Fast Branch Misprediction Recovery in Out-of-order Superscalar Processors

Peng Zhou Soner Önder Steve Carr
Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931-1295
{pzhou,soner,carr}@mtu.edu

ABSTRACT

Current trends in modern out-of-order processors involve implementing deeper pipelines and a large instruction window to achieve high performance. However, as pipeline depth increases, the branch misprediction penalty becomes a critical factor in overall processor performance. Current approaches to handling branch mispredictions either incrementally roll back to in-order state by waiting until the mispredicted branch reaches the head of the reorder buffer, or utilize checkpointing at branches for faster recovery. Rolling back to in-order state stalls the pipeline for a significant number of cycles and checkpointing is costly.

This paper proposes a fast recovery mechanism, called Eager Misprediction Recovery (EMR), to reduce the branch misprediction penalty. Upon a misprediction, the processor immediately starts fetching and renaming instructions from the correct path without restoring the map table. Those instructions that access incorrect speculative values wait until the correct data are restored; however, instructions that access correct values continue executing while recovery occurs. Thus, the recovery mechanism hides the latency of long branch recovery with useful instructions.

EMR achieves a mean performance improvement very close to a recovery mechanism that supports checkpointing at each branch. In addition, EMR provides an average of 9.0% and up to 19.9% better performance than traditional sequential misprediction recovery on the SPEC2000 benchmark suite.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures

General Terms

Design, Performance

Keywords

Branch misprediction, processor state, recovery, checkpoint

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS '05, June 20-22, Boston, MA, USA.
Copyright 2005 ACM 1-59593-167-8/06/2005 ...$5.00.

1. INTRODUCTION

Accurate branch prediction is crucial to the performance of modern out-of-order processors. As the trend of utilizing deeper pipelines to obtain higher clock rates for higher performance continues, the importance of high branch prediction accuracy magnifies because of the commensurate increase in branch misprediction penalty. Previous work has shown that branch mispredictions are the single largest contributor to performance degradation in modern superscalar processors [21]. Two options exist for solving this problem: improving prediction accuracy and speeding up the misprediction recovery process. Improving branch prediction accuracy is a well studied problem. In this paper, we examine the other approach by proposing a novel fast misprediction recovery mechanism that overlaps the misprediction recovery with the execution of useful instructions, thereby improving performance.

Typically, branch misprediction recovery requires stalling the front-end of the processor, repairing the architectural state, and then restarting the process of fetching and renaming instructions from the correct path. Is stalling the front-end necessary? Can the processor continue fetching and executing instructions before repairing the architectural state when it detects a misprediction, and still maintain the correct program semantics?

To address the above questions, we describe a fast recovery mechanism, called Eager Misprediction Recovery (EMR), that allows instructions accessing correct values to continue executing while forcing instructions that reference incorrect speculative values to wait until the correct data are restored. EMR makes three main contributions:

1. EMR provides a mechanism to identify precisely which values are speculative on a per register basis. It records which registers are modified after a branch prediction is made, giving the processor fine-grain information on whether a specific register contains a correct or speculative value.

2. EMR does not stall the front-end when a misprediction occurs. The processor continues fetching instructions down the correct path seamlessly and allows instructions that access only correct values to execute without waiting for all speculative values to be repaired.

3. EMR provides a mechanism to force instructions that reference speculative values to wait until those values are repaired. Instead of restoring map table entries to allow access to the correct value, the correct value is forwarded to the appropriate physical register recorded in the map table.

Overall, EMR focuses on incrementally restoring correct values to the registers recorded in the current map table, only when neces-

sary, rather than restoring the map table so that the entries point to the physical registers where the correct values reside. This approach allows EMR to overlap the recovery process with the execution of useful instructions down the correct path, improving ILP.

The remainder of this paper is organized as follows. Section 2 discusses current branch misprediction recovery techniques and motivates EMR. Section 3 presents the detailed design of EMR. Section 4 details our experimental evaluation, including detailed cycle-accurate simulations. Section 5 discusses related work, and finally, Section 6 gives our conclusions.

2. APPROACHES TO BRANCH MISPREDICTION RECOVERY

In order to illustrate the issues in dealing with branch mispredictions in an out-of-order issue processor that performs control speculation, we follow the terminology of in-order, speculative and architectural states [12]. Implementing correct program execution despite mis-speculations requires a control-speculative processor to be aware of these states in such a way that the processor always uses the correct state for any externally visible changes in data locations.

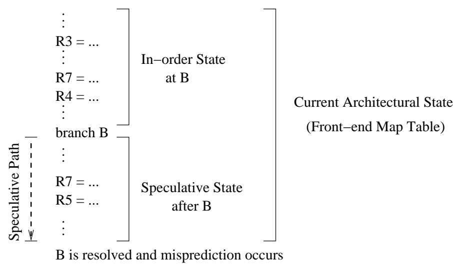


Figure 1: States upon Misprediction

Let us consider Figure 1 which illustrates the in-order, speculative and architectural states at the point of a branch misprediction. We define the in-order state as the state that would have been reached if the program were executed in program order, up to the point of interest, and the speculative state as the set of values produced that have not been committed. As should be clear, newer instructions should use values from the in-order state if the values have not been modified (i.e., are not part of the speculative state) and should use values from the speculative state otherwise. Defining the architectural state as the union of the in-order and speculative states conveniently describe the set of values to which any new speculatively fetched instruction should reference. For example, in Figure 1 the set of values produced before the mis-speculating branch are described as the *in-order state at B*, the set of speculative values produced after B is defined as the *speculative state after B*, and the architectural state at the point of misprediction is detected as the *current architectural state*. Obviously, the in-order state at B is the same state as the architectural state when the branch had been fetched.

Traditionally, there have been two main approaches to implementing the correct behavior when employing control speculation. The first approach involves mechanisms that rely on taking snapshots, or *checkpoints*, of the processor state at appropriate points and reverting back to the corresponding snapshot when a control mis-speculation is detected [10]. The second approach involves mechanisms that reconstruct the desired state by sequentially processing the in-flight instructions in program order until a deviation from the program order is reached (i.e., the mispredicting branch is encountered). Two well known mechanisms employed in the second approach are the history buffer and the future file mechanisms. We collectively refer to them as state-reconstructing mechanisms.

Obviously, a checkpointing processor does not have to wait too long upon a mis-speculation since a snapshot of the correct state is readily available; however, checkpointing limits the number of speculative in-flight branches due to the associated hardware costs. For example, the MIPS R10000 maintains a branch stack where each entry contains a complete copy of the integer and floating-point map tables [23]. At the point of recognizing a misprediction, the processor restores the front-end map table from the corresponding checkpoint. While checkpointing yields fast recovery, its cost can be prohibitive. Space to store the checkpoints limits the number of pending branches that can be in-flight. The MIPS R10000 allows only 4 pending branches to be in-flight since its branch stack has only 4 entries. Note that a CAM structured map table design will require smaller storage than a RAM structured map table for each checkpoint. In a RAM structured design, the total number of entries in the map table is equal to the number of logical registers, and each entry holds the renamed physical register designator. On the other hand, CAM-structured designs use a table in which the total number of entries is equal to the number of physical registers, and each entry keeps the logical register designator and a valid bit to indicate if it is the latest mapping [19]. A checkpoint of a CAM map table needs to copy only the valid bits, which is less costly than checkpointing in RAM map tables. For example, the Alpha 21264 [7], which uses the CAM map table, supports up to 80 checkpoints, in essence providing the capability to recover the state associated with any of the 80 in-flight instructions. Though checkpointing of CAM map tables is not too expensive, the CAM-structure itself may not scale well since higher degrees of ILP with increased issue widths require a large number of physical registers.

It is important to note that the correspondence between the number of in-flight branches and the number of checkpoint buffers is not mandatory. Although the number of checkpoints and branches in flight usually match, there has been recent work which illustrates that there may be significant benefits in incorporating a confidence mechanism and allowing more branches to be in-flight than the storage size dedicated to checkpointing [2, 14].

In contrast to checkpointing, state-reconstructing mechanisms can allow an arbitrary number of in-flight branches and, in addition, scale better due to the use of RAM-structured designs. Unfortunately, state-reconstructing mechanisms do not immediately restore the correct state, since that state is not immediately available. The state needs to be reconstructed by processing the in-flight instructions sequentially (typically through the retire logic). A commonly employed mechanism is to use a retirement map table called RMAP [9]. If a RMAP is used, when an instruction retires, it updates the retirement map table to indicate that the result register is in the in-order state. The retirement logic ensures that exceptions occur only if the operation causing the exception is the oldest, non-retired operation in the machine [9]. At this point, the retirement state is also the in-order state of this exception point. When a misprediction occurs, the processor restores the architectural state, or the front-end map table, from the retirement map table.

As can be seen, both approaches have their pros and cons. Although using a RMAP requires only one extra map table, recovery takes longer than necessary as renaming cannot start until all instructions prior to the mispredicted branch retire. If a long latency operation prior to the branch exists, *e.g.*, a cache miss, the misprediction penalty increases significantly. Akkary, *et al.*, discuss some optimizations when using a retirement map table [1, 2]. These optimizations walk through the reorder buffer to restore the map table without waiting for all prior instructions to retire. However,

the front-end stalls and instruction processing occurs sequentially, giving an $O(n)$ complexity where n is the number of instructions in-flight. Since more instructions appear in a processor with a large instruction window, this results in a significant increase in the misprediction penalty. A checkpointing mechanism on the other hand would either have to dedicate a large fraction of the chip area for checkpoint data, or limit the number of in-flight instructions, in essence limiting the amount of instruction level parallelism that can be exploited.

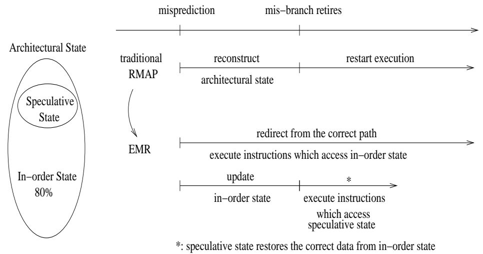


Figure 2: Overlapping Recovery with Useful Instructions

Despite the fact that these mechanisms appear to be drastically different, checkpointing and state-reconstructing mechanisms share a common property. Neither of these mechanisms allows resumption of the fetching and renaming of new instructions from the correct path until a known processor state is restored (albeit the time between the detection of the misprediction and resumption being considerably shorter in case of checkpointing). In this paper, we explore this opportunity that has not been previously considered. Our approach, EMR, *allows fetching and renaming instructions immediately upon a branch misprediction* and restores the state to the correct state *as the instruction fetching/renaming continues*. In other words, it effectively hides the state recovery latency in a RAM based map table design.

Key to the mechanism is selectively blocking the instructions which may reference values that are part of the incorrect speculative state, and allowing those instructions that do not to continue executing freely. Those blocked instructions can later be unblocked by some micro-architectural mechanism when the correct values that they need become available, allowing seamless misprediction handling in an out-of-order issue processor.

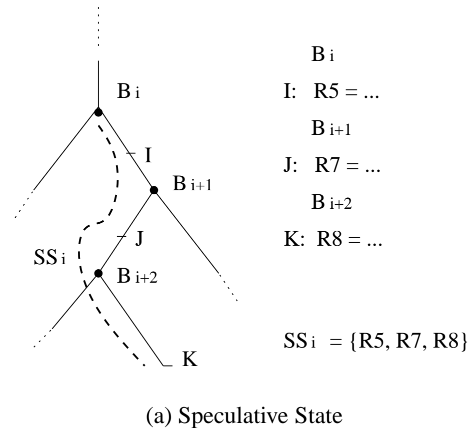
In order to better understand the performance potential of the mechanism, let us consider the processor states and the timing of the events in misprediction recovery shown in Figure 2. EMR draws from the observation that the incorrect speculative state is only a small part within the architectural state when a misprediction occurs. Our experiments show that on an average, the speculative state makes up about 20% of the architectural state upon a misprediction in the SPEC2000 benchmarks. Moreover, around 70% of the subsequent instructions from the correct path reference the in-order state whereas only 30% of the instructions need register values which have been damaged because of mis-speculation. In other words, a mechanism that employs EMR may need to block only a small percentage of the instructions from the correct path while the state reconstruction continues and can hide most of the time spent in reconstructing the correct architectural state.

3. EAGER MISPREDICTION RECOVERY

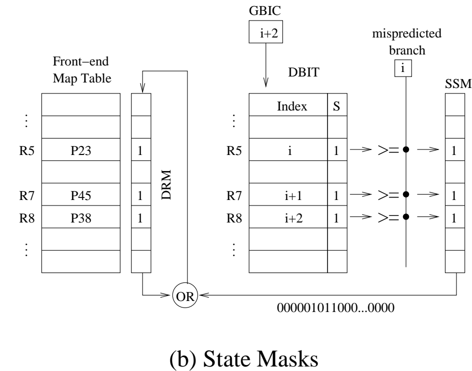
In this section, we present the design space of EMR. First, we give a technique to identify the speculative state when a misprediction occurs. Second, we discuss how to handle multiple mispredictions simultaneously. Third, we provide a mechanism to force instructions that access the speculative state to wait. Finally, we discuss how to restore the correct data to the incorrect speculative state to maintain the program's correctness.

3.1 Identifying Speculative State

The set of all registers that are defined on the speculative path comprises the speculative state. When the processor detects a branch misprediction, the speculative path consists of the set of instructions from the corresponding branch to the youngest instruction in the pipeline. Any register defined on the speculative path belongs to the speculative state of this mispredicted branch.



(a) Speculative State



(b) State Masks

Figure 3: Identifying Speculative State

To identify the speculative state, EMR maintains a Global Branch Index Counter (GBIC) for branches and a Dependent Branch Index Table (DBIT) for logical registers. The GBIC records the index of the youngest in-flight branch. When a branch is decoded, the GBIC is incremented by 1 and assigned to it. The DBIT is indexed by the logical register number, which includes two fields: one is the speculative (S) bit; the other is the branch index field. Initially, all S bits are reset. When a producer instruction is decoded, the current GBIC value is copied into the corresponding entry of its destination in the DBIT, and the S-bit is set. That indicates the destination register is speculative and it is dependent on the current youngest branch. When a producer instruction retires, if it is still the latest definition of its logical destination, the corresponding S-bit of the logical destination register is reset since it is in-order now and it does not depend on any branch. The DBIT can be accessed in par-

allel with the front-end map table and therefore it will not increase the cycle time of the decoding stage.

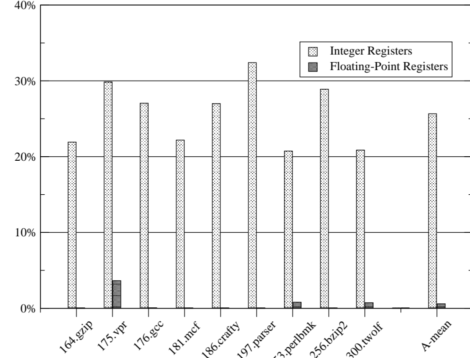
When a branch is mispredicted, in the DBIT, those registers whose S-bit is set and the index value is greater than or equal to this branch index are defined on its speculative path. They make up the speculative state for the mispredicted branch.

The index counter needs $\log_2 N$ bits if the maximum number of branches allowed to be in-flight is N. Since the counter zeroes when it overflows, an extra *color bit* is needed to handle the relative order of branches correctly. Once the counter overflows and zeroes, the *color bit* is flipped, from 0 to 1 or from 1 to 0. Each index is assigned both the counter value and the color bit. When two indices A and B are compared: A is greater than B if A's value is greater than B's and both color bits are the same. Or, A is greater than B if A's value is less than B's and their color bits are different. As a result, the GBIC and each branch index field in the DBIT need $\log_2 N + 1$ bits.

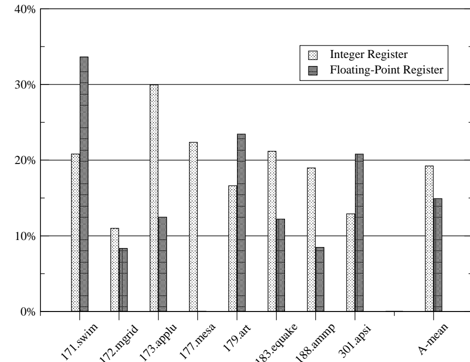
Figure 3 illustrates the speculative state identification process. The speculative state is represented by a mask of registers, called Speculative State Mask (SSM). Suppose when a misprediction occurs on the branch B_i, the branches B_{i+1} and B_{i+2}, and the producer instructions I, J and K have already been fetched and decoded speculatively, as shown in Figure 3(a). Figure 3(b) shows that the processor generates the SSM of B_i by comparing its index i with index values in the DBIT entries whose S bits are set. If a register's index value is greater than or equal to i, with respect to the circular order, then it is in the speculative state set of B_i, and the corresponding bit in the SSM is set. In this example, $R5$, $R7$ and $R8$ comprise the speculative state set of B_i. They are damaged and not available as the operands for subsequent instructions until the correct values are restored.

The speculative state represents exactly what registers need to be restored. Specifically, the recovery process only needs to recover those damaged registers contained in the SSM. If the speculative state only contains a few registers, the recovery process will be effectively hidden by the execution of useful instructions from the correct path.

Figure 4 shows that on an average the speculative state upon mispredictions for 17 SPEC2000 benchmarks accounts for around 20% of the architectural state. We obtain these results by running the benchmarks on our baseline micro-architecture model presented in Section 4.



(a) SPEC2000 INT



(b) SPEC2000 FP

Figure 4: Percentage of Speculative State upon Misprediction

3.2 Handling Mispredictions

Since EMR does not stall upon a misprediction, new mispredictions may occur before the current one is fully restored. Multiple mispredictions may be in-flight. To handle this situation, we use a global Damaged Register Mask (DRM), that is visible to new instructions, as shown in Figure 3(b). Once the speculative state is identified, we combine it with the DRM, $DRM = DRM \vee SSM$, to reflect the new global speculative state.

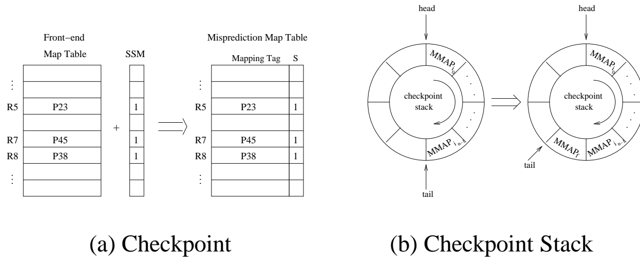
In order to rapidly restore the values to the damaged registers and correctly identify the physical registers associated with correct values, upon a misprediction EMR creates a copy of the current front-end map table and the SSM for the state recovery. Unlike the traditional checkpointing [10] and the recent work [1, 2, 14], which creates checkpoints on every branch or on some selected branches, EMR *creates checkpoints upon mispredictions*. It is called the Misprediction Map Table (MMAP).

MMAP has two fields, the Mapping Tag and the Speculative bit, as shown in Figure 5(a). The Speculative bit decides whether the corresponding logical register needs to be restored or not. The mapping tag shows the renaming register to which the correct value needs to be restored.

An N-entry circular queue of checkpoints is needed if as many as N pending mispredictions are allowed to be in-flight. Shown in Figure 5(b), the checkpoint queue is maintained as a circular buffer. The head pointer refers the first checkpoint, and the tail pointer always points the next free entry. Upon a misprediction, the MMAP of it is created and inserted into the tail of the checkpoint queue. When the first pending misprediction is recovered, the head pointer moves to the next entry towards the tail pointer and its allocated checkpoint entry is released. When the checkpoint queue is full, EMR stalls the front end until the first pending misprediction is fully recovered. We will discuss how to use the checkpointed state to handle the misprediction and restore the correct state in Section 3.4. To maintain the multiple mispredictions, we need to consider three cases when a misprediction occurs:

Case 1 No pending misprediction exists. Since the current misprediction is the only one in the pipeline, $DRM = 0$. Let B_f be the mispredicted branch. SSM is generated as described in Section 3.1, which represents the speculative state set of B_f. In this case, the DRM is set to the SSM since the set of damaged registers consists only of those on the speculative path of B_f. Thus,

$$DRM = DRM \vee SSM = SS_f \quad (1)$$



(a) Checkpoint (b) Checkpoint Stack

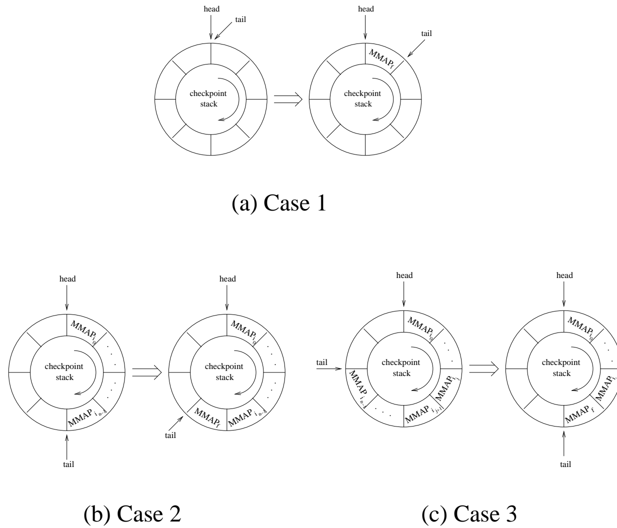
Figure 5: Checkpointing to Handle Multiple Mispredictions

The current front-end map table and the SSM are copied into MMAP_f and MMAP_f is inserted into the checkpoint queue as shown in Figure 6(a). B_f is the only misprediction in-flight.

Case 2 A misprediction occurs while the processor recovers from n earlier mispredictions. In this situation, the younger branch, B_f occurs while the processor is recovering from n previously mispredicted branches, $B_{i_0} \ldots B_{i_{n-1}}$. Here $i_0 < \ldots < i_{n-1} < f$. Then the DRM will be the union of the speculative state of B_f, SS_f in the SSM, and the speculative states of $B_{i_0} \ldots B_{i_{n-1}}$, which are $SS_{i_0} \ldots SS_{i_{n-1}}$. Since $SS_{i_0} \ldots SS_{i_{n-1}}$ have already been generated and are contained in the DRM,

$$DRM = DRM \vee SSM = SS_{i_0} \vee \ldots \vee SS_{i_{n-1}} \vee SS_f \quad (2)$$

In Figure 6(b), the copy of the current front-end map table and the SSM are copied into MMAP_f and MMAP_f is inserted into the checkpoint queue. There are $n+1$ mispredictions in flight after B_f is mispredicted.



(a) Case 1

(b) Case 2 (c) Case 3

Figure 6: Three Cases of Multiple Mispredictions

Case 3 A misprediction occurs while n mispredictions are in-flight. Assume that the processor is recovering from n mispredicted branches, $B_{i_0} \ldots B_{i_{n-1}}$. The DRM contains the union of $SS_{i_0} \ldots SS_{i_{n-1}}$. Since branches can be resolved out-of-order, it is possible the new misprediction is detected on a branch B_f, which is younger than B_{i_j} and older than $B_{i_{j+1}}$. Here $i_j < f < i_{j+1}$, and $i_0 \le i_{j+1} \le i_{n-1}$. Obviously, branches from $B_{i_{j+1}}$ to $B_{i_{n-1}}$ are on the speculative path of B_f. Any mispredictions of $B_{i_{j+1}}$ through $B_{i_{n-1}}$ are false mispredictions. The speculative state SS_f generated in the SSM contains the speculative states $SS_{i_{j+1}} \ldots SS_{i_{n-1}}$. Thus,

$$\begin{aligned} DRM &= DRM \vee SSM \\ &= SS_{i_0} \vee \ldots \vee SS_{i_j} \vee SS_{i_{j+1}} \vee \ldots \vee SS_{i_{n-1}} \vee SS_f \\ &= SS_{i_0} \vee \ldots \vee SS_{i_j} \vee SS_f \end{aligned} \quad (3)$$

The DRM now represents the new union of the speculative state of B_f and the speculative states of $B_{i_0} \ldots B_{i_j}$. Any false mispredictions that are caused by invalid branches through the speculative path of B_f are covered by the misprediction of B_f. In Figure 6(c), the MMAPs for $B_{i_{j+1}} \ldots B_{i_{n-1}}$ are flushed from the checkpoint queue. The MMAP for B_f is created and inserted into the queue.

The DRM always represents the complete set of all damaged registers of the multiple in-flight mispredictions. With the DRM, the processor can easily distinguish those instructions from the correct path that reference any incorrect state.

3.3 Handling Consumer Instructions of Speculative State

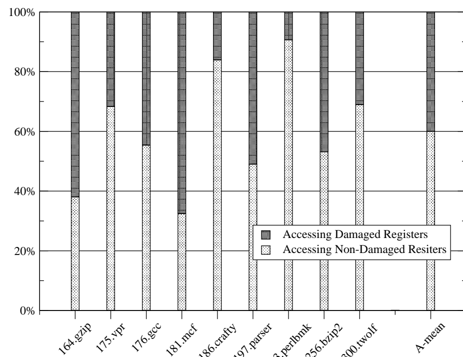
After the processor identifies the speculative state, it changes the PC to the correct target of the mispredicted branch. New instructions from the correct path are continuously fetched and renamed. Within the front-end map table, some logical registers belong to the in-order state, and others belong to the speculative state. If an instruction only accesses the in-order state, which is not damaged, it can execute normally without any problem. If an instruction references the incorrect speculative state, EMR does not allow the instruction to execute since it will access an incorrect value.

Traditionally, when the processor dispatches an instruction into the instruction window, the ready bit of an operand is set to valid if the operand has already been computed [20]. In our mechanism, if an operand belongs to the in-order state, the ready bit will be set normally, depending on whether this value is computed or not. If it belongs to the incorrect speculative state, the ready bit should be set as invalid even if the value has already been computed. Using the DRM, simple logic is enough to handle both cases: $R = \overline{D}_i \wedge V_j$, where D_i is the ith bit in the DRM, corresponding the ith logical register, R is the operand ready bit and V_j is the value ready bit of the physical register allocated to the ith logical register. If D_i is 0, this operand is not damaged and is ready if the value has already been computed. If D_i is 1, this operand is damaged and is not ready.

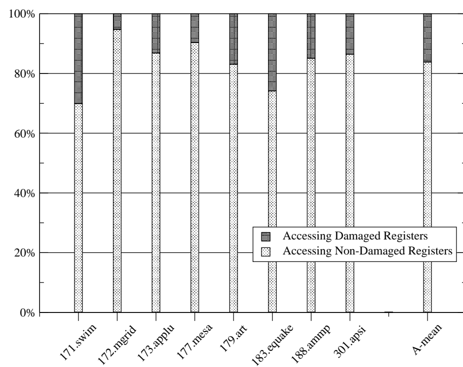
During the renaming stage, each producer instruction resets the D-bit of its logical destination in the DRM. Any subsequent instruction that needs that logical register as an operand will reference the new, undamaged state.

When the D-bit is set to 1, instructions that reference this damaged speculative state wait in the reservation stations until the correct state is restored. Instructions that access only undamaged registers proceed without waiting.

Figure 7 shows the percentage of instructions that access damaged and undamaged registers from 17 programs of the SPEC2000 benchmark suite. As can be seen, on average only 18% and 40% of all instructions reference damaged registers in CFP2000 and CINT2000, respectively. Using EMR, instructions referencing undamaged registers never wait unnecessarily because of a branch



(a) SPEC2000 INT



(b) SPEC2000 FP

Figure 7: Damaged/Non-Damaged Register Distribution

misprediction as new instructions are fetched and renamed using the current map table values without interruption.

3.4 Repairing Incorrect Speculative State

To maintain the program's correctness, EMR needs to repair the speculative state by restoring the correct data from the in-order state. Then those instructions which reference the damaged state can be executed correctly and the program semantics is maintained.

Like RMAP, EMR uses a retirement map table to construct the in-order state at the misprediction point sequentially through the retire logic. When an instruction retires, it updates the retirement map table to indicate that the result register is in the in-order state. When a mispredicted branch reaches the head of the reorder buffer, the retirement state is also the in-order state of this mispredicted branch. EMR records the speculative state in the MMAP when a misprediction happens. When the mispredicted branch causing it reaches the head of the reorder buffer, all previous mispredictions should have already been recovered. The first entry of the checkpoint queue contains the MMAP of this misprediction.

With the retirement map table, RMAP, and the MMAP popped from the checkpoint queue, EMR can restore the correct data from the in-order state to the speculative state.

Figure 8 illustrates the recovery process for the misprediction of B_f. The S-bit in the MMAP indicates whether a logical register is in the speculative state or not. For example, the S-bit of R_i is

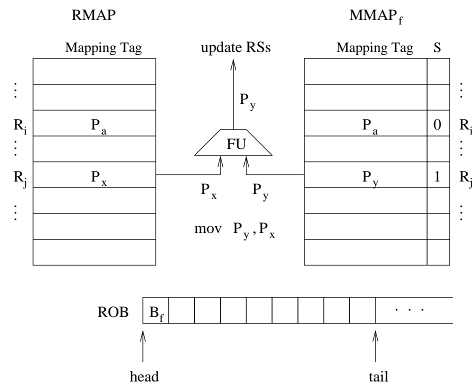


Figure 8: Restoring Speculative State

not set indicating that R_i is not damaged through the speculative path of B_f. As a result, the corresponding entries are the same both in the RMAP and the MMAP (P_a). On the contrary, the S-bit of R_j is set. That indicates it belongs to the speculative state of B_f. The correct data needs to be restored from the in-order state to the speculative state: $P_x \longrightarrow P_y$. After restoring the correct data value, EMR broadcasts the tag of P_y to the reservation stations to wake up blocked instructions that need this value. If P_y is still the latest renaming tag of R_j in the front-end map table, the corresponding D-bit of R_j in the DRM is reset. Doing so will permit subsequent instructions which may access R_j as their operand to continue normally.

After all registers in the speculative state are restored the recovery of the current misprediction is done. At this point, retirement continues normally and the first entry in the checkpoint queue is released.

Besides correctness, we need to consider the complexity of the proposed mechanism as well. The recovery of each damaged register would need one read-port and one write-port of the physical register file. Also, a tag bus would be needed to update the reservation stations. Given that in current superscalar processors the register file complexity is a significant issue, dedicating separate hardware for restoring may not be acceptable. In order not to increase both the demand for register file ports and the complexity of tag buses of the reservation stations, EMR implements recovery through the functional units. Since each functional unit has two read-ports and one write-port to the register file, and one result tag bus to the reservation stations, the functional unit can process the recovery operation for one damaged register without increasing the processor's complexity. Simply stated, EMR issues copy operations of the form *mov* P_y, P_x, into the free functional units to restore the correct data. Using move instructions in this manner achieves the desired effect seamlessly. The copy operations execute as normal instructions as they read from and write to the register file, and wake up dependent instructions blocked in the reservation stations. Moreover, the copy operations update the retirement map table and the DBIT after they restore the data just as normal producer instructions do. A uniform pipeline design can be used for normal execution and recovery operations. Each cycle, EMR can restore as many damaged registers as there are free functional units. Note that, if there are not many free functional units, this implies that the newly fetched instructions reference the undamaged state. In contrast, when there are many free functional units the newly fetched instructions reference the damaged state. In the former case we can afford to be slow in the recovery; in the latter we can quickly restore values and unblock waiting instructions rapidly.

EMR needs 1 DBIT, 1 RMAP, and *m* MMAPs if *m* pending mispredictions are allowed to be in-flight. In Section 4, our experiment shows that a small *m* is enough to extract high performance.

3.5 Optimization

So far the fundamental design space of EMR has been discussed. Although EMR allows execution of instructions which do not reference damaged values, it cannot start repairing damaged values before a known in-order state is obtained. This state is obtained by waiting until the mispredicted branch reaches the head of the reorder buffer and under normal circumstances this may not be a significant problem. However, when the head of the reorder buffer is blocked by a long latency operation such as a cache miss, the time for the mispredicted branch to reach the head of the reorder buffer may become significant. During this time, the likelihood of finding instructions which do not reference the damaged state will rapidly diminish and the processor will eventually stall.

We therefore augment our basic technique with an appropriate variation of WALK algorithms [1, 2]. Both RMAP+WALK and HISTORY+WALK are optimizations on the basic RMAP mechanism and both methods walk through the reorder buffer entries to reconstruct the in-order state without waiting for all instructions prior to the mispredicted branch to retire. This technique is orthogonal to EMR and EMR can also be improved by incorporating the WALK scheme. We refer to the combined technique as EMR+WALK. As illustrated in Figure 9, the technique requires some additional fields in the MMAP.

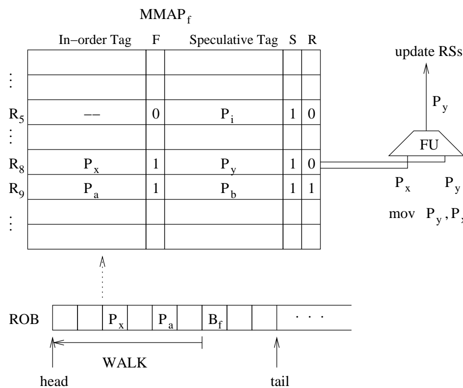


Figure 9: EMR+WALK

In order to repair the damaged speculative state in the MMAP, EMR+WALK needs to restore the correct value for each damaged register from the latest definition of the same logical destination prior to the misprediction point. EMR+WALK walks from the mispredicted branch towards the head of the reorder buffer to retrieve the latest definition information from each ROB entry. When a definition ROB entry is scanned, it is the latest definition of the speculative destination register prior to the misprediction point if the corresponding *S*-bit (Speculative) is 1 and the *F*-bit (Found) is 0 in the MMAP. If this is the case, the renaming tag of the destination register is put into the $In-order\ Tag$ field and the *F*-bit is set. After EMR+WALK walks to the head of the ROB, if there is any entry with $S=1$ and $F=0$ left, its $In-order\ Tag$ can be retrieved from the retirement map table.

Since this *WALK* process is independent from the retirement logic, restoring correct values can be started as early as possible, without waiting for all instructions prior to the mispredicted branch to retire. Any entry in the MMAP with $S=1$, $F=1$ and $R=0$ (Recovered) will trigger a move operation: $In-order\ Tag \longrightarrow Speculative\ Tag$, if the correct value is ready and there is a free functional unit. After the correct value is restored, its *R*-bit is set.

Since there can be multiple mispredictions in-flight simultaneously, multiple walk units are needed and the walk process of younger misprediction may cross older ones. All make the implementation complicated. To simplify the implementation, only one walk process is allowed for the first pending misprediction. Once a misprediction becomes the oldest one, its walk process and restoring process can start immediately.

4. EXPERIMENTAL EVALUATION

4.1 Experimental Methodology

In order to evaluate the performance of EMR, we have collected results from 17 benchmarks of the SPEC2000 benchmark suite. To limit the simulation time, all benchmarks are run to completion using the reduced reference inputs from the MinneSPEC workload [13]. The benchmarks are compiled with the gcc 3.2.3 cross compiler targeting the MIPS IV instruction set. Since we do not have the appropriate cross-compiler, Fortran 90 and C++ benchmarks in the SPEC2000 suite have been excluded. The architectural simulators used in this study are written in the ADL language [16] and automatically generated by the FAST simulation system. Simulators model the basic superscalar pipeline shown in Figure 10 and are cycle accurate.

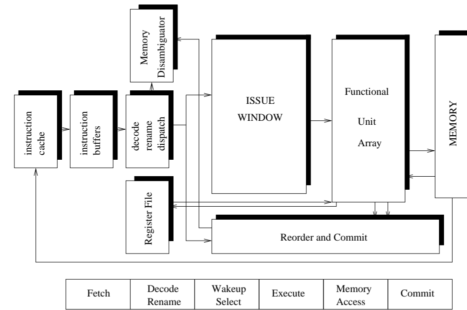


Figure 10: Machine Model

The parameters of the baseline model are shown in Table 1. Both load and store instructions are allowed to issue out-of-order [6, 17] using the store set memory dependence predictor. Five models with different misprediction recovery mechanisms are evaluated and compared:

1. *RMAP*, the traditional state-reconstructing method. A retirement map table is used to restore the state.

2. *RMAP+WALK*, the optimization of the above method. With the retirement map table, it walks from the head of ROB towards to the misprediction point to restore the state.

3. *EMR (M=i)*, our proposed fast misprediction recovery mechanism, which can handle *i* pending mispredictions.

4. *EMR+WALK (M=i)*, the optimization of EMR, which is combined with the WALK method.

5. *UL_CHK(UNLIMITED CHECKPOINTS)*, in which a checkpoint is made with every branch. It can immediately restore the correct state from the checkpoint when a misprediction is detected.

We kept the above five machines identical in all aspects except the branch misprediction recovery scheme. In two WALK models, RMAP+WALK and EMR+WALK, the walking step matches the machine's issue width, 8/cycle.

Parameter	Configuration
Issue/Fetch/Retire width	8/8/8
Instruction window size	128
Reorder buffer size	256
Register file (unique)	256
Functional units	Issue width Symmetric
Branch predictor	16K gshare
BTB	1024-entry
Return-address stack	32-entry
Dcache	L1: 32KB, 4-way, 64B/line, 2 cycles L2: 512KB, 8-way, 64B/line, 10 cycles
Memory	8B/line, 40 cycles first chunk, 4 cycles inter-chunk.

Table 1: Machine Configurations

4.2 Performance Results

The instructions per cycle (IPC) for each program in the benchmark suite using the 5 recovery models stated previously is shown in Figure 11. EMR/+WALK are implemented using $M = 4$, handling at most 4 branch mispredictions simultaneously. We will discuss the selection of different values of M in Section 4.3. As can be seen from Figure 11, EMR outperforms the traditional RMAP mechanism across all benchmarks, while EMR+WALK performs better than RMAP+WALK. Furthermore, EMR+WALK performs nearly as well as UL_CHK.

To help understand the performance results for the 5 different models, Figure 12 illustrates the percent speedup over RMAP of the other four models. As shown in Figure 12, RMAP+WALK obtains a 2.9% and 0.4% harmonic mean improvement over RMAP on CINT2000 and CFP2000, respectively. EMR achieves a 4.7% and 1.0% improvement over RMAP. Recall as discussed in Section 3.5, the restoring process of EMR can be delayed significantly due to some long latency operations, such as cache misses or floating point operations. Long latency operations cause EMR to perform worse than RMAP+WALK on several CFP2000 benchmarks and on 181.mcf, where the cache miss rate is relatively high. On the other hand, EMR+WALK utilizes the advantage from the WALK method and overcomes this shortcoming. Thus, it outperforms RMAP+WALK across all benchmarks. As shown in Figure 12(a), on the nine integer benchmarks, EMR+WALK outperforms RMAP by an average of 9.0% with a maximum improvement of 19.9% (175.vpr). The best method, UL_CHK, improves the performance by a harmonic mean of 10.2%. In other words, EMR+WALK achieves $(1+9.0\%)/(1+10.2\%) = 99\%$ of the harmonic mean performance of UL_CHK.

Although EMR+WALK obtains a lower performance improvement on CFP2000 compared to CINT2000, our technique obtains an arithmetic mean improvement of 4.8% and a harmonic mean improvement of 1.3% on the eight CFP2000 benchmarks. As shown in Figure 12(b), EMR+WALK achieves the same harmonic mean performance as UL_CHK. In most cases, floating-point programs have relatively better branch prediction accuracies using advanced branch prediction techniques. Therefore, they are less sensitive than integer programs to the misprediction recovery mechanisms.

4.3 Misprediction-under-Misprediction

This section evaluates the performance of EMR/+WALK when the number of allowed outstanding mispredictions varies. In order to achieve a good trade-off between performance and the hardware cost associated with the misprediction checkpoints, EMR implementations need to choose a reasonable value for M. Figure 13 illustrates the respective performance of different EMR implementations where the number of misprediction maps is varied from $M = 1$ to $M = 16$. We only present the harmonic mean IPC for our entire suite of SPEC2000 benchmarks and omit the details of individual benchmarks. As can be seen, both performance lines of EMR and EMR+WALK have a steep gradient from $M = 1$ to $M = 2$. After each method reaches the value $M = 2$, performance levels off. Recall that the front-end is stalled when the ith misprediction is detected in EMR/+WALK with $M = i$. When $M = 1$, EMR and EMR+WALK stall fetching new instructions until the current misprediction is recovered resulting in poor performance. As the value of M is increased EMR and EMR+WALK can better hide the latency of state recovery.

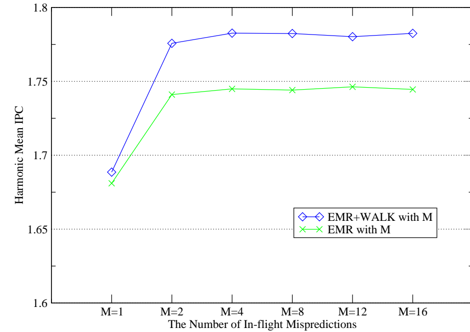


Figure 13: Performance of EMR/+WALK with Different M

With a highly accurate branch predictor, the probability of having many mispredictions in succession diminishes. Under such circumstances allowing many mispredictions to be in-flight will not provide significant performance improvements. As can be seen from Figure 13, selecting $M = 4$ provides the best trade-off between performance and hardware complexity for both recovery models. Using 4 checkpoints, EMR/+WALK can fully exploit its advantage.

Theoretical analysis verifies the experimental results. In both EMR models, checkpoints are created only upon mispredictions. If the number of in-flight branches is B, then

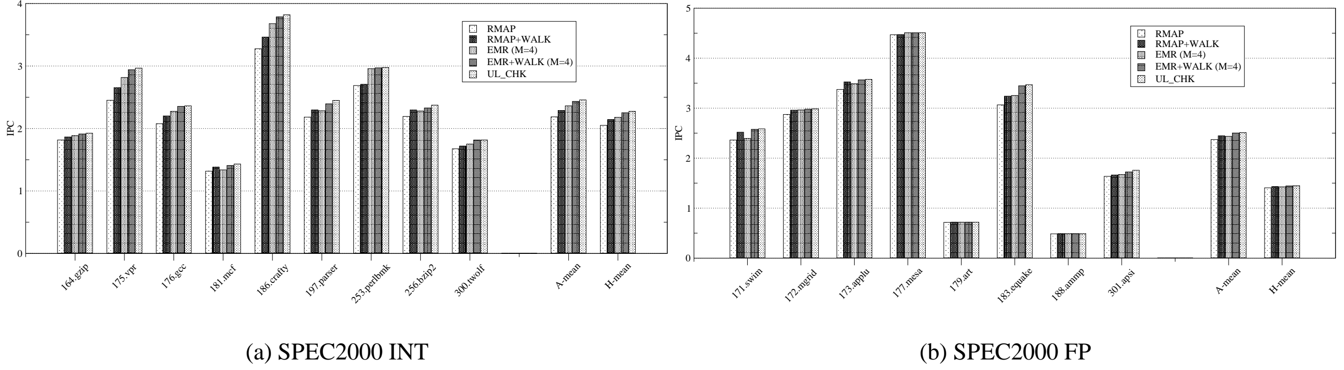
$$M = B * misprediction\ rate.$$

In our experimental models, the number of total in-flight instructions is 256. Given that on an average every 3-5 instructions a branch is encountered, B is around 50. Assume that a gshare predictor, utilized in the experiment, has less than 10% misprediction rate, then $M \approx 4$. In contrast, UL_CHK would need about 50 checkpoints as each branch would require a checkpoint. Our EMR mechanism roughly requires *misprediction rate*% of the hardware cost of UL_CHK while capturing 99% performance.

4.4 Towards a Large Instruction Window

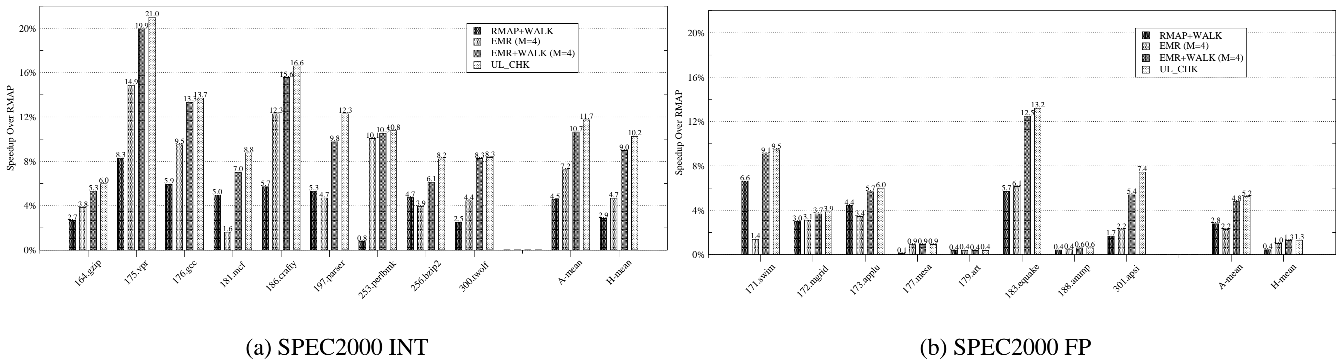
This section studies the performance variation of the five recovery methods when the instruction window size and the reorder buffer size increase. Figure 14 shows the harmonic mean IPCs when the instruction window size varies from 32 to 256. To focus the performance study on the misprediction recovery mechanism exclusively, the physical register file size is kept idealized in this group of experiments.

As shown in Figure 14, all five models obtain performance improvement due to an increased instruction window size. However,



(a) SPEC2000 INT (b) SPEC2000 FP

Figure 11: Performance of five models



(a) SPEC2000 INT (b) SPEC2000 FP

Figure 12: Speedup of RMAP+WALK, EMR/+WALK (M=4) and UL_CHK over RMAP

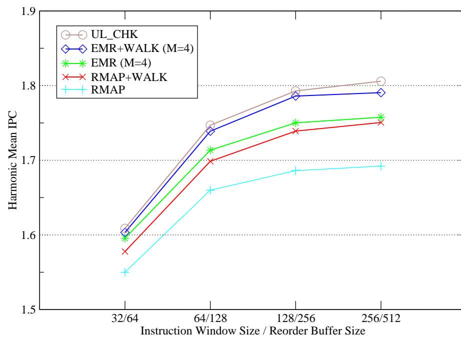


Figure 14: Performance of 5 Models with Different IW/ROB sizes

the strides of the improvement are not equal. As can be seen, the performance gap between RMAP and UL_CHK becomes larger as instruction window size increases. The performance of RMAP reduces from 96% of the performance of UL_CHK down to 93% as the instruction window size increases from 32 to 256. This phenomenon indicates that misprediction recovery is more critical for large instruction window processors. In contrast, EMR+WALK always achieves within 99% performance of UL_CHK across all window sizes. EMR+WALK is more scalable than traditional state-reconstructing recovery methods.

5. RELATED WORK

In [1, 2], Akkary *et al.* use selective checkpoints at low-confidence branches to recover from branch mispredictions. Selective checkpointing provides better scalability as the instruction window becomes larger. However, as the size of the instruction window is increased, the distance between a valid checkpoint and the current instruction pointer increases, which in-turn increases the possibility of re-executing already executed instructions since the confidence estimator cannot be perfect.

Gandhi *et al.* [8] propose Selective Branch Recovery (SBR) to reduce branch misprediction penalty by exploiting a frequently occurring type of control flow independence, called exact convergence. The results of some convergent instructions computed on the mispredicted path can be reused. Thus, the recovery penalty is reduced since those convergent instructions do not need to be fetched/renamed again. Non-convergent instructions on the mispredicted path are re-issued as move operations. Each such move operation copies the value from the previous renaming physical register of its destination to its renaming physical register. Thus the correct value of each logical destination is restored one by one through the definition chain similar to EMR state recovery.

In [3], Aragon *et al.* analyze the performance loss due to branch mispredictions. They break the misprediction penalty into three

subcategories: pipeline-fill penalty, window-fill penalty, and serialization penalty. They propose a Dual Path Instruction Processing (DPIP) to reduce the pipeline-fill penalty. In DPIP, a low-confidence branch is forked and both paths are fetched and renamed, however, the alternative path is not executed. A checkpoint of the map table is created upon the low-confidence branch to support the dual path processing. Thus, when a misprediction happens, some instructions from the correct path have already been fetched and renamed in the pipeline. DPIP can only fork once since only two active paths are allowed at the same time.

A significant body of research has provided us with increasingly better branch prediction accuracies [24, 15, 22, 5, 11]. Although the type of branch predictor is orthogonal to the EMR technique, EMR will provide diminishing returns as the accuracy of branch prediction increases. Similarly, it provides significant performance benefits as branch predictor accuracy decreases. EMR may tend to blur the differences between different branch predictors and hence may favor less accurate but faster branch predictors.

Armstrong *et al.* [4] propose to reduce performance degradation caused by branch misprediction. They propose a mechanism to leverage wrong path events (WPEs), which occur during periods of misprediction, such as a NULL pointer memory access. WPEs can be used to detect whether a branch was mispredicted before it is executed. Thus, the time for detecting misprediction is reduced. When a wrong path event occurs, misprediction recovery can be initiated early. Utilization of WPEs is orthogonal to EMR.

6. CONCLUSIONS

As pipeline depth increases, branch misprediction becomes a primary bottleneck in obtaining high performance. We have presented a fast recovery mechanism, EMR, that reduces the latency of branch mispredictions by immediately starting to process instructions from the correct target without waiting for the processor state to be restored. Our technique stores the fine-grain processor state in the checkpoint, MMAP, upon each misprediction and forwards values to blocked instructions by using free functional units, making EMR a complexity-effective approach.

EMR+WALK obtains an average performance speedup of 9.0% over the traditional RMAP on CINT2000. Moreover, it achieves 99% of the performance obtained by an unlimited checkpoint recovery method using only 4 checkpoints.

7. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments. This work is supported in part by a NSF CAREER award (CCR-0347592) to Soner Önder.

8. REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 423–434, December 2003.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. An analysis of a resource efficient checkpoint architecture. *ACM Transactions on Architecture and Code Optimization*, Volume 1:418–444, December 2004.

[3] J. L. Aragon, J. Gonzalez, A. Gonzalez, and J. E. Smith. Dual path instruction processing. In *Proceedings of the 2002 International Conference on Supercomputing*, pages 220–229, June 2002.

[4] D. N. Armstrong, H. Kim, O.Mutlu, and Y. N. Patt. Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*, pages 119–128, Portland, Oregon, 2004.

[5] I.-C. K. Chih-Chieh Lee and T. N. Mudge. The bi-mode branch predictor. In *The 30th Annual IEEE-ACM International Symposium on Microarchitecture*, pages –, December 1997.

[6] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Conference on Computer Architecture*, pages 142–153, June 1998.

[7] COMPAQ. Alpha 21264 microprocessor hardware reference manual. July 1999.

[8] A. Gandhi, H. Akkary, and S. T. Srinivasan. Reducing branch misprediction penalty via selective branch recovery. *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, pages 254–264, February 2004.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. In *Intel Technology Journal*, February 2001.

[10] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, June 1987.

[11] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, January 2001.

[12] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[13] A. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, Volume 1, June 2002.

[14] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*, pages 3–14, Istanbul, Turkey, November 2002.

[15] S. McFarling. Combining branch predictors. Technical Report WRL-TN-36, Digital Western Research Laboratory, 1993.

[16] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, May 1998.

[17] S. Önder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *32nd Annual IEEE-ACM International Symposium on Microarchitecture*, pages 170 – 176, November 1999.

[18] S. Palacharla, N. P. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 206–218, June 1997.

[19] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-96-1328, University of Wisconsin Technical Report, 1996.

[20] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures, A Design Space Approach*. ADDISON-WESLEY, 1997.

[21] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 25–34, May 2002.

[22] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor:a mechanism for reducing negative branch history interference. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 284–291, 1997.

[23] K. C. Yeager. The MIPS R10000 superscalar microprocessor. In *IEEE Micro*, pages 28–44, April 1996.

[24] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th International Conference on Computer Architecture*, pages 124–134, 1992.