in *function()*. The branch predictor predicts the return address for *return(value)* by popping the RAS. Unfortunately, the address on the top of the RAS is **RA**, which is the return address for the subroutine call to *subroutine()* instead of the return address for the subroutine call to *function()*. Even more unfortunate, the remaining addresses on the RAS do not provide correct predictions for the subroutine returns to which they correspond.

```
int function()              void subroutine()
{                           {
  ...                         ...

  if (<cdtn>)               }
  {
    subroutine();
  }
  return(value);
}
```

Example 1: Effect of Mispredicted-Path Execution on the Return Address Stack.

No published studies have reported on the effects of wrong-path instructions on the branch prediction structures. All published studies on branch prediction have used trace-driven simulations, and have assumed perfect recovery mechanisms for the branch prediction structures. For instance, [17] reports on the use of *golden registers* to deal with wrong-path instructions while using trace-driven simulations. However, the author assumed that branches were resolved in order. Resolving branches in order simplifies the recovery mechanisms but impacts the performance [7]. Note that the performance degradation of 3% reported in [7] assumes a moderate amount of speculative execution. As the amount of speculative execution increases, the performance degradation becomes more severe.

This paper focuses on the effect of wrong-path instructions on the branch prediction structures. There are two options as to when to update these structures. The first option, the updates occur speculatively in the early stages of the machine's pipeline. While this allows the branch predictor to use the most recent branch history information to make the next prediction, it also requires recovery mechanisms for those prediction structures which are adversely affected by execution of instructions on the wrong-path. The second option, the updates occur non-speculatively at instruction retirement. Since the branch prediction structures are not updated speculatively, recovery mechanisms are not required for these structures. However, this significantly impairs the predictor's performance since it does not use the most recent branch history information to make predictions [15, 5]. To achieve the highest performance, speculative updates of the branch prediction structures and out-of-order branch resolution must be done. This may require complex recovery mechanisms. To justify the extra cost of implementing recovery mechanisms should wrong-path execution significantly impair the performance of the speculative update model, experiments were conducted to determine the appropriateness of such recovery mechanisms for each of the four structures of the Two-Level Adaptive Branch Predictor.

This paper is organized as follows. Sections 2 and 3 describe the execution model and the simulation process. Section 4 presents recovery mechanisms that can be used to repair branch prediction structures that have been updated incorrectly due to wrong-path execution. Simulation results concerning the effectiveness of recovery mechanisms are reported in section 5. We provide concluding remarks in section 6.

2 Execution Model

The model of execution used for this study exploits instruction level parallelism through speculative execution and dynamic scheduling. We call this model of execution the High Performance Substrate (HPS) [11]. Many elements of HPS are embodied in today's high end microprocessors, for example, the Intel P6 [4] and the PowerPC 620 [3].

Execution in HPS flows as follows: each cycle multiple instructions are issued, and, using the information in the current copy of the Register Alias Table, the instructions are merged into the reservation stations, which we call Node Tables, much like the Tomasulo algorithm merges operations into reservation stations of the IBM 360/91 [16]. Associated with each instruction (node) are the source operands for that instruction (or identifiers for obtaining the operands), and destination information. Each node is stored in its proper node table independent of and decoupled from all other nodes currently awaiting dependencies in the datapath until all its operands are available, at which point the node is eligible for firing. Each cycle, the oldest firable node of each node table is scheduled, i.e. it is shipped to a pipelined functional unit for execution. Each cycle, functional units complete execution of nodes and distribute the results to nodes waiting for these results, which then may become firable.

For memory operations, a node is firable only if all of its operands are available, if it is not dependent on any previous memory operations, and if there are no previous memory operations to unknown addresses that may interfere with the execution of the node. This dynamic memory disambiguation requires that, in the case of load operations, no previous stores are to un-

known addresses. Likewise, for store operations, any previous loads or stores to unknown addresses will stall the store operation. In addition, stores are always performed non-speculatively.

The processor simulated has an issue width of 8. We modeled a 32k byte, 8-way set-associative instruction cache with a 64 byte line size, as in the PowerPC 620. The data cache modeled was 32k byte, non-blocking, write back, write no-allocate, 8-way set-associative, with a 16 byte line size. Banking of the caches was not modeled. A perfect second level unified cache was assumed. In the event of a first level cache miss, eight cycles were required to return the data from the second level cache.

A checkpointing mechanism was used to repair the machine state in the event of a branch misprediction [6, 1]. An issue packet is a group of consecutive instructions within the dynamic instruction stream. A packet is fetched from the instruction cache using at most one prediction from the branch predictor, and issued as a whole into the node tables. Each fetch attempted to access two consecutive instruction cache lines and pull the desired instructions from these lines. Instruction issue was restricted to issuing one packet of instructions per cycle. The issue packet was broken after either the first control flow instruction, or the eighth instruction, depending on which came first. Node tables had a capacity of 32 issue packets, that is, 32 by 8 instructions, or 256 instructions. Issue stalled when node tables were at capacity. Issue packets were removed (retired) from the node tables in the order in which they were issued. All instructions in the issue packet had to complete execution before the packet could be retired.

Inst. Class	Lat.	Description
Integer	1	INT add, sub and logic ops
Bit Field	1	Shift, and bit testing
FP Add	3	FP add, sub, and convert
Multiply	3	FP mul and INT mul
Divide	8	FP div and INT div
Load	2	Memory loads
Store	2	Memory Stores
Branch	1	Control instructions

Table 1: Instruction Classes and Latencies

Eight fully pipelined execution units were modeled. Table 1 shows the instruction classes and their simulated execution latencies, along with a description of the instructions that belong to that class. Table 2 shows the functional unit configuration simulated, where the functional units are defined by what instruction classes they can execute. Functional units handling memory operations were capable of performing an address calculation in parallel with another non-address calculation. Thus, both an address calculation node and a non-address calculation could be fired to a functional unit in the same cycle.

Finally, the branch predictor was a modified Two-Level Global Adaptive Branch Predictor (GAg [17]) scheme which exclusive-ORs a global history with the fetch address to select the appropriate PHT entry (gshare [10]). We used a 16-bit global BHR. This results in a 64k-entry PHT. The RAS featured 32 entries. Unless otherwise stated, we used a 2048-entry, 4-way set-associative BTB.

Functional Unit Number	Instruction Classes Executed
1	FP Add, Integer
2	Multiply, Integer
3	Divide, Integer
4	Branch, Integer
5	Branch, Integer
6	Load/Store, Bit Field, Integer
7	Load/Store, Bit Field, Integer
8	Load/Store, Bit Field, Integer

Table 2: Simulated Machine Configuration

3 Simulation Methodology

3.1 Benchmarks

Five benchmarks from the SPECint92 suite were used. All benchmarks were compiled for the Motorola 88k instruction set using the gcc v2.4.3 compiler with all optimizations turned on. Due to our time consuming simulation technique, benchmarks were only simulated for the first 25 million instructions. Table 3 lists the five benchmarks, their data sets, the number of conditional branches and return instructions simulated along the correct-path, and the number of conditional branches and return instructions fetched while following a wrong path (w-p). The wrong-path statistics were gathered using the 16-bit gshare predictor, with perfect recovery mechanisms provided for each of the branch prediction structures. Therefore, for these statistics, the branch prediction structures were not affected by wrong-path execution. These results show that the number of predictions made while processing wrong-path instructions is significant.

3.2 Simulation Environment

Two simulators were used for the study: an instruction level simulator provided by Motorola (*archsim*) and our HPS execution driven simulator (*fullsim*). *Fullsim* is a stand-alone simulator which reads in the executable image of a benchmark, and then performs a cycle by cycle simulation of the executable. This simulation includes the execution of any instructions along mispredicted paths. (This simulation technique is much more time consuming than trace driven simulation.) *Archsim* was used to verify the correctness of *fullsim*. *Archsim* produces a trace containing the instruction addresses and the corresponding instruction data for the instructions along the correct path of

Benchmark	Data Set	Inst	Cond.	Cond. (w-p)	%	Ret.	Ret. (w-p)	%
008.espresso	bca	25 M	5.3 M	50,900	0.95	0.1 M	9794	8.13
022.li	li-input.lsp	25 M	3.7 M	105,806	2.75	0.8 M	59,765	7.24
023.eqntott	int_pri_3.eqn	25 M	4.8 M	80,411	1.64	0.1 M	7,567	5.55
026.compress	in	25 M	3.2 M	375,643	11.59	0.2 M	18,554	7.38
085.gcc	stmt.i	25 M	4 M	144,753	3.49	0.3 M	52,482	13.71

Table 3: Benchmark Summary

execution. As instructions were committed from the node tables of *fullsim* (only instructions along the correct path of execution commit), they were compared to the corresponding instruction in the trace produced by *archsim*. Any difference in instruction data or instruction address caused *fullsim* to abort the simulation.

4 Recovery Mechanisms

In this section, we first introduce state recovery mechanisms required to correctly resume execution in dynamically-scheduled processors. The state recovery mechanisms presented include the *history buffer* [14], *the reorder buffer* [14], *the future file* [14], and the *checkpoint repair mechanism* [6]. In dynamically-scheduled processors, logic must be provided to keep track of the location of each architectural register. Additionally, logic must also be provided to restore the architectural state of the register file should a misprediction occur. In the next few paragraphs, we will briefly describe how each of these recovery mechanisms can be used to repair this architectural state. Following that, we will describe how these mechanisms can be applied to the BTB, the PHTs, the BHR, and the RAS to discard the effect of wrong-path execution.

The history buffer is a stack which contains a record of older architectural register locations. Whenever the location for an architectural register changes, the previous location is recorded in the history buffer. Should a misprediction occur, the recovery process consists of restoring the architectural register locations from the history buffer. Once done, execution can resume. The major drawback of such a recovery mechanism is that it requires several cycles to restore the architectural state from the entries in the history buffer. [2] reports that this significantly impacts performance.

The reorder buffer is a queue which contains the speculatively allocated architectural register locations. The committed register file maintains the location of each non-speculative architectural register. The architectural state is maintained by both the reorder buffer and the committed register file: an associative look-up over the reorder buffer is required to find the most recent location for a given architectural register. To recover from a misprediction, the processor flushes subsequent recorded locations from the reorder buffer. For wide-issue dynamically-scheduled processors, a large number of reorder buffer entries may be required (over 100 [8]). Additionally, the number of read ports is twice the issue width. Both these factors may adversely affect the cycle time.

An alternative to avoid the costly associative look-ups in the reorder buffer is to explicitly identify the architectural state by means of a future file. Associative look-ups are no longer required. On misprediction, this architectural state must be repaired. The straightforward way to repair is to wait for the retirement of all the remaining speculative instructions. Hence, in addition to the extra space needed for the future file, this scheme requires several cycles to start the recovery process. However the recovery process is immediate since the committed register file is the architectural state required to resume. [2] reports that the delay before recovery impairs performance as much as the history buffer scheme.

The checkpoint repair mechanism establishes snapshots or *checkpoints* of the architectural state whenever a branch is predicted. If misprediction occurs, the checkpoint established for that branch will become the architectural state. When using this mechanism to repair the architectural state of the register file, the recovery process is immediate. However, this mechanism is space-consuming since each checkpoint records the architectural state of the register file. Several optimizations are based on the fact that the contents of these checkpoints differ by only a few locations, and mappings are cheaper to record than register values [1, 13, 9].

In the following sections, we describe how these recovery mechanisms can be applied to the BTB, the PHTs, the BHR, and the RAS to discard the effect of wrong-path execution.

4.1 Branch Target Buffer

Each BTB entry contains a valid bit, an address tag, a taken target address, a fall-through target address, and the branch type (unconditional branch, conditional branch, subroutine call, or subroutine return). The BTB is accessed in parallel with the instruction cache. The BTB determines whether branches are present in the block of instructions being fetched from the instruction cache. As stated in the introduction, the effect of mispredicted path execution on the BTB is the allocation and the replacement of BTB entries which would have not occurred had the machine executed only correct-path instructions. This effect can be either beneficial or detrimental. Mispredicted path execution may serve as a form of BTB prefetching, in-

creasing the BTB hit rate. On the other hand, the replacement of BTB entries results in the loss of information about the branch. Thus, the misprediction rate may increase. It is possible for both the BTB hit rate and the branch misprediction rate to increase as a result of mispredicted path execution.

Checkpointing the BTB to avoid the effects of mispredicted path execution is not viable because of the large amount of information needed for each checkpoint. Providing buffers to maintain the pending BTB entry updates (reorder buffer) or to record the overwritten BTB entries (history buffer) is a way to discard the effects of mispredicted path execution on the BTB. However, it comes at the expense of the extra buffer and the additional logic required either to read the buffer in parallel with the BTB (reorder buffer), or to handle the recovery of the BTB (history buffer). Depending on the impact of mispredicted path execution, it might be more cost-effective to provide more entries in the BTB.

4.2 Pattern History Tables

Each PHT entry contains a saturating 2-bit counter. PHTs are updated at retirement time for a correctly predicted branch. PHTs do not need to be updated speculatively, because subsequent conditional branches accessing the same PHT entry will be predicted in the same way due to the 2-bit counter algorithm. Thus, delaying the PHT update until retirement gives the same up-to-date information as would updating the PHT entry speculatively. PHTs are updated at execute time for a mispredicted branch. Since only correct-path branches can retire, mispredicted path execution does not affect the PHTs for branches that have been resolved as correct. However, if branches are executed out-of-order, the effects of mispredicted path execution on the PHTs can be observed when a mispredicted wrong-path branch is executed prior to the mispredicted correct-path branch. Note that since misprediction rates are low in Two-Level Adaptive Branch Prediction Schemes, this occurs infrequently. Furthermore, recovering from this pollution effect is simple if required. Since only a few incorrect updates occur, a history buffer based recovery mechanism is appropriate. To resume execution, the processor does not wait for the history buffer to restore the PHTs, as is required for the register file. The PHT entries in the history buffer can be restored whenever there is a cycle in which no branch is retired. Therefore, an extra write port to the PHTs is not required.

4.3 Branch History Registers

In a global scheme, the BHR maintains the history of past conditional branches. [5] reports that the predictor should use the most up-to-date history to achieve low misprediction rates. Therefore, the BHR is updated immediately after the prediction is made. The update is speculative and thus the effect of mispredicted branches can be observed if no recovery mechanism is provided. If no recovery mechanism is provided, each wrong-path conditional branch that is fetched inserts a bit into the BHR. These wrongly inserted bits remain in the BHR, resulting in poor conditional branch prediction accuracy. A recovery mechanism for a global predictor is simple to implement.

The history buffer used to restore the architectural state of the register file can also be used to restore the BHR. Conditional branch instructions do not specify a destination register. The history buffer entry that would have been allocated to hold the result of the branch instruction can be used instead to hold the BHR. During the restore process, both architectural register and the BHR are corrected in the same way. A similar technique can be used if the architectural state is maintained with a reorder buffer. However, the most appropriate recovery mechanism for the BHR is checkpointing, since there is little information to record. An alternative to the common checkpointing scheme is to use a wider circular BHR which maintains the outcomes of speculative branches and non-speculative history. Based on the checkpoint number corresponding to the checkpoint holding the mispredicted branch, the subsequent speculative predictions can be shifted out from this wider history register on misprediction.

The history buffer used to restore the architectural state of the register file can still be used to restore the BHRs for the per-address prediction scheme [17]. The similar technique can still be used if the architectural state is maintained with a reorder buffer. Checkpointing will be costly because of the amount of information required for each checkpoint (number of BTB entries times the width of the history registers). Therefore, for the per-address scheme, a reorder buffer or a history buffer is more appropriate.

4.4 Return Address Stack

The RAS is used to predict the targets of return instructions. For each subroutine call, the return address is pushed onto the RAS. For each subroutine return, the target is predicted by popping the RAS. To provide a recovery mechanism for the RAS, we can checkpoint the pointers used to access the RAS. In the following paragraphs, we explain the design of such a Checkpointed RAS.

Figure 1 is a diagram of a Checkpointed RAS. A Checkpointed RAS contains three components: register file RAS, register TOS (top of stack), and register NEXT. RAS contains the return addresses for the most recent subroutine calls. TOS points to the entry in RAS needed to predict the next subroutine return. NEXT points to the entry in RAS to be written for the next subroutine call. Each entry in RAS has two fields: NOS (next on stack) and ADDRESS. The NOS field of an entry points to the RAS entry that is logically next on the stack after that entry. For example, the NOS field of the RAS entry pointed to by TOS is the RAS entry for the second item on the stack.

The NEXT counter is incremented for each subroutine call. The NEXT counter is not decremented for subroutine returns. If the RAS contains 8 entries, as in the figure, a unique RAS entry will be allocated for the 8 most recently encountered subroutine calls. On overflow, the NEXT counter wraps around to point to the entry which was allocated for the least recently

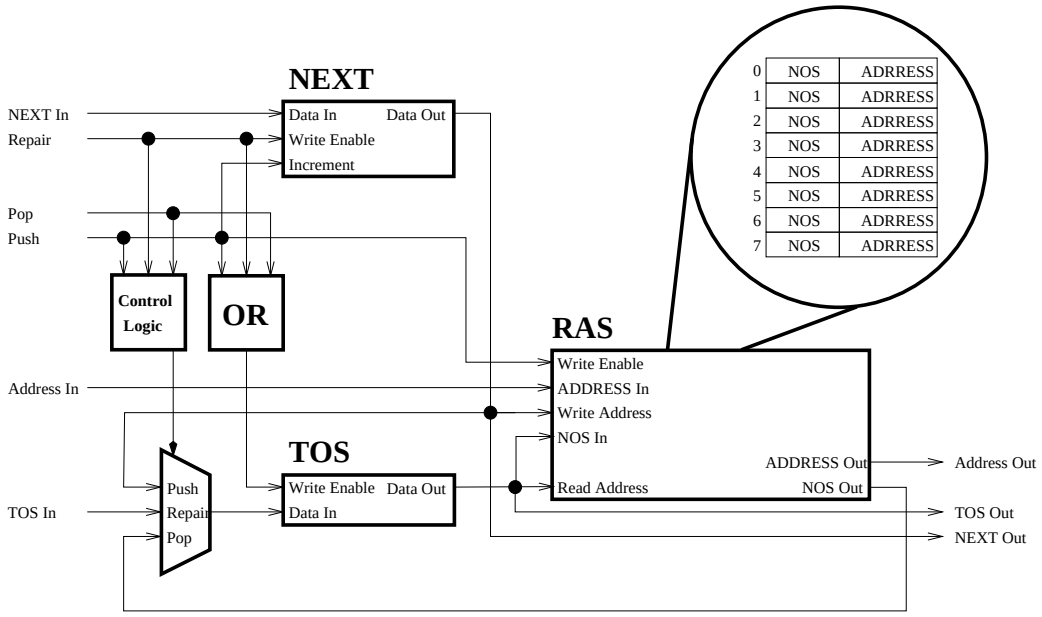

Figure 1: Checkpointed RAS.

encountered subroutine call in the RAS.

Consider what would happen if the NEXT counter was decremented for subroutine returns. If the NEXT counter was decremented for subroutine returns, the NEXT pointer would always be equal to the TOS pointer. This would result in behavior identical to that of a non-checkpointed RAS. Suppose the following branches are encountered in the following order: a subroutine call, a conditional branch, a subroutine return, and another subroutine call. If the NEXT counter was decremented for subroutine returns, the second subroutine call will be allocated the same RAS entry as the first subroutine call. Thus, the return address for the first subroutine call will be overwritten with the return address for the second subroutine call. If the conditional branch is mispredicted, the return address for the first subroutine call will be needed to predict the subroutine return. Unfortunately, this return address is longer present in the RAS. The problem is solved by not decrementing the NEXT counter for subroutine returns.

For a subroutine call, the following steps are taken:

1. *Return address of the subroutine call is written into the ADDRESS field of the RAS entry indicated by NEXT.*
2. *TOS is written into the NOS field of the RAS entry indicated by NEXT. Thus, the NOS field provides a link to the old top of stack.*
3. *NEXT is copied into TOS.*
4. *NEXT is incremented.*

For a subroutine return:

1. *Predict the return address using the ADDRESS field of the RAS entry indicated by TOS.*
2. *Copy NOS field of the RAS entry indicated by TOS into TOS.*

The values of TOS and NEXT are used to checkpoint the state of the Checkpointed RAS. Associated with each branch prediction are the values of TOS and NEXT that were valid before the branch prediction was made. To recover from a branch misprediction, the values TOS and NEXT values associated with the mispredicted branch are reloaded into the TOP and NEXT registers.

5 Simulation Results

In the previous section, we investigated several recovery mechanisms for the different structures in the Two-Level Adaptive Branch Predictor. In this section, we report the impact that providing these recovery mechanisms has on the processor performance. For all experiments, we used a 16-bit gshare scheme with a 32-entry RAS. Unless otherwise stated, we used a 2048-entry 4-way set-associative BTB. Figures 2 and 3 show our experimental results.

Figure 2 (a) compares the overall performance, expressed in *Instructions Retired per Cycle* (IPC), between a processor with recovery mechanisms for all the branch prediction structures, and a processor without any recovery mechanism for the branch prediction structures. It also shows the conditional branch misprediction rate (due to both misprediction and target address misfetch), the return misprediction rate, and the BTB miss rate. Only correct-path instructions are used to calculate these rates because only correct-path instructions actually contribute to performance. From the IPC numbers, we observe that without any recovery mechanisms for its branch prediction structures, the processor suffers a significant

Figure 2: Impact on Performance of Recovery Mechanisms for Branch Prediction Structures (1).

performance loss (on average, performance drops from 2.77 IPC to 1.97 IPC). This is a consequence of the large increases in the conditional branch misprediction rate (5.42% to 23.08%) and the return misprediction rate (3.03% to 53.04%). It is interesting to note the prefetching effect due to wrong-path execution for cache-like structures mentioned in [4] and [13] is also observed for the BTB when running gcc. This indicates that wrong-path branch instructions actually allocate BTB entries that are subsequently accessed by correct-path branches. For gcc, the BTB miss rate drops significantly from 3.46% to 2.75% when no recovery mechanism is provided; however, this does not offset the negative impact of the increased misprediction rate. Since other benchmarks do not have large number of static branches, their BTB miss rates remain about the same.

From the above results, we conclude that recovery mechanisms for branch prediction structures are necessary to avoid performance degradation. In the remaining part of this section we will take a closer look at the effect of providing recovery mechanisms for each of the structures of the Two-Level Adaptive Branch Predictor.

The results in figure 2 (b) present the performance difference for BTBs with and without a recovery mechanism. To isolate the effect of wrong-path execution on the BTB, we do not pollute the BHR, the PHTs, and the RAS when following a wrong-path. For all benchmarks, not providing a recovery mechanism for the BTB only slightly degrades the IPC (1.3%). This suggests that it is not worthwhile to implement a recovery mechanism for BTBs. Again, as observed in figure 2 (b) for gcc, wrong-path execution prefetches BTB entries. This prefetching reduces the BTB miss rate. On the other hand, the replacement of BTB entries results in the loss of information about the branch. Thus, the misprediction rate may increase.

We also studied a smaller, 2-way set associative BTB with 512 entries. These results are shown in Figure 2 (c). Except for gcc and xlisp, a BTB without a recovery mechanism performs about as well as a BTB with a recovery mechanism. For gcc, wrong-path execution results in higher conditional branch and return misprediction rates. This is caused by the increased address misfetch rate caused by the replacement of BTB entries when following the wrong-path. For xlisp, which has a large number of subroutine calls and returns, the increase in return mispredictions degrades performance. Another observation is that as BTB becomes smaller, the replacement of BTB entries by wrong-path execution starts to offset the benefit of prefetching into BTB. Prefetching is counterproductive unless there are enough BTB entries to hold the prefetched items without evicting entries that are likely to be needed in the near future. As a result, gcc no longer has a reduced BTB miss rate due to wrong-path execution, as was the case for the larger BTB. In summary, our simulation results do not justify the use of a recovery mechanism for the BTB. For those benchmarks that are more sensitive to the detrimental effects of wrong-path execution such as gcc and xlisp, it might be more cost-effective to increase the BTB size rather than to implement a costly recovery mechanism.

The effect of wrong-path execution on the PHTs is shown by Figure 3 (a). For this experiment, we assumed recovery mechanisms for the BHR and the RAS, but not for the BTB. Enabling wrong-path pollution of the PHTs does not significantly change the IPC. In fact, the IPC improves (2.734 to 2.760 on average) due to the reduction in the conditional branch misprediction rate. The return mispredicted rate and BTB miss rate remain roughly the same since only the PHTs are affected. As described in section 4.2, our model updates the PHTs for a mispredicted branch at the execute stage. We believe that a branch along a mispredicted path, that is resolved as being mispredicted, trains the PHTs for the subsequent predictions of that branch. This results in better overall performance. Given these facts, a recovery mechanism is not required for PHT.

Figure 3 (b) shows the effect of wrong-path execution on the RAS. For this experiment, a recovery mechanism was provided for the BHR, but not for the BTB and PHTs. As pointed out in example 1 of the introduction, we would expect a significant increase in the return misprediction rate if a recovery mechanism was not provided. In fact, the average return misprediction rate increases from 3.31% to 21.84% when a recovery mechanism is not provided. The conditional branch misprediction rate and the BTB miss rate remain approximately unchanged. For benchmarks with a small number of subroutine calls and returns, the overall IPC is not significantly affected even when there is a high return misprediction rate. On the other hand, for benchmarks that have a large number of subroutine calls and returns, such as gcc and xlisp, high return misprediction rates can lead to a significant performance degradation. Since a recovery mechanism for the RAS is simple to implement (see section 4.4), we suggest that it be included in the branch predictor for higher performance.

As can be seen in figure 3 (c), wrong-path execution significantly reduces performance if no recovery mechanism is provided for the BHR. For this experiment, a recovery mechanism was provided for the RAS, but not for the BTB and PHTs. On average, the conditional branch misprediction rate grows from 5.59% to as high as 20.74%, leading to a drop in the IPC from 2.76 to 2.12 (23% decrease). Based on this result, it is necessary to provide a recovery mechanism for the BHR.

Besides speculatively updating the BHR, and providing a recovery mechanism for the BHR in the case of a misprediction, we also have the option of updating the BHR at instruction retirement. Figure 3 (c) shows that updating at retirement causes an average decrease of 9.3% in the IPC as compared to the scheme with speculative update and a recovery mechanism. This confirms the results obtained in [5], where it was found that the predictor should use the most up-to-date BHR in order to make an accurate prediction. Nevertheless, updating the BHR at instruction retirement still outperforms the scheme with speculate update of the BHT but without a recovery mechanism for the BHT. This is because only correct-path instructions retire, and therefore no wrong-path branches update the BHR.

Figure 3: Impact on Performance of Recovery Mechanisms for Branch Prediction Structures (2).

6 Concluding Remarks

In this paper, we have examined the effects of mispredicted path execution on the four structures of the Two-Level Adaptive Branch Predictor: the BTB, the PHTs, the BHR, and the RAS. We have proposed appropriate recovery mechanisms to disable the effects of mispredicted path execution on each of these prediction structures. We have run experiments to justify the extra cost of implementing recovery mechanisms should the performance be affected by wrong-path execution. We have shown that the performance drops by an average of 29% if no recovery mechanisms are provided for the branch prediction structures. We have also shown that recovery mechanisms for the BHR and the RAS should be provided to achieve good performance. When no recovery mechanism is provided for the BTB, pollution effects outweigh any prefetching effects; when no recovery mechanism is provided for the BTB, IPC decreases by 1.3%. The PHTs are not adversely affected by wrong-path execution. For xlisp, we found that a recovery mechanism would be beneficial for the RAS. We found that mispredicted path execution severely affects the BHR, resulting in a 23% degradation in IPC. Finally, we described simple recovery mechanisms for the RAS and the BHR.

An interesting result found in this study is that wrong-path execution affects the PHTs in a way that improves branch prediction accuracy. We are now investigating relevant heuristics to train the predictor, including a heuristic that updates the PHT entries using branches that have been resolved as correct, but are along a mispredicted path.

Acknowledgments

This research was supported in part by gifts from Intel Corporation and NCR Corporation. Stephan Jourdan's stay at the University of Michigan was funded by the University of Toulouse. Tse-Hao Hsing was supported by CNPq - The Brazilian Research Council. We greatfully acknowledge all of the above support.

References

[1] M. G. Butler and Y. N. Patt, "An Area-Efficient Register Alias Table For Implementing HPS," *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[2] M. G. Butler, "Aggressive Execution Engines for Surpassing Single Basic Block Execution," *PhD Thesis*, University of Michigan, 1993.

[3] S. Ewedemi, D. Todd, and J. Yen, "Design Issues of the High Performance PowerPC 620 Microprocessor," *Somerset Design Center*, December 1994.

[4] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, Vol. 9 Num. 2, 1995.

[5] E. Hao, P-Y Chang, and Y. N. Patt, "The effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, November 1994.

[6] W. Hwu and Y. N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines," *IEEE Transactions on Computers*, December 1987.

[7] M. Johnson, "Superscalar Microprocessor Design," *Prentice-Hall*, 1991.

[8] S. Jourdan, P. Sainrat, and D. Litaize, "Exploring Configurations of Functional Units in an Out-of-Order Superscalar Processor," *Proceedings of the 22nd Annual Symposium on Computer Architecture*, June 1995.

[9] Mips Technologies Incorporated, "R10000 Microprocessor Product Overview," *Technical Report*, October 1994.

[10] S. McFarling, "Combining Branch Predictors," *Technical Report TN-36*, Digital Western Research Laboratory, June 1993.

[11] Y. N. Patt, W. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proceedings of the 18th Annual Workshop on Microprogramming*, December 1985.

[12] J. E. Pierce, "Cache Behavior in the Presence of Speculative Execution - The Benefits of Misprediction," *PhD Thesis*, University of Michigan, 1995.

[13] S. Simone et al., "Implementation Trade-offs in Using a Restricted Data Flow Architecture in a High Performance RISC Microprocessor," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[14] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985.

[15] A. R. Talcott, W. Yamamoto, M. J. Serrano, R. C. Wood, and M. Nemirovsky, "The Impact of Unresolved Branches on Branch Prediction Scheme Performance," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[16] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, Vol. 11, January 1967.

[17] T. Yeh, "Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors," *PhD Thesis*, University of Michigan, 1993.