July 1993

UILU-ENG-93-2229
CRHC-93-16

Center for Reliable and High-Performance Computing

APPLICATION OF COMPILER-ASSISTED MULTIPLE INSTRUCTION ROLLBACK RECOVERY TO SPECULATIVE EXECUTION

N.J. Alewine
W. K. Fuchs
W.-M. Hwu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified

1b. RESTRICTIVE MARKINGS: None

2a. SECURITY CLASSIFICATION AUTHORITY:

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE:

3. DISTRIBUTION / AVAILABILITY OF REPORT: Approved for public release; distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S): UILU-ENG-93-2229 CRHC-93-16

5. MONITORING ORGANIZATION REPORT NUMBER(S):

6a. NAME OF PERFORMING ORGANIZATION: Coordinated Science Lab, University of Illinois

6b. OFFICE SYMBOL *(If applicable)*: N/A

7a. NAME OF MONITORING ORGANIZATION: National Aeronautics and Space Administration

6c. ADDRESS *(City, State, and ZIP Code)*: 1101 W. Springfield Avenue, Urbana, IL 61801

7b. ADDRESS *(City, State, and ZIP Code)*: Moffitt Field, CA

8a. NAME OF FUNDING / SPONSORING ORGANIZATION: 7a

8b. OFFICE SYMBOL *(If applicable)*:

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER:

8c. ADDRESS *(City, State, and ZIP Code)*: 7b

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*: Application of Compiler-Assisted Multiple Instruction Rollback Recovery to Speculative Execution

12. PERSONAL AUTHOR(S): ALEWINE, N. J, W. K. Fuchs, and W.-M. Hwu

13a. TYPE OF REPORT: Technical

13b. TIME COVERED: FROM ______ TO ______

14. DATE OF REPORT *(Year, Month, Day)*: 1993 July 12

15. PAGE COUNT: 18

16. SUPPLEMENTARY NOTATION:

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*: rollback recovery, compiler-assisted multiple instruction, transient processor failures, instructional level parallelism

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Speculative execution is a method to increase instruction level parallelism which can be exploited by both super-scalar and VLIW architectures. The key to a successful general speculation strategy is a repair mechanism to handle mispredicted branches and accurate reporting of exceptions for speculated instrucitons. Multiple instruction rollback is a technique developed for recovery from transient processor failure. Many of the difficulties encountered during recovery from branch misprediction or from instruction re-execution due to exception in a speculative execution architecute are similar to those encountered during multiple instruction rollback.

This paper investigates the applicability of a recently developed compiler-assisted multiple instruciton rollback scheme to aid in speculative exectuion repair. Extensions to the ocmpiler-assisted scheme to support branch and exception repair are presented along with performance measurements across ten application programs.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION: Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL:

22b. TELEPHONE *(Include Area Code)*:

22c. OFFICE SYMBOL:

TO APPEAR: WORKSHOP ON HARDWARE AND SOFTWARE ARCHITECTURES FOR FAULT TOLERANCE: PERSPECTIVES AND TOWARDS A SYNTHESIS JUNE 14-16, 1993
LE MONT SAINT-MICHEL, FRANCE

APPLICATION OF COMPILER-ASSISTED MULTIPLE INSTRUCTION ROLLBACK RECOVERY TO SPECULATIVE EXECUTION

N. J. Alewine*, W. K. Fuchs, W.-M. Hwu

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Abstract

Speculative execution is a method to increase instruction level parallelism which can be exploited by both super-scalar and VLIW architectures. The key to a successful general speculation strategy is a repair mechanism to handle mispredicted branches and accurate reporting of exceptions for speculated instructions. Multiple instruction rollback is a technique developed for recovery from transient processor failures. Many of the difficulties encountered during recovery from branch misprediction or from instruction re-execution due to exceptions in a speculative execution architecture are similar to those encountered during multiple instruction rollback.

This paper investigates the applicability of a recently developed compiler-assisted multiple instruction rollback scheme to aid in speculative execution repair. Extensions to the compiler-assisted scheme to support branch and exception repair are presented along with performance measurements across ten application programs.

1 Introduction

Super-scalar and VLIW architectures have been shown effective in exploiting instruction level parallelism (ILP) present in a given application [1–3]. Creating additional ILP in applications has been the subject of study in recent years [4–6]. Code motion within a basic block is insufficient to unlock the full potential of super-scalar and VLIW processors with issue rates greater than two [3]. Given a trace of the most frequently executed basic blocks, limited code movement across block boundaries can create additional ILP at the expense of requiring complex compensation code to ensure program correctness [7]. Combining multiple basic blocks into *superblocks* permits code movement within the superblock without the compensation code required in standard trace scheduling [3].

General upward and downward code movement across trace entry points (joins) and general downward code motion across trace exit points (branches, or forks) is permitted without the need for special hardware support [7]. Sophisticated hardware support is required, however, for unrestricted upward code motion across a branch boundary. Such code motion is referred to as *speculative execution* and has been shown to substantially enhance performance over non-speculated architectures [8–10]. This paper focuses on the support hardware for speculative execution, which ensures correct operation in the presence of excepting speculated instructions (referred to as exception repair) and of mispredicted branches (referred to as branch repair). It is shown that data hazards which result from exception and branch repair are very similar to data hazards that result from multiple instruction rollback, and that techniques used to resolve rollback data hazards are applicable to exception and branch repair.

The remainder of the paper is organized as follows. Section 2 gives a brief overview of a compiler-assisted multiple instruction rollback (MIR) scheme to be used as a base for application to speculative execution repair (SER). Section 3 describes speculative execution and the requirements for exception repair and branch repair. Section 4 introduces a *schedule reconstruction* scheme and extends the compiler-assisted rollback scheme. Section 5 describes *read buffer* flush costs and Section 6 presents performance impacts which result

*International Business Machines Corporation, Boca Raton, Fl.

[1] This research was supported in part by the National Aeronautics and Space Administration (NASA) under grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), and in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283.

from read buffer flushes.

2 Compiler-Assisted Multiple Instruction Rollback Recovery

2.1 Hazard Classification

Within a general error model, data hazards resulting from instruction retry are of two types [11–13]. On-path hazards are those encountered when the instruction path after rollback is the same as the initial path and branch hazards are those encountered when the instruction path after rollback is different than the initial path. As shown in Figure 1, r_x represents an on-path hazard where during the initial instruction se-

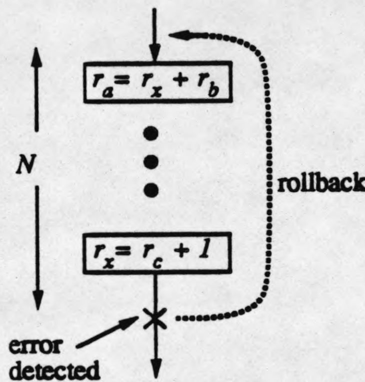


Figure 1: On-path data hazard.

quence r_x is written and after rollback is read prior to being re-written. As shown in Figure 2, r_y represents a branch hazard where the initial instruction sequence writes r_y and after rollback r_y is read prior to being re-written however this time not along the original path.

2.2 On-path Hazard Resolution Using a Read Buffer

Hardware support consisting of a read buffer of size $2N$, as shown in Figure 3, has been shown to be effective in resolving on-path hazards [11–13]. The read buffer maintains a window of register read history. If an on-path hazard is present, then prior to writing over the old value of the hazard register, a read of that value must have taken place within the last N instructions (else after rollback of $\leq N$, a read of the hazard register would not occur before a redefinition). Key to this scenario is the fact that the original path is repeated. Branch hazard resolution is left to the

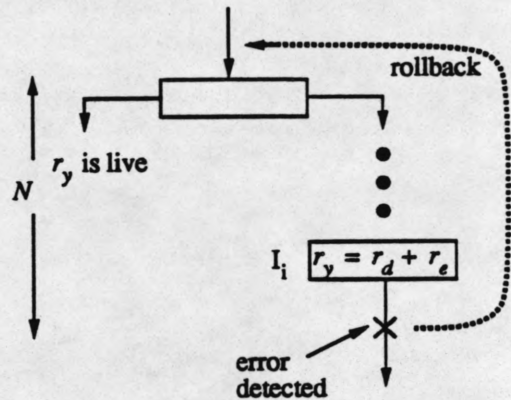


Figure 2: Branch data hazard.

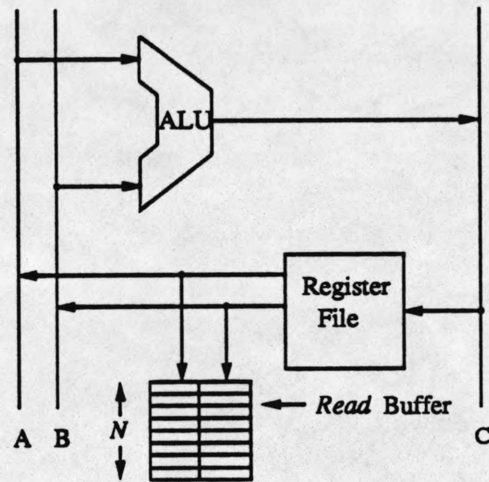


Figure 3: *Read* buffer.

compiler. At rollback, the read buffer is flushed back to the general purpose register file (GPRF), restoring the register file to a restartable state. The primary advantage of the read buffer is that it does not require an additional read port as with a history buffer, replication of the GPRF as with the future file, or bypass logic as with the reorder buffer or delayed write buffer [14, 15].

2.3 Branch Hazard Removal Compiler Transformations

Compiler transformations have been shown to be effective in resolving branch hazards [11, 12]. Branch hazard resolution occurs at three levels; 1) pseudo code, 2) machine code, and 3) post-pass. Resolution at the pseudo code level would be accomplished by renaming the pseudo register r_y of instruction I_i (Fig-

ure 2) to r_z. Node splitting, loop expansion and loop protection transformations aid in breaking pseudo register equivalence relationships so that renaming can be performed. After the pseudo registers are mapped to physical registers, some branch hazards could reappear. This is prevented at the machine code level by adding hazard constraints to live range constraints prior to register allocation. Branch hazards that remain after the first two levels can be resolved by either creating a "covering" on-path hazard or by inserting nop (no operation) instructions ahead of the hazard instruction until the rollback is guaranteed to be under the branch. Given the branch hazard of Figure 2, a covering on-path hazard is created by inserting an MOV r_y, r_y instruction immediately before the instruction in which r_y is defined. This guarantees that the old value of r_y is loaded into the read buffer and is available to restore the register file during rollback.

3 Speculative Execution

Figures 4 and 5 illustrate the two basic problems which are encountered when attempting upward code motion across a branch. As shown in Figure 4, if the

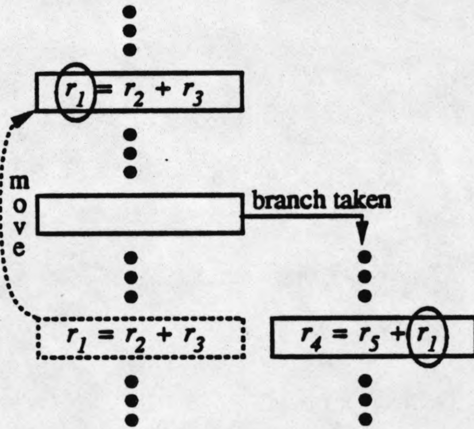


Figure 4: r_1 in live_out of taken path.

speculated instruction (i.e., an instruction moved upward past one or more branches) modifies the system state, and due to the branch outcome the speculated instruction should not have been executed, program correctness could be affected. Figure 5 illustrates that if the speculated instruction causes an exception, and again due to the branch outcome, the excepting instruction should not have been executed, program performance or even program correctness could be affected.

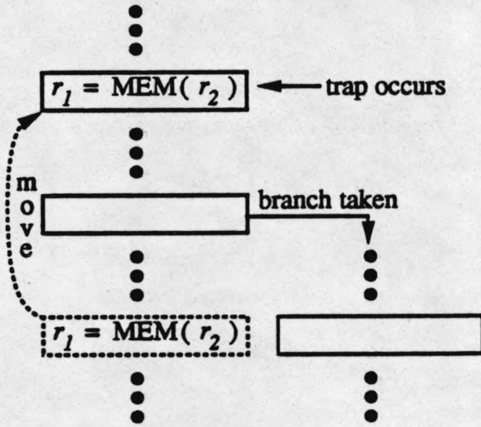


Figure 5: Speculated instruction traps.

3.1 Branch Repair

Figure 6 shows an original instruction schedule and a new schedule after speculation. Instructions d, i, and f have been speculated above branches c and g from their respective fall-through paths.[2] Speculated instructions are marked "(s)." The motivation for such a schedule might be to hide the load delay of the speculated instructions or to allow more time for the operands of the branch instructions to become available. If c commits to the taken path (i.e., it is mispredicted by the static scheduler), some changes to the system state that have resulted from the execution of d, i, and f, may have to be undone. No update is required for the PC; execution simply begins at j. If instead, c commits to the fall-through path but g commits to the taken path, then only i's changes to the system state may have to be undone.

Not all changes to the system state are equally important. If for example, d writes to register r_x and $r_x \notin live_in(j)$ (i.e., along the path starting at j, a redefinition of r_x will be encountered prior to a use of r_x [16]), then the original value of r_x does not have to be restored. Inconsistencies to the system state as a result of mispredicted branches exhibit similarities to branch hazards in multiple instruction rollback [11, 12]. Given this similarity between branch hazards due to instruction rollback and branch hazards due to speculative execution, compiler-driven dataflow manipulations, similar to those developed to eliminate branch hazards for MIR [11, 12], can be used to resolve branch hazards that result from speculation. Such compiler transformations have been proposed for

[2]For this example it is assumed that the fall-through paths are the most likely outcome of the branch decisions at c and g.

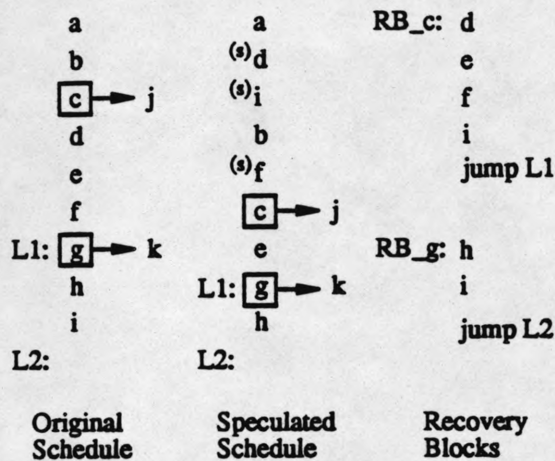


Figure 6: Branch repair.

branch misprediction handling [9]. Since re-execution of speculated instructions is not required for branch misprediction, compiler resolution of branch hazards becomes a sufficient branch repair technique.

3.2 Exception Repair

Figure 6 also demonstrates the handling of speculated trapping instructions. If *d* is a trapping instruction and an exception occurred during its execution, handling of the exception must be delayed until *c* commits so that changes to the system state are minimized, and in some cases to ensure that repair is possible in the event that *c* is mispredicted. If *c* commits to the taken path, the exception is ignored and *d* is handled like any other speculated instruction given a branch mispredict. If *c* was correctly predicted, three exception repair strategies are possible. The first is to undo the effects of only those instructions speculated above *c* (i.e., *d*, *i*, and *f*) and then branch to a recovery block *RB_c* [10] as shown in Figure 6. The address of the recovery block can be obtained by using the PC value of the excepting instruction as an index into a hash table. This strategy ensures precise interrupts [14, 17] relative to the nonspeculated schedule but not relative to the original schedule. Recovery blocks can cause significant code growth [10]. The second strategy undoes the effects of all instructions subsequent to *d* (i.e., *i*, *b*, and *f*), handles the exception, and resumes execution at instruction *i* [9]. This latter strategy provides restartable states and does not require recovery blocks. A third exception repair strategy undoes the effects of only those subsequent instructions that are speculated above *c* (i.e., only *i* and *f*), handles the exception, and resumes execution at instruction *i*, however, this time only executing speculated instructions until *c* is reached. The improved efficiency of strategy 3 over that of strategy 2 comes at the cost of slightly more complex exception repair hardware.

When a branch commits and is mispredicted, the exception repair hardware must perform three functions: 1) determine whether an exception has occurred during the execution of a speculated instruction, 2) if an exception has occurred, determine the PC value of the excepting instruction, and 3) determine which changes to the system state must be undone. Functions 1 and 2 are similar to error detection and location in multiple instruction rollback. Function 3 is similar to on-path hazard resolution in multiple instruction rollback [11, 12, 18]. On-path hazards assume that after rollback the initial instruction sequence from the faulty instruction to the instruction where the error was detected is repeated.

Figure 7 illustrates the speculation of a group of

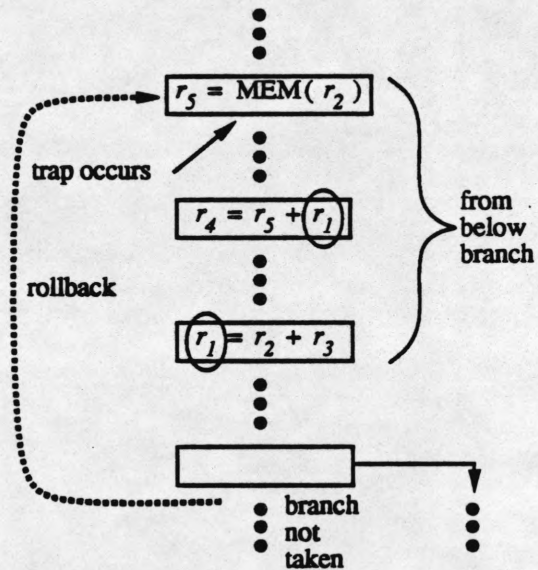


Figure 7: Exception repair.

instructions and re-execution strategy 3. The load instruction traps, but the exception is not handled until the branch instruction commits to the fall-through path. Control is then returned to the trapping instruction. This scenario is identical to multiple instruction rollback where an error occurs during the load instruction and is detected during the branch instruction. For this example, only r_1 must be restored during rollback since r_4 and r_5 will be rewritten prior to use during re-execution. Figure 7 shows that exception repair

hazards in speculative execution are the same as on-path hazards in multiple instruction rollback, and a read buffer as described in Section 2 can be used to resolve these hazards. The depth of the read buffer is the maximum distance from I_b to I_n along any backwards walk[3], where I_n is a trapping instruction that was speculated above branch instruction I_b.

3.3 Schedule Reconstruction

Assumed in Figures 6 and 7 are mechanisms to identify speculative instructions, determine the PC value of excepting speculated instructions, and determine how many branches a given instruction has been speculated above. An example of the latter case is shown in Figure 6 where instructions d, i, and f, are undone if c is mispredicted; however, only i must be undone if g is mispredicted.

If the hardware had access to the original code schedule, the design of these mechanisms would be straightforward. Unfortunately, static scheduling reorders instructions at compile-time and information as to the original code schedule is lost. To enable recovery from mispredicted branches and proper handling of speculated exceptions, some information relative to the original instruction order must be present in the compiler-emitted instructions. This will be referred to as *schedule reconstruction*.

By limiting the flexibility of the scheduler, less information about the original schedule is required. For example, if speculation is limited to one level only (i.e., above a single branch), a single bit in the opcode field is sufficient to indicate that the instruction has been moved above the next branch [8]. The hardware would then know exactly which instruction effects to undo (i.e., the ones with this bit set). Also, removing branch hazards directly with the compiler permits general speculation with no schedule reconstruction for branch repair [9].

4 Implicit Index Schedule Reconstruction

Implicit index scheduling supports general speculation of regular and trapping instructions. The scheme was inspired by the handling of stores in the sentinel scheduling scheme [9] and was designed to exploit the unique properties of the read buffer hardware design described in Section 2. Schedule reconstruction is accomplished by marking each instruction *speculated* or *nonspeculated* by including a bit in the opcode field, and using this encoding to maintain an operand history of speculated instructions in a FIFO queue called a speculation read buffer (SRB). The SRB operates similar to a read buffer with additional provisions for exception handling.

4.1 Exception Repair Using a Speculation Read Buffer

Figure 8 shows an original code schedule and two speculative schedules, along with the contents of the SRB at the time branches I_c and I_g commit. Instructions I_d and I_f have been speculated above branch instruction I_c, and I_i has been speculated above both I_g and I_c. The encoding of speculated instructions informs the hardware that the source operands are to be saved in the SRB, along with the source operand values, corresponding register addresses, and the PC of the speculated instruction.

Speculated instructions execute normally unless they trap. If a speculated instruction traps, the exception bit in the SRB which corresponds to the trapping instruction is set and program execution continues. Subsequent instructions that use the result of the trapping instruction are allowed to execute normally.

A *chk_except(k)* instruction is placed in the home block of each speculated instruction. Only one *chk_except(k)* instruction is required for a home block. As the name implies, *chk_except(k)* checks for pending exceptions. The command can simultaneously interrogate each location in the SRB by utilizing the bit field k. As shown in schedule 1 of Figure 8, *chk_except(001111)* in I_c' checks exceptions for instructions I_d and I_d. If a checked exception bit is set, the SRB is flushed in reverse order, restoring the appropriate register and PC values. Execution can then begin with the excepting instruction.

Figure 8 illustrates several on-path hazards which are resolved by the SRB. In schedule 1, if I_i traps and the branch I_c commits to the taken path, I_i has corrupted r_2 and I_f has corrupted r_7. Flushing the SRB up through I_i restores both registers to their values prior to the initial execution of I_i. Note that register r_6 is also corrupted but not restored by the SRB, since after rollback r_6 will be rewritten with a correct value before the corrupted value is used.

As an alternative to checking for exceptions in each home block, the exception could be handled when the exception bit reaches the bottom of the SRB. This is similar to the reorder buffer used in dynamic scheduling [14] and eliminates the cost of the *chk_except(k)* command, however, increases the exception handling

[3] A *walk* is a sequence of edge traversals in a graph where the edges visited can be repeated [19].

Original Schedule

I_a: $r_1 = r_2 * r_3$
I_b: $r_3 = r_4 + r_5$
I_c: bne r_1, r_3, I_j
I_d: $r_6 = r_7 * r_8$
I_e: $r_8 = r_8 + 4$
I_f: $r_7 = r_7 + 4$
I_g: bne r_8, r_7, I_k
I_k: $r_6 = r_6 + 4$
I_i: $r_2 = \mathrm{MEM}(r_2)$
⋮

Speculated Schedule 1

I_a: $r_1 = r_2 * r_3$
I_i: $r_2 = \mathrm{MEM}(r_2)$
I_d: $r_6 = r_7 * r_8$
I_b: $r_3 = r_4 + r_5$
I_f: $r_7 = r_7 + 4$
I_c: bne r_1, r_3, I_j
I'_c: chk_except(001111)
I_e: $r_8 = r_8 + 4$
I_g: bne r_8, r_7, I_k
I'_g: chk_except(110000)
I_k: $r_6 = r_6 + 4$
⋮

Speculated Schecule 2

I_a: $r_1 = r_2 * r_3$
I_d: $r_6 = r_7 * r_8$
I_i: $r_2 = \mathrm{MEM}(r_2)$
I_b: $r_3 = r_4 + r_5$
I_f: $r_7 = r_7 + 4$
I_c: bne r_1, r_3, I_j
I'_c: chk_except(110011)
I_e: $r_8 = r_8 + 4$
I_g: bne r_8, r_7, I_k
I'_g: chk_except(001100)
I_k: $r_6 = r_6 + 4$
⋮

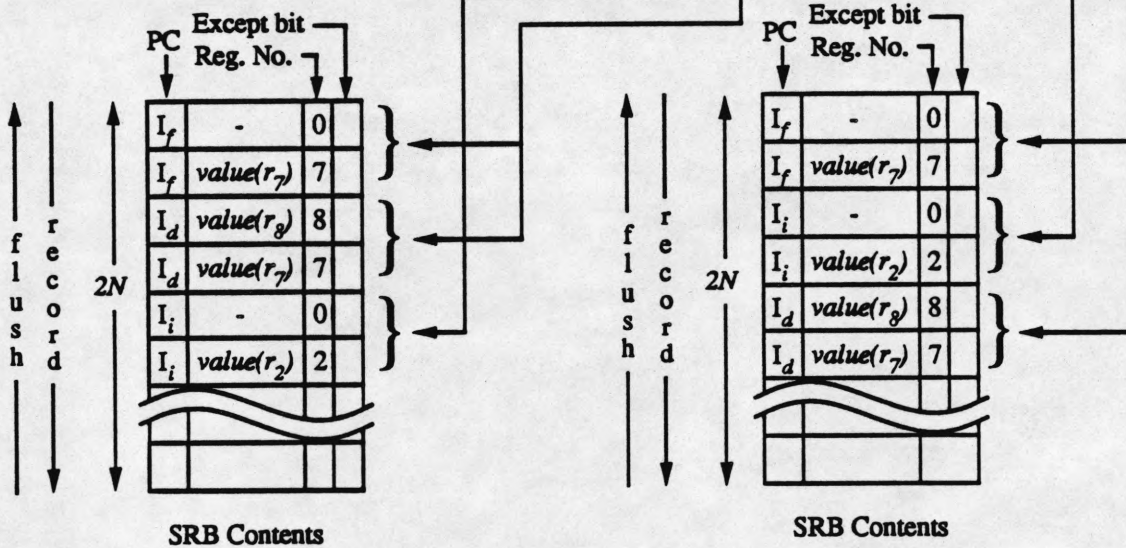


Figure 8: Exception repair using a speculation read buffer (SRB).

latency which can impact performance depending on the frequency of exceptions.

Implicit index scheduling derives its name from the ability of the compiler to locate a particular register value within the SRB. This is possible only if the dynamically occurring history of speculated instructions is deterministic at branch boundaries. Superblocks guarantee this by ensuring that the sole entry into the superblock is at the header and by limiting speculation to within the superblock. For standard blocks, bookkeeping code [7] can be used to ensure this deterministic behavior.

4.2 Branch Repair Using a Speculation Read Buffer

As described in Section 2, branch repair can be handled by resolving branch hazards with the compiler. Branch hazard resolution in multiple instruction rollback can be assisted by the read buffer when covering on-path hazards are present, reducing the performance cost of variable renaming [11, 12]. In a similar fashion, the SRB can assist in branch repair. Figure 9 shows the original code schedule and the two speculative schedules of Figure 8. For this example, it is assumed that r_2, r_3, r_6, and r_7 are elements in both $live_in(I_j)$ and $live_in(I_k)$.

As shown in schedule 1, if branch instruction I_c commits to the taken path, r_2, r_6, and r_7, which were modified in I_i, I_d, and I_f, respectively, must be restored. If instead, I_c commits to the fall-through path and I_g commits to the taken path, only r_2 must be restored. Registers r_2 and r_7 are rollback hazards that result from exception repair; therefore, the SRB contains their unmodified values. By including a *flush(k)* command at the target of I_c and I_g, the SRB can be used to restore r_2 and/or r_7 given a misprediction of I_c or I_g.

The *flush(k)* command selectively flushes the appropriate register values given a branch misprediction. For example, in schedule 2 of Figure 9, if I_c is predicted correctly and I_g is mispredicted, the SRB is flushed in reverse order up through I_i, restoring $value(r_2)$ from I_i but not restoring $value(r_7)$ from I_f. Since speculation is always from the most probable branch path, the *flush(k)* command is always placed on the most improbable branch path, minimizing the performance penalty. Not all branch hazards are resolved by the presence of on-path hazards. These remaining hazards can be resolved with compiler transformations.

5 SRB Flush Penalty

The examples of Section 4 demonstrate that compiler-assisted multiple instruction rollback can be applied to both branch repair and exception repair in a speculative execution architecture. The flush penalty of the read buffer is not a key concern in multiple instruction rollback applications since instruction faults are typically very rare. In application to exception repair in speculative execution, the SRB flush penalty is also not a major concern due to the infrequency of exceptions involving speculated instructions. However, in application to branch repair, the SRB flush penalty could produce significant performance impacts. Studies of branch behavior show a conditional branch frequency of 11% to 17% [20]. Static branch prediction methods result in branch mispredictions in the range of 5% to 15%. This results in a branch repair frequency as high as 2.5%. Assuming a CPI (clock cycles per instruction) rate of one and an average SRB flush penalty of ten cycles, the performance overhead of the flush mechanism would reach 22.5%. This indicates the importance of minimizing the amount of redundant data stored in the SRB so that the flush penalty is reduced.

Recently, a technique was proposed to reduce the amount of redundant data in a read buffer so that the read buffer size could be reduced [12, 13]. A similar technique can be used to assure that only the data required for branch and exception repair is stored in the SRB. In the implicit index scheme of Section 4, a bit indicating whether an instruction is speculated is added to the opcode field. By expanded this field to two bits, operand storage requirements can be specified. Figure 10 shows the reduced contents of the SRB given schedule 1 of Figure 9. In the modified scheme, only the first read of r_7 must be maintained. Register r_8 is not required since it was not modified. The improved scheme also eliminates blank spaces in the SRB. For this example, the misprediction of I_c in schedule 1 of Figure 9 results in four less variables to flush.

The coding of the two speculation bits would be as follows: 00) no save required, 01) save operand 1, 10) save operand 2, and 11) save both operands. If neither operand of a speculated instruction has be saved in the SRB, the instruction is not marked as speculated. This is not a problem for branch repair: however, if such an instruction traps, the hardware would have no way of knowing not to handle the exception immediately. There would also be no entry in the SRB for the exception bit or for the corresponding PC value. One solution to the problem would be to add another bit to

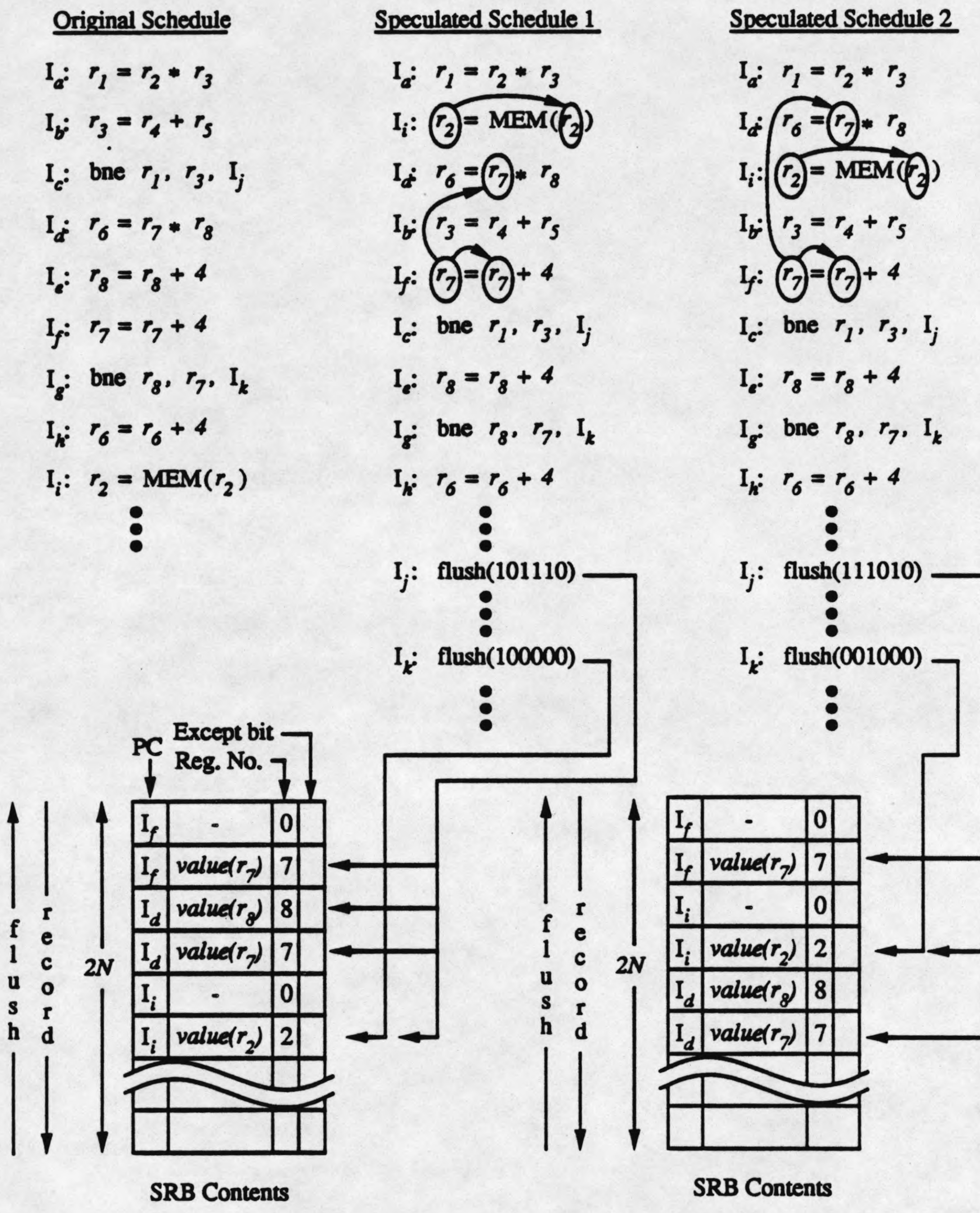


Figure 9: Branch repair using a speculation read buffer (SRB).

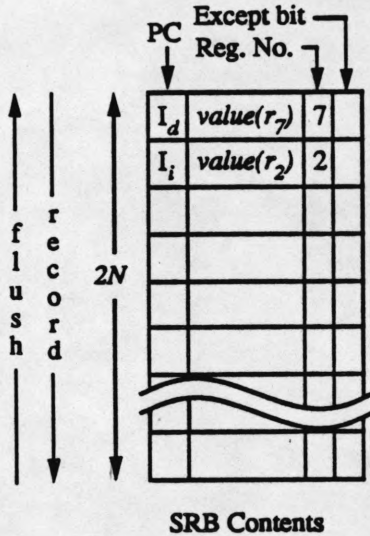


Figure 10: SRB with reduced content.

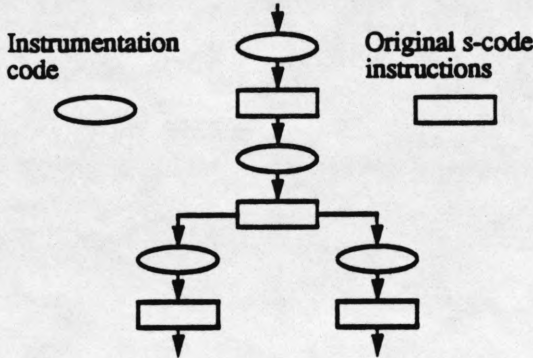


Figure 11: Instrumentation code placement.

the opcode field which marks speculated trapping instructions. A better solution is to code all speculated trapping instructions which have no operands to save as 01. This will indicate that exception handling is to be delayed and cause a reservation of an entry in the SRB, and also will slightly increase the flush penalty during branch repairs.

6 Performance Evaluation

6.1 Evaluation Methodology

In this section, results of a read buffer flush penalty evaluation are presented. The instrumentation code segments of Figure 11 call a branch error procedure which performs the following functions:

1. Update the read buffer model.
2. Force actual branch errors during program execution, allowing execution to proceed along an incorrect path for a controlled number of instructions.
3. Terminate execution along the incorrect path and restore the required system state from the simulated read buffer.
4. Measure the resulting flush cycles during the branch repair.
5. Begin execution along the correct path until the next branch is encountered.

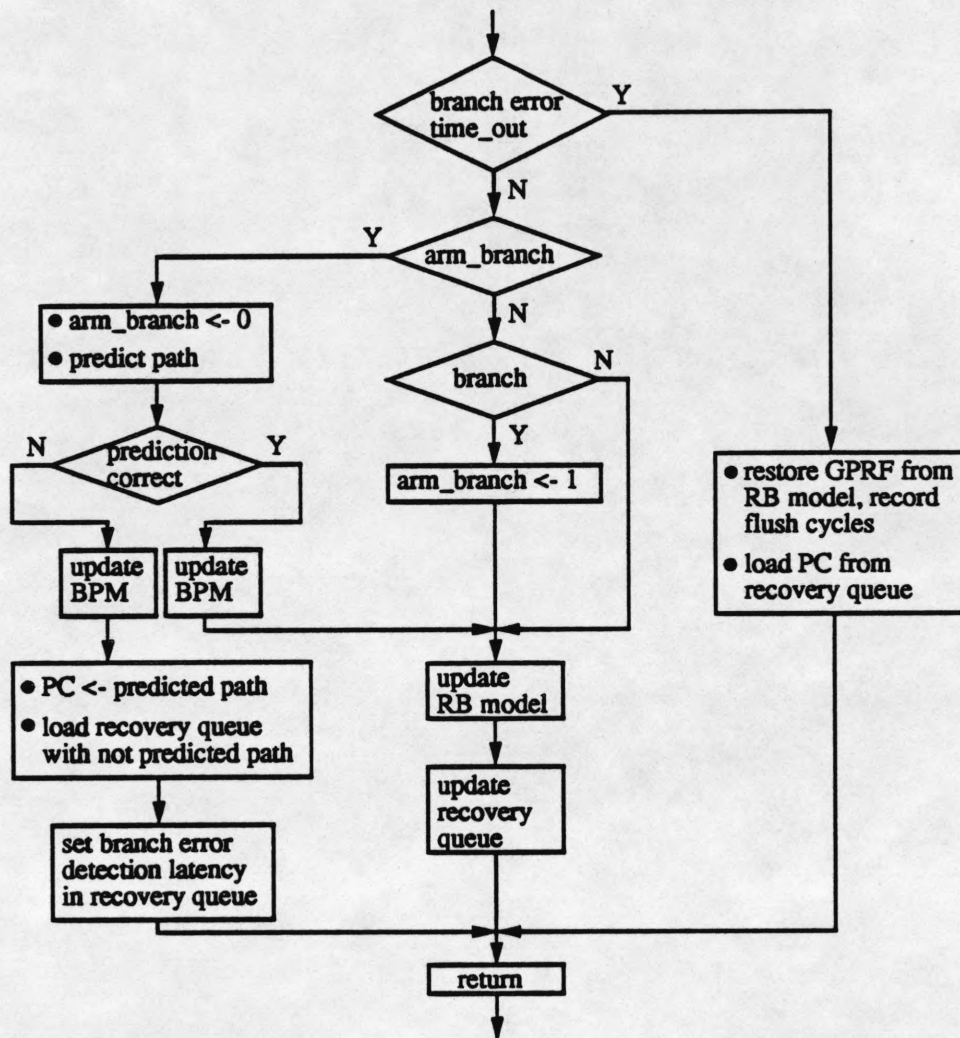
An example instrumentation code segment is shown in Figure 12. Parameters, such as operand saving information, current PC, branch fall-though PC, and branch target PC values, are passed by the instrumentation code to the branch error procedure. An additional miscellaneous parameter contains instruction type and information used for debugging.

Figure 13 gives a high level flow of operation for the branch error procedure. When a branch instruction in the original application program is encountered, an *arm_branch* flag is set. Prior to the execution of the next application instruction, the *arm_branch* flag is checked, and if set, the branch decision made by the application program is set aside. The branch is then predicted by the branch prediction model. Four models are used in the evaluation: 1) predict taken, 2) predict not taken, 3) dynamic prediction, and 4) static prediction from profiling information. The dynamic prediction model is derived from a two bit counter branch target buffer (BTB) design [21] and is the only model that requires updating with each prediction outcome.

After the branch is predicted, the prediction is checked against the actual branch path taken by the application program. If the prediction was correct, execution proceeds normally. If the prediction was incorrect, the correct branch path is loaded into the recovery queue along with a branch error detection (BED) latency, and the predicted path is loaded into the PC. The BED latency indicates how long the execution of instructions is to continue along the incorrect path. The *branch error time_out* flag is set when the BED latency is reached. When a branch error is detected, the register file state is repaired using the read buffer contents. The PC value of the correct branch path is obtained from the recovery queue. During branch error rollback recovery, the number of cycles required to flush the read buffer during branch repair is recorded.

```
$_simlb_2_24_0:
 # instruction 24
 #    Begin brsim_sim hook: s1 = 16, s2 = 0: normal
        subu    $sp,     44
        la      $at,      $_simlb_2_24_0 <---- hook address
        sw      $at,      20($sp)
        la      $at,      $_simlb_2_24_1 <---- instruction adress
        sw      $at,      24($sp)
        la      $at,      $_simlb_2_25_0 <---- next hook address
        sw      $at,      28($sp)
        li      $at,      8216 <-------------- miscellaneous
        sw      $at,      32($sp)
        li      $at,      16 <---------------- directs read buffer to save
        sw      $at,      40($sp)              register 16
        move    $at,      $sp
        j       brsim_save
 #    End brsim_sim hook.
$_simlb_2_24_1:

      addu      $16,     $16,     4  <-------- original instruction

$_simlb_2_25_0:
 # instruction 25
 #    Begin brsim_sim hook: s1 = 16, s2 = 9: branch
        subu    $sp,     44
        la      $at,      $_simlb_2_25_0 <---- hook address
        sw      $at,      20($sp)
        la      $at,      $_simlb_2_25_1 <---- instruction adress
        sw      $at,      24($sp)
        la      $at,      $_main_6 <---------- next hook address
        sw      $at,      28($sp)
        li      $at,      532505 <------------ miscellaneous
        sw      $at,      32($sp)
        la      $at,      $_main_5 <---------- target address
        sw      $at,      36($sp)
        li      $at,      304 <--------------- directs read buffer to save
        sw      $at,      40($sp)              registers 16 and 9
        move    $at,      $sp
        j       brsim_save
 #    End brsim_sim hook.
$_simlb_2_25_1:

      bne       $16,     $9,      $_main_5 <-- original instruction

$_main_6:
```

Figure 12: Instrumentation code sequences.



PC - program counter

GPRF - general purpose register file

RB - read buffer

BPM - branch prediction model

Figure 13: Branch error procedure operation.

Table 1: Application programs.

Program	Static Size	Description
QUEEN	148	eight-queen program
WC	181	UNIX utility
QSORT	252	quick sort algorithm
CMP	262	UNIX utility
GREP	907	UNIX utility
PUZZLE	932	simple game
COMPRESS	1826	UNIX utility
LEX	6856	lexical analyzer
YACC	8099	parser-generator
CCCP	8775	preprocessor for gnu C compiler

It is assumed for this evaluation that two read buffer entries can be flushed in a single cycle. This corresponds to a split-cycle-save assumption of the general purpose register file [12]. Performance overhead due to read buffer flushes (% increase) is computed as

$$Flush_OH = 100 * \frac{flush_cycles}{total_cycles}$$

All instructions are assumed to require one cycle for execution. This assumption is conservative since the MIPS processor used for the evaluation requires two cycles for a load. The additional cycles would increase the *total_cycles* and thereby reduce the observed performance overhead. In addition to accurately measuring flush costs, the evaluation verifies the operation of the read buffer and its ability to restore the appropriate system state over a wide range of applications.

The instrumentation insertion transformation operates on the s-code emitted by the MIPS code generator of the IMPACT C compiler [3]. The transformation determines which operands require saving in the read buffer and inserts calls to the *initialization*, *branch error*, and *summary* procedures. The resulting s-code modules are then compiled and run on a DECstation 3100. For the evaluation, BED latencies from 1 to 10 were used. Table 1 lists the ten application programs evaluated. *Static Size* is the number of assembly instructions emitted by the code generator, not including the library routines and other fixed overhead.

6.2 Evaluation Results

Experimental measurements of read buffer flush overhead (*Flush OH*) for various BED latencies are shown in Figures 14 through 23. The four branch prediction strategies used for the evaluation are: 1) predict taken (*P_Taken*), 2) predict not taken (*P_N_Taken*), 3) dynamic prediction based on a branch target buffer (*Dyn_Pred*), and 4) static branch prediction using profiling data (*Prof_Pred*).

Flush costs were closely related to branch prediction accuracies, i.e., the more often a branch was mispredicted, the more often flush costs were incurred. In a speculative execution architecture, branch prediction inaccuracies result in performance impacts in addition to the impacts from the branch repair scheme. Branch misprediction increases the base run time of an application by permitting speculative execution of unproductive instructions. Increased levels of speculation increase the performance impacts associated with branch prediction inaccuracies. Only the performance impacts associated read buffer flushes are shown in Figures 14 through 23.

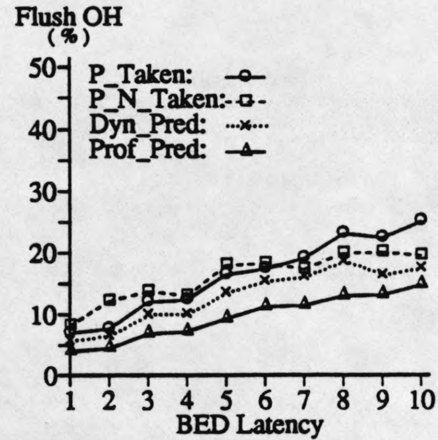


Figure 14: Flush penalty: QUEEN.

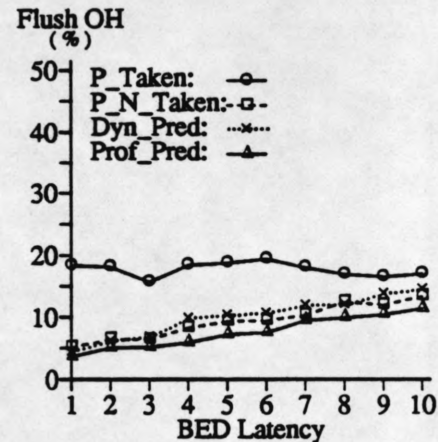


Figure 15: Flush penalty: WC.

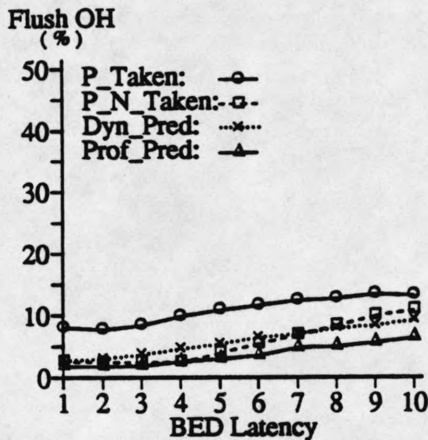


Figure 16: Flush penalty: COMPRESS.

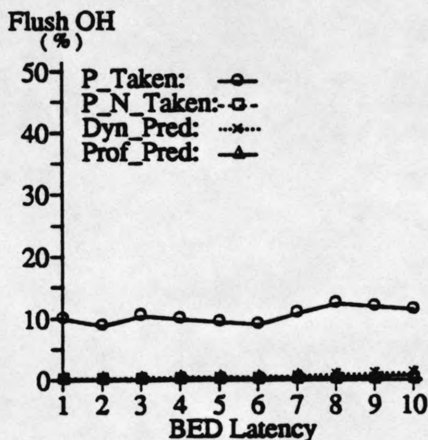


Figure 17: Flush penalty: CMP.

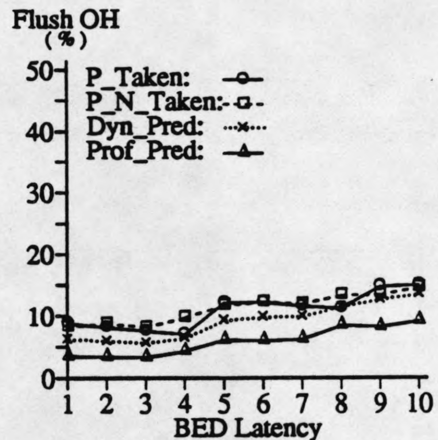


Figure 18: Flush penalty: PUZZLE.

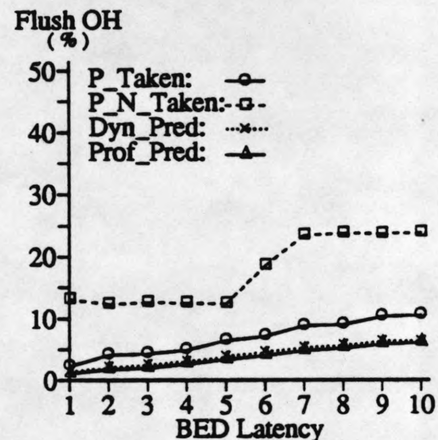


Figure 19: Flush penalty: QSORT.

For nine of the ten applications, *P_N_Taken* was significantly more accurate or marginally more accurate in predicting branch outcomes than *P_Taken*. For QSORT, *P_Taken* was significantly more accurate than *P_N_Taken*. This result demonstrates that in a speculative execution architecture, it is difficult to guarantee optimal performance across a range of applications given a choice between predict-taken and predict-not-taken branch prediction strategies.

For all but one application, *Prof_Pred* was more accurate than either *P_Taken* or *P_N_Taken*. For CMP, *Prof_Pred*, *P_N_Taken*, and *Dyn_Pred* were nearly perfect in their prediction of branch outcomes. *Prof_Pred* marginally outperformed *Dyn_Pred* in all applications except LEX.

The purpose of measuring read buffer flush costs given the recovery from injected branch errors is to establish the viability of using a read buffer design for branch repair for speculative execution. Although in such a speculative schedule only static prediction strategies would be applicable, the *Dyn_Pred* model was included to better assess how varying branch prediction strategies impact flush costs. Overall, the accuracy of *Dyn_Pred* fell between *P_Taken*/*P_N_Taken* and *Prof_Pred*.

Over the ten applications studied, read buffer flush overhead ranged from 49.91% for the P_Taken strategy in CCCP to .01% for the *P_N_Taken* strategy for CMP given a BED of ten. It can be seen from Figures 14 through 23 that a good branch prediction strategy is key to a low read buffer flush cost. The results show that given a static branch prediction strategy using profiling data, an average BED of ten produces flush costs no greater than 14.8% and an average flush cost of 8.1% across the ten applications studied. This performance overhead is comparable to the overhead

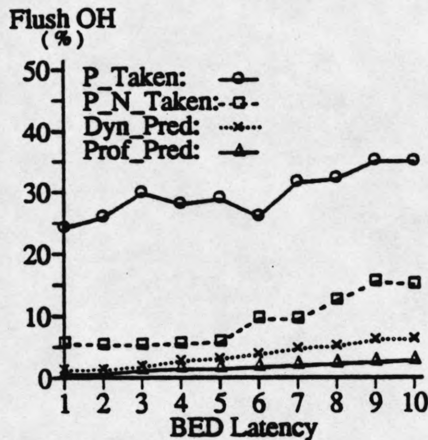


Figure 20: Flush penalty: GREP.

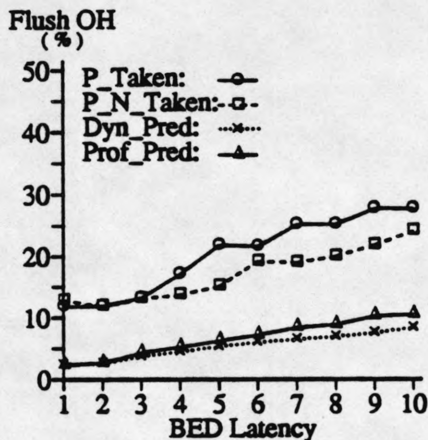


Figure 21: Flush penalty: LEX.

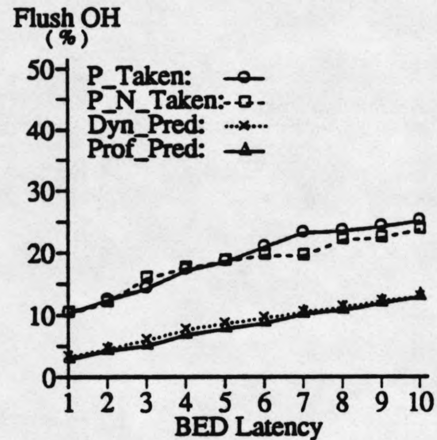


Figure 22: Flush penalty: YACC.

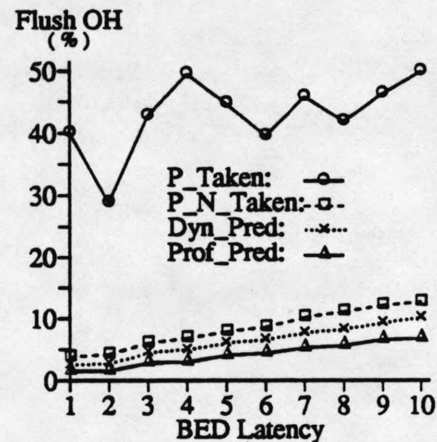


Figure 23: Flush penalty: CCCP.

expected from a delayed write buffer scheme with a maximum allowable BED of ten [15]. Given a maximum BED of ten and an average BED of less than ten, the flush costs of the read buffer would be less than that of a delayed write buffer, since a delayed write buffer is designed for a worst-case BED and the flush penalty of a read buffer is based on the average BED. The observed flush costs are small in comparison to the substantial performance gain of speculated architectures over that of nonspeculated architectures [8–10].

The BED for a given branch in this evaluation corresponds to the number of instructions moved above a branch in a speculative schedule. The results of the evaluation indicate that if the average number of instructions speculated above a given branch is ≤ 10, then the read buffer becomes a viable approach to handling branch repair.

7 Summary

Speculative execution has been shown to be an effective method to create additional instruction level parallelism in general applications. Speculating instructions above branches requires schemes to handle mispredicted branches and speculated instructions that trap.

This paper showed that branch hazards resulting from branch mispredictions in speculative execution are similar to branch hazards in multiple instruction rollback developed for processor error recovery. It was shown that compiler techniques previously developed for error recovery can be used as an effective branch repair scheme in a speculative execution architecture. It was also shown that data hazards that result in rollback due to exception repair are similar to on-path hazards suggesting a read buffer approach to exception

repair.

Implicit index scheduling was introduced to exploit the unique characteristics of rollback recovery using a read buffer approach. The read buffer design was extended to include PC values to aid in rollback from excepting speculated instructions.

Read buffer flush penalties were measured by injecting branch errors into ten target applications and measuring the flush cycles required to recover from the branch errors using a simulated read buffer. It was shown that with a static branch prediction strategy using profiling data, flush costs under 15% are achievable. The results of these evaluations indicate that compiler-assisted multiple instruction rollback is viable for branch and exception repair in a speculative execution architecture.

8 Acknowledgements

The authors wish to thank Shyh-Kwei Chen and C.-C. Jim Li for their help with the compiler aspects of this paper. We would like to thank Scott Mahlke, William Chen, and John Christopher Gyllenhaal for their excellent technical suggestions and assistance with the IMPACT C compiler. Finally, we express our thanks to Janak Patel for his contributions to this research.

References

[1] R. P. Colwell, R. P. Nix, J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," in *Proc. 2nd Int. Conf. Architecture Support Programming Languages and Operating Syst.*, pp. 105–111, Oct. 1987.

[2] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped Loop Support in the Cydra 5," in *Proc. 3rd Int. Conf. Architecture Support Programming Languages and Operating Syst.*, pp. 26–38, April 1989.

[3] P. Chang, W. Chen, N. Warter, and W.-M. W. Hwu, "IMPACT: An Architecture Framework for Multiple-Instruction-Issue Processors," in *Proc. 18th Annu. Symp. Comput. Architecture*, pp. 266–275, May 1991.

[4] B. R. Rau and C. D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," in *Proc. 20th Annu. Workshop Microprogramming Microarchitecture*, pp. 183–198, Oct. 1981.

[5] M. S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proc. ACM SIGPLAN 1988 Conf. Programming Language Design Implementation*, pp. 318–328, June 1988.

[6] A. Aiken and A. Nicolau, "Optimal Loop Parallelization," in *Proc. ACM SIGPLAN 1988 Conf. Programming Language Design Implementation*, pp. 308–317, June 1988.

[7] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Comput.*, vol. c-30, no. 7, pp. 478-490, July 1981.

[8] M. D. Smith, M. S. Lam, and M. Horowitz, "Boosting Beyond Scalar Scheduling in a Superscalar Processor," in *Proc. 17th Annu. Symp. Comput. Architecture*, pp. 344–354, May 1990.

[9] S. A. Mahlke, W. Y. Chen, W.-M. W. Hwu, B. R. Rao, and M. S. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors," in *Proc. 5th Int. Conf. Architecture Support Programming Languages and Operating Syst.*, pp. 238–247, Oct. 1992.

[10] M. D. Smith, M. A. Horowitz, and M. S. Lam, "Efficient Superscalar Performance Through Boosting," in *Proc. 5th Int. Conf. Architecture Support Programming Languages and Operating Syst.*, pp. 248–259, Oct. 1992.

[11] N. J. Alewine, S.-K. Chen, C.-C. J. Li, W. K. Fuchs, and W.-M. W. Hwu, "Branch Recovery with Compiler-Assisted Multiple Instruction Retry," in *Proc. 22th Int. Symp. Fault-Tolerant Comput.*, pp. 66–73, July 1992.

[12] N. J. Alewine, *Compiler-assisted Multiple Instruction Rollback Recovery using a Read Buffer*. PhD thesis, Tech. Rep. CRHC-93-06, University of Illinois at Urbana-Champaign, 1993.

[13] N. J. Alewine, S.-K. Chen, W. K. Fuchs, and W.-M. W. Hwu, "Compiler-assisted Multiple Instruction Rollback Recovery using a Read Buffer," Tech. Rep. CRHC-93-11, Coordinated Science Laboratory, University of Illinois, May 1993.

[14] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Trans. Comput.*, vol. 37, pp. 562-573, May 1988.

[15] Y. Tamir and M. Tremblay, "High-Performance Fault-Tolerant VLSI Systems Using Micro Rollback," *IEEE Trans. Comput.*, vol. 39, pp. 548-554, Apr. 1990.

[16] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[17] M. Johnson, *Superscalar Microprocessor Design.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

[18] C.-C. J. Li, S.-K. Chen, W. K. Fuchs, and W.-M. W. Hwu, "Compiler-Assisted Multiple Instruction Retry," Tech. Rep. CRHC-91-31, Coordinated Science Laboratory, University of Illinois, May 1991.

[19] J. A. Bondy and U. Murty, *Graph Theory with Applications.* London, England: Macmillan Press Ltd., 1979.

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

[21] J. K. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, vol. 17, no. 1, pp. 6-22, Jan. 1984.