International Journal of Parallel Programming, Vol. 25, No. 5, 1997

Recovery Requirements of Branch Prediction Storage Structures in the Presence of Mispredicted-Path Execution

Stéphan Jourdan,[1,2] Jared Stark,[1]
Tse-Hao Hsing,[1] and Yale N. Patt[1]

Execution along mispredicted paths may or may not affect the accuracy of subsequent branch predictions if recovery mechanisms are not provided to undo the erroneous information that is acquired by the branch prediction storage structures. In this paper, we study four elements of the Two-Level Branch Predictor: the Branch Target Buffer (BTB), the Branch History Register (BHR), the Pattern History Tables (PHTs), and the Return Address Stack (RAS). For each we determine whether a recovery mechanism is needed, and, if so, show how to design a cost-effective one. Using five benchmarks from the SPECint92 suite, we show that there is no need to provide recovery mechanisms for the BTB and the PHTs, but that performance is degraded by an average of 30% if recovery mechanisms are not provided for the BHR and RAS.

KEY WORDS: Architecture; processor design; branch prediction; speculative execution; superscalar.

1. INTRODUCTION

Speculative execution, even with a great branch predictor, from time to time results in instruction execution along a mispredicted path. We refer to

[1] Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan 48109-2122.
[2] Institut de Recherche en Informatique de Toulouse, Université Paul Sabatier, 31062 Toulouse, France.

0885-7458/97/1000-0363$12.50/0

the execution of such instructions as wrong-path instructions. Instructions executed along the correct path we call correct-path instructions.

Wrong-path instructions can affect machine performance substantially, sometimes for good, sometimes for not good. Caches, for example, will cause pollution due to load misses that never should have been processed. The effect of this pollution, however, is sometimes beneficial.[1-3]

Wrong-path instructions may compete with correct-path instructions for functional units, thus delaying the execution of correct-path instructions. This is particularly annoying (although not particularly harmful[1]) in the case of nonpipelined functional units where a scarce resource can be tied up for an inordinate amount of time. Because our execution model uses oldest-ready dynamic scheduling[4] and pipelined functional units, correct-path instructions are always given scheduling priority over wrong-path instructions, and so correct-path instructions are never delayed.

Wrong-path instructions also affect the storage mechanisms of the Two-Level Branch Predictors.[5] Two-Level Branch Predictors use two levels of history to make branch predictions. The first-level history records the outcomes of the most recently executed branches and the second-level history keeps track of the more likely direction of a branch when a particular pattern is encountered in the first level history. The 2-level branch predictor uses one or more k-bit shift registers, called Branch History Registers (BHRs), to record the branch outcomes of the most recent k branches. It uses one or more arrays of 2-bit saturating up-down counters, called Pattern History Tables (PHTs), to keep track of the more-likely direction for branches. The lower bits of the branch address select the appropriate PHT and the value in the BHR selects the appropriate 2-bit counter to use within that PHT. Processing wrong-path instructions causes the storage mechanisms of the Two-Level Branch Predictors to acquire incorrect branch information. This can pollute the Branch Target Buffer (BTB), the BHRs, the PHTs, and the Return Address Stack (RAS).

Example 1 illustrates the effect of wrong-path instructions on the RAS if no misprediction recovery mechanism is provided.

If the conditional branch *if* (⟨*condition*⟩) is predicted taken, control is redirected to *subroutine*( ) and its return address (say, **RA**) is pushed on the RAS. Suppose this prediction is incorrect, and further, that it is discovered before *subroutine*( ) completes. Control proceeds down the correct path, i.e., to *return* (*value*). The branch predictor predicts the return address for *return* (*value*) by popping RAS. Unfortunately, the address at the top of RAS is **RA**, which is the return address for *subroutine*( ), and not for *function*( ). What is even worse is that ALL the remaining entries on the RAS will produce incorrect predictions.

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

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

Example 1. Effect of mispredicted-path execution on the Return Address Stack.

We are not aware of any heretofore published studies that have reported on the effects of wrong-path instructions on branch prediction. The studies which we are aware of use trace-driven simulation, and assume perfect recovery mechanisms. One of the most comprehensive studies of branch prediction[6] uses golden registers to deal with wrong-path effects, and assumes that branches are resolved in order. Resolving branches in order simplifies recovery but degrades performance.[7] Although a moderate amount of speculative execution only degrades performance a small amount (3% reported in Ref. 7), wider issue widths and deeper pipelines will result in increased speculative execution, which should make the performance degradation become much more severe.

This paper focuses on the recovery requirements of branch prediction storage structures due to adverse effects of wrong-path instructions. Branch prediction storage structures can be updated speculatively as the predictions occur, or nonspeculatively at retirement time. In the first case, branches can use the most recent history information in making subsequent predictions. However, recovery mechanisms must be provided for those structures that are adversely affected by wrong-path instructions. In the second case, recovery mechanisms are not needed, but the predictor suffers badly from not being able to use the latest history information.[8,9]

The bottom line is that since high performance requires speculative updates to the branch prediction storage structures, complex recovery mechanisms must be provided in those cases where it matters. In this paper, we determine which of the four structures present in the Two-Level predictor require complex recovery mechanisms.

The paper is organized as follows: Section 2 describes the execution model. Section 3 describes the simulation process. Section 4 describes recovery mechanisms. Section 5 reports the measured effectiveness of these recovery mechanisms. Section 6 provides concluding remarks.

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).[10] Many elements of HPS are embodied in today's high end microprocessors, for example, the Intel P6,[11] the PowerPC 620,[12] and the Digital Alpha 21264.[13]

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.[14] 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 unknown addresses. Likewise, for store operations, any previous loads or stores to unknown addresses will stall the store operation. In addition, stores are always performed nonspeculatively.

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, nonblocking, 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.[15, 16] 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.

Eight fully pipelined execution units were modeled. Table I shows the instruction classes and their simulated execution latencies, along with a description of the instructions that belong to that class. Table II 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 nonaddress calculation. Thus, both an address calculation node and a nonaddress calculation could be fired to a functional unit in the same cycle.

Table I. Instruction Classes and Latencies

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 II. Simulated Machine Configuration

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

Finally, the branch predictor was a modified Two-Level Global Adaptive Branch Predictor ($GAg^{(5)}$) scheme which exclusive-ORs a global history with the fetch address to select the appropriate PHT entry ($gshare^{(17)}$). We used a 16-bit global BHR. This results in a 64k-entry PHT. The RAS featured 32 entries. We used a 2048-entry, 4-way set-associative BTB.

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 100 million instructions, except for compress, which completes in 86.4 million instructions. Table III lists the five benchmarks, their data sets, and the number of instructions simulated.

Table III. Benchmark Summary

Benchmark	Data Set	Inst
008.espresso	bca	100 M
022.li	li-input.lsp	100 M
023.eqntott	int_pri_3.eqn	100 M
026.compress	in	86.4 M
085.gcc	stmt.i	100 M

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 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*, *the reorder buffer*, *the future file*,[18] the *checkpoint repair mechanism*.[14] 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. Butter and Patt[19] report 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 of 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[20]). Additionally, the number of read ports is twice the issue width. Both these factors may adversely affect the cycle time.

An alternative to avoiding 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. Butler and Patt[19] report 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.[16, 21]

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 not have 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, increasing 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. Hao *et al.*[9] report 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 could (depending on the implementation) 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.[20] A 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 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 noncheckpointed 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 would be allocated the same RAS entry as the first subroutine call. Thus, the return address for the first subroutine call would 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 would no longer be 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.

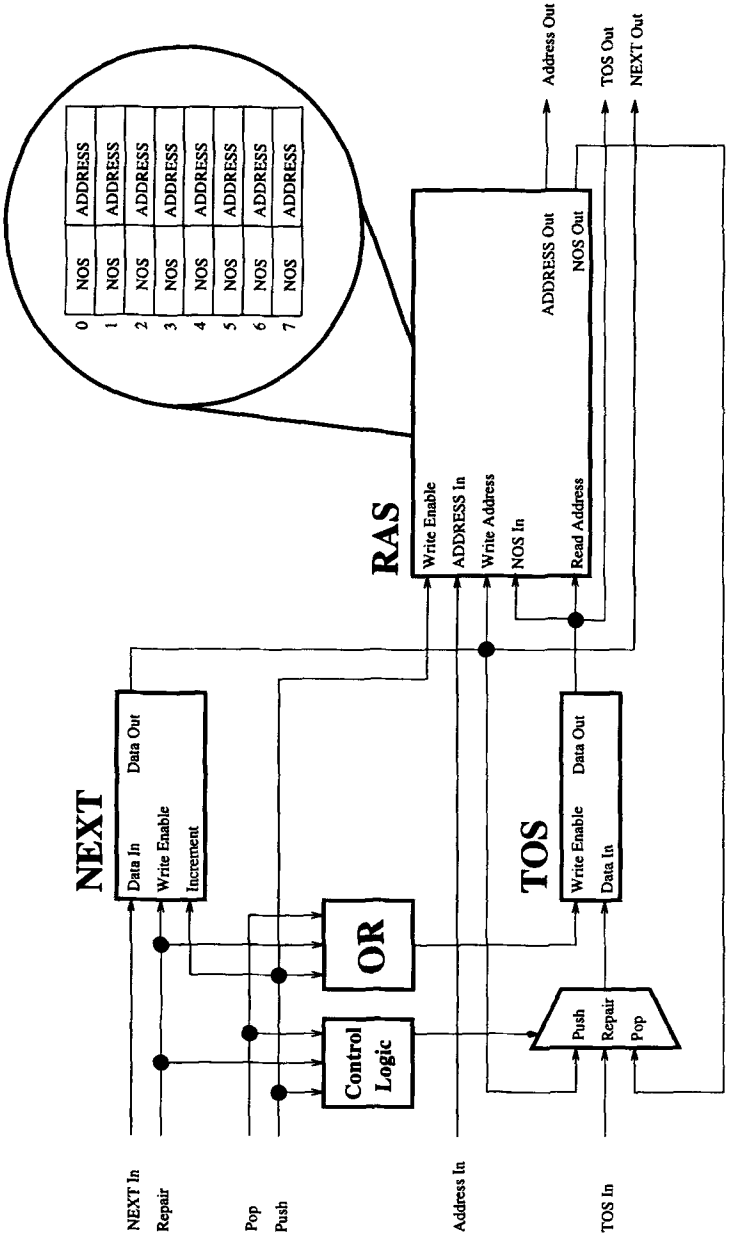


Fig. 1. Checkpointed RAS.

For a subroutine return:

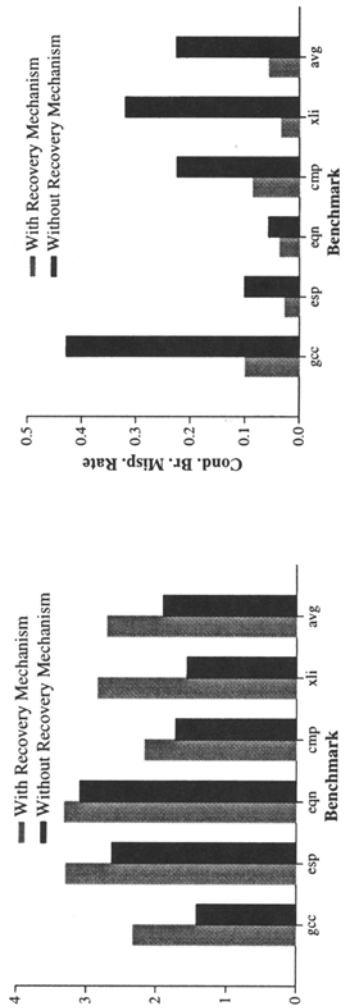
1. Predict the return address using the ADDRESS field of the RAS entry indicated by TOS.
2. Copy NOS field of the RAS entry indicated by TOS into TOS.

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, and a 2048-entry 4-way set-associative BTB. Figures 2–5 show our experimental results.

Figure 2 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 performance loss (on average, performance drops from 2.70 IPC to 1.90 IPC). This is a consequence of the large increases in the conditional branch misprediction rate (5.60% to 22.62%) and the return misprediction rate (2.80% to 44.17%). It is interesting to note the prefetching effect due to wrong-path execution for cache-like structures mentioned in Refs. 1–3, 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 from 2.58% to 2.20% 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 numbers of



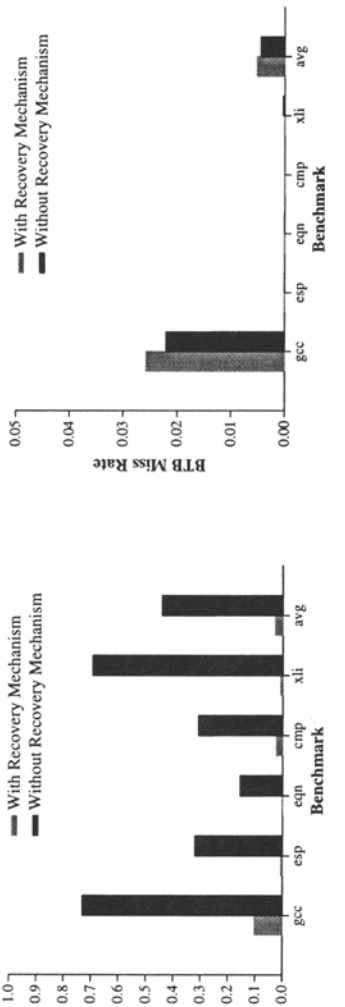


Fig. 2. Performance difference between a processor with recovery mechanisms for all of its branch prediction structures, and a processor without any recovery mechanisms for its branch prediction structures.

static branches, their BTB miss rates remain about the same. From these 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 Fig. 3 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. This suggests that it is not worthwhile to implement a recovery mechanism for BTBs. Again, as observed in Fig. 3 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.

The effect of wrong-path execution on the PHTs is shown by Fig. 4. 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 degrade the IPC. The IPC decreases from 2.701 to 2.699 on average. The conditional branch misprediction rate, the return mispredicted rate, and BTB miss rate remain roughly the same since only the PHTs are affected. From these results, we conclude that a recovery mechanism is not required for PHTs.

Figure 5 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 2.86% to 19.08% 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 Fig. 6, 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

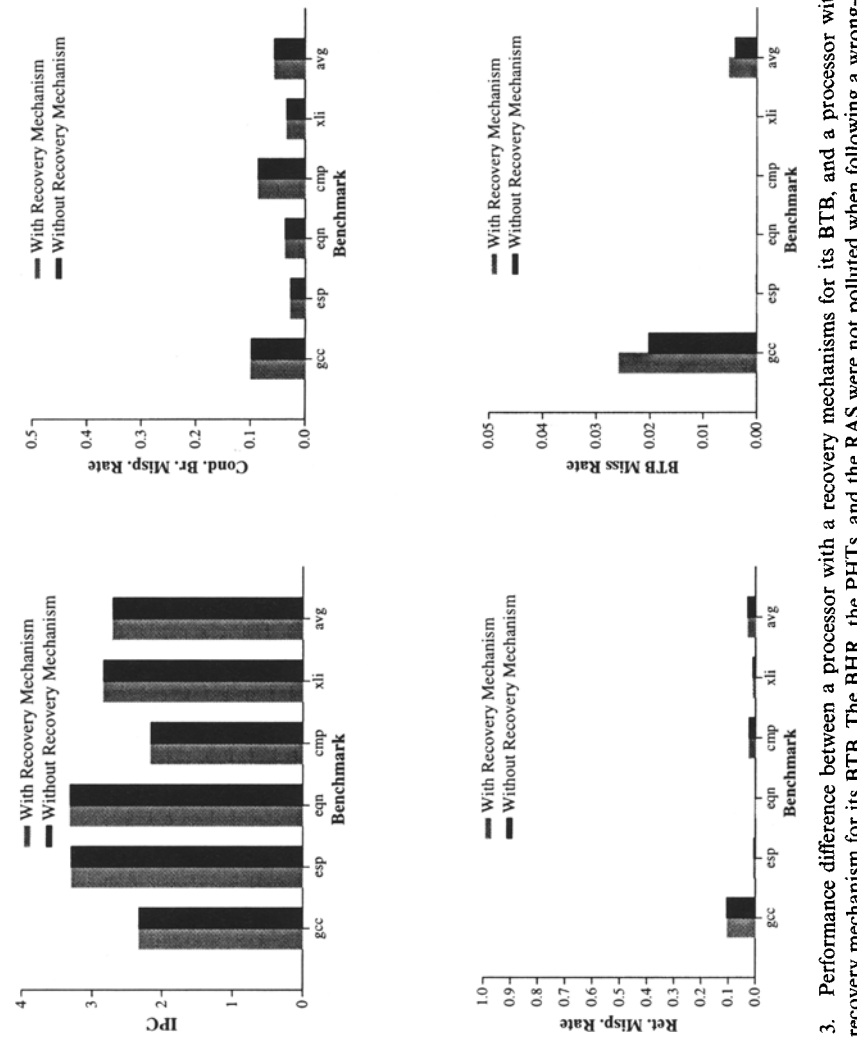


Fig. 3. Performance difference between a processor with a recovery mechanisms for its BTB, and a processor without any recovery mechanism for its BTB. The BHR, the PHTs, and the RAS were not polluted when following a wrong-path.

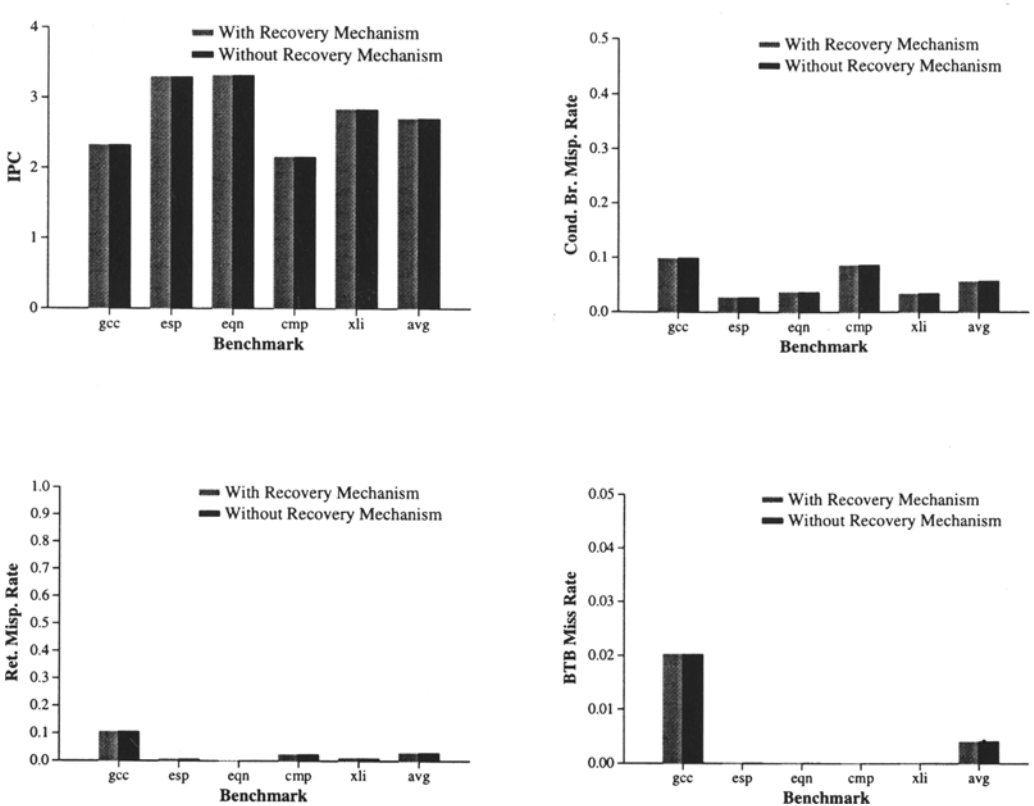


Fig. 4. Performance difference between a processor with a recovery mechanisms for its PHTs, and a processor without any recovery mechanism for its PHTs. The BTB was polluted when following a wrong-path. The BHR and the RAS were not polluted when following a wrong-path.

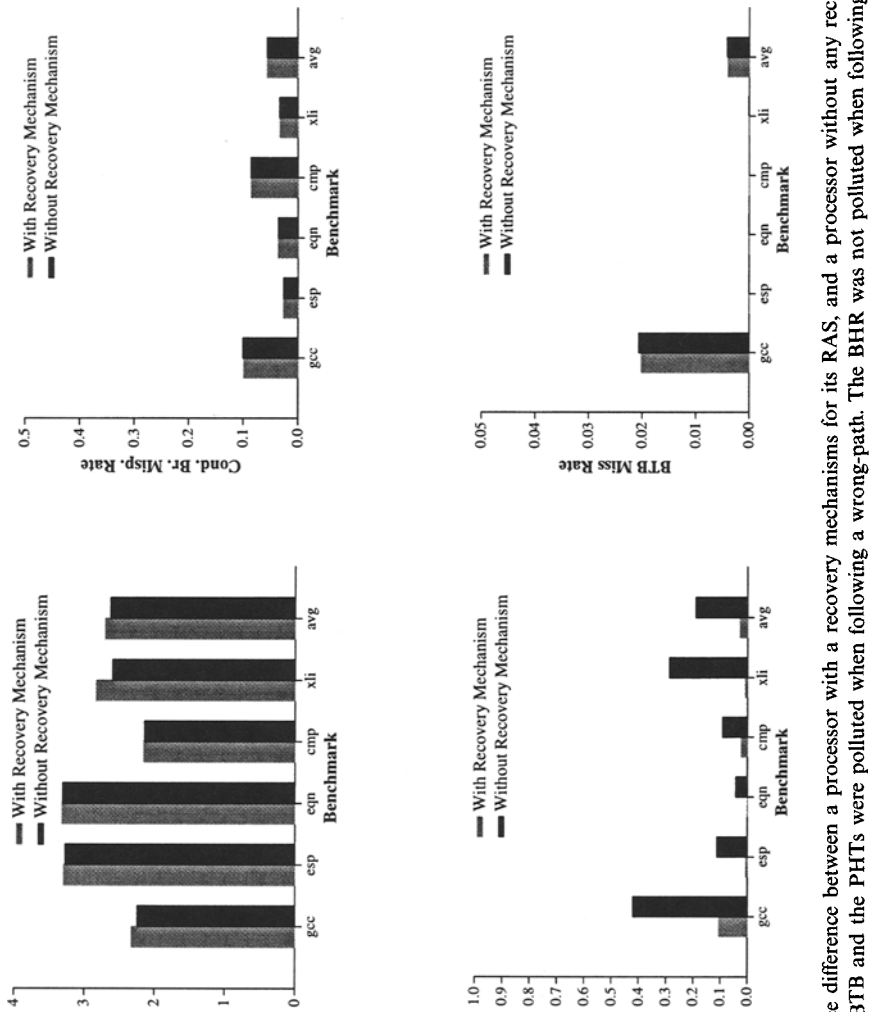


Fig. 5. Performance difference between a processor with a recovery mechanisms for its RAS, and a processor without any recovery mechanism for its RAS. The BTB and the PHTs were polluted when following a wrong-path. The BHR was not polluted when following a wrong-path.

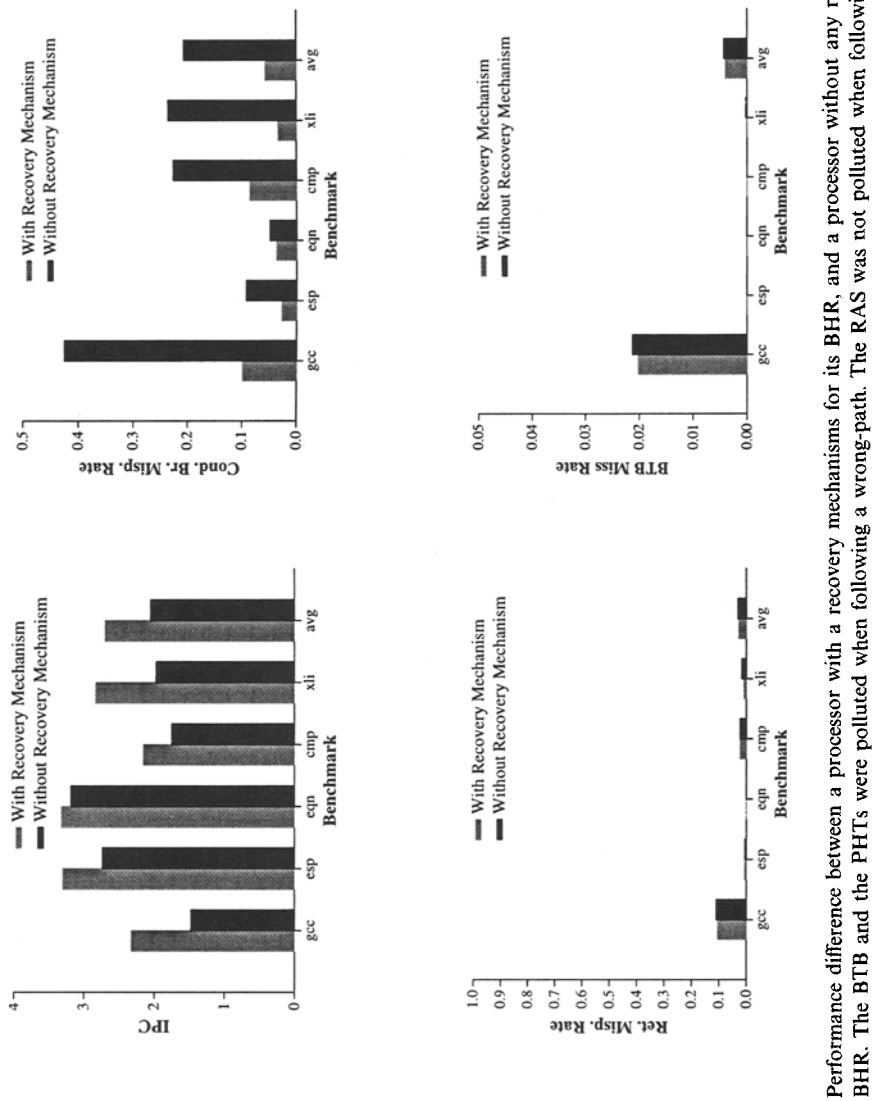


Fig. 6. Performance difference between a processor with a recovery mechanisms for its BHR, and a processor without any recovery mechanism for its BHR. The BTB and the PHTs were polluted when following a wrong-path. The RAS was not polluted when following a wrong-path.

the BTB and PHTs. On average, the conditional branch misprediction rate grows from 5.62% to 20.58%, leading to a drop in the IPC from 2.70 to 2.05 (24% decrease). Based on this result, it is necessary to provide a recovery mechanism for the BHR.

6. CONCLUDING REMARKS

In this paper, we have examined the effects of mispredicted path execution on the four storage 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 30% 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 slightly outweigh any prefetching effects. However, the performance degradation caused by the BTB pollution is small enough that a recovery mechanism is not warranted for the BTB. 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 24% degradation in IPC. Finally, we described simple recovery mechanisms for the RAS and the BHR.

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, Aggressive Execution Engines for Surpassing Single Basic Block Execution, Ph.D. Thesis, University of Michigan, 1993.
2. J. Pierce and T. Mudge, The Effect of Speculative Execution on Cache Performance, *Proc. Int'l Parallel Processing Symp.* (April 1994).
3. J. Pierce and T. Mudge, Wrong-Path Instruction Prefetching, *Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture* (December 1996).

4. M. G. Butler and Y. N. Patt, An Investigation of the Performance of Various Dynamic Scheduling Techniques, *Proc. 25th Ann. ACM/IEEE Int'l Symp. Microarchitecture*, (December 1992).
5. T. Yeh and Y. N. Patt, Alternative Implementations of Two-Level Adaptive Branch Prediction, *Proc. 19th Ann. Symp. on Computer Architecture* (May 1992).
6. T. Yeh, Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors, Ph.D. Thesis, University of Michigan (1993).
7. M. Johnson, Superscalar Microprocessor Design, Prentice-Hall (1991).
8. A. R. Talcott, W. Yamamoto, M. J. Serrano, R. C. Wood, and M. Nemirovsky, The Impact of Unresolved Branches on Branch Prediction Scheme Performance, *Proc. 21st Ann. Int'l Symp. Computer Architecture* (April 1994).
9. E. Hao, P-Y Chang, and Y. N. Patt, The effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited, *Proc. 27th Ann. Int'l Symp. Microarchitecture* (November 1994).
10. Y. N. Patt, W. Hwu, and M. Shebanow, HPS, A New Microarchitecture: Rationale and Introduction, *Proc. 18th Ann. Workshop on Microprogramming* (December 1985).
11. L. Gwennap, Intel's P6 Uses Decoupled Superscalar Design, *Microprocessor Report*, Vol. 9 No. 2, 1995.
12. S. Ewedemi, D. Todd, and J. Yen, Design Issues of the High Performance PowerPC 620 Microprocessor, Somerset Design Center (December 1994).
13. L. Gwennap, Digital 21264 Sets New Standard, *Microprocessor Report*, Vol. 10, No. 14 (1996).
14. R. M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM J. Res. Develop.*, Vol. 11 (January 1967).
15. W. Hwu and Y. N. Patt, Checkpoint Repair for Out-of-Order Execution Machines, *IEEE Trans. Comp.* (December 1987).
16. M. G. Butler and Y. N. Patt, An Area-Efficient Register Alias Table For Implementing HPS, *Proc. Int'l Conf. Parallel Processing* (1990).
17. S. McFarling, Combining Branch Predictors, Technical Report TN-36, Digital Western Research Laboratory (June 1993).
18. J. E. Smith and A. R. Pleszkun, Implementation of Precise Interrupts in Pipelined Processors, *Proc. 12th Ann. Int'l Symp. Computer Architecture* (June 1989).
19. M. G. Butler and Y. N. Patt, A Comparative Performance Evaluation of Various State Maintenance Mechanisms, *Proc. 26th Ann. ACM/IEEE Int'l Symp. Microarchitecture* (December 1993).
20. S. Jourdan, P. Sainrat, and D. Litaize, Exploring Configurations of Functional Units in an Out-of-Order Superscalar Processor, *Proc. 22nd Ann. Symp. Computer Architecture* (June 1995).
21. S. Simone, A. Essen, A. Ike, A. Krishnamoorthy, N. Patker, M. Ramaswami, and V. Thirumalaiswamy, Implementation Trade-offs in Using a Restricted Data Flow Architecture in a High Performance RISC Microprocessor, *Proc. 22nd Ann. Int'l Symp. Computer Architecture* (June 1995).