NASA-CR-193102

April 1993

UILU-ENG-93-2215
CRHC-93-06

Center for Reliable and High-Performance Computing

NASA
N-61-CR
166346
P. 149

COMPILER-ASSISTED MULTIPLE INSTRUCTION ROLLBACK RECOVERY USING A READ BUFFER

Neal Jon Alewine

(NASA-CR-193102) COMPILER-ASSISTED MULTIPLE INSTRUCTION ROLLBACK RECOVERY USING A READ BUFFER Ph.D. Thesis (Illinois Univ. at Urbana-Champaign) 149 p

N93-27239

Unclas

G3/60 0166346

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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-93-2215 CRHC-93-06	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 &Space Admin.	
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Moffett 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)

Compiler-Assisted Multiple Instruction Rollback Recovery Using a Read Buffer

12. PERSONAL AUTHOR(S) ALEWINE, Neal Jon

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	1993 APRIL 22	

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

multiple instruction rollback, read buffer, compiler-assisted

19. ABSTRACT

Multiple instruction rollback (MIR) is a technique to provide rapid recovery from transient procesor failures and has been iplemented in hardware by researchers and slo in mainframe computers. Hardware-based MIR designs eliminate rollback data hazards by providing data redundancy implemented in hardware. Compiler-based MIR designs have also been developed which remove rollback data hazards directly with data flow manipulations, thus eliminating the need for most data redundancy hardware.

This theis focuses on compiler-assisted techniques to ahieve multiple instruction rollback recovery. We observe that data some hazards resulting from instruciton rollback can be resolved more efficiently by providing hardware redundancy while others are resolved more efficiently with compiler transformations. A compiler-assisted multiple instructionrollback scheme is developed which combines hardware-implemented data redundancy with compiler-driven hazard removal transformations. Experimental performance evaluations were conducted which indicate improved efficiency over previous hardware-based and compiler-based schemes. Various enhancements to the compiler transformations and to the data redundancy hardware developed for the compiler-assisted MIR scheme are described and evaluated. The final topic of this thesis deals with the application of compiler-asisted MIR techniques to aid in exception repair and branch repair in a speculative execution architecture.

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

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

COMPILER-ASSISTED MULTIPLE INSTRUCTION ROLLBACK RECOVERY USING A READ BUFFER

BY

NEAL JON ALEWINE

B.S., Florida Atlantic University, 1980
M.S., Florida Atlantic University, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

MAY 1993

WE HEREBY RECOMMEND THAT THE THESIS BY

NEAL JON ALEWINE

ENTITLED COMPILER-ASSISTED MULTIPLE INSTRUCTION ROLLBACK USING A READ BUFFER

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

[illegible]
Director of Thesis Research

N. Narayana Rao
Head of Department

Committee on Final Examination†

[illegible] Chairperson

[illegible]

[illegible]

[illegible]

† Required for doctor's degree but not for master's.

O-517

COMPILER-ASSISTED MULTIPLE INSTRUCTION ROLLBACK RECOVERY USING A READ BUFFER

Neal Jon Alewine, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1993
W. Kent Fuchs, Advisor

Multiple instruction rollback (MIR) is a technique to provide rapid recovery from transient processor failures and has been implemented in hardware by researchers and also in mainframe computers. Hardware-based MIR designs eliminate rollback data hazards by providing data redundancy implemented in hardware. Compiler-based MIR designs have also been developed which remove rollback data hazards directly with data flow manipulations, thus eliminating the need for most data redundancy hardware.

This thesis focuses on compiler-assisted techniques to achieve multiple instruction rollback recovery. We observe that data some hazards resulting from instruction rollback can be resolved more efficiently by providing hardware redundancy while others are resolved more efficiently with compiler transformations. A compiler-assisted multiple instruction rollback scheme is developed which combines hardware-implemented data redundancy with compiler-driven hazard removal transformations. Experimental performance evaluations were conducted which indicate improved efficiency over previous hardware-based and compiler-based schemes. Various enhancements to the compiler transformations and to the data redundancy hardware developed for the compiler-assisted MIR scheme are described and evaluated. The final topic of this thesis deals with the application of compiler-assisted MIR techniques to aid in exception repair and branch repair in a speculative execution architecture.

DEDICATION

Dedicated to Kuky, Joey, and Larry.

ACKNOWLEDGEMENTS

On the technical side, I would like to thank my thesis advisor Professor W. Kent Fuchs for his guidance and support throughout my time here at the University of Illinois. I would also like to express my appreciation to my committee members, Professor Wen-mei Hwu, Professor Janak H. Patel, Professor Prithviraj Banerjee, and Professor Chung Laung Liu, for their support and suggestions. I extend special thanks to Scott Mahlke, William Chen, Roger A. Bringmann, and John Christopher Gyllenhaal, for their excellent technical suggestions, and especially to Dr. Chung-Chi Jim Li and Shyh-Kwei Chen, for their help with the compiler aspects of this thesis.

On the support and human understanding side, I would like to thank my friends from IBM, Dick Smith, Sue Parliament, Bill Burger, John Klein, and particularly Mike Kelly, for their unfailing interest in my well-being. I would also like to thank my new friends, in addition to those already mentioned, Jonathan Simonson, Grant Edward Haab, John G. Holm, Yi-Min Wang, Paul Chen, Antoine Mourad, Bob Janssens, Professor Saab, and Vicki McDaniel, to name only a few.

Finally, I would like to thank my wife, Kuky, and my sons, Joey and Larry, for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Motivation

Instruction retry is an effective technique to allow rapid recovery from transient faults in a processing system. Multiple instruction rollback recovery may be appropriate when error detection latencies or when error reporting latencies are greater than a single instruction cycle. Single and multiple instruction rollback recovery has been implemented in hardware by researchers and main-frame computer designers. In general, complex implementations strive to minimize impacts to system performance while less complex implementations permit some performance impact. Compiler-assisted multiple instruction rollback has also been developed which replaces dedicated data redundancy hardware with compiler transformations that remove rollback data hazards. This thesis introduces a multiple instruction rollback recovery scheme, which uses the compiler to remove some rollback data hazards and uses dedicated data redundancy hardware to remove all remaining hazards. The scheme results in less hardware than required for the least complex hardware implementations and performance approaching that of the most complex hardware implementations.

1.2 Thesis Contributions

The contributions of this thesis are grouped into four topics. The first extends previous compiler-based multiple instruction rollback to a broad class of code execution failures. Data hazards that result from instruction rollback are classified and shown to be of two types: 1) on-path hazards, and 2) branch hazards. Previous compiler-driven data-flow manipulations resolve on-path hazards only. These transformations are extended to resolve both on-path and branch hazards. Evaluations of the extended compiler-based scheme indicate slightly increased performance impacts over transformations that resolve only on-path hazards. Using the hazard classification, a new compiler-assisted scheme is proposed which utilizes a hardware implemented read buffer to remove on-path hazards and compiler transformations to remove branch hazards. Performance evaluations of the new compiler-assisted multiple instruction rollback scheme indicate a lower performance penalty than with either a compiler-based approach or a comparable hardware-based approach.

The second develops enhancements to previous compiler transformations used for rollback hazard removal. A one-pass node splitting algorithm is developed which uses the concept of conflicting parents and graph coloring to eliminate constraints that forced previous node splitting algorithms to operate iteratively. Experimental evaluations show that the one-pass node splitting algorithm reduces code growth and achieves a compile-time speedup of 30 over iterative algorithms. To complement the one-pass node splitting algorithm, a one-pass static loop protection algorithm was also developed along with a dynamic loop protection algorithm incorporated into the one-pass node splitting algorithm. The use of profile data to aid in loop protection decisions was evaluated and found to be effective for some cases in improving application run-time performance.

The third studies the read buffer size requirement. A flexible evaluation methodology is developed and used to study ten read buffer configurations. The evaluation methodology used updates a read buffer model at dynamically occurring application instructions boundaries. It was found that a 55% read buffer size reduction is achievable with an average reduction of 39.5% over the ten applications evaluated, given the most efficient read buffer configuration, but that additional control logic to handle read buffer overflows may limit the overall hardware savings.

The final topic of the thesis studies the application of compiler-assisted multiple instruction rollback to aid in speculative execution repair. It is shown that the handling of speculated excepting instructions and the handling of mispredicted branches in a speculative execution architecture are similar to the handling of on-path and branch hazards, respectively, in multiple instruction rollback. A speculative scheduling method referred to as implicit index scheduling is proposed which utilizes a modified read buffer to remove on-path hazards and compiler transformations to remove branch hazards. The viability of the read buffer to aid in branch repair is also investigated and shown to be contingent on the expected read buffer flush penalty. Estimates of flush penalties are obtained using the same evaluation methodology developed for the read buffer size study. Evaluation results indicate that read buffer flush costs under 15% are achievable.

1.3 Thesis Organization

Chapter 2 presents a background of error detection strategies and rollback recovery schemes. Chapter 3 describes the error model and classifies data hazards that result from instruction rollback. Compiler transformations that remove rollback hazards are presented, along with resulting performance evaluations of a compiler-only multiple instruction rollback scheme. Chapter 3 also presents a new compiler-assisted multiple instruction rollback recovery scheme along with experimental evaluations of the new scheme. Chapter 4 describes enhancements to compiler transformations used for rollback hazard removal. These enhancements focus on reducing compile times and improving run-time performance. Chapter 5 assesses the minimum size requirements of the read buffer proposed in Chapter 3 and gives performance evaluations for ten read buffer configurations. Chapter 6 investigates the viability of applying the new compiler-assisted multiple instruction rollback scheme to aid in speculative execution repair. Chapter 7 contains summary remarks, limitations, and future research directions.

2. BACKGROUND

2.1 Error Detection

Various error detection strategies have been studied, resulting in a range of efficiencies in both implementation complexities and error detection latencies. Error detection schemes that utilize error correcting codes [1] or low level functional redundancy [2] typically detect errors within the same cycle, however, can result in increased cycle times. Some time redundant [3], algorithm-based [4], high level functional redundancy [5], and control-flow checking [6, 7] error detection schemes achieve error detection latencies of a few cycles without impacting system cycle times. In designs for which error detection latencies are greater than a single instruction cycle, multiple instruction rollback recovery may be appropriate.

2.2 Rollback Recovery

2.2.1 System-level checkpointing and recovery

System-level checkpointing is a well-understood method for implementing rollback recovery when system errors occur [8–10]. In the case of a detected fault, the system is rolled back to a previous checkpoint containing a *consistent state* of the system [11]. System-level checkpointing is implemented in software (typically included in the operating system) with the checkpointed system state being stored on stable media such as the system disk. To minimize overall system performance impacts given the significant overhead associated with taking a checkpoint, checkpoint intervals must be great (from from minutes to hours). This strategy permits long error detection latencies, however, has the disadvantage of long recovery times and significant lost work during repair.

2.2.2 Multiple instruction rollback

When transient processor errors occur, multiple instruction rollback (also referred to as multiple instruction retry or simply instruction retry) can be an effective alternative to system-level checkpointing and rollback recovery [12, 13]. Multiple instruction retry within a sliding window of a few instructions [12], or re-execution of a few cycles [5], can be implemented in parallel with concurrent, algorithm-based, or control-flow error detection methods for rapid recovery from transient processor errors. Rapid error detection ensures minimal system state changes between detection and rollback and allows hardware to efficiently save and restore the required system state. Multiple instruction rollback recovery is feasible only when error detection latencies are sufficiently small.

The issues associated with instruction retry are similar to the issues encountered with exception handling in an out-of-order instruction execution architecture. If an instruction is to write to a register and N is the maximum error detection latency (or exception latency), two copies of the data must be maintained for N cycles. Hardware schemes such as reorder buffers, history buffers, future files [14], and micro-rollback [12] differ in where the updated and old values reside, circuit complexity, CPU cycle times, and rollback efficiency.

2.3 Hardware Implemented Instruction Retry

Multiple instruction retry and system level checkpointing are similar concepts differing in implementation (i.e., hardware versus software) and scope (i.e., the error detection latency and amount of system state involved). Similar to system level checkpointing, hardware implemented instruction retry schemes belong to one of two groups: 1) full checkpointing and 2) incremental checkpointing. Full checkpointing maintains "snapshots" of the required system state space at regular, or predetermined, intervals. Upon error detection, the system can be rolled back to the appropriate checkpointed system state. Incremental checkpointing maintains changes to the system state in a "sliding window". Upon error detection the system state is restored by undoing, or "backing-out" the system state changes up to the instruction in which the error occurred.

Several examples from each instruction retry group will be discussed. The examples are drawn from research, commercial processors, and patent applications. These examples, along with a discussion of compiler-assisted instruction retry, will serve as a background and comparison for the proposal of a new multiple instruction rollback approach presented in Chapter 3.

2.3.1 The IBM 4341

The IBM 4341 supports the capability for single instruction retry by making use of a level sensitive scan design (LSSD), which was originally proposed to provide increased observability and controllability in LSI circuits and also to make sequential logic operations independent of circuit delays and wire delays [15]. Figure 2.1 illustrates both of these features.[1]

During normal operation CLK_a and CLK_b operate as interleaved, nonoverlapping system clocks. CLK_a latches state changes and provides stable inputs to the second stage of the latch pair. CLK_b modifies the system state using these stable values. LSSD ensures that the steady-state output is independent of the sequence of input signals or signal rise/fall times. Scanning is accomplished by substituting CLK_s for CLK_a in the normal operation. In this way, a complete system state image can be loaded serially through the $SCAN\_in$ line. Likewise, the current system state image can be obtained serially through the $SCAN\_out$ line.

[1]Derived from pp. 438, *"Logic Design Principles"*, E. J. McCluskey [16].

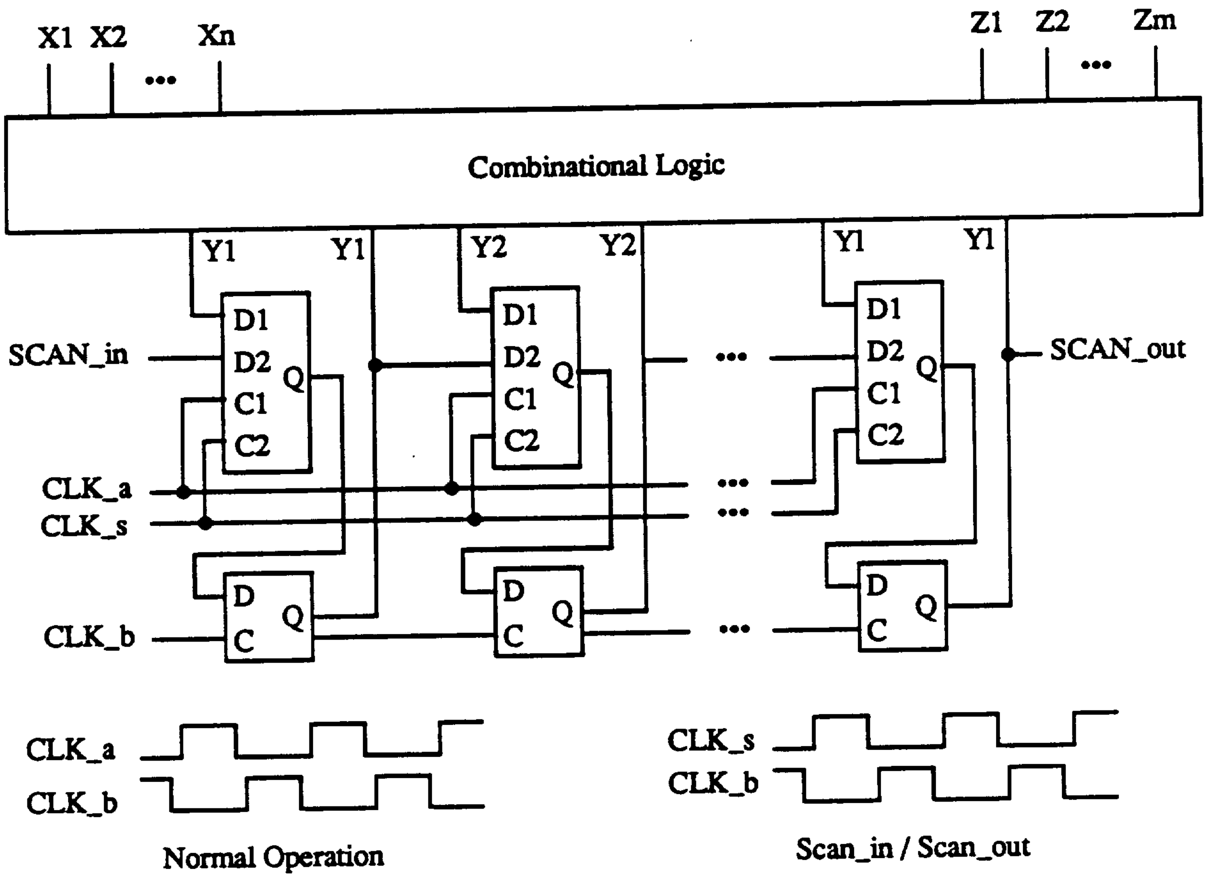


Figure 2.1: LSSD double latch design.

The 4341 incorporates three error detection strategies; 1) duplication and compare, 2) odd-parity, and 3) special error conditions such as invalid combination of control lines. When one of these mechanisms detects an error, the system clocks are "frozen." The current system state is then scanned out by a separate service processor which makes the proper adjustments to the system state. The updated state is then scanned back into the processor where the faulty instruction is retried. A more complete description of fault handling on the IBM 4341 can be found in [17]. The 4341 is best classified as a full checkpointed rollback recovery scheme, with a checkpoint interval of one instruction (via LSSD state duplication) and a rollback distance of one.

2.3.2 The IBM 3081

The instruction unit of the IBM 3081 establishes a checkpoint every 10 to 20 instructions. This checkpoint is then used as a re-entry point for rollback repair in the event that an error is detected. The local cache array includes hardware which saves the old value of an updated cache line into a push-down array. Flushing the push-down array in reverse order restores the cache array to its state at the time of the last checkpoint. This method is similar to the *history buffer* which is described in Section 2.3.8 [14].

The 3081 uses LSSD to read and write system state information. Scanning out the entire system state, including the general purpose register file (GPRF) and local cache arrays, can be very time-consuming. The hardware support previously mentioned obviates full scanning. The IBM 3081 implements full checkpointed rollback recovery with a checkpoint interval of 10 to 20 instructions and corresponding rollback distance. A more complete description of fault handling on the IBM 3081 can be found in [13, 18].

2.3.3 The VAX 8600

The VAX 8600 uses parity checking on internal buses including the arithmetic logic unit (ALU) and shifter. The instruction fetch and decode unit (I box), floating point unit (F box), and execution unit (E box) each have a copy of the GPRF. Writes to one GPRF cause simultaneous writes to the other two GPRF's in order to maintain consistency. If a parity error is detected in one of the GPRF's, the other GPRF's can be used to correct the invalid GPRF.

For performance reasons, parity checking and correction in the local cache (M box) occur after the data has been sent to the requester. If an error is detected by the M box, it is corrected with Error-Correcting Code (ECC) logic and an error signal is sent to the E box so that the instruction can be retried. The VAX 8600 is a pipelined complex instruction-set architecture which does not guarantee that all updates to the system state are held until the write state of the pipeline. If the CPU has not performed an operation that makes retry impossible, the instruction is retried. The ability to retry is determined by the abort bit which is set when the current instruction: 1) is an I/O read, 2) is a memory write, or 3) results in a modification of the system state by the E box. The VAX 8600 can be classified as a full checkpointed rollback recovery scheme, with a checkpoint interval of one instruction and a rollback distance of one. A more complete description of fault handling on the VAX 8600 can be found in [19].

2.3.4 The VAX 9000

The VAX 9000 system is organized such that all storage elements, i.e., latches and flip flops, are connected to form a "visibility chain." The system state is therefore visible and can be read and written by the service processor through a serial diagnostic bus. The visibility chain operates much like a scan ring design [15, 16]. The VAX 9000 is a pipelined complex instruction-set architecture. The user visible system state (i.e., memory contents and register values) is well-defined at macroinstructions boundaries; however, during actual operation, several instructions will be executing at once, with each instruction at a different stage in the pipeline. This make identification of macroinstructions boundaries difficult. Also, errors cannot be expected to occur at well-defined instruction boundaries.

The VAX 9000 execution unit (E-box) accepts operands, computes the result, and delivers the result for storage. An error interfering with one or more of these operations causes a trap when the E-box requests data from the faulty subsystem. If the program visible state of the machine has not been modified by the current instruction, the instruction is backed up to the beginning and restarted. This is usually possible since the system state typically changes in the final stages of the pipeline. If memory or register values were modified early in the pipeline, status flags are updated so that instruction retry can be disabled.

The VAX 9000 can best be classified as a full checkpointed rollback recovery scheme, with a checkpoint interval of one instruction and a rollback distance of one. A more complete description of fault handling on the VAX 9000 can be found in [20].

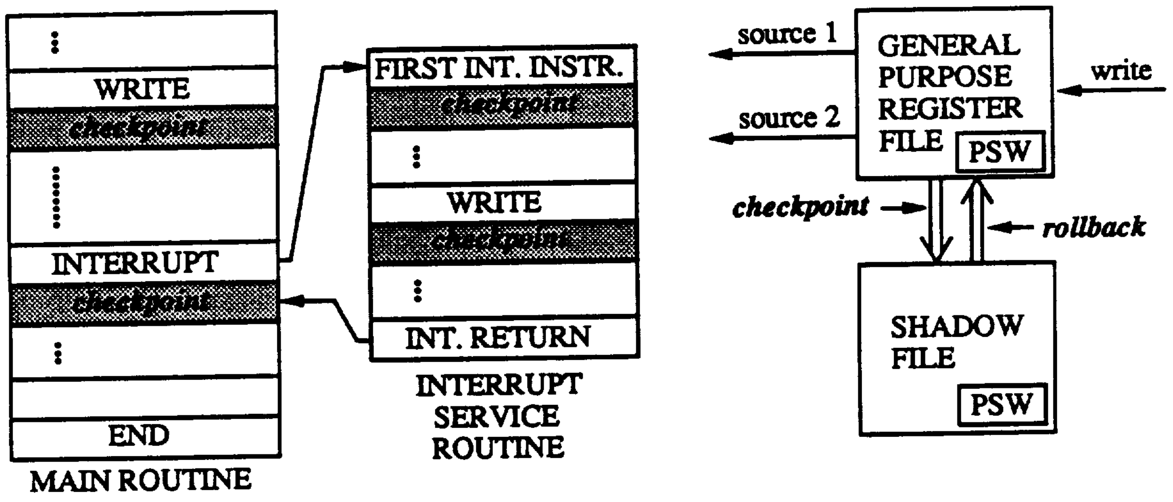


Figure 2.2: Checkpoint retry mechanism: U.S. patent number 4,912,707.

2.3.5 IBM Patent number 4,912,707

The checkpoint retry mechanism described in U.S. patent number 4,912,707 deals primarily with checkpoint placement in a hardware implemented multiple instruction rollback scheme. Many instruction retry schemes have a rollback distance of one instruction making checkpoint placement trivial. Nontrivial checkpoint placements are handled by software in system level checkpointing but are more difficult in hardware implementations without precise placement rules.

As shown in Figure 2.2, this mechanism automatically takes checkpoints at one of three points in a dynamic instruction stream: 1) immediately after a *write* instruction, 2) immediately after a return from interrupt, or 3) immediately after the first instruction in an interrupt handler routine. This approach avoids the need to take a checkpoint after every instruction and ensures that a rollback will never cross a write or interrupt boundary.

The checkpoint is taken by storing a copy of the GPRF and program status word (PSW) into a shadow file, as shown in Figure 2.2. For rollback, the GPRF is restored from the shadow file. This checkpoint retry mechanism implements full checkpointed rollback recovery with a variable checkpoint interval and corresponding rollback distance. A more complete description of the mechanism can be found in U.S. patent number 4,912,707 [21].

2.3.6 IBM Patent number 4,044,337

The instruction retry mechanism described in U.S. patent 4,044,337 resolves the problem of register file and cache memory corruption due to a defective instruction cycle.[2] The mechanism maintains duplicate copies of such data to be used for restoration in the event of corruption.

For each register in the GPRF, two duplicate locations are maintained, as shown in Figure 2.3. Each time register r_x is written in the GPRF by instruction I_y, one of these two duplicate registers, r'_x or r''_x, is also written with the same value. The duplicate locations used, alternate with instruction count, that is, I_y writes to r'_x, I_{y+1} writes to r''_x, I_{y+2} writes to r'_x, etc. If I_y writes r_x more than once, each subsequent write of I_y overwrites the previous value in the duplicate location leaving only the last written value of r_x in the location upon completion of I_y. This scheme ensures that, given an error in I_{y+1}, the GPRF contents which existed prior to the execution of I_{y+1} can be recovered from the duplicate register locations.

[2]U.S. patent 4,044,337 refers to register file storage as local store and cache storage as buffer store.

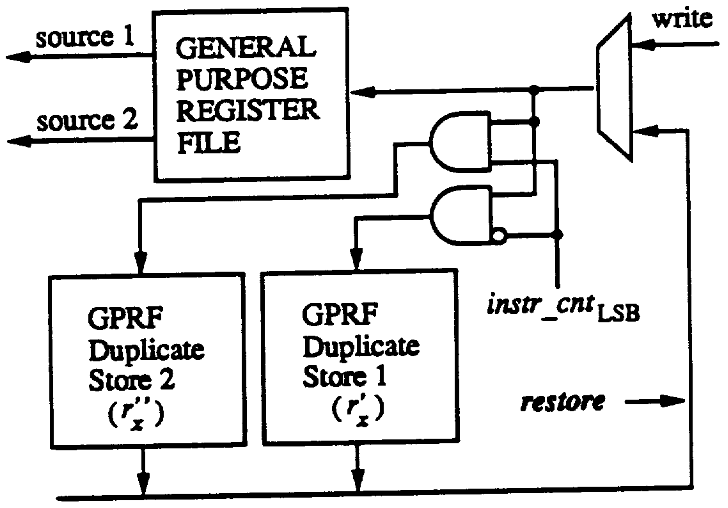

source 1
source 2
GENERAL PURPOSE REGISTER FILE
write
GPRF Duplicate Store 2 (r''_x)
GPRF Duplicate Store 1 (r'_x)
$instr\_cnt_{LSB}$
restore

Figure 2.3: GPRF duplicate store mechanism: U.S. patent number 4,044,337.

A mechanism similar to the one used for GPRF duplication is used for the cache duplication. Since triplication of the cache results in a significant circuit overhead, the total cache duplicate store size is significantly smaller than the cache size. As shown in Figure 2.4, a store controller accepts the cache address and maps it into the smaller duplicate stores. The duplicate stores operate the same as the register file duplicate stores, except that in addition to the cache data, cache addresses are also saved. If a cache store occurs when the duplicate stores are full, a replacement algorithm in the store controller submits a request for the cache to write out an appropriate cache line to main memory. This makes a corresponding duplicate store pair available for the current cache write.

The checkpoint retry mechanism, described in U.S. patent number 4,044,337, implements incremental checkpointing and rollback recovery, where changes to the system state are maintained and backed out during recovery. The mechanism supports a maximum

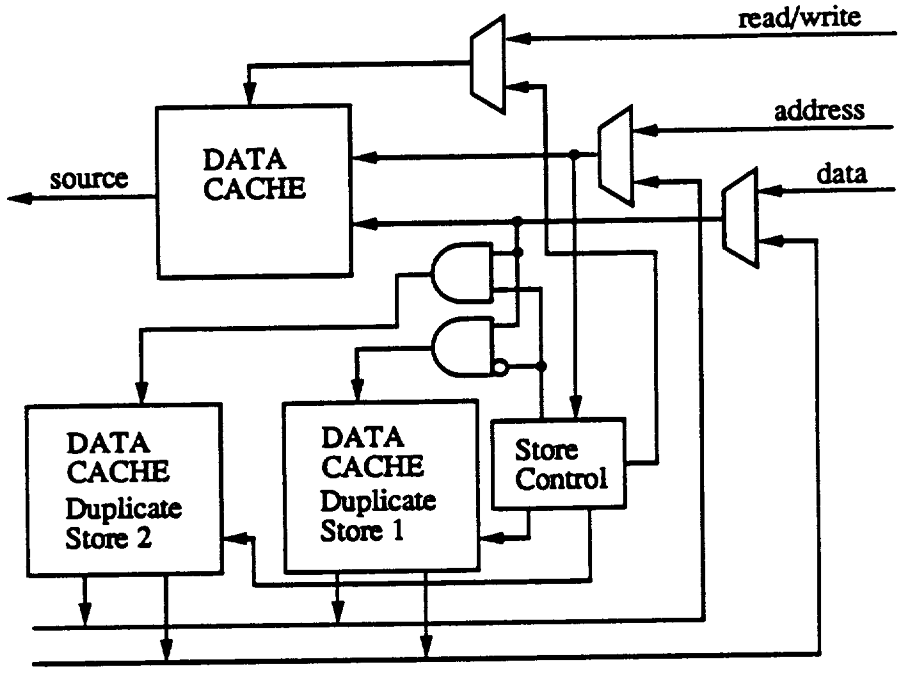


Figure 2.4: Cache duplicate store mechanism: U.S. patent number 4,044,337.

rollback distance of one instruction. A more complete description of this checkpoint retry mechanism can be found in [22].

2.3.7 Delayed write buffer

The delayed write buffer (DWB), also referred to as micro-rollback [12], establishes data redundancy to aid in rollback recovery by delaying writes to the appropriate storage location. Given a GPRF DWB of depth N, the last N register writes are contained in the DWB while the unmodified copies are maintained in the GPRF. Figure 2.5 gives an example of a high-level DWB design, where the DWB is organized as a first-in-first-out (FIFO) queue. Included in the illustration is a sample code segment and the resulting

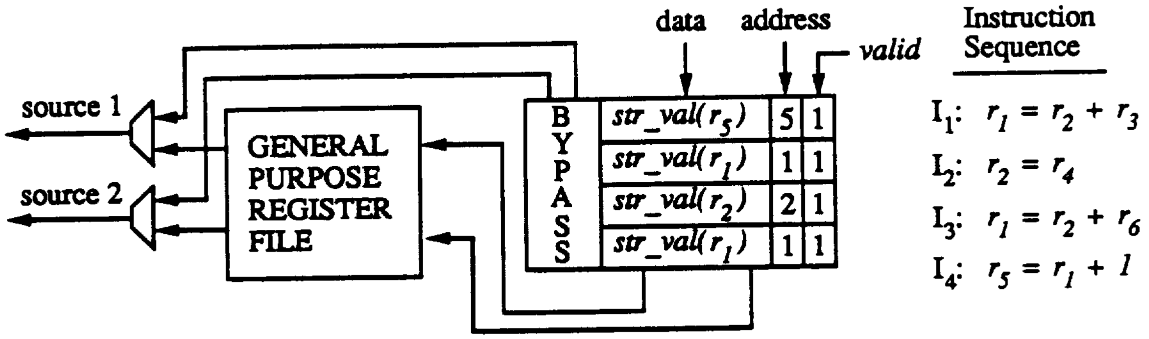


Figure 2.5: Delayed write buffer.

DWB contents. For simplicity, it will be assumed that the example processor is a simple pipelined load/store machine, executing one instruction each machine cycle.

A value written to register r_x in the GPRF is denoted as $str\_val(r_x)$ and is shown in the DWB along with the register address x. Since the most recent value of a register may be contained in the DWB, bypass logic is included to inspect the DWB during all register accesses to determine if a more recent value is available. If more than one copy of an accessed register is present in the DWB, priority logic contained in the bypass unit forwards the most recent value to the appropriate source bus. This latter case occurs during instruction I_4 where r_1 is accessed and two copies are present in the DWB. The delayed write buffer is very similar in operation to the reorder buffer proposed to aid in exception repair for out-of-order execution architectures [14].

As long as an error is detected within N cycles (for this example one instruction is completed each cycle), the contents of the DWB can be invalidated, restoring the GPRF to a *precise state* [11] prior to the faulty instruction. The DWB can be applied to any system storage unit, however, it is best suited to units that are accessed through

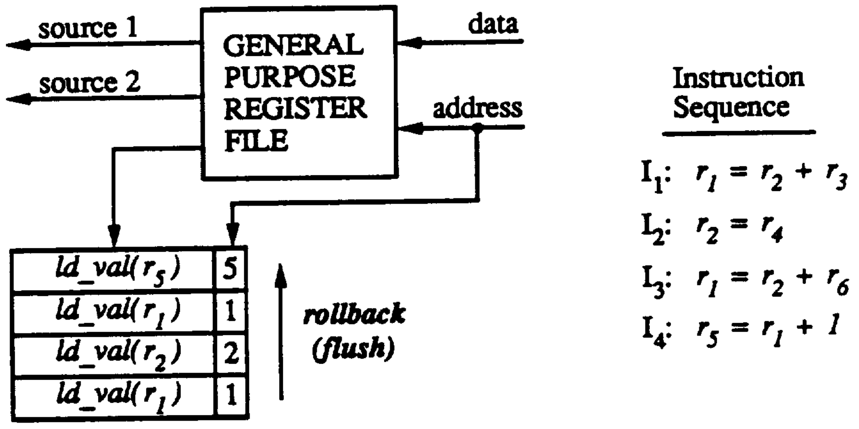


Figure 2.6: History buffer.

an address. For storage units that can be accessed directly, e.g., the program counter, instruction counter, and program status word, a state history can be maintained in a simple first-in-first-out (FIFO) queue. The delayed write buffer can be classified as an incremental checkpointing and recovery scheme.

2.3.8 History buffer

The history buffer (HB) is derived from the reordering version proposed as an aid in exception repair for out-of-order execution architectures [14]. Figure 2.6 illustrates a high-level HB design to aid in rollback recovery, along with an example instruction sequence and the corresponding HB contents. Each time register r_x is written in the GPRF, the old contents of r_x, denoted as $ld\_val(r_x)$, are read out and placed in the HB. The HB is organized as an FIFO queue.

In an HB organization, the most recent values of all registers are maintained in the GPRF; therefore, bypass logic is not required as with the DWB. In a pipelined

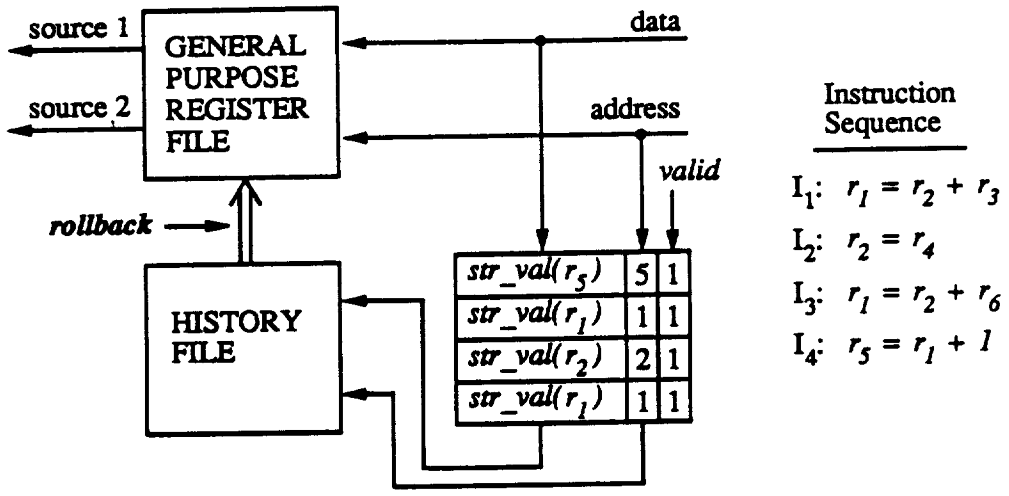


Figure 2.7: History file.

architecture, reads from both ports of the GPRF in each cycle are common. An efficient HB design requires an extra GPRF read port, complicating the GPRF design. Rollback to a precise state is accomplished by flushing the HB in reverse order, up to and including the faulty instruction. The flush operation requires many cycles in contrast to the single cycle invalidate of the DWB. The HB is classified as an incremental checkpointing and recovery scheme.

2.3.9 History file

The history file (HF) is derived from the future file proposed to aid exception repair in out-of-order execution architectures [14]. Figure 2.7 shows a high-level HF design consisting of the system GPRF and a duplicate register file. A write buffer depth of N, delays writes to the HF and ensures that the HF state is the precise state of the system that existed N cycles in the past. Since the most recent register values are contained in the GPRF and therefore no bypass logic is required. Unlike the HB, no additional read port is required for the GPRF, instead a full duplication of the GPRF is required along with the write buffer.

Rollback of exactly N instructions is very efficient, requiring a global load from the HF to the GPRF and an invalidate of the write buffer. These two operations can be performed in a single cycle. Variable rollback requires that the write buffer be flushed to the HF, up to but not including the faulty instruction prior to the global load from the HF to the GPRF. Like the DWB and HB, the HF is an incremental checkpointing and recovery scheme.

2.3.10 The IBM ES/9000

Prior to the invention of cache memory, main memory load latency was a significant performance limiter in high-end processing systems. In an effort to maximize CPU productivity during main memory accesses, IBM developed out-of-order execution [23]. The out-of-order execution feature was dropped by IBM in 1968 after the introduction of the cache.

Due to the availability of higher density technology and the need for higher performance, IBM has once again incorporated out-of-order execution into its high-end ES/9000 system. The virtual register management and branch misprediction repair schemes include data redundancy useful for multiple instruction retry. Although the design point for the ES/9000 is to detect errors within the current machine cycle, through ECC, parity, or checking, the reporting and recording of the error may take several cycles.

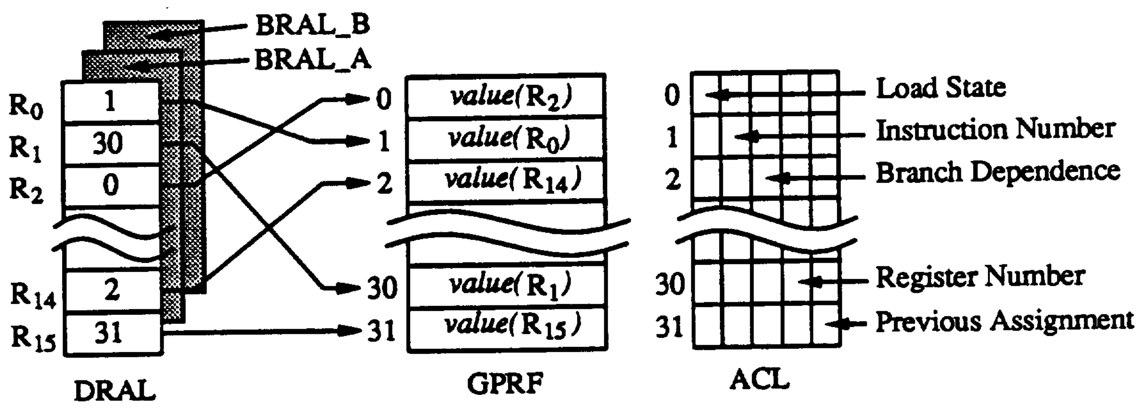


Figure 2.8: Virtual register management for the IBM ES/9000.

The ES/9000 has considerable fault-tolerance capability; however, only the instruction retry facility will be discussed here. For a more detailed description of the fault-tolerant characteristics of the ES/9000, see [24]. For a complete description of virtual register management, see [25,26].

The virtual register management system (RMS) maps 16 architectural registers into 32 physical registers. Architectural registers identified in a decoded instruction locate pointers in the decode-time register assignment list (DRAL) shown in Figure 2.8. These pointers locate the appropriate physical register from the GPRF. The array control list (ACL) has entries which correspond to the entries in the GPRF. The ACL contains the required physical register status such as the load state of the physical register (i.e., available, pending but not loaded, pending and loaded, assigned), instruction number, branch dependence, architectural register assignment, and previous assignment. The branch register assignment lists, BRAL_A and BRAL_B, contain exact images of the DRAL at the time a branch path is predicted. If the branch is mispredicted, the DRAL can be restored to its value prior to the branch. The two BRAL's allow a total of two pending branch predictions. A third branch prediction causes a stall until one of the two pending branches is resolved.

The RMS supports multiple instruction retry by holding the availability of physical registers until the appropriate instructions have been determined to be fault free. In this way, a faulty instruction can never overwrite the contents of an architectural register. Figure 2.9 illustrates this instruction retry feature.[3] Out-of-order execution forces

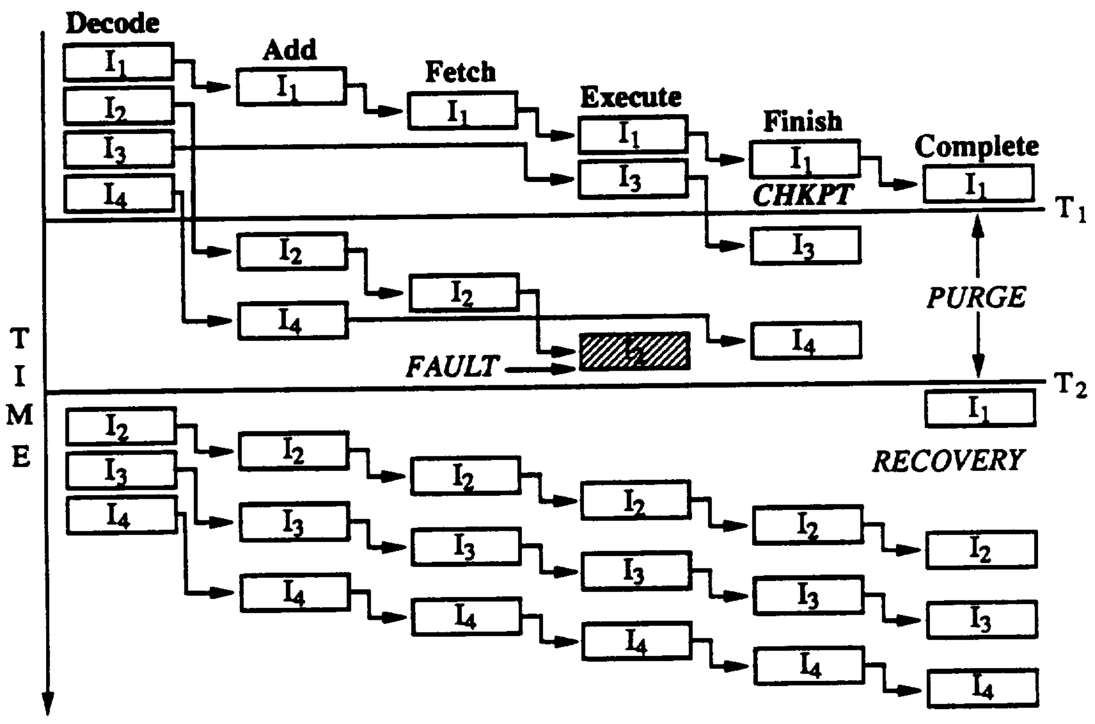


Figure 2.9: Instruction retry and recovery for the IBM ES/9000.

instructions to *complete* in order but allows instructions to *finish* in any order. Only upon completion is the physical register marked available for reassignment. In Figure

[3]Figure 2.9 source: Proc. 22th Int. Symp. Fault-Tolerant Comput. [24].

2.9, instructions 3 and 4 (I_3, I_4) *finish* prior to I_2. I_1 *completes* at time T1, and a fault in I_2 is recorded at time T2. Registers modified by I_2 and I_3 have not been released for reassignment; therefore, their contents are available for the retry sequence shown. The instruction rollback and recovery scheme of the ES/9000 can best be described as an incremental checkpointing scheme with a variable rollback distance.

2.4 Compiler-Assisted Rollback Recovery

2.4.1 Compiler-assisted checkpoint placement

Recently, compiler-based approaches to checkpointing and recovery have been investigated. The studies have been conducted in two areas: 1) system/application level checkpointing and recovery, and 2) multiple instruction rollback recovery. As an alternative to system and application level checkpointing, a compiler-assisted checkpointing and recovery scheme has been proposed [27]. The scheme uses compile-time information to create checkpoints adaptively. Efficient use of compile-time information allows for the determination of optimal checkpoint placements, the minimization of checkpoint sizes by exploiting large variations in memory usage, and the generation of sparse checkpoint code. A training technique was also developed resulting in checkpoints with lower cost and higher coverage. The compiler-assisted checkpointing scheme was shown to result in reduced checkpoint size while maintaining transparency at the programmer, operating system and hardware levels.

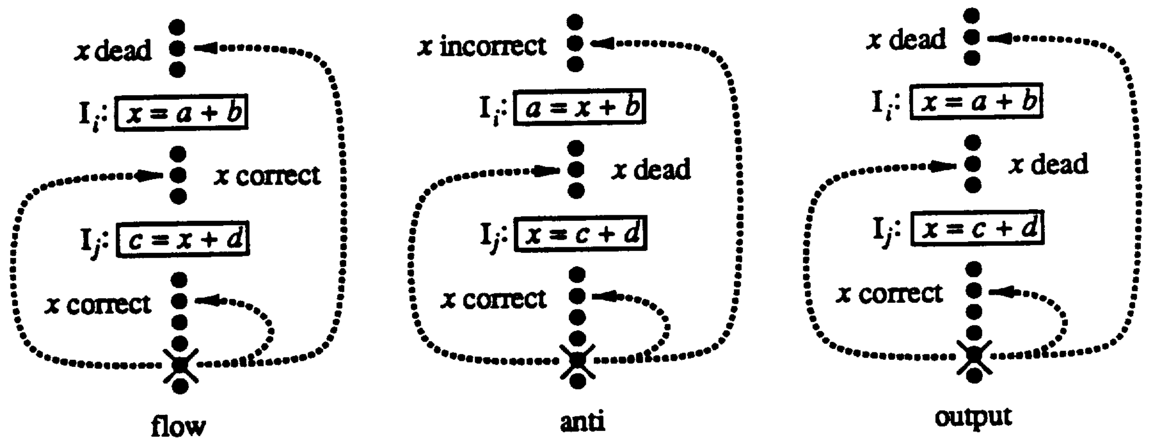


Figure 2.10: Dependencies and their impact during rollback.

2.4.2 Compiler-assisted multiple instruction rollback

In contrast to system level checkpointing, compiler-assisted multiple instruction rollback supports the rollback of a few instructions by using compiler-driven data-flow manipulations to remove hazards that result from rollback.

Figure 2.10 illustrates three data dependencies relative to variable x and their effect on rollback hazards.[4] For the *flow dependency*, the instruction I_i writes variable x and then the subsequent instruction I_j reads x. If an error is detected after I_j and rollback is below I_j, x has not been modified and there is no hazard. If the rollback is between I_i and I_j, again x is correct and no rollback hazard exists. If the rollback is to a position above I_i, x has been corrupted since it was modified by I_i. In this latter case, there is still no data hazard since x will be rewritten by I_i prior to its use in I_j, i.e., the variable x is dead. Given an *output dependency*, if an error is detected after I_j and rollback is

[4]For a complete presentation of data-flow properties and manipulation methods, see [28].

below I_j, x is correct and no rollback hazard is present. If the rollback is either between I_i and I_j or above I_i, x is dead an no hazard exists.

A data hazard is present, however, given an *antidependency*, an error detection below I_j, and a rollback to a position above I_i. In this case I_j corrupts x and after rollback, I_i uses the corrupted x value. Hardware rollback schemes maintain a redundant copy of x so that it can be restored to the correct value in the event of a rollback. By using compiler transformations to remove all antidependencies of length $\leq N$, where N represents the maximum rollback distance, the compiler-assisted rollback scheme removes data hazards and the requirement for redundancy hardware.

Antidependencies are removed at three levels: 1) pseudo code, 2) machine code, and 3) post-pass. Pseudo code is the code level prior to variables being assigned to physical registers. The primary compiler transformation to remove antidependencies is variable renaming. For the antidependency case of Figure 2.10, variable x of instruction I_j would be renamed x', requiring all subsequent uses of x to be renamed x'. Variable renaming becomes difficult when the renaming of x in I_j results in the need to rename x in I_i. This occurs when backedges exist (i.e., loops) and also through the *equivalence* relationships of variables [29]. These difficulties are handled with *node splitting*, *loop expansion*, and *loop protection* transformations [28, 29].

Once all antidependencies have been removed at the pseudo level, register allocation may result in the re-emergence of some antidependencies. An example of this would be if renamed variable x' of instruction I_j in Figure 2.10 and variable x of I_i were assigned to the same physical register. To prevent this, arcs are added to the dependence graph used in the register allocation algorithm.

Due to register spills and register save/restore conventions at procedure boundaries, some antidependencies remain in the compiler emitted code. These hazards are resolved by a post-pass transformation which inserts *no-operation* (nop) instructions to increase the antidependency distance to $> N$. The post-pass transformation carries a significant performance penalty since up to N nop's could potentially be inserted in a frequently executed portion of the code. For this reason, all possible antidependencies are removed prior to the post-pass level. The overall performance of the compiler-assisted multiple instruction rollback scheme is comparable to that of hardware schemes, with the primary advantage being the reduced hardware requirement and the ability to select the rollback distance at compile time.

2.5 Summary

Several full checkpointing and incremental checkpointing instruction retry schemes have been presented as background for the development of a new multiple instruction rollback approach. These schemes demonstrate the variety, design trade-offs and continued viability of multiple instruction rollback as a key fault tolerance feature.

This thesis focuses on multiple instruction rollback techniques and specifically compiler-based techniques similar to those presented in Section 2.4.2. Previous compiler-based schemes produced average performance impacts similar to comparable hardware implemented schemes such as the delayed write buffer [12]. The performance impact of the compiler-based scheme, however, varied greatly between applications. This thesis extends compiler-based multiple instruction rollback recovery to a broad class of code execution failures and reduces the average and peak performance impacts observed in previous schemes. The new compiler-assisted scheme combines compiler-driven rollback hazard removal with hardware implemented hazard removal.

3. COMPILER-ASSISTED MULTIPLE INSTRUCTION ROLLBACK

3.1 Introduction

This chapter extends the compiler-based [29] instruction retry scheme discussed in Chapter 2 to include a broad class of code execution failures.[1] Given a more general error model, data hazards which occur as a result of multiple instruction rollback are formally classified. This classification proves useful in the development of two new multiple instruction retry schemes. The first scheme extends the compiler-based scheme while the second scheme combines compiler-driven hazard removal with hardware data redundancy techniques. The new compiler-assisted approach exploits the unique characteristics of different hazard types. Ten benchmarks were used to measure the performance penalty of hazard resolution. Experimental results indicate that the enhanced compiler-based approach can achieve overall performance consistent with existing hardware and compiler-based approaches, and that the new compiler-assisted resolution strategy can

[1]Portions of Chapter 3 were derived from [30].

achieve superior performance to either a hardware-only or compiler-based instruction retry scheme.

3.2 Error Model and Hazard Classification

3.2.1 Rollback data hazard model

The following are assumptions used in the error model:

1. The maximum error detection latency is N instructions.
2. Memory and I/O have delayed write buffers and can rollback N cycles.
3. The states of the program counter and program status word (PSW) are preserved by an external recording device or by shadow registers as described in the micro rollback scheme [12].
4. The CPU state can be restored by loading the correct contents of the register file, program counter, and PSW.

Given the above assumptions, any error which does not manifest itself as an illegal path in the control-flow graph (CFG) of the program is allowed provided that the following conditions are satisfied:

1. Register file contents do not spontaneously change.
2. Data can not be written to an incorrect register location.

The following is a list of targeted errors:

1. CPU errors such as those caused by an ALU.
2. Incorrect values being read from I/O, memory, the register file, or external functional units such as the floating point unit.
3. Correct/incorrect values being read from incorrect locations within the I/O, memory, or register file.
4. Incorrect branch decisions resulting from error types 1, 2, or 3.

3.2.2 Hazard classification

The code can be represented as a CFG $G(V, E)$, where V is the set of nodes denoting instructions and E is the set of edges denoting control-flow. If there is a direct control-flow from instruction i, denoted I_i, to I_j, where $I_i \in V$ and $I_j \in V$, then there is an edge $(I_i, I_j) \in E$. Let $d_{min}(I_i, I_j)$ denote the smallest number of instructions along any path from I_i to I_j.

The hazard set H_{regs} of the error model is defined as the set of pseudo registers (or machine registers) whose values are inconsistent during different executions of an instruction sequence due to retry. A formal classification of hazard set H_{regs} follows.

Property 1: $x \in H_{regs}$ *iff* there exists a sequence of instructions $I_1, I_2, \ldots, I_N$ which form a legal walk[2] in G such that x is *live* at I_1, and x is defined during the walk.

[2]A *walk* is a sequence of edge traversals in a graph where the edges visited can be repeated [31].

Proof: For the *if* case, an error occurring in I_1 will be detected by I_N. During the retry of I_1, x will be in an inconsistent state since it was defined during the walk. Since x is *live* at I_1, there is some path along which x is used prior to its redefinition, and since x is in an inconsistent state, $x \in H_{regs}$. For the *only if* case, we suppose the contrary. Assume that among all legal walks of length N in G, either x is not live at the beginning, or x is not defined during the walk. It then follows that x either has no use, or x is not changed. (The error model does not allow a write to a wrong location and the contents of register x can not spontaneously change.) Therefore there is no inconsistency problem for x, which implies $x \notin H_{regs}$.

Property 2: All hazards can be classified as one of two types: 1) those that appear as antidependencies of length $\leq N$ in $G(V, E)$, referred to as *on-path* hazards, and 2) those that appear at branch boundaries, referred to as *branch* hazards. These two hazard types may overlap.

Proof: Since $x \in H$, there exists a legal walk $W_1 = I_1, I_2, \ldots, I_N$ in G, such that x is live at I_1, and after the execution of $I_1, I_2, \ldots, I_N$ in sequence, x has a different value. The latter implies that there is at least one instruction defining x along W_1 (the error model does not allow a write to a wrong location and the content of register x can not spontaneously change). Let i be the largest index that I_i defines x, where $i \in \{1, 2, \ldots, N\}$. Property 1 implies that there exists a legal walk W_2 in G, beginning with I_1, such that the first instruction I_j along W_2 referring x is a use. Case 1: if $W_2 \subseteq W_1$, instructions I_j and I_i constitute an antidependency of length $\leq N$, and there

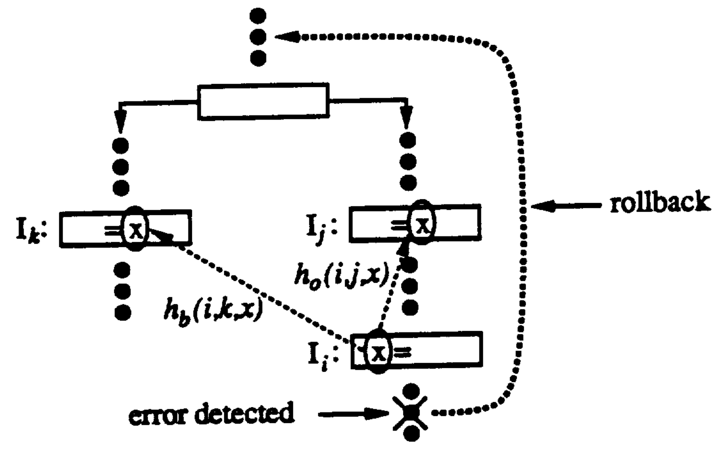


Figure 3.1: On-path and branch hazards.

is an on-path hazard on x. Case 2: if $W_2 \not\subseteq W_1$, there exists a branch instruction I_k between I_1 and I_{i-1}. Since $d_{min}(I_k, I_i) \leq N$, there is a hazard on x at a branch boundary.

3.2.3 Definitions and terminology

An on-path or branch data hazard occurs when I_i defines variable x, and after rollback, I_j uses the corrupted x value prior to its being redefined. To simplify subsequent discussion, such on-path and branch hazards will be denoted $h_o(i,j,x)$ and $h_b(i,j,x)$ respectively. Figure 3.1 illustrates this hazard notation. A few definitions are now presented to simplify subsequent discussions.[3]

[3]A complete description of data-flow terminology can be found in *"Compilers: Principles, Techniques, and Tools"*, Aho et al., [28]. More on equivalence can be found in [29].

Definitions:

1. If I_i defines variable x, then *def(i)* $= x$.
2. If the k^{th} operand of I_i uses variable x, then $use_k(i) = x$.
3. If there is some path beginning with I_i which encounters a use of x prior to a definition of x, then $x \in live\_in(i)$.
4. If there is some path from I_i to I_j which does not encounter a redefinition of *def(i)*, then *def(i) reaches j*.
5. If *def(i) reaches j* and *def(i)* $= use_k(j)$ for some k, then *def(i) reaches* $use_k(j)$.
6. If *def(i) reaches j* then $i \in reaching\_in(j)$.
7. If renaming *def(i)* requires the renaming of $use_k(j)$ for any k, then $equiv(i,j) = 1$.

3.3 Compiler Resolution of On-path and Branch Hazards

Previously developed compiler transformations restrict hazard resolution to on-path hazards [29]. The transformations are performed in four phases. Phase 1 resolves pseudo register hazards, phase 2 resolves machine register hazards, phase 3 resolves interprocedural register hazards, and phase 4 uses nop insertion to resolve the remaining hazards. The expanded error model of Section 3.2 permits branch hazards in addition to on-path hazards. What follows is a discussion of the viability of these same compiler transformations in application to branch hazard resolution.

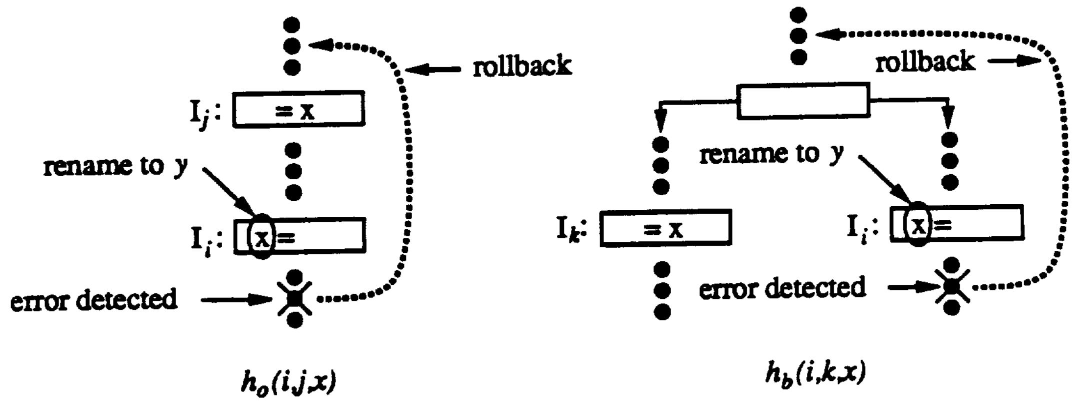


Figure 3.2: Register renaming.

3.3.1 Pseudo register renaming

The basic compiler transformation to remove hazards is register renaming. Figure 3.2 shows how hazard $h_o(i,j,x)$ can be removed by renaming *def(i)* from x to y. It can be seen from Figure 3.2 that register renaming is equally effective in resolving branch hazard $h_b(i,k,x)$ by renaming *def(i)* from x to y. If $h_o(i,j,x)$ and $h_b(i,k,x)$ coexist as in Figure 3.1, both hazards are resolved simultaneously, i.e., given $h_o(i,j,x)$ and $h_b(i,k,x)$, resolution of $h_o(i,j,x)$ through renaming resolves $h_b(i,k,x)$. In addition to renaming x to y in I_i, some uses of x in other instructions must also be renamed to y. The variables requiring renaming are determined by the equivalence property [29]. If $equiv(i,j) = 1$ in the examples of Figure 3.2, then $use_1(j)$ would ultimately be renamed to y, negating the hazard resolution. Equivalence can negate both on-path hazard resolution and branch hazard resolution.

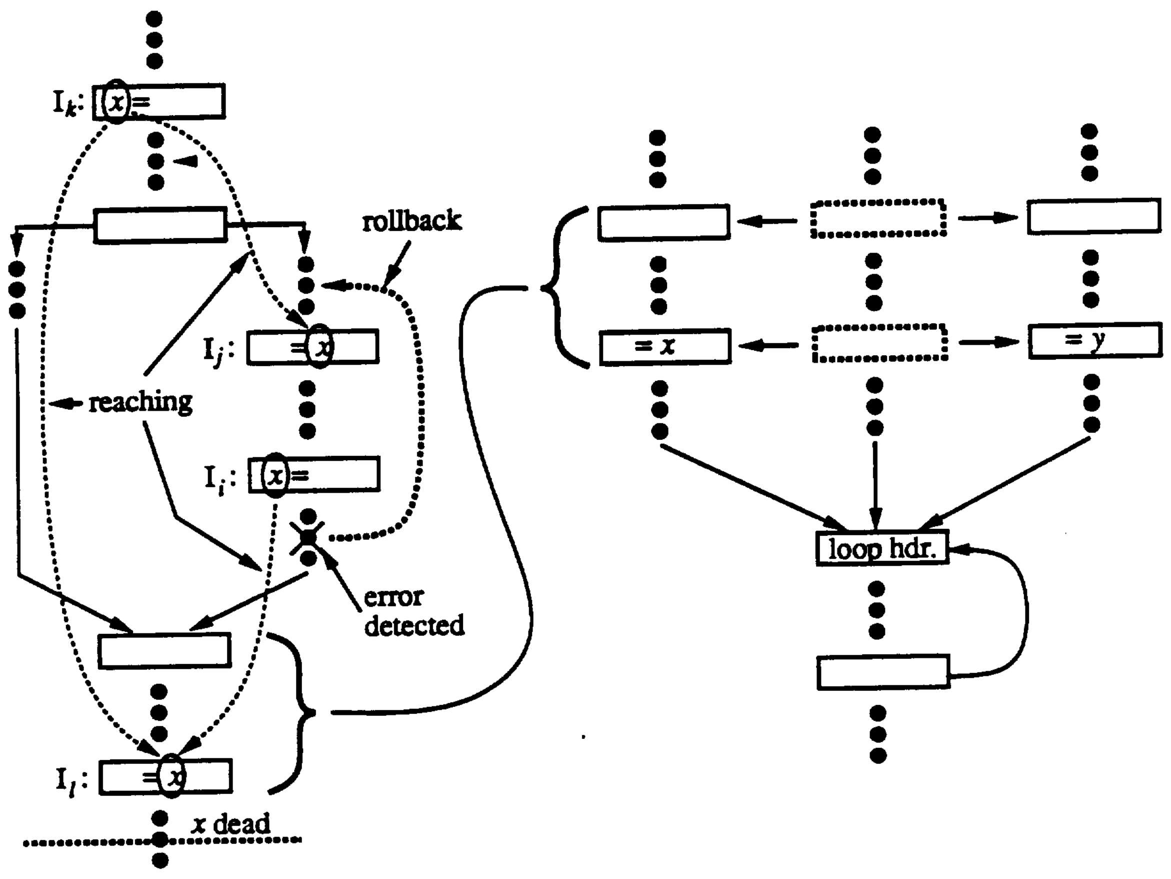


Figure 3.3: Node splitting.

To break the equivalence relationship, ***node splitting*** and ***loop expansion*** transformations are used. A ***loop protection*** transformation ensures that loop integrity is maintained during the node splitting and loop expansion transformations.

3.3.2 Node splitting

Given $h_o(i,j,x)$ or $h_b(i,j,x)$, node splitting forces $equiv(i,j) = 0$. Figure 3.3 shows an example of data dependence requiring node splitting and the result of a node splitting algorithm. Since *def(i)* reaches $use_1(l)$, renaming x in I_i to y forces the the renaming of x

```
disable back edges;
calculate hazards;
while($changed$), do;
    $changed$ = 0;
    for all $x$ in $H_{regs}$, do;
        for all V $\in$ G(V,E), do;
            if $x \notin live\_in$(V)
                continue;
            if multiple definitions of $x$ reach V, do;
                split(V);
                $changed$ = 1;
        endfor
    endfor
endwhile
enable back edges;
```

Figure 3.4: Node splitting algorithm.

to y in I_l. Since *def(k)* also reaches $use_1(l)$, *def(k)* must be renamed to y. Finally, *def(k)* reaches $use_1(j)$, requiring $use_1(j)$ to be renamed to y. Hazard $h_o(i,j,x)$ has changed to $h_o(i,j,y)$ but it has not been resolved. Register renaming cannot resolve $h_o(i,j,x)$ or $h_b(i,j,x)$ given $equiv(i,j) = 1$. The simple node splitting algorithm shown in Figure 3.4 forces $equiv(i,j) = 0$ given $h_o(i,j,x)$ or $h_b(i,j,x)$.

When two definitions of a hazard variable reach a node, the node is split. The effect is an "unzipping" of instructions which stops when the hazard variable becomes dead or when a loop header is reached. In the former case, the equivalence relationship can no longer be affected by the instruction. In the latter case, a split of the loop header would compromise the integrity of the loop. It would be possible to treat the loop as a node and duplicate the entire loop; however, this would result in significant code growth. The loop protection algorithm is responsible for ensuring that no loop header is split.

After node splitting, a hazard node i [4] has a "personalized" path to each of the use nodes it reaches. More formally, given $h_o(i,j,x)$ or $h_b(i,j,x)$, no $use_k(m)$ that is reached by *def(i)* is reached by *def(n)*, where *def(n)* = *def(i)*. Node splitting does not break direct equivalence. Direct equivalence is $equiv(i,j) = 1$ such that *def(i)* reaches $use_k(j)$. For $h_o(i,j,x)$ and $h_b(i,j,x)$ this occurs when *def(i)* reaches $use_k(j)$ by traversing a loop back edge.[5] For this reason, the node splitting algorithm of Figure 3.4 is run with the back edges of $G(V,E)$ disabled. On-path hazards that remain after node splitting are resolved with loop expansion.

3.3.3 Loop expansion

Loop expansion involves unrolling a loop in an effort to remove on-path hazards which remain after node splitting. Figure 3.5 gives an example of a loop which requires expansion due to an on-path hazard. It can be seen that the rollback traverses the loop back edge. For this example, the loop is unrolled once by duplicating the loop body one time. The second copy of the loop body has all occurrences of x renamed to y. The length of the loop-carried antidependency present in the original loop has increased from

[4]Hazard node i is defined as the node representing I_i in $G(V,E)$ given the existence of either $h_o(i,j,x)$ or $h_b(i,k,x)$.

[5]In contrast to on-path hazards, direct equivalence for branch hazards can exist without loop back edges; however, was observed to be infrequent for the eleven application studied. The handling of these hazards is presented in Chapter 4.

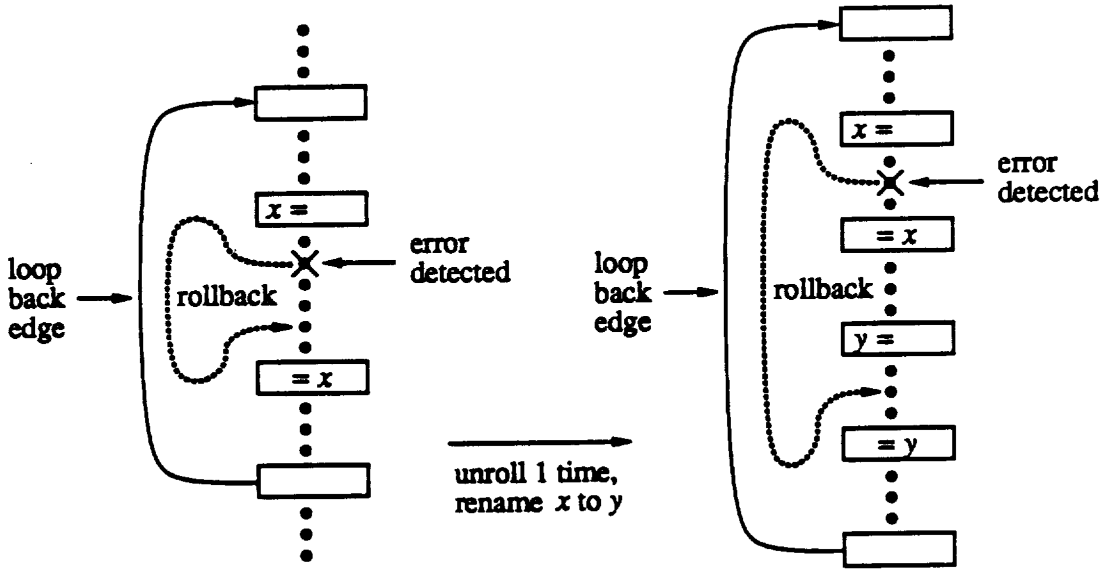


Figure 3.5: On-path hazard traversing a loop back edge.

$\leq N$ to $> N$. Loop expansion to resolve on-path hazards results in significant code growth, reaching 350% for some applications [29].

Although on-path hazards which traverse loop back edges are common, we have experimentally observed a low rate of occurrence of branch hazards traversing loop back edges. This is due to three factors. The first is that branch hazards are less common than on-path hazards. Figure 3.6 shows the percentage of all nodes that are on-path and branch hazard nodes given various rollback distances for the QUEEN and PUZZLE applications. Details on the evaluation methodology can be found in Section 3.5. The second factor is that resolving $h_o(i, j, x)$ through renaming resolves $h_b(i, k, x)$. Since on-path hazards are resolved prior to the resolution of branch hazards, many branch hazards which traversed loop back edges are no longer present. Finally, the most common code structure which results in a branch hazard traversing a loop back edge also causes the

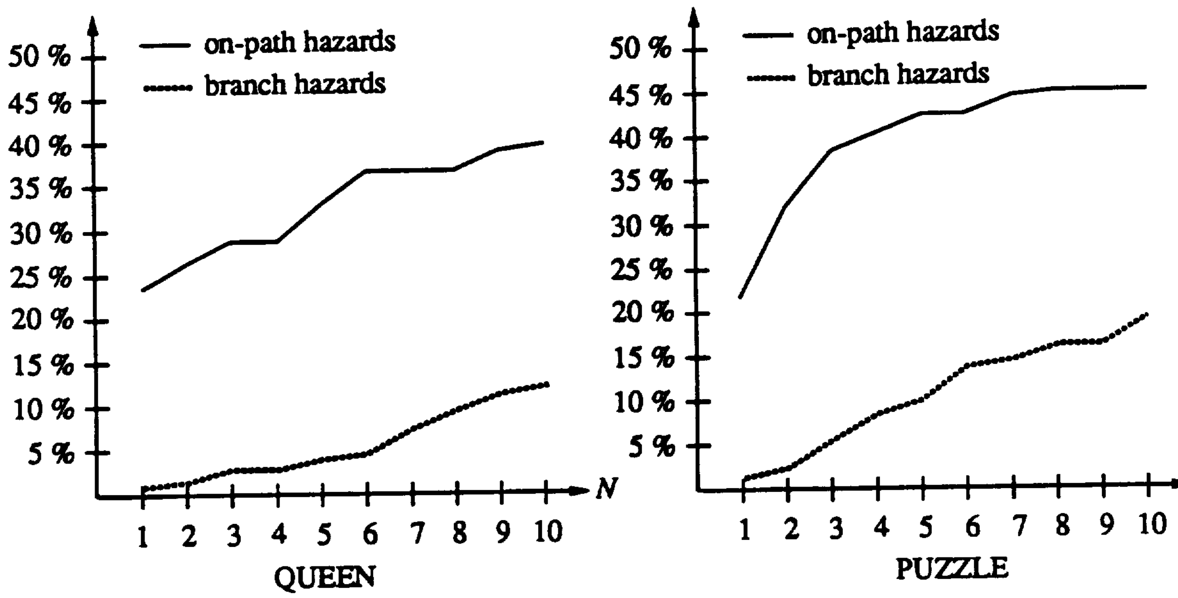


Figure 3.6: Percentage of total I_i's that are on-path and branch hazards.

loop to be protected by inserting a save/restore pair around the loop. The save/restore pair breaks the direct equivalence of the branch hazard and thereby resolves it without the need for loop expansion. Due to the significant code growth potential of loop expansion and the infrequency of branch hazards traversing loop back edges, all such hazards are left to be resolved in the nop insertion phase.

3.3.4 Loop protection

Figure 3.7 demonstrates how loop l is protected from hazard node i, where $def(i) = x$. The loop header will not be split since $x \notin live\_in(hdr\_node(l))$, where $hdr\_node(l)$ represents the header node of loop l. The loop protection transformation operates on two inputs. The first is the pseudo register x which is defined in I_i given hazard $h_o(i,j,x)$. The second is the live-out analysis of the CFG. The loop protection transformation is

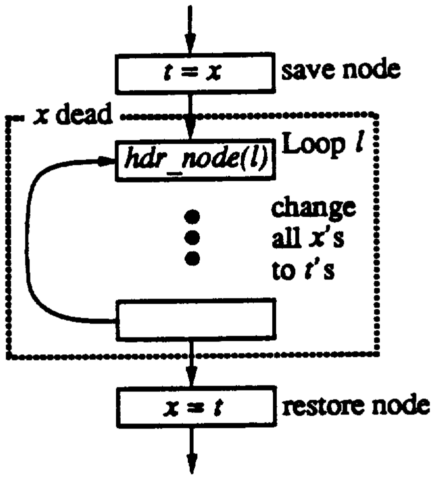


Figure 3.7: Loop protection from hazard variable x.

not dependent on the type of hazard which identifies the pseudo register and will protect loops from header splitting that would occur as a result of branch hazards.

3.3.5 Machine registers

Once hazards have been eliminated through renaming, they may reappear as physical registers are assigned. It is also possible that new hazards will emerge. Figure 3.8 shows the elimination of on-path and branch hazards by adding arcs to the dependency graph used for register allocation.

3.3.6 Interprocedural hazards

Interprocedural register saving conventions can create immediate on-path hazards. For example, if register r_k is read and saved prior to a procedure call, and then initialized in the called procedure, an antidependency is created. Previous work used a disjoint register block scheme to guarantee that any read prior to a procedure call and any

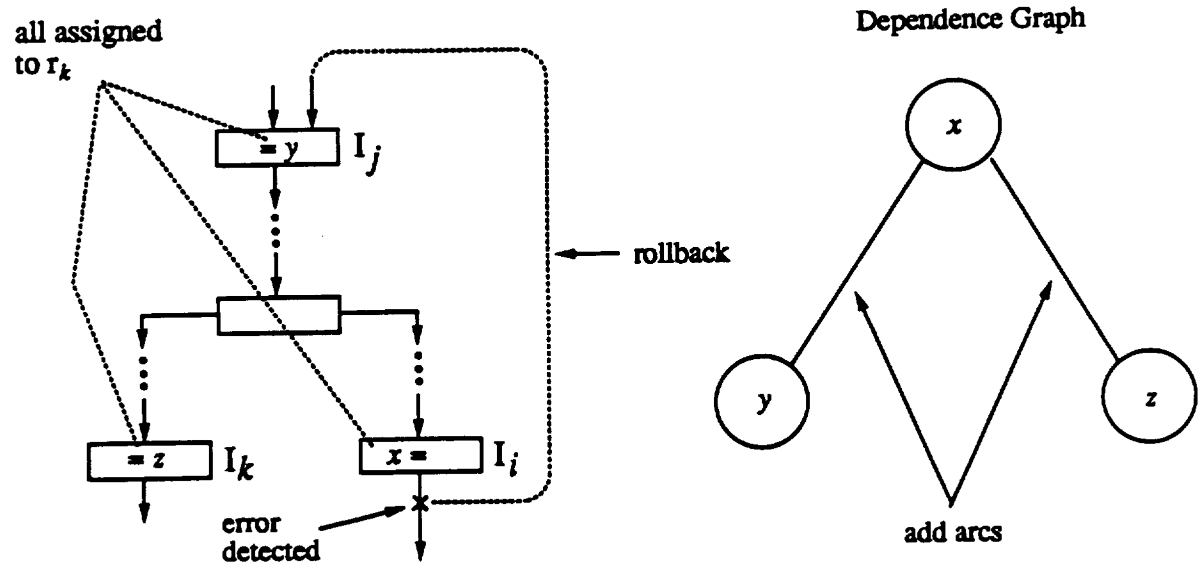


Figure 3.8: On-path and branch machine register hazards.

definition during procedure initialization use registers from different blocks [29]. Branch hazards are not immediately created at procedural boundaries. All remaining branch hazards are resolved in the nop insertion phase described in the next section or by the post-pass transformation described in Section 3.4.3 (p. 44).

3.3.7 Nop insertion

Spill code as a result of register allocation can create on-path and branch hazards. A similar problem exists with the stack pointer and frame pointer. Some branch hazards may also remain that were unresolved with the loop expansion transformation. On-path hazards are resolved by inserting nop instructions directly before the hazard instruction so that the rollback will be below the last use of the hazard register. This technique does not work for branch hazards since the distance between the definition and the use instructions is not relevant. Instead, nop insertion is used to increase the distance from

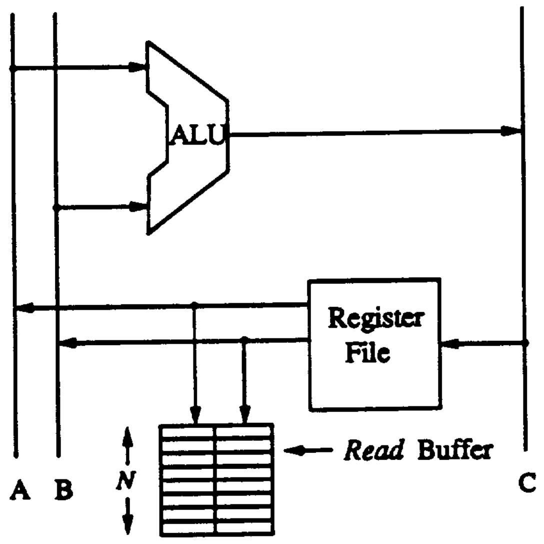


Figure 3.9: *Read* buffer.

the hazard instruction to its nearest predecessor branch. In this case, a rollback will be below the branch.

3.3.8 Summary

It has been shown that compiler techniques previously developed to resolve on-path hazards are equally effective in resolving branch hazards. A compiler-based multiple instruction rollback recovery scheme utilizing theses transformations was developed and evaluated. The results of the evaluations are presented in Section 3.5.

3.4 Hardware-Assisted Hazard Resolution

3.4.1 The read buffer

Figure 3.9 shows a hardware scheme to resolve on-path hazards. A *read buffer* is attached to the output ports of the register file. Each time a register is used it appears on the read port and is saved in the read buffer. If a register r_k is defined in I_i and it is an on-path hazard, then r_k must have been read within the last N cycles. In this case, the read buffer will contain the old value and it is permissible to write the new value into the register file. In the event of a rollback of N instructions, the contents of the read buffer are flushed in reverse order and stored back to the register file. For an on-path hazard, the path taken after the rollback will be the same as the path taken prior to rollback and each read of r_k will produce the same value as before. Branch hazards will be removed by the compiler transformation presented in Section 3.3. It is assumed that the read buffer is an integral part of the register file and any error in the system does not corrupt the transfer to the read buffer or its contents.

In contrast to a history buffer which forces a read of r_k prior to writing r_k , the read buffer monitors the register file ports and stores only the values read as part of the normal program flow and, therefore, should not significantly impact the register file performance or CPU cycle time. The read buffer is twice the width of a register with a depth of N. This is twice the size of a delayed write buffer, but eliminates the requirement for complex bypassing and prioritization logic.

3.4.2 Covering on-path hazards

In addition to resolving all on-path hazards, the read buffer will resolve some branch hazards. Figure 3.10 shows an on-path hazard and a branch hazard both with definitions of x in I_i and uses of x, after rollback, in instructions I_j and I_j, respectively. Note that if path l is initially taken, the read buffer will contain the old value of x and rollback

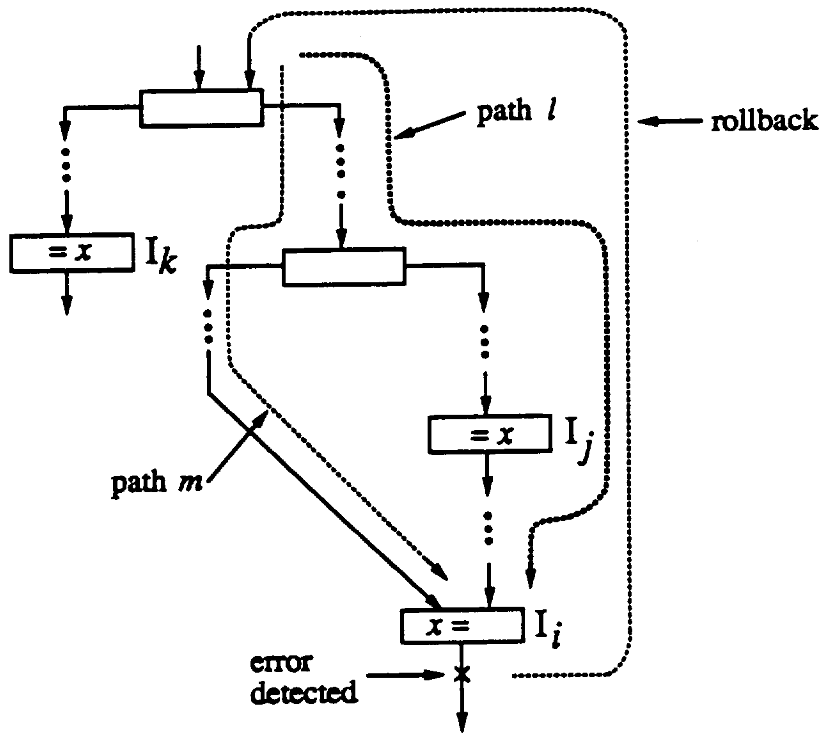


Figure 3.10: Covering on-path hazard.

would be successful. However if path m is taken, the read buffer will not contain the old value of x and rollback would be unsuccessful. If only paths such as l exist, the presence of the on-path hazard assures successful rollback or "covers" the branch hazard. In this case, resolution of the branch hazard using compiler techniques is not necessary.

3.4.3 Post-pass transformation

Given the efficiency of the read buffer in resolving on-path hazards, a post-pass transformation on assembler-level code becomes possible as a replacement for the nop insertion transformation described in Section 3.3.7 (p. 41). The post-pass transformation creates on-path hazards when necessary to assure that all branch hazards are resolved by the read buffer. Given one such branch hazard which defines physical register r_k at instruction I_i, the transformation inserts an MOV r_k, r_k instruction immediately before I_i. This guarantees that all paths leading to I_i are like path l in Figure 3.10.

3.5 Performance Evaluation

3.5.1 Implementation

The transformation algorithms presented in Section 3.3 have been implemented in the MIPS code generator of the IMPACT C compiler [32]. Transformations resolving pseudo register hazards (loop protection, node splitting, and loop expansion) are called just before register allocation. Transformations resolving machine register hazards are called after the live range constraints have been generated and before physical register allocation. The nop insertion algorithm, or post-pass algorithm, is called before the assembly code output routine.

3.5.2 Application programs

Table 3.1 lists the eleven application programs used in the evaluations. The applications were cross-compiled on a SPARCserver 490 and run on a DECstation 3100. *Size* is the number of assembly instructions emitted by the code generator, not including the library routines and other fixed overhead.

The results are summarized in Figures 3.11 through 3.21 (pp. 50 through 55). Each figure contains two plots, the first plot shows the percent of run-time overhead (*Time*

Table 3.1: Application programs: run-time and code size overhead evaluation.

Program	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
TBL	8197	table formatting preprocessor
CCCP	8775	preprocessor for gnu C compiler

OH) of the referenced hazard resolution scheme, and the second plot shows the percent of code growth overhead (*Size OH*) relative to the base values in Table 3.1.

Four hazard resolution techniques were evaluated. *Compiler 1* resolves on-path hazards only, using the compiler-driven data-flow manipulations presented in Chapter 3. *Compiler 2* extends the compiler transformations to resolve both on-path and branch hazards. *PP* (post-pass) disables the compiler transformations and relies solely on the post-pass transformation presented in Section 3.4. *Comp/PP* uses compiler transformations to resolve branch hazards, assumes a read buffer to resolve on-path hazards, and uses the post-pass transformation to remove remaining branch hazards.

Due to the excessive compile times of *Compiler 1* and *Compiler 2*, for large applications, the evaluations of these schemes were restricted to applications QUEEN, WC, COMPRESS, CMP, PUZZLE, and QSORT. The compiler transformations to resolve branch hazards for *Comp/PP* have been enhanced to reduced compile times. These enhancements are described in Chapter 4. Both *Comp/PP* and *PP* were evaluated for all eleven applications.

3.5.3 Performance analysis

Compiler transformations used for the removal of data hazards can impact performance in several ways. Loop protection inserts save/restore operations at the head and tail of the loop. This increases the path length and, therefore, the run time. Additional arcs in the dependency graph can cause more spill code to be generated, increasing memory references and cache misses. Nop insertion can be costly since up to N nops could be inserted for each unresolved hazard. The insertion of MOV r_k, r_k instructions to create covering on-path hazards in the post-pass transformation also increases path lengths, although typically less than with nop insertions. Finally, the increase in code size, mainly due to loop expansion, may cause more run-time cache misses.

The loop expansion transformation can improve performance over a compiler that does not have this optimization technique [33] as demonstrated by the negative run-time overhead measurements for COMPRESS, CMP, and PUZZLE, shown in Figures 3.13, 3.14, and 3.15 (pp. 51 and 52), respectively. Once the loop is expanded, some condition checks and index operations can be eliminated. Also the save/restore operations from loop protection shorten the live ranges of some registers thus allowing more efficient register allocation. Only the latter optimization is implemented in the current software.

3.5.4 Results: *Compiler 2*

As can be seen in Figures 3.11 through 3.16 (pp. 50 through 52), extending the compiler hazard resolution scheme to include branch hazards introduces little incremental performance impact or code growth overhead. Given a rollback distance of 10, resolving both on-path and branch hazards using compiler transformations resulted in a maximum performance impact of 32.6% and an average performance impact of 12.6%. This compares with maximum and average impacts of 35.4% and 15.4%, respectively, for compiler-driven on-path hazard resolution only. The maximum code size overhead measured for the extended compiler-based was 328% with an average overhead of 207%, for a rollback distance of 10. This compares with a maximum and average overhead of 372% and 225%, respectively, for the unextended compiler-based scheme.

These results indicate a small incremental run-time performance overhead and a small code size overhead given compiler-based branch hazard removal compared to compiler-based on-path hazard removal alone. Three factors account for these small incremental impacts. First, on-path hazards dominate in frequency of occurrence. Second, resolving an on-path hazard at instruction I_i through renaming can sometimes resolve a branch hazard at instruction I_i. Third, resolving on-path hazards with nop insertion may resolve a corresponding branch hazard by increasing the distance between the hazard node and its nearest predecessor branch node.

3.5.5 Results: *PP*

Figures 3.11 through 3.21 (pp. 50 through 55) show the run-time and code size overheads for each application studied assuming the read buffer to resolve on-path hazards and the post-pass transformation described in Section 3.4 to cover all branch hazards. The results are worst case in that many of the branch hazards could have been resolved with no performance impact using the compiler techniques of Section 3.3. Instead, they are resolved by the insertion of MOV instructions which cause a guaranteed, although small, performance impact. Given a rollback distance of 10, the post-pass transformation produced a maximum performance impact of 7.69% with an average performance impact of 2.43%, significantly below the levels produced by the compiler-based scheme. Code growth overhead measurements were correspondingly lower with a maximum overhead of 13.0% and an average overhead of 8.59%.

3.5.6 Results: *Comp/PP*

The compiler-assisted scheme achieved consistently low performance overheads across all applications and slightly better performance than with the post-pass transformation only. Given a rollback distance of 10, the compiler-assisted scheme produced a maximum performance impact of 6.57% with an average performance impact of 2.03%, and a maximum code growth overhead of 51.2% with and an average overhead of 15.5%. The run time results of PUZZLE, YACC, and CCCP indicate that compiler techniques are still useful in reducing run-time performance penalties. These compiler techniques, however, have the disadvantage of requiring recompilation and additional code growth. The primary advantage of the compiler-assisted and post-pass schemes are their utilization of the read buffer to resolve the more frequent on-path hazards.

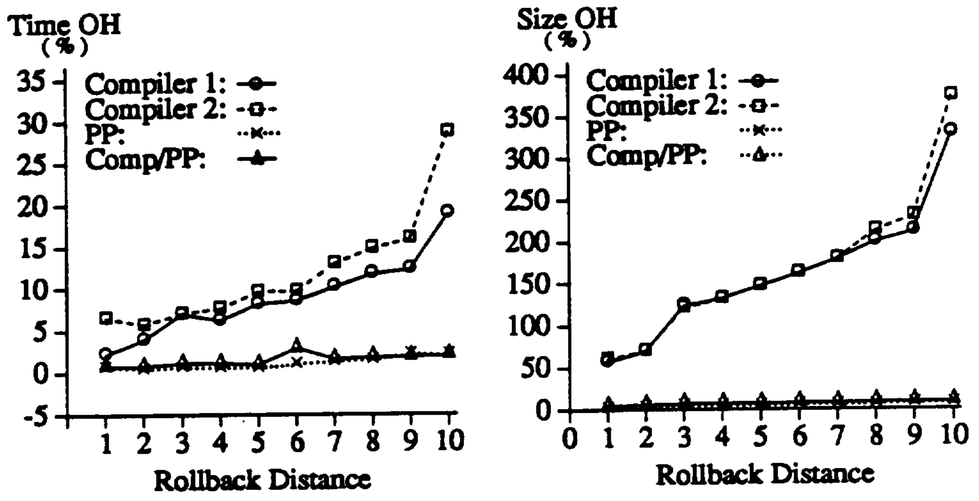


Figure 3.11: Run-time overhead and code size overhead: QUEEN.

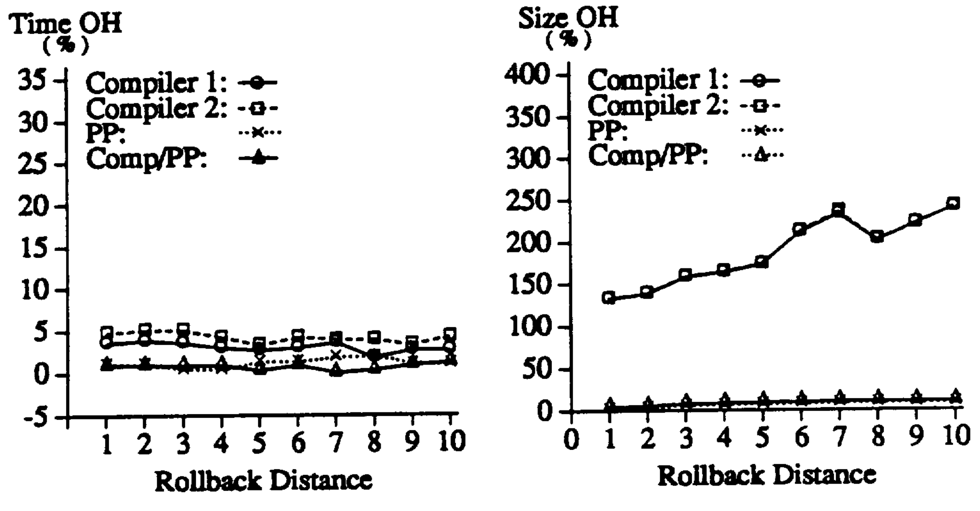


Figure 3.12: Run-time overhead and code size overhead: WC.

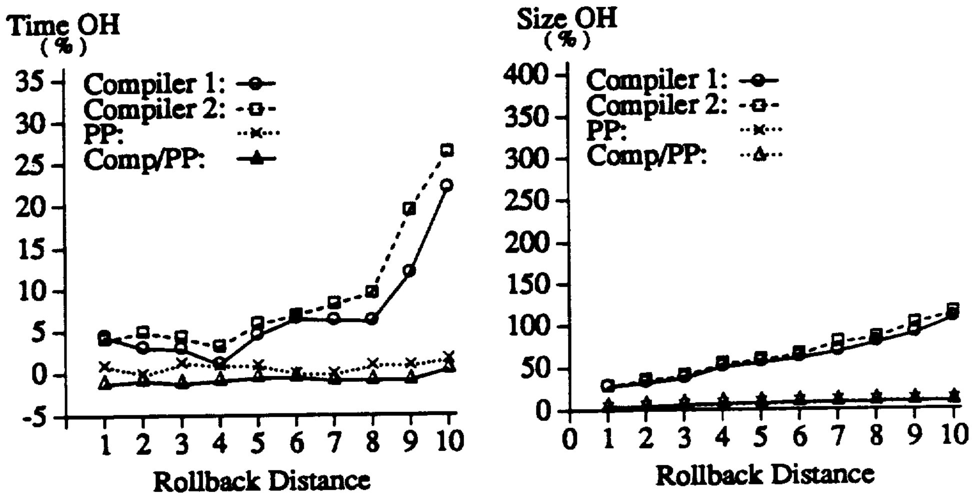


Figure 3.13: Run-time overhead and code size overhead: COMPRESS.

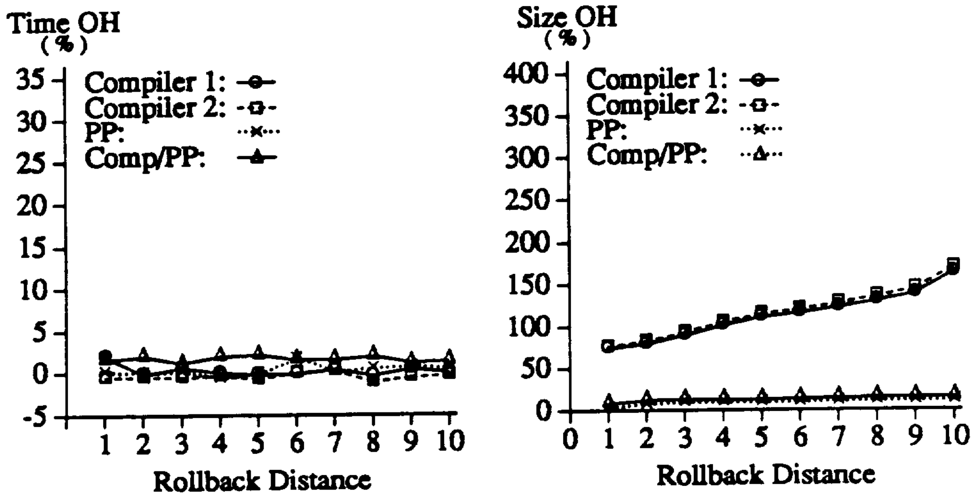


Figure 3.14: Run-time overhead and code size overhead: CMP.

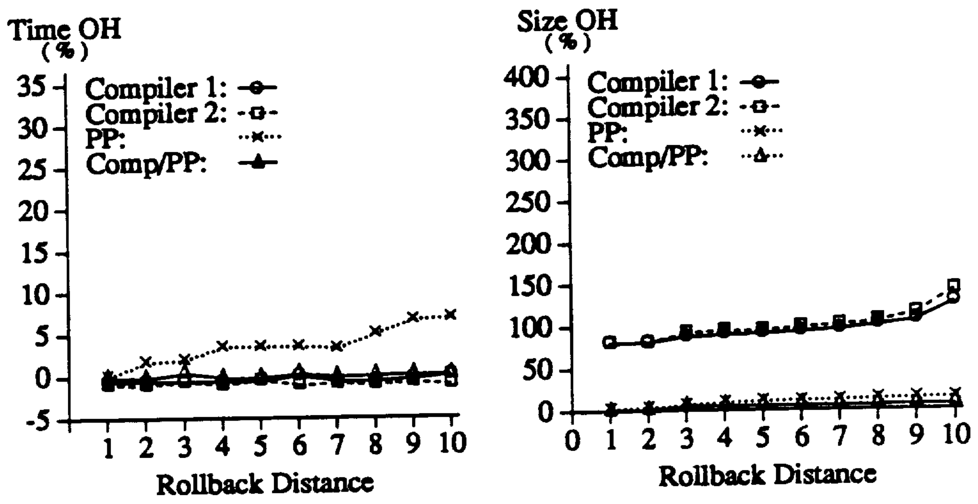


Figure 3.15: Run-time overhead and code size overhead: PUZZLE.

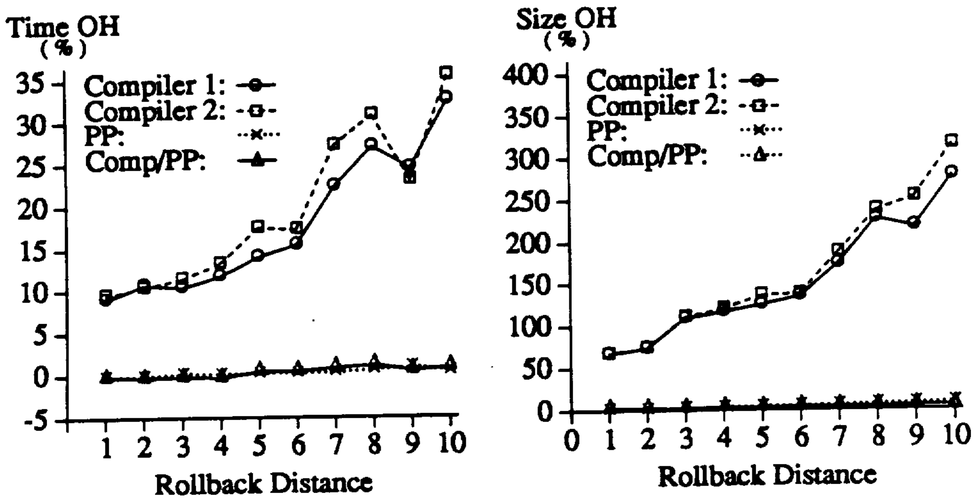


Figure 3.16: Run-time overhead and code size overhead: QSORT.

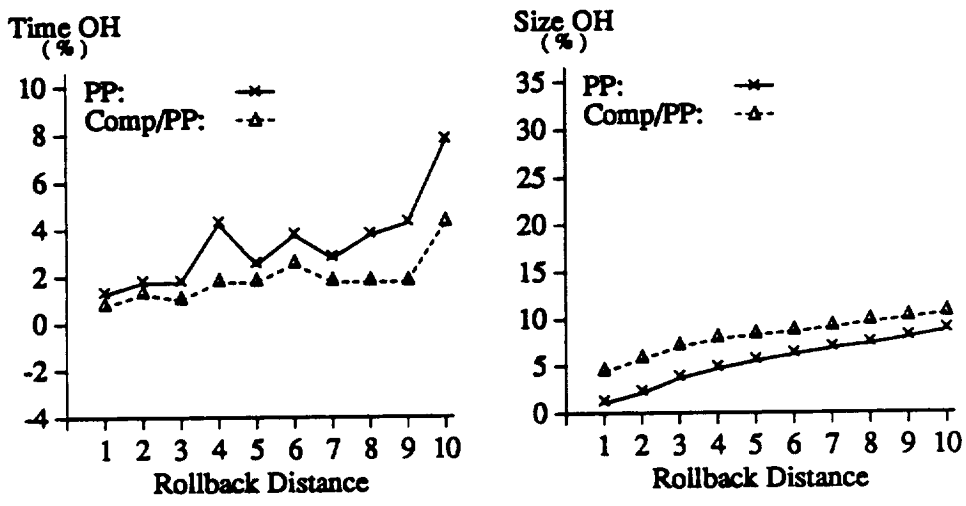


Figure 3.19: Run-time overhead and code size overhead: YACC.

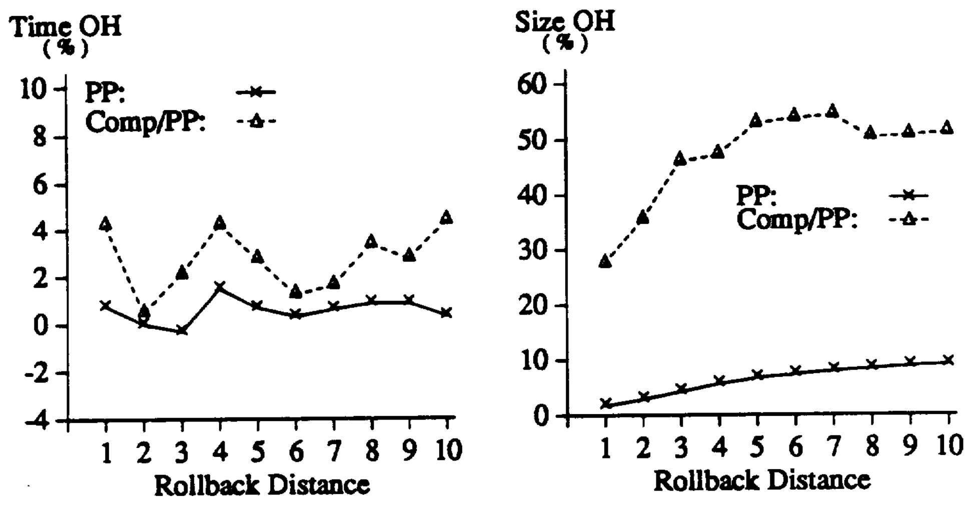
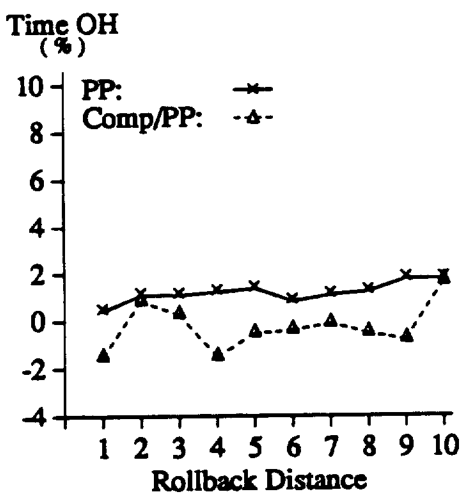


Figure 3.20: Run-time overhead and code size overhead: TBL.



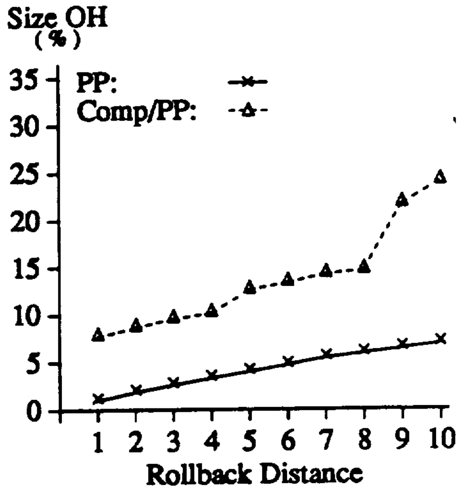


Figure 3.21: Run-time overhead and code size overhead: CCCP.

3.6 Concluding Remarks

A compiler-based and a compiler-assisted scheme have been described which support multiple instruction rollback with branch recovery. Hazard classification has proved useful in construction of the compiler-assisted scheme. Compiler transformations such as pseudo register renaming, node splitting, loop protection, and loop expansion were shown to be effective in resolving on-path and branch hazards with little performance impacts over resolving on-path hazards alone. The compiler-based approach yields performance impacts consistent with previous compiler techniques [29] and hardware techniques [12]. A hardware assisted scheme was introduced to resolve on-path hazards by maintaining a window of instruction read history.

The hardware assisted scheme introduces little performance impact and reasonable additional circuitry. Compiler techniques are used to resolve the remaining branch hazards with a modest increase in overall compile time. The performance measurements indicate that the compiler-assisted scheme can achieve lower performance impact than either a compiler-based scheme or a delayed write hardware scheme. It should be noted that the combined scheme applies only to the CPU and requires additional hardware to maintain the states of the program counter, program status word, etc.. The read buffer is twice the size of a delayed write buffer but avoids the requirement for bypassing and prioritization logic.

4. TRANSFORM ENHANCEMENTS

4.1 Introduction

In Chapter 3, a compiler-assisted multiple instruction rollback recovery scheme was presented. The scheme uses an operand read buffer to resolve on-path rollback hazards and uses compiler-driven data-flow manipulations to remove branch rollback hazards. This chapter presents enhancements to previously proposed compiler transformations used for hazard resolution [29]. These enhancements result in improved compile times and improved application run times.

4.2 Node Splitting

4.2.1 Iterative node splitting algorithm

As discussed in Chapter 3, node splitting breaks equivalence relationships which would prevent pseudo register renaming, i.e., given $h_o(i,j,x)$ or $h_b(i,j,x)$, node splitting forces

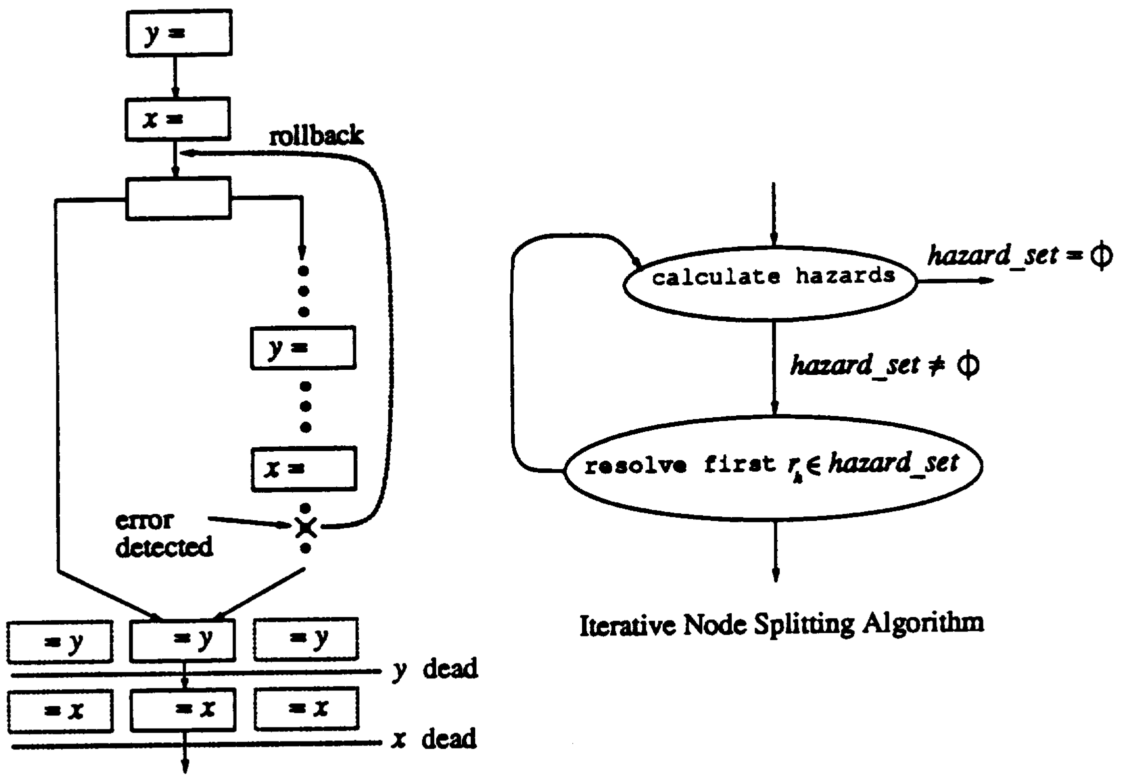


Figure 4.1: Iterative node splitting algorithm.

$equiv(i, j) = 0$. Figure 4.1 shows an example code sequence requiring splitting and an iterative node splitting algorithm.

The code segment of Figure 4.1 contains two branch hazards. The first hazard involves pseudo register x and the second involves pseudo register y. When two definitions of a hazard variable reach a node in which the hazard variable is live, the node is split. In this case, node splitting to resolve the hazard variable x also resolves the hazard variable y. This implies that the hazard set should be recalculated after splitting takes place for each hazard register. Previous node splitting algorithms used this iterative algorithm to avoid unnecessary node splitting [29].

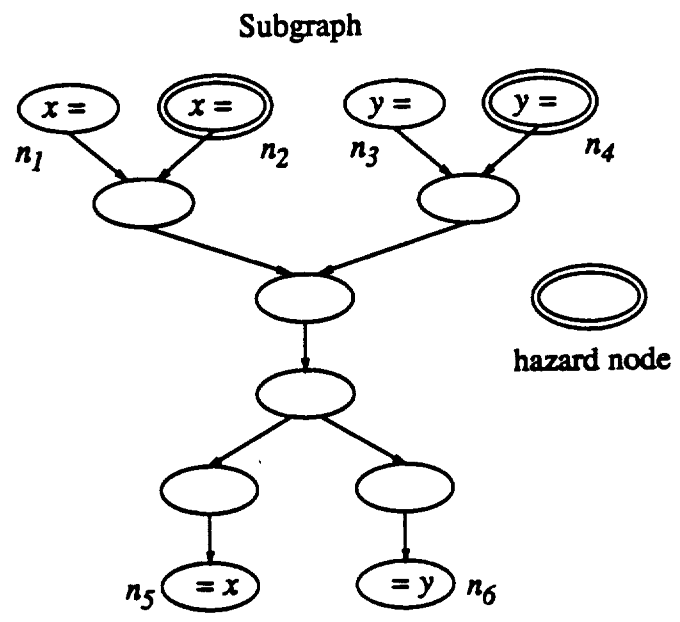


Figure 4.2: Node splitting: original subgraph.

Figures 4.2 through 4.4 demonstrate the effect of the iterative node splitting algorithm on an example subgraph. Node splitting relative to hazard variable x ensures that the definition of x in node n_1 and the definition of x in node n_2 do not both reach the same use of x in node n_5. Node splitting relative to y ensures that the definition of y in node n_3 and the definition of y in node n_4 do not both reach the same use of y in node n_6.

Figure 4.3 shows the subgraph after splitting relative to hazard variable x, and Figure 4.4 shows the subgraph after splitting relative to hazard variables x and y. Although the iterative algorithm was initially intended to prevent excessive node splitting, this example demonstrates that excessive node splitting is still possible. Figure 4.5 shows an optimal subgraph which resolves both hazards with less splitting than produced by the iterative algorithm.

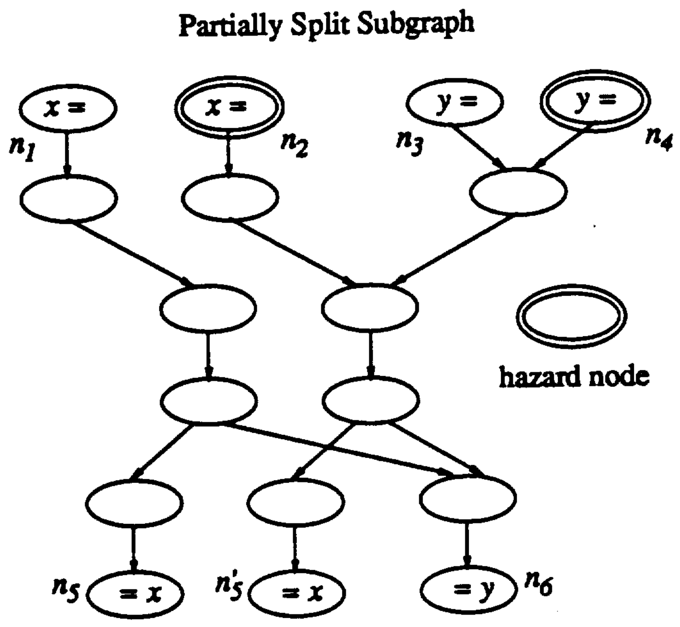


Figure 4.3: Node splitting relative to hazard variable x.

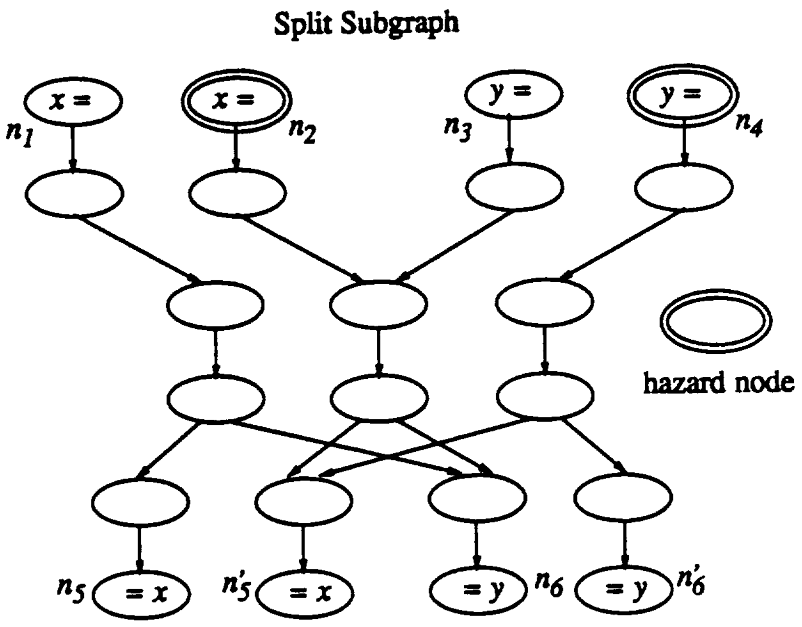


Figure 4.4: Node splitting relative to hazard variables x and y.

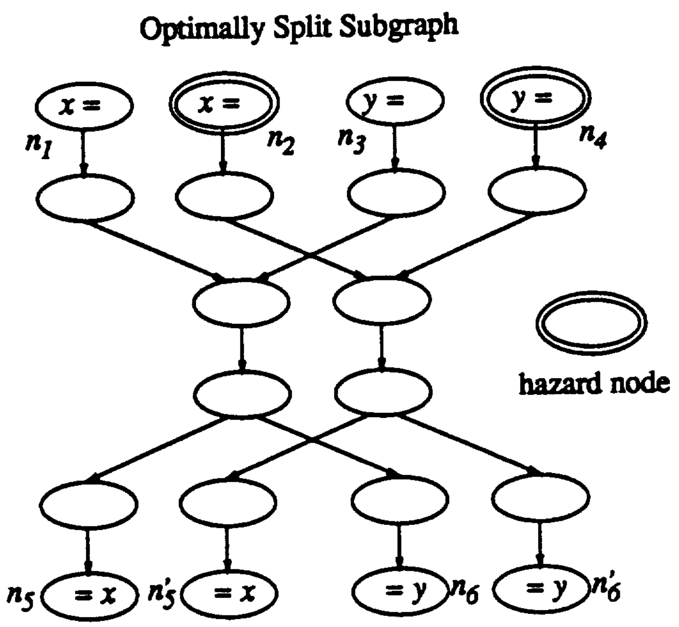


Figure 4.5: Optimal node splitting relative to hazard variables x and y.

4.2.2 Conflict definition

To ensure minimal splitting, a new node splitting algorithm is developed using the concept of conflicting parents. Given a CFG with back edges disabled, let H_{nodes} represent the set of all hazard hazard nodes[1] present in a CFG with back edges disabled. A conflict exists between node n's parent nodes, p_a and p_b, if

- $m \in H_{nodes} \cap reaching\_out(p_a)$ for some m, and
- $l \in reaching\_out(p_b)$ for some $l \neq m$, and
- $def(m) = def(l)$, where $def(m) \in live\_in(n)$

[1]A hazard node n is defined as the node representing I_n in $G(V, E)$ given the existence of either $h_o(n, m, x)$ or $h_b(n, l, x)$. See Chapter 3 for notation details.

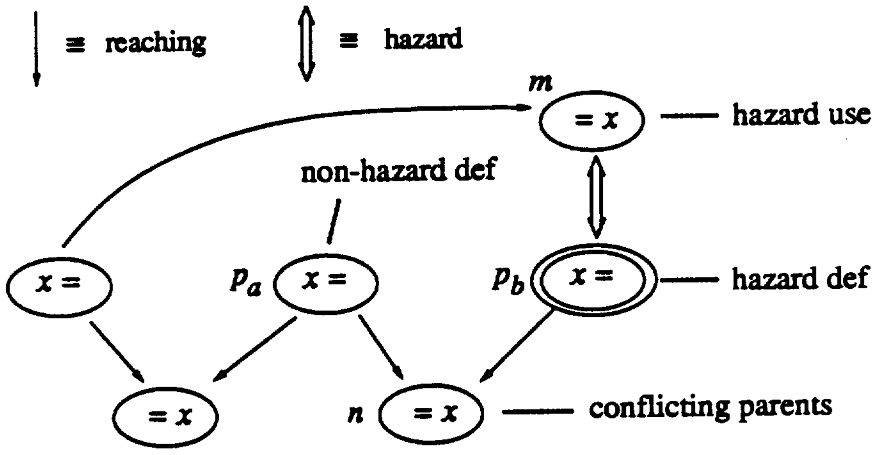


Figure 4.6: Conflict definition.

Any node with one or more conflicting parents must be split. Note that parent conflicts are not based on a single hazard variable.

Figure 4.6 illustrates the conflict definition. Double arrows represent a hazard pair, where x is defined in node p_b, and after rollback, is used prior to re-definition in node m. Single arrows represent reaching definitions and show that if variable x in node p_b is renamed to z, then x in node m would ultimately require renaming to z. Node n, of Figure 4.6, has conflicting parents p_a and p_b. Ensuring that node n does not have conflicting parents enables resolution of the hazard using variable renaming.

4.2.3 Node splitting using graph coloring

Given the definition of conflicting parents, the node splitting strategy for a particular node is to group the parents of that node such that elements within a group do not conflict. Each group becomes a parent node for a duplicate of the original node. For example, if node n has six parent nodes and these nodes can be organized into three

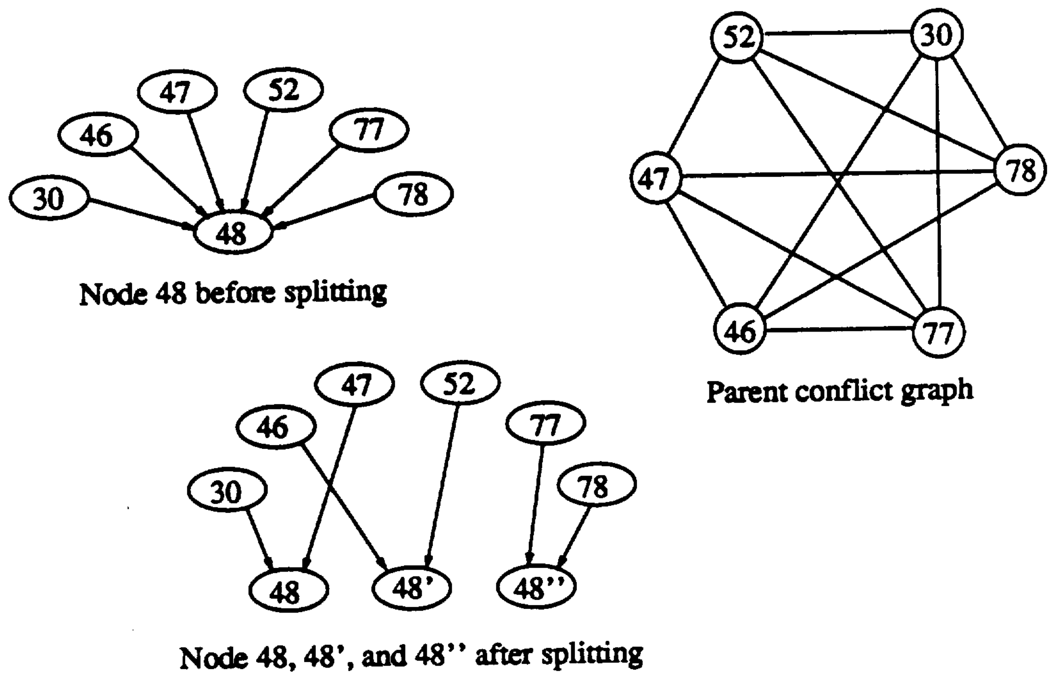


Figure 4.7: Node splitting using graph coloring; QSORT.

nonconflicting groups, then only three total copies of n are required. Figure 4.7 shows node 48 from *Lcode* emitted by the IMPACT compiler for the QSORT application shown in Table 3.1 (p. 46). Node 48 has six parent nodes prior to splitting. These nodes can be arranged in a parent conflict graph, where each arc of the graph represents two nodes which conflict. Establishing groups can be achieved by finding the minimum coloring of the parent conflict graph, i.e., coloring the nodes such that no two nodes connected by an arc have the same color. For the example shown in Figure 4.7, three colors are sufficient to cover the parent conflict graph, resulting in the splitting of node 48 into nodes 48, 48' and 48".

Determining whether a graph is k-colorable is NP-complete in general: however, linear-time heuristics have been developed. Figure 4.8 shows the heuristic used for col-

```
Given parent_conflict_graph(V,E)

int color_graph(parent_conflict_graph)
graph_struct parent_conflict_graph;
{
   int i, j, k;
   graph_struct temp_graph;
   node_struct v[MAX_PARENTS];

   temp_graph = parent_conflict_graph;
   while (temp_graph != null) {
      v[i] = min_degree_node{all V in temp_graph(V,E)};
      k = degree_of(v[i]);
      delete v[i] and all edges of v[i] from temp_graph(V,E);
      ++i;
   }
   ++k;
   for (j=i; j<0; --j)
      color v[j] in parent_conflict_graph with one of k colors;
   return(k);
}
```

Figure 4.8: Parent conflict graph coloring heuristic.

oring the parent conflict graph. The heuristic is a modified version of an algorithm used for register allocation [28]. The algorithm selects the node with the fewest edges, records the node, and then removes it from the parent conflict graph. This process continues until the parent conflict graph is empty. If node n has the fewest edges (i.e., k edges), then at least $k+1$ colors are required to color the graph. One color is required for node n and k colors are required for the nodes connected to n. Node n can be removed leaving a subgraph. Once the subgraph is colored with k colors, then node n can be colored with the remaining color. The reverse order of the node recordings can therefore be used to color the parent conflict graph.

```
disable backedges;
calculate hazards;
for all nodes n, in a topological traversal, do {
   compute reaching_in set for n from reaching_out set of n's
    parents;
   build a parent conflict graph (PCG);
   return to k the # of colors required to color the PCG;
   color PCG with k colors;
   delete node n;
   create k-1 duplicates of n;
   use coloring to connect parent nodes to n and duplicates;
   if n was a hazard node, add duplicate nodes to hazard list;
   compute reaching_out set for n and duplicates;
 }
enable backedges;
```

Figure 4.9: One-pass node splitting algorithm.

4.2.4 One-pass node splitting algorithm

Both *live_in(n)* and *reaching_out(n)* analyses are required to identify conflicting parent nodes. A one-pass node splitting algorithm becomes possible by precalculating *live_in* and H_{nodes}, and, then, beginning with the root node, splitting in a topological traversal of the CFG. The one-pass node splitting algorithm is shown in Figure 4.9. A topological traversal ensures than when processing node n, all ancestors of n have been processed and no descendants of n have been processed. This latter case ensures that the presplit calculation of *live_in(n)* can be used for parent conflict identification when processing a given node. Unlike *live_in(n)*, *reaching_out(n)* is affected by the splitting of ancestor nodes. Since *reaching_out(n)* is based solely on node n and its ancestors, *reaching_out(n)* can be calculated as node splitting proceeds. If a hazard node is split, each duplicate of the node must be added to the H_{nodes} set. Since the root node does not have conflicting parents, a topological traversal of the CFG using the graph coloring node splitting technique ensures that no node in the resulting graph has conflicting parents.

Table 4.1: Node splitting algorithm comparisons: COMPRESS.

- Iterative Algorithm run time = 614.0 seconds
- One-pass Algorithm run time = 20.3 seconds
- Speedup = 30.2

Orig. Node Cnt.	Iterative Alg.	% Increase	One-pass Alg.	% Increase
547	601	9.9	566	3.5
461	499	8.2	496	7.6
144	147	2.1	147	2.1
181	209	15.5	207	14.4
75	80	6.7	80	6.7
21	28	33.3	27	28.6
45	79	75.6	48	6.7

Table 4.1 illustrates the improvement of the one-pass node splitting algorithm over the iterative algorithm for the COMPRESS application. The COMPRESS application was compiled on a SPARCserver 490 using the IMPACT C compiler [32] and a rollback distance of 10. Node count values represent pseudo (*Lcode*) instructions created by the IMPACT C compiler before and after splitting. Seven of the 14 COMPRESS functions which required splitting are listed. Algorithm run times represent the overall compile times given each node splitting algorithm.

Table 4.1 shows a marginal overall code growth reduction for the one-pass algorithm. Although one function demonstrated a significant code growth reduction (6.7% down from 75.6%), the function is small and has minimal effect on the overall code size. The improvement in compile-time of the one-pass algorithm is more dramatic, resulting in a speedup of 30.2. The compile-time improvement can be explained as follows. If 60 hazard variables are present in a given function, the iterative algorithm may require up to 60 passes through the CFG of that function, including 60 data-flow analysis and hazard calculations. Although processing a given node in the one-pass algorithm is slightly more complex, a single data-flow analysis calculation and a single pass through the CFG are sufficient.

4.3 Loop Protection

As discussed in Chapter 3, due to the significant code growth potential of loop expansion and the infrequency of branch hazards traversing loop back edges, all such hazards are left to be resolved by the post-past transformation. Node splitting therefore becomes the dominant pseudo level hazard removal transformation, eliminating the need for loop protection to aid in loop expansion. A new loop protection algorithm, aimed at maintaining loop integrity only during node splitting, is now developed.

A similar approach to the conflict definition of Section 4.2 can be used to determine if a loop has to be protected. The header node of loop l will be defined as *hdr_node(l)*. The requirement to split loop l due to hazard node n will be defined as *split(l,n) = 1*, and this occurs given the following conditions:

- $n \in H_{nodes} \cap reaching\_in(hdr\_node(l))$ for some n, and
- $m \in reaching\_in(hdr\_node(l))$ for some $m \neq n$, and
- $def(n) = def(m)$, where $def(n) \in live\_in(hdr\_node(l))$

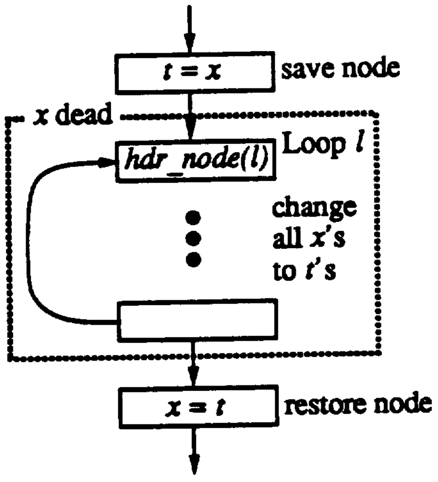


Figure 4.10: Loop protection from hazard variable x.

Given the one-pass node splitting algorithm of Section 4.2 and no loop protection requirement for loop expansion, loop l will require protection from hazard node n if *split(l,n)* = *1*. Figure 4.10 demonstrates how loop l is protected from hazard node n, where *def(n)* = x. The loop header will not be split since $x \notin live\_in(hdr\_node(l))$. A loop protection algorithm, referred to as static loop protection, is shown in Figure 4.11, where *outer(l)* indicates the outer loop of loop l. The static loop protection algorithm is executed prior to node splitting. Each loop is processed twice: the first time to record the hazard variables for which the loop must be protected, and the second time to protect the loop.

4.3.1 Dynamic loop protection

Since the static loop protection algorithm shown in Figure 4.11 is executed prior to node splitting, it does not predict loop header splits that result from new hazard nodes created during ancestor splitting. Given the topological traversal of the one-pass node

```
disable backedges;
calculate hazards;
for all loops l, from outer loops to inner loops, do {
   for each hazard node n, in hazard_node_set, do {
      if (split(l,n) == l), do {
         if (n is not in loop_hazard_set(outer(l))), do {
            add n to loop_hazard_set(l);
         }
      }
   }
}
for all loops l, do {
   for all nodes n in loop_hazard_set(l), do {
      protect loop l for hazard variable def(n);
   }
}
enable backedges;
```

Figure 4.11: Static loop protection algorithm.

splitting algorithm, loop protection can be performed dynamically when a loop header is encountered.

The set containing all of the nodes in loop l is defined as *loop_nodes*(l). Protection of loop l relative to hazard variable x can affect *live_in*(n) for all $n \in$ *loop_nodes*(l). Every exit node of loop l in which x is live will have a restore node placed between it and its children nodes as shown in Figure 4.10. Changes to *live_in*(n) for $n \in$ *loop_nodes*(l) is therefore contained to loop l. Figure 4.12 shows a simple dynamic loop protection algorithm which includes updating of the presplit *live_in* analysis result.

The creation of restore nodes during loop protection can occasionally result in additional branch hazards. Static loop protection ensures that these additional hazards are identified and removed by the node splitting algorithm. Dynamic loop protection can create branch hazards that are not guaranteed to be identified and removed by the node splitting algorithm. The approach used for branch hazard removal therefore is

```
dyn_loop_protect(l,x)

   insert save node s ahead of node hdr_node(l);
   copy live_in(hdr_node) into live_in(save_node);
   copy s into loop_nodes(l);
   for all exit nodes e of loop l, do {
      insert restore node r;
      copy live_in(e) into live_in(r);
      copy r into loop_nodes(l);
     }
   for all nodes n in loop_nodes(l), do {
      if (x in live_in(n)), do {
         delete x from live_in(n);
         add t to live_in(n);
        }
     }

return;
```

Figure 4.12: Dynamic loop protection algorithm.

to: 1) execute static loop protection, 2) use dynamic loop protection within the node splitting algorithm, and 3) re-execute the node splitting algorithm if previous dynamic loop protection resulted in additional hazards. Experimental results have shown that re-execution of the node splitting algorithm is rarely necessary. Two executions of the node splitting algorithm were sufficient to remove all required branch hazards for the eleven applications shown in Table 3.1 (p. 46).

4.4 Performance Enhancements Through Profiling

4.4.1 Post-pass transformation versus loop protection

After hazards are removed by the compiler, some hazards remain and must be removed using the post-pass transformation. Previous post-pass transformations used nop insertions to increase all antidependency distances to $> N$ [29]. Since nop insertion can be

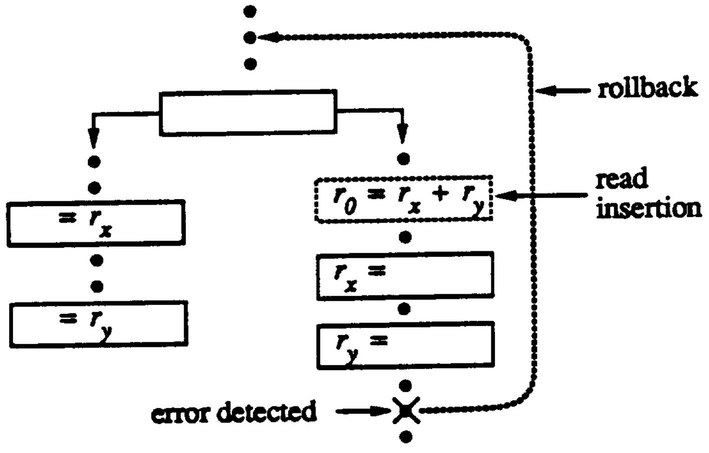


Figure 4.13: Post-pass hazard removal using read insertion.

costly to performance, previous compiler transformations removed all hazards possible, leaving only unresolvable hazards to be removed by the post-pass transformation.

In Chapter 3, a new post-pass transformation was introduced in which nop insertion was replaced by read insertions as the primary hazard removal technique. As illustrated in Figure 4.13, up to two branch hazards can be removed by a single read instruction. The new post-pass transformation is very efficient and in some cases can resolve branch hazards with less performance impact than pseudo-level transformations. Figures 4.14 and 4.15 show performance overhead comparisons between compiler-driven data-flow manipulations and the new post-pass transformation. *Comp/PP* indicates that hazards are resolved by the compiler where possible, with the remaining hazards being resolved at the post-pass level. *PP* (post-pass) indicates that compiler transformations have been disabled and that all hazards are removed at the post-pass phase. Performance evaluations were obtained using the methods described in Chapter 3. The TBL application is a table formatting preprocessor for *nroff*, a text processing facility. PUZZLE is a game.

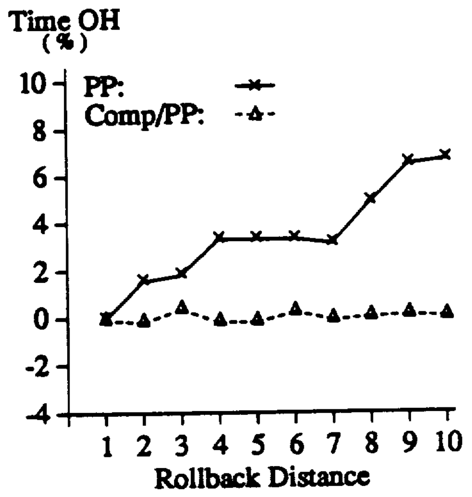


Figure 4.14: Run-time overhead: PUZZLE.

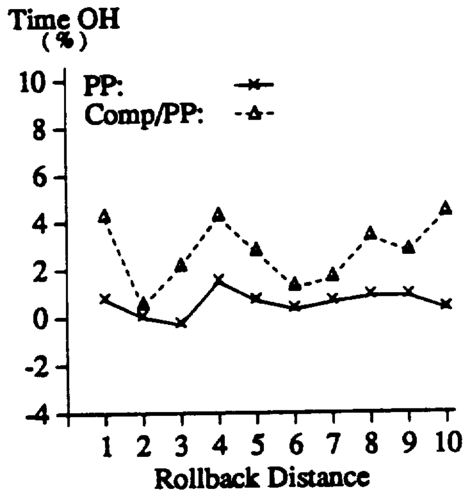


Figure 4.15: Run-time overhead: TBL.

For the PUZZLE application, compiler transformations produce better performance than the post-pass transformation alone. For the TBL application, shown in Figure 4.15, using the post-pass transformation to remove all hazards produces slightly better performance than the combination of compiler and post-pass transformations. Hazard elimination via read insertion introduces a guaranteed but small performance impact due to the longer instruction path length. As demonstrated in Figure 4.14, pseudo register renaming can eliminate hazards without impacting performance when loop protection is infrequent. The save/restore operations of loop protection can result in more performance impact than read insertion when loop protection is frequent, as demonstrated in Figure 4.15.

Figure 4.16 illustrates the potential effect on performance given the following two types of hazard removal: 1) hazard removal using register renaming that results in loop protection, and 2) hazard removal using read insertion. If the protected loop is executed 20 times and the hazard instruction is executed two times, loop protection would require the execution of 40 additional instructions, where read insertion would require the execution of only two additional instructions. If the loop and hazard instruction execution frequencies were reversed, then read insertion would produce more performance impact than loop protection. As shown in Figure 4.16, profiling data can be used to aid in loop protection decisions.

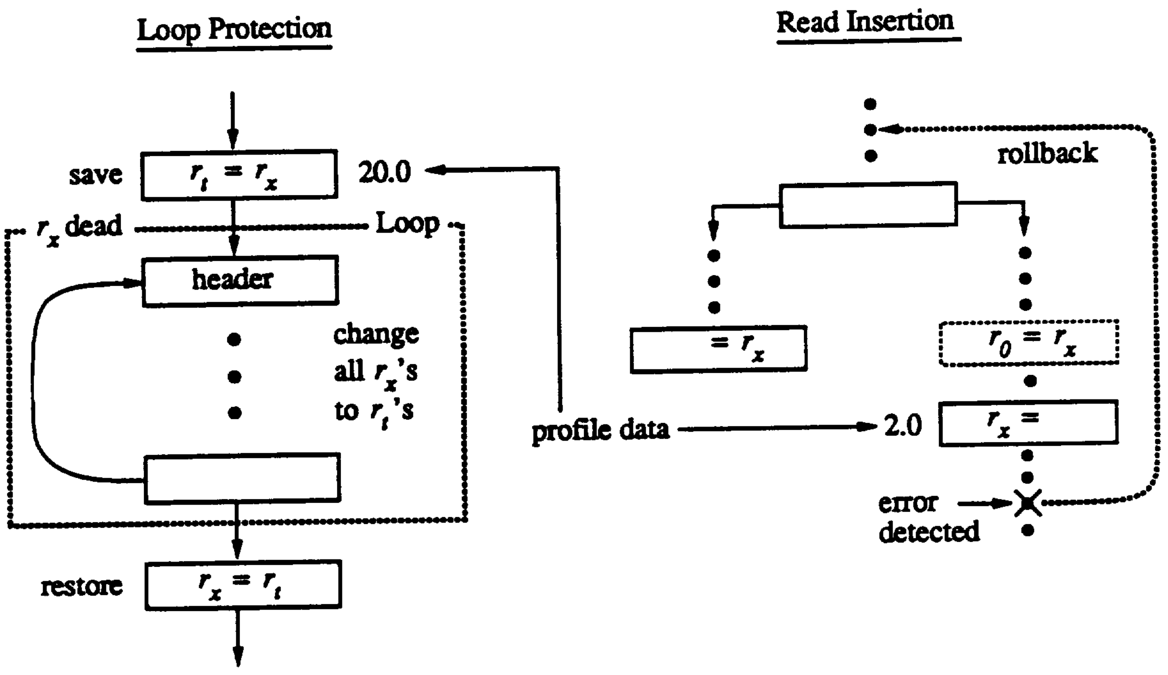


Figure 4.16: Loop protection versus read insertion.

4.4.2 Profiling effectiveness

Profiled data was included in the pseudo-level transformations of Chapter 3. The profile data is comprised of both dynamic profile sampling and static prediction. The static prediction is used as a supplement for areas of the application code that are unexecuted during profile sampling. For static profiling, a loop is assumed to iterate ten times. Inner loops, therefore, iterate multiples of 10 times depending on the depth of loop nesting. All loop header nodes and hazard nodes are assigned weights based on the profile data.

Protection of loop l due to hazard node n_h is required based on the following condition: if $n_h\_weight > 3 * (hdr\_node(l)\_weight)$, then protect loop l. The constant 3 adjusts the weights to account for both direct and indirect loop protection costs. Direct loop protection costs result from the save/restore instruction pair shown in Figure 4.16.

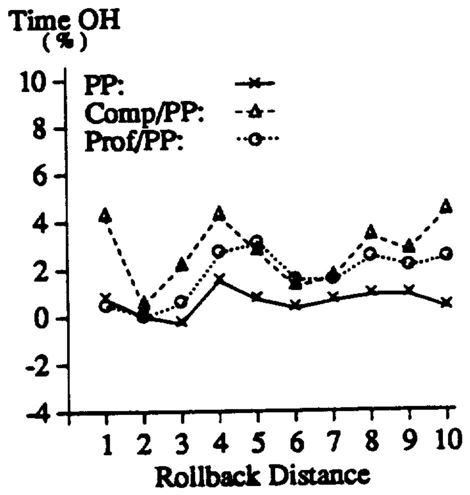


Figure 4.17: TBL: profile data used for loop protection decisions.

Indirect loop protection costs result from: 1) an increased number of hazards which in turn required more node splitting and more loop protection, and 2) increased register usage due to the save/restore instructions which can result in additional register spills. Figure 4.17 shows the run-time overhead for the TBL application with rollback distances from 1 to 10. *Prof/PP* indicates that profiling data was used in loop protection decisions.

The results show that the use of profile data can improve application performance by postponing some hazard resolutions until the post-pass phase. Using profile data to aid in loop protection decisions did not produce performance equal to that for the post-pass transformation, for the TBL application. As an extension to this work, profile data can be used to aid in register allocation. As discussed in Chapter 3, hazards that are present after pseudo register renaming are resolved by adding arcs to the register allocation dependency graph. These additional constraints can cause additional register pillage and impact performance. Similar techniques to those developed for loop protection can be used to enhance register allocation decisions.

4.5 Summary

In this chapter, compiler transformations used for the removal of branch hazards have been enhanced, resulting in reduced compile times and increased application performance. A one-pass node splitting algorithm was developed which uses the concept of conflicting parents to reduce the number of duplicate nodes required. A graph coloring heuristic was developed to connect split nodes to parents. For the COMPRESS application, the one-pass node splitting algorithm resulted in marginally reduced code growth and a compile-time speedup of 30 over previous iterative node splitting algorithms. Similar techniques used in the node splitting algorithm were used to develop a one-pass static loop protection algorithm. Due to the splitting of hazard nodes, it was shown that the static loop protection algorithm did not predict all loop header splitting. A dynamic loop protection algorithm was developed which allows loops to be protected as they are encountered by the node spitting algorithm. It was also shown that read insertion used in the post-pass hazard removal phase could produce less performance impact than pseudo register renaming when the latter results in loop protection. Profiling was shown to be effective in making better loop protection decisions, resulting in improved overall application performance.

5. READ BUFFER SIZE REQUIREMENT

5.1 Introduction

In Chapter 3, a compiler-assisted approach to multiple instruction rollback in which a *read buffer* of size $2N$ (where N represents the maximum instruction rollback distance) was used to aid in hazard removal. This chapter examines the size and design of the read buffer. A practical lower bound and average size requirement for the read buffer are established by modifying the design to save only the data required for rollback. The study measures the effect on the performance of ten application programs using six read buffer configurations with varying read buffer sizes. Two alternative configurations are shown to be the most efficient and differed depending on whether split-cycle-saves are assumed.

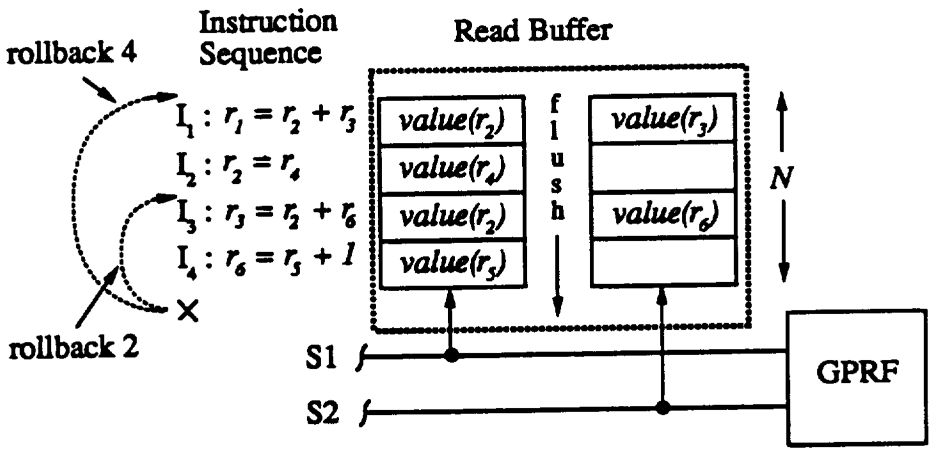


Figure 5.1: Read buffer of size $2N$.

5.2 Read Buffer Configurations

Given a read buffer configuration as shown in Figure 5.1, rollback is accomplished by first flushing the read buffer back to the general purpose register GPRF in the reverse order of which the values were saved. Figure 5.1 shows the two FIFO read buffers above the source 1 (S1) and source 2 (S2) buses to better illustrate the buffer's content given the instruction sequence shown. As long as the depth of the dual FIFO read buffers are N, redundant copies of the appropriate register values (denoted $value(r_x)$) are available to restore the register file given a rollback of $\leq N$.

The read buffer size requirement of $2N$ is the worst case. The buffer maintains the last N register reads from the GPRF, assuring data redundancy for all values required. The read buffer may also save data which is not required during rollback. Register reads that must be saved can be determined at compile time. If this information is added to the instruction encoding (e.g., as an extra bit field for source 1 and for source 2), then the

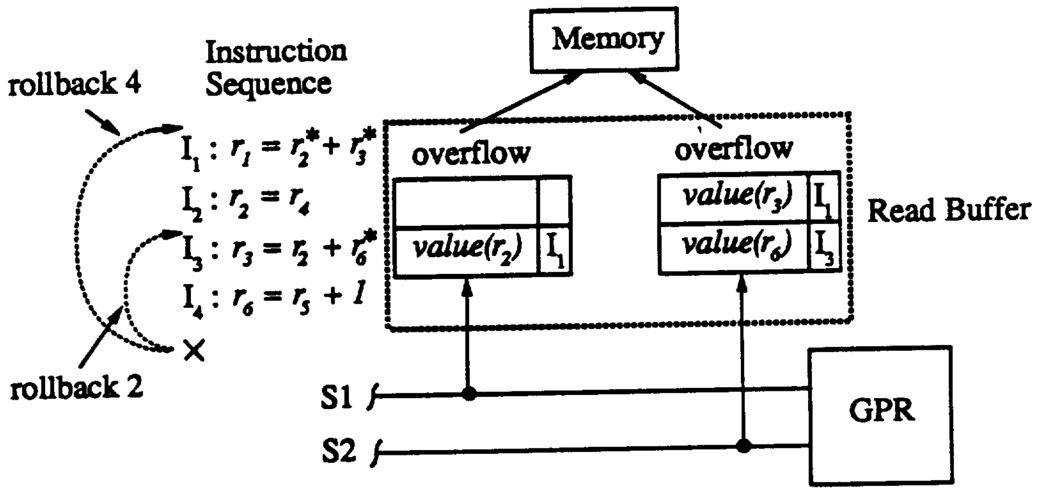


Figure 5.2: Read buffer of size < $2N$.

read buffer can be designed to save only those values required. As long as the required values are maintained for N cycles, a less than $2N$ read buffer size design is possible.

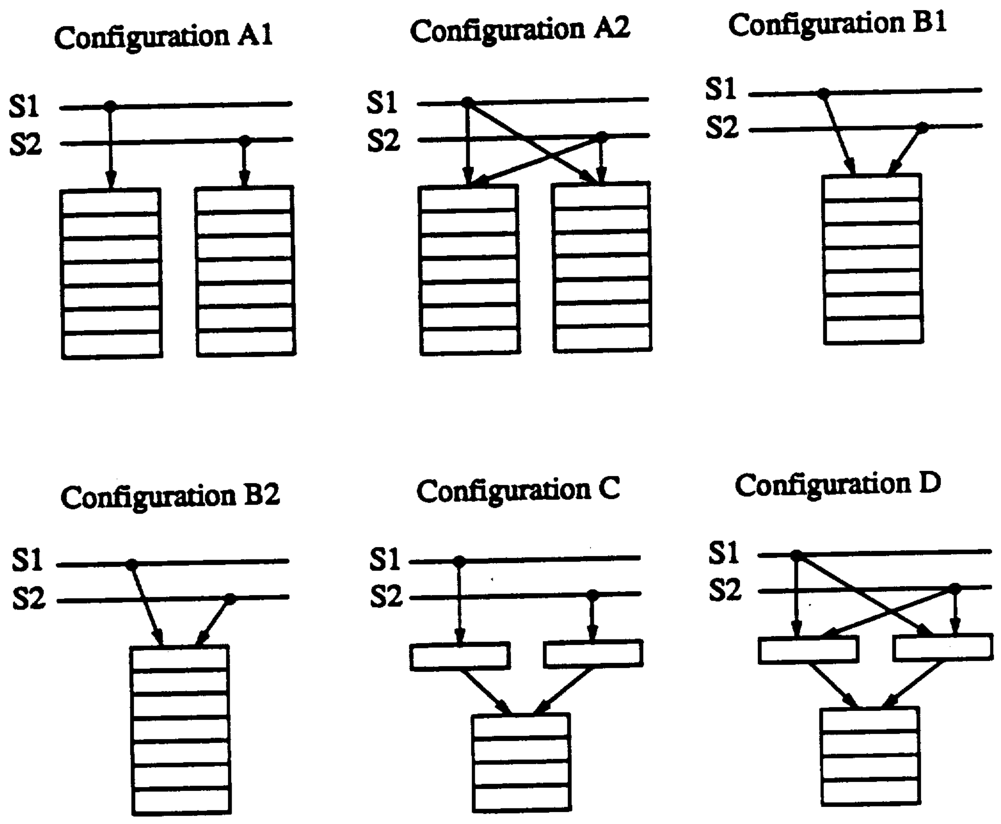
Figure 5.2 illustrates a case in which all register reads do not have to be placed in the read buffer. The registers required to be saved are marked with an "*." Since only the required values are saved, the read buffer total size can now potentially be less than N. In this case, however, the instruction count must also be saved so that the value can be maintained for at least N cycles. In the event that the read buffer overflows, the oldest value in the buffer must be pushed to memory and a record kept so that during rollback the value can be retrieved from memory. Given a dual FIFO depth of M, memory would serve the function of the remaining $N - M$ of the two FIFOs. This read buffer design reduces the buffer size while introducing potential performance impacts due to buffer overflows.

A key to the evaluation of a given read buffer design is the set of assumptions made relative to overflow handling. For example, if a memory store buffer were assumed, there would be no stall if a single FIFO overflowed and the store buffer was available, given that the current instruction were not a store. However, if the store buffer were full or if the current instruction were a store, then a stall would occur. The problem with including a store buffer in the model is that the performance impact measured would depend on the store buffer size, clouding the performance impact due to the read buffer alone. The same difficulty arises if a cache is included in the model.

It is assumed in this evaluation that a read buffer overflow will always cause a stall of one cycle. If both FIFOs overflow, a stall of two cycles will occur. This simplifying assumption is pessimistic relative to a store buffer which may have empty locations, while optimistic relative to a full store buffer requiring a write to cache. These assumptions guarantee that all measured performance impacts are directly due to changes in the read buffer size or configuration.

5.2.1 Read buffer designs

The most straightforward design for the read buffer is that of configuration A1, shown in Figure 5.3. The obvious problem with configuration A1 is that if the FIFO connected to S1 is full and the current S1 value must be saved, a stall occurs due to overflow even though the FIFO connected to S2 may have an available entry. Configuration A2 in Figure 5.3 resolves this inefficiency by giving both S1 and S2 access to either FIFO.



◇ Configuration B1: Can store buses S1 and S2 simultaneously.

◇ Configuration B2: Must stall on second store to single buffer.

◇ Configurations C & D: Assumes stall on second store to single buffer.

Figure 5.3: Read buffer configurations.

Configuration B1 also resolves the inefficiency of configuration A1 by having a single FIFO with both S1 and S2 connected to it. Configuration B1 assumes that the S1 value and the S2 value can be saved within the same cycle. This would be possible if the S1 value is saved during the first half of the cycle and the S2 value is saved during the second half of the cycle. This split-cycle-save assumption is consistent with the design of register files which write back during the first half of the cycle and read during the second half of the cycle [34].

Configuration B2 is identical to configuration B1 except that two saves during the same cycle are not permitted. If two saves are required during the same cycle (e.g., an instruction of the form: $r_x = r_y^* + r_z^*$), then a stall to save the second value occurs.

Configuration C attempts to lessen the impact due to the bottleneck in configuration B2 by adding two single level queues between the S1 and S2 buses and the single FIFO. Configuration C can absorb one simultaneous save, processing the first in the current cycle and the second in the next cycle assuming the next instruction does not also require a simultaneous save. Configuration D extends configuration C to allow both S1 and S2 access to either queue.

5.3 Application Program Execution and Read Buffer Simulation

5.3.1 Simulation approach

The read buffer is simulated at the instruction level. Prior to each instruction execution, a procedure is called to update the read buffer model. Parameters such as which

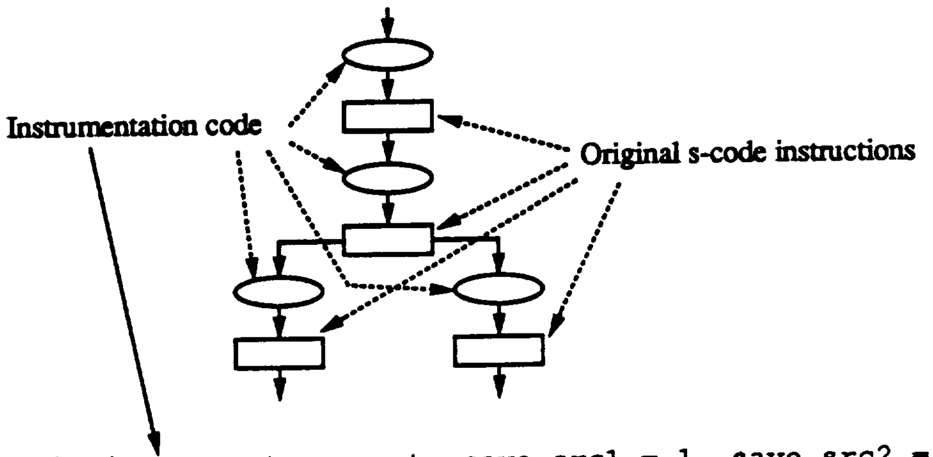
C-2

register reads to save and instruction type are passed to the simulation program. The drawbacks to this approach are the code growth in the original application program and the reduction in application run time.

The instructions inserted to branch to the simulation procedure prior to each original application instruction cannot be added in the high level language. If this were done, the one-to-one correspondence between original instructions and simulation procedure calls would be lost. Also the simulation procedure cannot be permitted to affect the original application by changing register assignments, live ranges, etc. For this reason, calculation of hazards and subsequent determination of which register reads should be saved are performed at the s-code level (after register assignment) and the appropriate s-code level instructions inserted prior to each original s-code instruction of the application program.

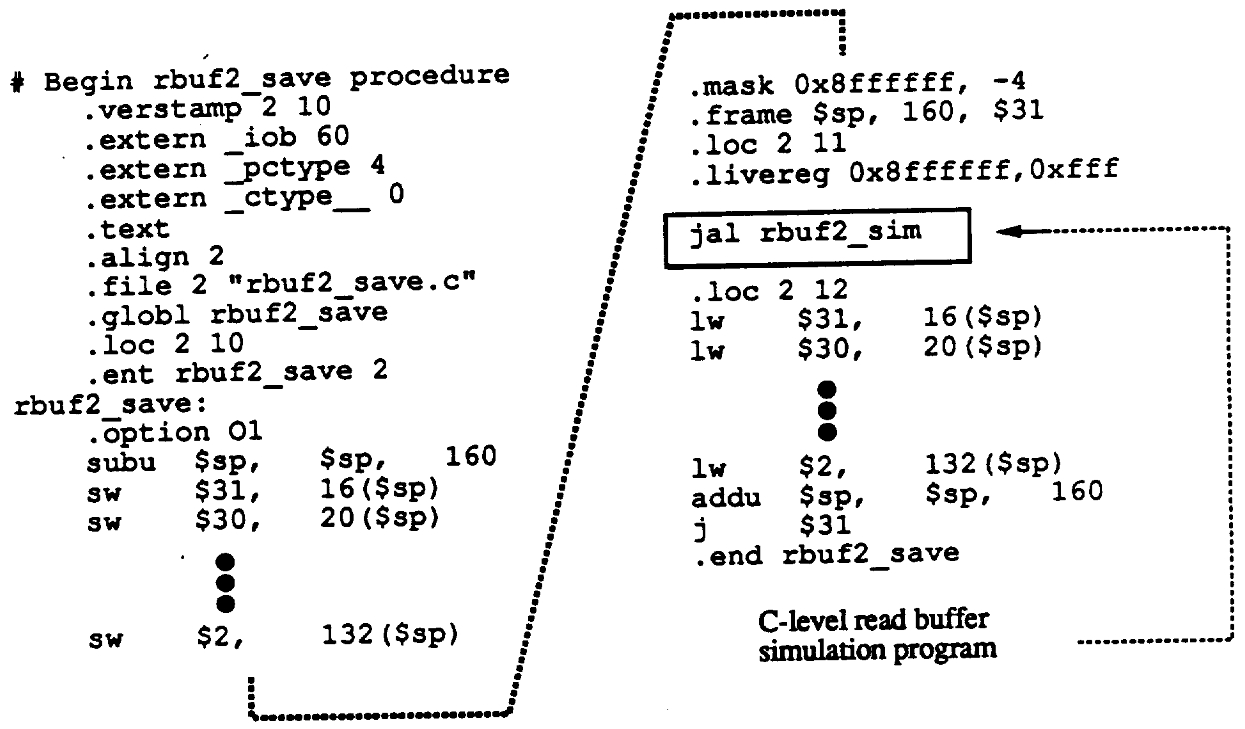
5.3.2 Implementation

To minimize the application code growth, a simple s-code sequence (written for the MIPS 2000/3000 architecture) shown in Figure 5.4 is inserted prior to each instruction. The code segment pushes register 31 and register 4 on the stack, loads register 4 with information relative to the saving of S1 or S2 for this particular instruction, calls *rbuf2_save*, and then upon return from *rbuf2_save* restores registers 31 and 4. Register 31 is used as a return address during procedure calls and therefore will be corrupted. Register 4 is used to pass parameters in the MIPS compiler convention.



```
# Begin instrument segment: save_src1 = 1, save_src2 = 0
    subu  $sp,     28
    sw    $31,     20($sp)
    sw    $4,      24($sp)
    li    $4,      1   <———  directs read buffer to
    jal   rbuf2_save          save source 1 value
    lw    $31,     20($sp)
    lw    $4,      24($sp)
    addu  $sp,     28
# End instrument segment.
    addu  $25,     $23,     $8   <——  original instruction
```

Figure 5.4: Instrumentation code segment.



```
# Begin rbuf2_save procedure
    .verstamp 2 10
    .extern _iob 60
    .extern _pctype 4
    .extern _ctype__ 0
    .text
    .align 2
    .file 2 "rbuf2_save.c"
    .globl rbuf2_save
    .loc 2 10
    .ent rbuf2_save 2
rbuf2_save:
    .option O1
    subu  $sp,    $sp,    160
    sw    $31,    16($sp)
    sw    $30,    20($sp)
          ●
          ●
          ●
    sw    $2,     132($sp)
```

```
    .mask 0x8ffffff, -4
    .frame $sp, 160, $31
    .loc 2 11
    .livereg 0x8ffffff,0xfff
    jal rbuf2_sim
    .loc 2 12
    lw    $31,    16($sp)
    lw    $30,    20($sp)
          ●
          ●
          ●
    lw    $2,     132($sp)
    addu  $sp,    $sp,    160
    j     $31
    .end rbuf2_save
```

Figure 5.5: *rbuf2_save* code segment.

The code sequence of Figure 5.4 only saves the two registers necessary to branch to a procedure. Prior to calling the read buffer simulation procedure, the remaining registers which are used must be saved. This was not done in the code segment of Figure 5.4 to limit application code growth. The code sequence, *rbuf2_save*, shown in Figure 5.5 conservatively saves all remaining registers on the stack. Both *callee* and *caller* saved registers are saved since the standard conventions are corrupted by the code insertion. The read buffer simulation procedure, *rbuf2_sim*, is called from the code segment shown in Figure 5.5. *rbuf2_sim* can now be modified and re-compiled without a corresponding modification to the application program or the two previous s-code segments.

Similar s-code segments handle initialization and summary calculations. The *initialization* procedure call is placed in the "*main*" module prior to the first instruction. The *summary* procedure calls are placed prior to all "*jal exit*" instructions in all modules and prior to the "*j $31*" instructions in the "*main*" module. Performance impact (% increase) is computed as

$$100 * \frac{stall\_cycles}{base\_cycles}$$

Stall cycles result from read buffer overflows. All instructions are assumed to require one cycle to complete in a pipelined architecture. This is a pessimistic assumption for performance impact measurement since load and branch delays would give the read buffer an extra cycle to handle an overflow. The assumption is again made to help isolate read buffer effects on performance from those of various delay slot filling strategies.

The hazard analysis transformation operates on the s-code emitted by the MIPS code generator of the IMPACT C compiler [32]. The transformation determines which register reads should be saved by the read buffer and inserts calls to the *initialization*, *simulation*, and *summary* procedures as described earlier. The resulting s-code modules are then compiled and run on a DECstation 3100. For the study, a rollback distance of 10 was selected. Given a rollback distance of 10, a read buffer size of 20 (for configurations A1, A2, and B1) will produce zero performance impact. Table 5.1 lists the ten application programs studied. *Size* is the number of assembly instructions emitted by the code generator, not including the library routines and other fixed overhead.

Table 5.1: Application programs: read buffer size study.

Program	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

5.4 Results and Analysis

5.4.1 Detailed analysis: QUEEN

Figure 5.6 shows changes in performance overhead (Cycles OH) for various read buffer sizes and configurations running the QUEEN application. Looking at Figure 5.6, configuration A1, it can be seen that significant performance impact is incurred even with a modest reduction in read buffer size. As can be seen from the other application runs, shown in Figures 5.8 through 5.16 (pp. 93 through 97), configuration A1 is consistently the least efficient of the six configurations studied.[1] This is due to the fact that the dual FIFO's are dedicated to a single source bus. In many cases saving S1 will cause an overflow because the S1 FIFO is full, even though there is room in the S2 FIFO. Configuration A1 does allow for simultaneous saves of S1 and S2, given sufficient room in each, but this feature does not compensate for the latter inefficiency. Configuration A2 demonstrates

[1]An efficient configuration is one with a low performance overhead given a small read buffer size.

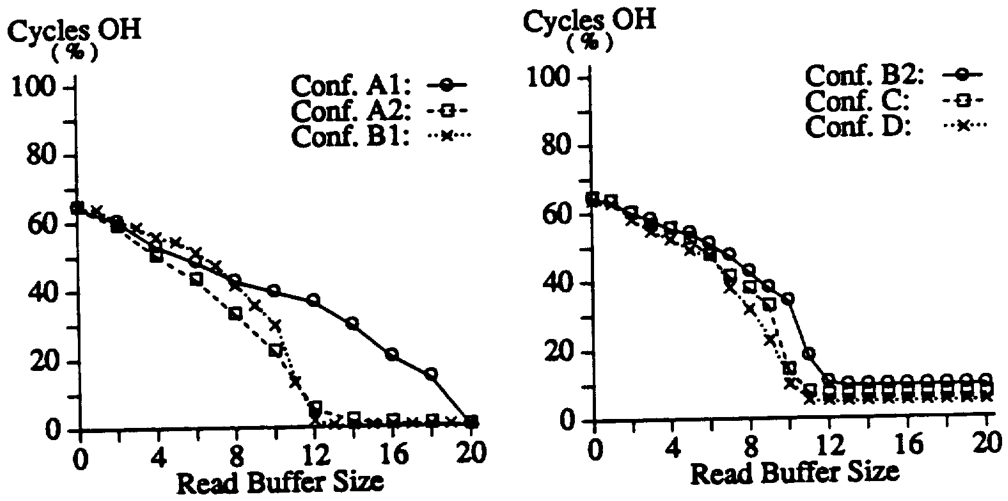


Figure 5.6: Cycle overhead: QUEEN.

the improvement gained by allowing either source bus access to either FIFO. Configuration B1 was the most efficient of the six configurations for the QUEEN application. In this configuration a total read buffer size of 13 would produce zero performance impact with a 35% reduction in read buffer size.

Comparing configurations A2 and B1 of Figure 5.6, it can be seen that configuration A2 out-performs configuration B1 at the smaller buffer sizes while B1 performs slightly better at the larger buffer sizes. Figure 5.7 illustrates the operation of read buffer configurations A2 and B1 given an example instruction sequence. Instruction operands that require saving are marked with an "*." For the instruction sequence shown, a read buffer size of two, and a maximum rollback distance of four, it can be seen that configuration B1 results in one extra overflow. In configuration A2, $value(r_b)$ is loaded into the S2 buffer and remains there during subsequent loads of the S1 buffer. After instruction I_4, $value(r_b)$ becomes invalid since a rollback of four instructions would be to I_2. A similar

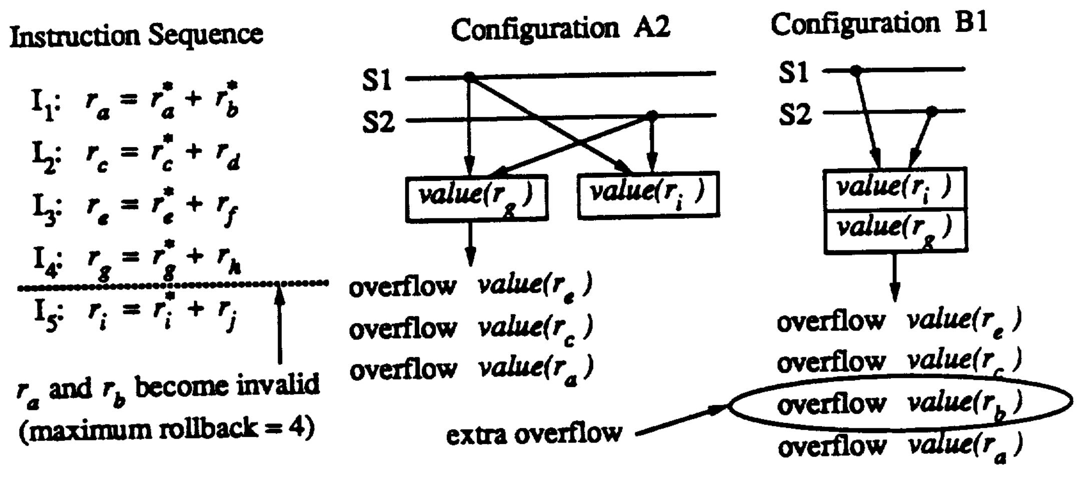


Figure 5.7: Read buffer configurations A2 and B1: buffer size = 2.

scenario is possible with configurations A1. Due to the arrangement of configuration B1, *value*(r_b) overflows before it becomes invalid, resulting in one extra overflow. As the buffer size increases, values in the B1 buffer have more time to become invalid before they reach the *head* position; extra overflows become less frequent.

Configuration B1 can also produce less overflows than configuration A2. When two operands require saving in the buffer of configuration B1, the *head* position and the *head* $-$ 1 position of the B1 buffer are checked to see if those positions contain valid data. When two operands require saving in the buffers of configuration A2, the *head* positions of the S1 buffer and S2 buffer are checked. Configuration A2 can require and extra overflow if one of its two *head* positions have valid data and one of the *head* $-$ 1 positions has invalid data. The advantage of configuration B1 (relative to configuration A2) becomes visible at large read buffer sizes where the previously mention disadvantage of configuration B1 (relative to configuration A2) diminishes.

This characteristic of configuration A2 versus configuration B1 is present in most of the application results. It should be noted that configuration B1 assumes that simultaneous saves of S1 and S2 can be handled within the same cycle. If this latter assumption is invalid, Figure 5.6, configuration B2, shows that no less than 9.4% performance impact is achieved regardless of the read buffer size. The "leveling off" of B2 is due to the bottleneck at the single FIFO entry point and not the depth of the FIFO. The flat part of the curve shows the percent of instructions requiring simultaneous saves of S1 and S2 in the QUEEN application.

Figure 5.6, configuration C, shows how a single level dual queue placed between the source bus and the single FIFO can alleviate some of the bottleneck effects. The dual queue can absorb a single simultaneous save of S1 and S2, distributing the saves over multiple cycles. A nonzero minimum performance overhead is still present due to cases in which the dual queue has not emptied before the next simultaneous save occurs.

Figure 5.6, configuration D, shows the results of an improved queue structure which permits saves from either bus into either queue. This configuration avoids stalls in some cases (e.g., S2 must be saved while the queue dedicated to S2 in configuration C is full and the other queue is empty). Configuration D also has a nonzero minimum performance overhead but gives better performance than configuration C.

The simulation results for QUEEN show that configuration A1 is the least efficient and that given the ability to do split-cycle-saves, configuration B1 is the most efficient. Without the split-cycle-save capability, configuration D is the best of the single FIFO

Table 5.2: Result Summary.

	RB_size		*OH_level (%)*	
Program	A2	B1	A2	B1
QUEEN	14	12	1.66	1.36
WC	10	8	0.00	2.54
QSORT	16	15	2.28	0.94
CMP	12	11	0.00	0.00
GREP	10	10	0.18	0.18
PUZZLE	10	9	2.87	0.32
COMPRESS	12	12	2.87	1.12
LEX	12	12	2.73	1.55
YACC	16	15	1.07	0.00
CCCP	12	12	2.34	1.74

designs resulting in a minimum performance overhead of 4.5%, and configuration A2 is the best of the dual FIFO designs resulting in a 1.7% performance overhead with a read buffer size of 14. For configurations B1, B2, C, and D, a total read buffer size of 13 is sufficient to maximize performance.[2]

5.4.2 Evaluation of all application programs

Results for the other nine application programs are similar to those for QUEEN and can be found in Figures 5.8 through 5.16 (pp. 93 through 97). The differences between the application results are the points at which the curve "levels off" (i.e., the buffer size) and, in the case of configurations B2 through D, at what level the performance overhead stabilizes. Table 5.2 summarizes measurements obtained for the ten applications given the two most efficient configurations, A2 and B1. It is assumed for this study that

[2]Two must be added to each read buffer size value in C and D to account for the queues.

minimal performance overhead can be tolerated as a result of read buffer size reduction. For this reason, configuration comparisons are made at read buffer size values which produce low values of performance overhead. Configuration A2 does not level off like configuration D and does not rapidly approach zero like configuration B1. For a better comparison of configurations A2 and B1, Table 5.2 gives the read buffer size value where the performance overhead value drops below 3%. The read buffer size value is referred to as *RB_size* and the performance overhead value is referred to as *OH_level*.

It can be seen from Table 5.2 that the read buffer size requirement is roughly the same, per application, regardless of the split-cycle-save assumption (i.e., comparing configurations A2 and B1). The size requirement is application dependent - from 8 for WC, to 15 for QSORT and YACC. The measurements show that a considerable reduction in read buffer size is achievable. Given the split-cycle-save assumption and configuration B1, a minimum of 25%, a maximum of 60%, and an average of 42% reduction was achieved. For configuration A2 and no split-cycle-save assumption, a minimum of 20%, a maximum of 50%, and an average of 38.0% reduction was achieved. The measurements indicate that care should be taken relative to the ultimate selection of read buffer size. Given the steepness of the B1 curve around the *RB_size* value, small decreases in size can produce large performance overheads.

As seen in these results, the full $2N$ read buffer size is not required for full on-path hazard resolution and negligible performance overhead given a wide variety of application programs. Slightly smaller read buffer sizes are possible given the split-cycle-save capability. As indicated by our measurements, placing a single level queue between the source buses and a single FIFO (configurations C and D) was not as effective as a dual FIFO where each source bus has access to each FIFO (configuration A2). When the split-cycle-save capability was not assumed and a single FIFO was used, QUEEN, QSORT, COMPRESS, LEX, YACC, and CCCP showed moderate performance overheads regardless of buffer size.

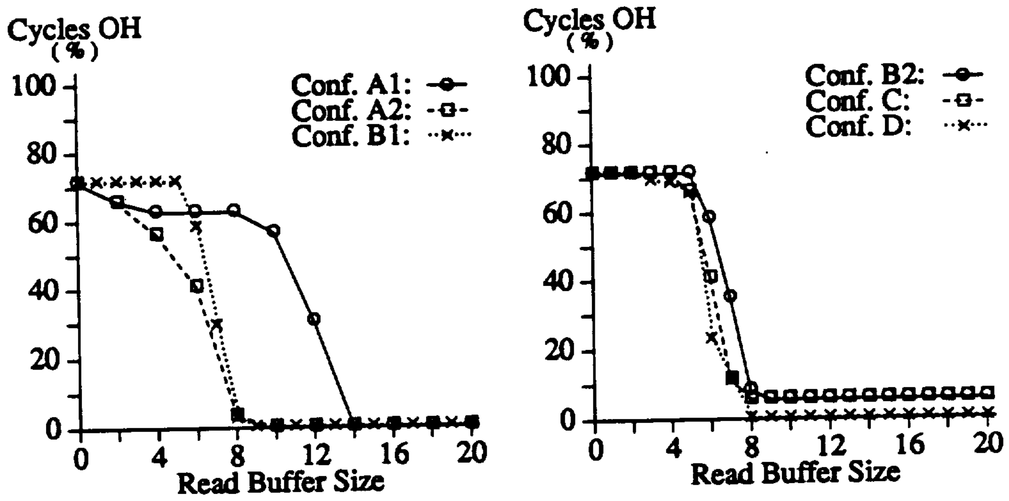


Figure 5.8: Cycle overhead: WC.

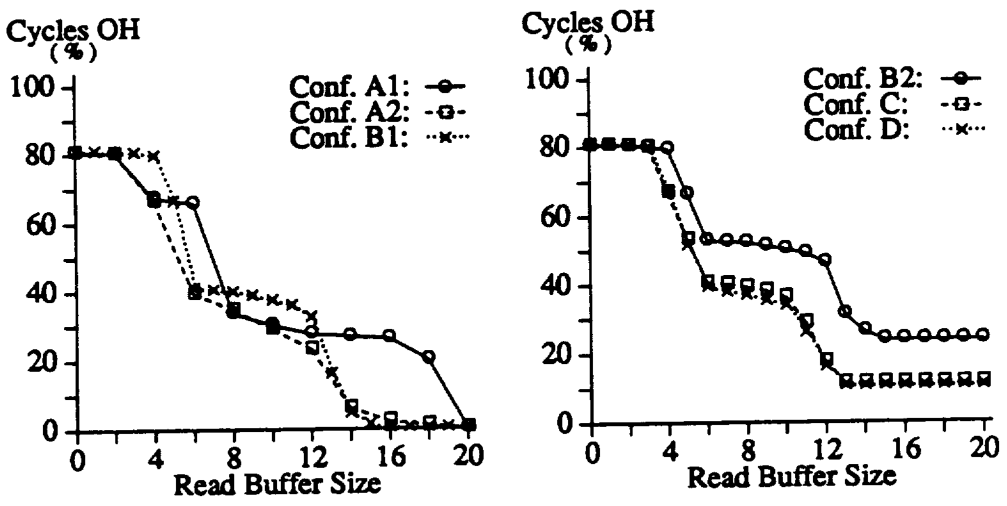


Figure 5.9: Cycle overhead: QSORT.

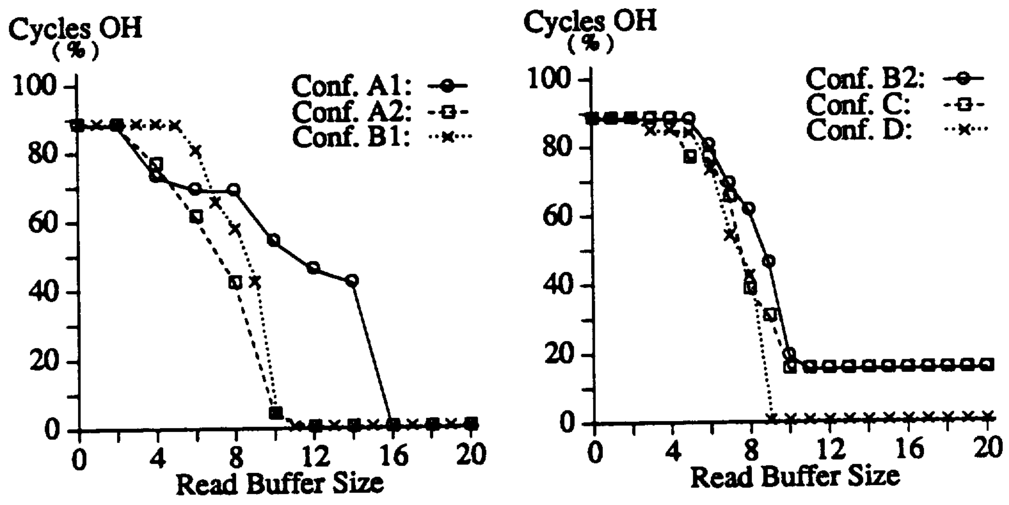


Figure 5.10: Cycle overhead: CMP.

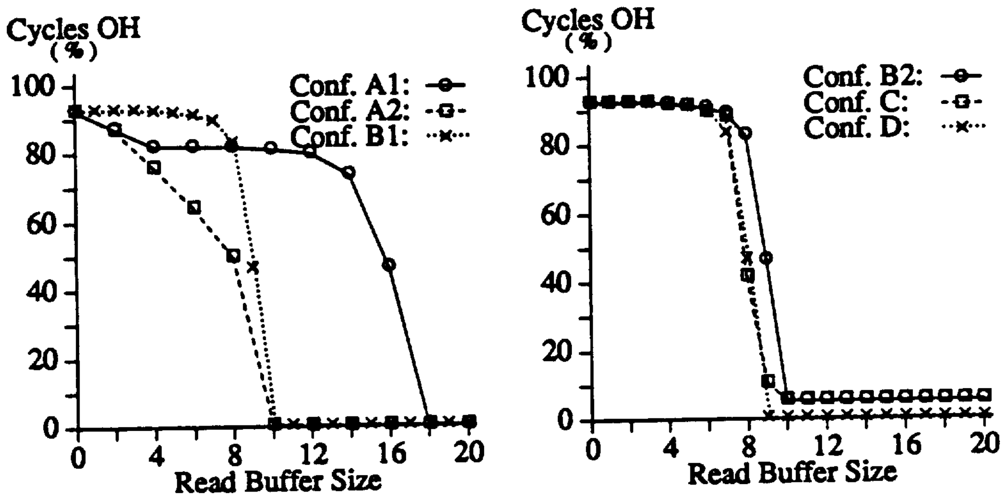


Figure 5.11: Cycle overhead: GREP.

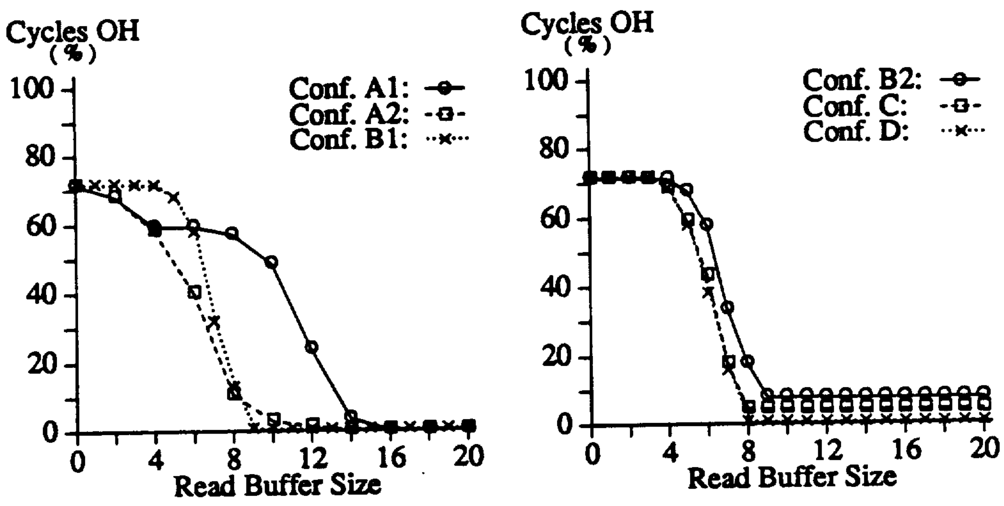


Figure 5.12: Cycle overhead: PUZZLE.

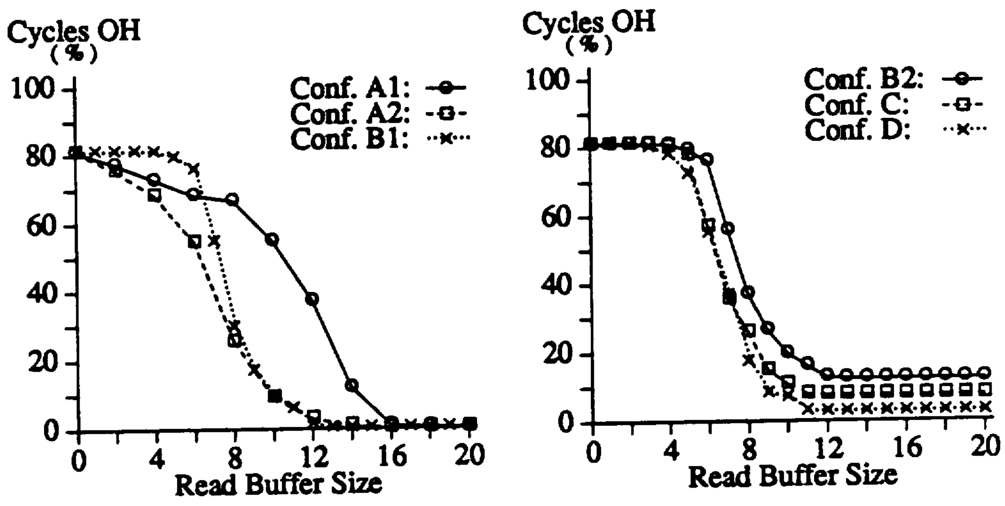


Figure 5.13: Cycle overhead: COMPRESS.

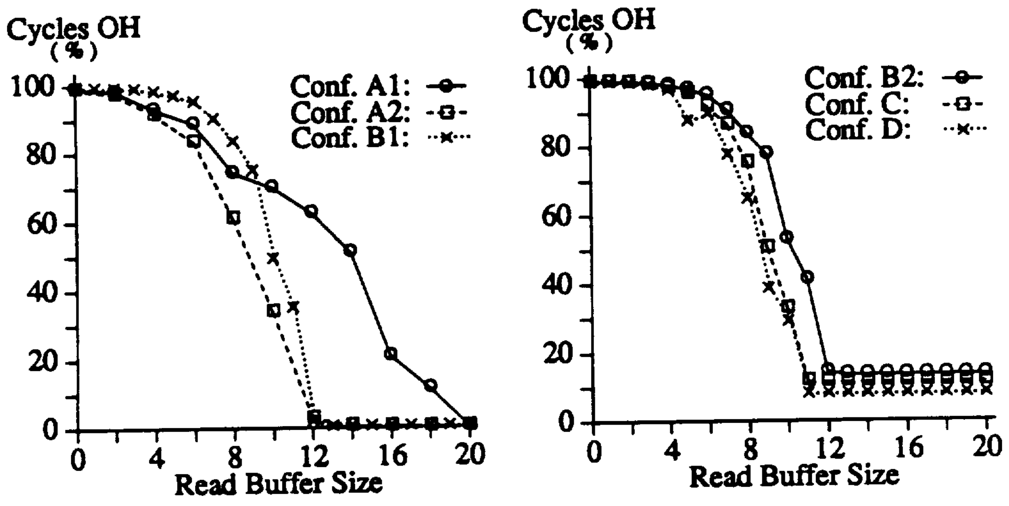


Figure 5.14: Cycle overhead: LEX.

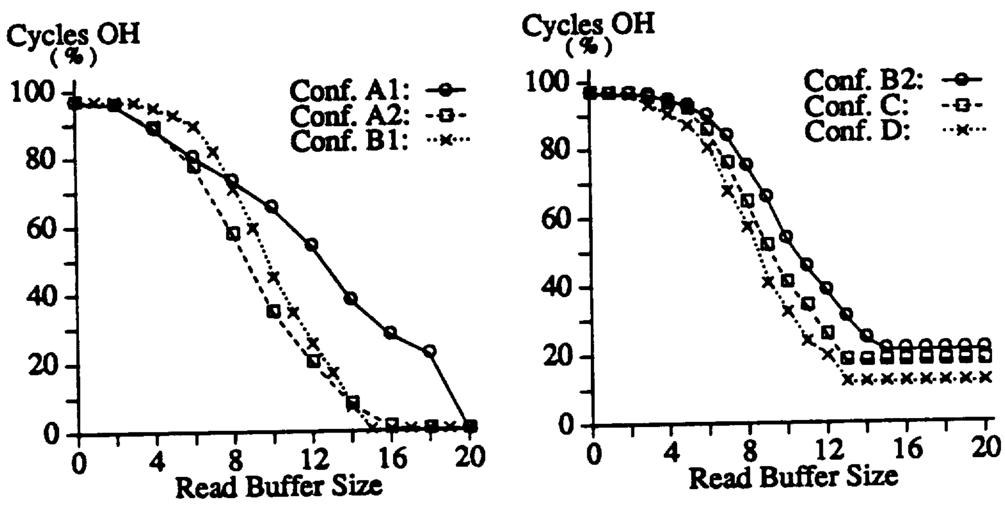


Figure 5.15: Cycle overhead: YACC.

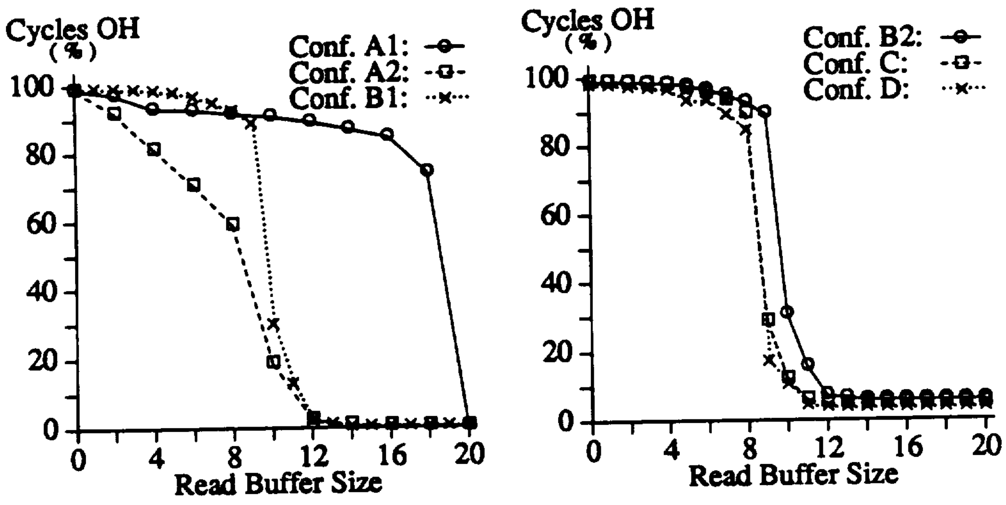


Figure 5.16: Cycle overhead: CCCP.

5.5 Summary

By adding extra bits to the operand field for source 1 and source 2, it becomes possible to design the read buffer proposed in Chapter 3 to save only those values required, thus reducing the read buffer size requirement. The performance cost of the buffer size reduction is occasional read buffer overflows which result in stall cycles. Results show that two read buffer configurations were the most efficient. The dual FIFO with source bus access to each and the single FIFO with the split-cycle-save capability consistently out-performed the other configurations. There were moderate variances between the buffer sizes required for minimum performance impact between the ten applications and the performance stabilization value assuming no split-cycle-save capability. Up to a 55% read buffer size reduction is achievable with an average reduction of 39.5% given the most efficient read buffer configuration for the applications. It was also found that given the split-cycle-save assumption and single FIFO configuration, significant changes in the performance overhead result from small changes in the read buffer size. This indicates that care should be taken in the final selection of read buffer size in any given design.

6. MIR TECHNIQUES APPLIED TO SPECULATIVE EXECUTION REPAIR

6.1 Objectives

Speculative execution is an effective 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. Speculative execution repair (SER) strategies have been proposed which trade-off speculation scope, hardware complexity, and software complexity. Many of the difficulties encountered during recovery from branch misprediction, or from instruction re-execution due to exceptions, are similar to those encountered during multiple instruction rollback (MIR). This chapter investigates the applicability of compiler-assisted instruction rollback to aid in SER.

6.2 Introduction

Super-scalar and VLIW architectures have been shown effective in exploiting instruction level parallelism (ILP) present in a given application [32,35,36]. Creating additional ILP in applications has been the subject of much study in recent years [37–39]. 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 [32]. 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 [40]. Combining multiple basic blocks into *superblocks* permits code movement within the superblock without the compensation code required in standard trace scheduling [32].

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 [40]. Sophisticated hardware support is required, however, for 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 nonspeculated architectures [41–43]. This chapter focuses on the support hardware for speculative execution, which is responsible to ensure correct operation in the presence of excepting speculated instructions and mispredicted branches.

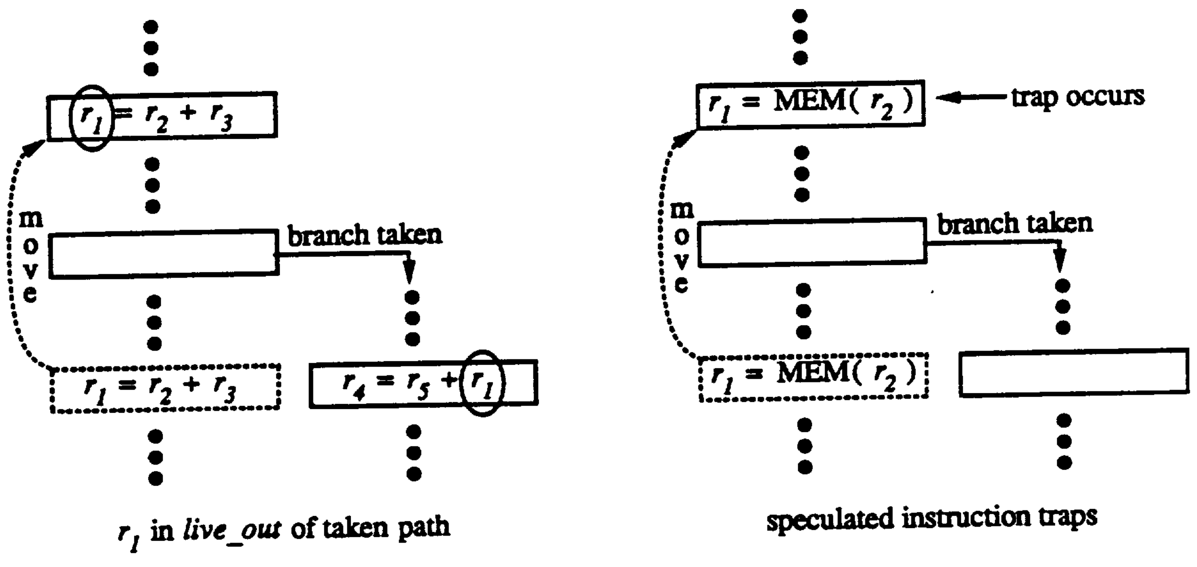


Figure 6.1: Speculative execution.

6.3 Speculative Execution

Figure 6.1 illustrates the two basic problems which are encountered when attempting upward code motion across a branch. First, if the 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. Second, 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.

6.3.1 Branch repair

Figure 6.2 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

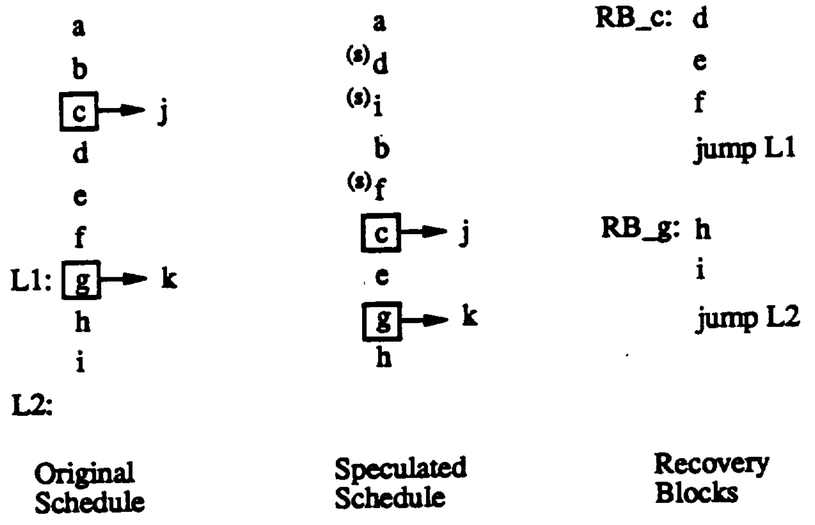


Figure 6.2: Branch repair.

fall-through paths.[1] 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 $r_x \notin live\_in(j)$, 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

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

to branch hazards in multiple instruction rollback. The two differences in how branch hazards are determined for speculative execution are: 1) the walk to record variable assignments, described in Properties 1 and 2 of Section 3.2, begins at the immediate predecessor of the branch in question (I_b) and proceeds in a backwards progression, i.e., ${}^{(s)}I_{b-1}$, ${}^{(s)}I_{b-2}$, ... ${}^{(s)}I_{b-N}$, and 2) only speculated instructions are considered in the walk. The walk distance N for speculative execution is the maximum distance from ${}^{(s)}I_{b-1}$ to ${}^{(s)}I_n$ along any backwards walk, where I_n was speculated above I_b. A branch hazard $h_b(d, j)$ exists in Figure 6.2 if d writes to register r_x and j reads r_x.

Given the similarity between branch hazards due to instruction rollback and branch hazards due to speculative execution, compiler-driven data-flow manipulations, similar to those presented in Chapter 3, can be used to resolve branch hazards that result from speculation. Such compiler transformations have been proposed for branch misprediction handling [42]. Since re-execution of speculated instructions is not required for branch misprediction, compiler resolution of branch hazards becomes a sufficient branch repair technique.

6.3.2 Exception repair

Figure 6.2 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* [43] as shown in Figure 6.2. The address of the recovery block can be be obtained by using the PC value as an index into a hash table. This strategy ensures precise interrupts [14, 44] relative to the nonspeculated schedule but not relative to the original schedule. Recovery blocks can cause significant code growth [43]. 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* [42]. 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

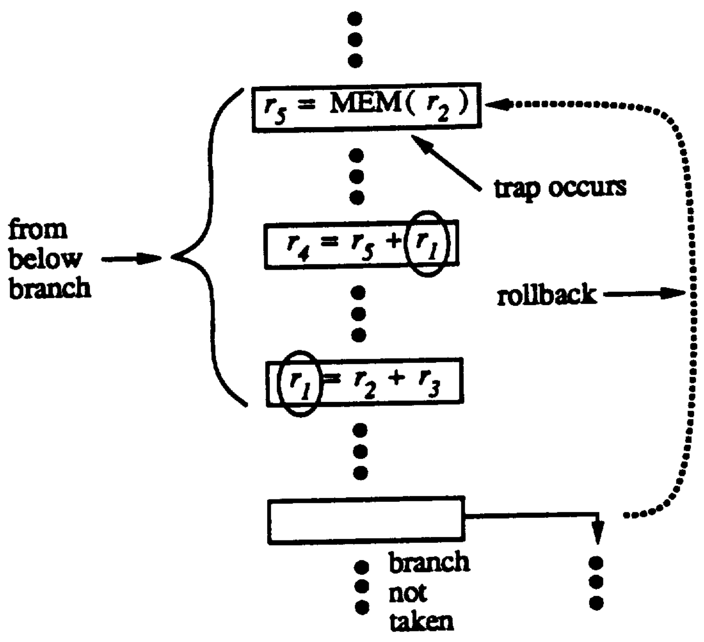


Figure 6.3: Exception repair.

multiple instruction rollback. Function 3 is similar to on-path hazard resolution in multiple instruction rollback. As discussed in Chapter 3, 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 6.3 illustrates the speculation of a group of 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 6.3 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 Chapter 3 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, where I_n is a trapping instruction and was speculated above branch instruction I_b.

6.3.3 Schedule reconstruction

Assumed in Figures 6.2 and 6.3 are mechanisms to: 1) identify speculative instructions, 2) determine the PC value of excepting speculated instructions, and 3) determine how many branches a given instruction has been speculated above. An example of the latter case is shown in Figure 6.2 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 [41]. 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 compiler-driven data-flow manipulations permits general speculation with no schedule reconstruction for branch repair [42].

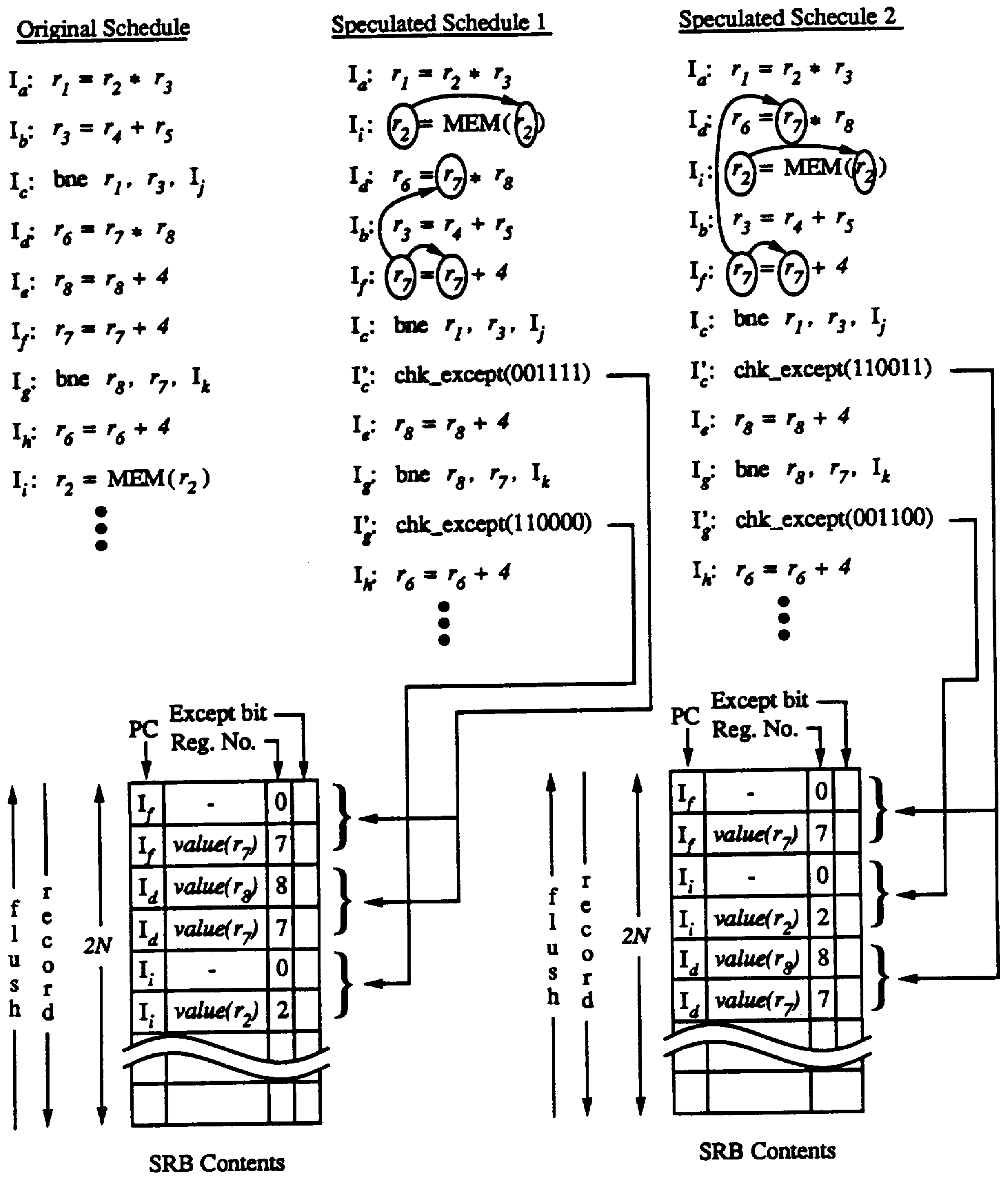
6.4 Implicit Index Schedule Reconstruction

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 [42] and was designed to exploit the unique properties of the read buffer hardware design described in Chapter 3. Schedule reconstruction is accomplished by marking each instruction *speculated* or *nonspeculated* and using this marking 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 some additional provisions for exception handling.

6.4.1 Exception repair using the speculation read buffer

Figure 6.4 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. Speculated instructions are marked. This marking identifies the source operands to be saved in the SRB. Along with the source operand values and corresponding register addresses, the PC of the speculated instruction is also recorded in the SRB.

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



PC		Reg. No.	Except bit
I_f	-	0	
I_f	value(r_7)	7	
I_d	value(r_8)	8	
I_d	value(r_7)	7	
I_i	-	0	
I_i	value(r_2)	2	

PC		Reg. No.	Except bit
I_f	-	0	
I_f	value(r_7)	7	
I_i	-	0	
I_i	value(r_2)	2	
I_d	value(r_8)	8	
I_d	value(r_7)	7	

Figure 6.4: Exception repair using a speculation read buffer (SRB).

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 6.4, *chk_except(001111)* in I_c' checks for 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 6.4 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_8 is also corrupted but not restored by the SRB, since after rollback r_8 will be rewritten with a correct value before the corrupted value is used.

Instead of checking for exceptions in each home block, the exception could be handled when the exception bit reaches the bottom of the SRB. This is very similar to the reorder buffer used in dynamic scheduling [14]. This eliminates the cost of the *chk_except(k)* command, however, and increases the exception handling latency which can impact performance depending on the frequency of exceptions. In addition, the technique guarantees that exceptions will be processed in the original home block order. For example, in schedule 2 of Figure 6.4, if both I_i and I_f trap, *chk_except(k)* ensures that I_f will be handled first.

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 [40] can be used to ensure this deterministic behavior.

6.4.2 Branch repair using the speculation read buffer

Branch repair can be handled by removing branch hazards with the compiler. As shown in Chapter 3, branch hazard removal 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. In a similar fashion, the SRB can assist in branch repair. Figure 6.5 shows the original code schedule and the two speculative schedules of Figure 6.4. 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;

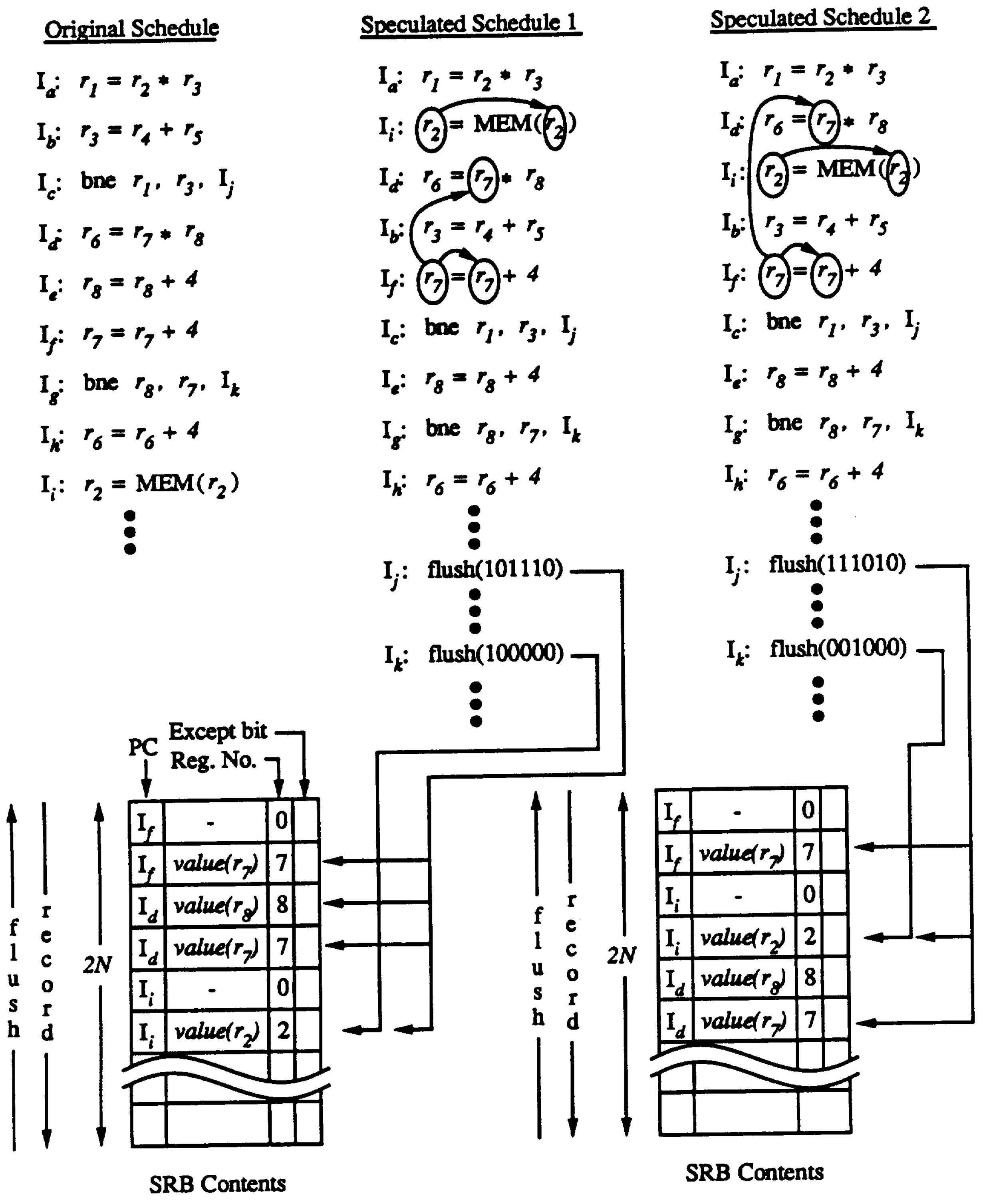


Figure 6.5: Branch repair using a speculation read buffer (SRB).

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.

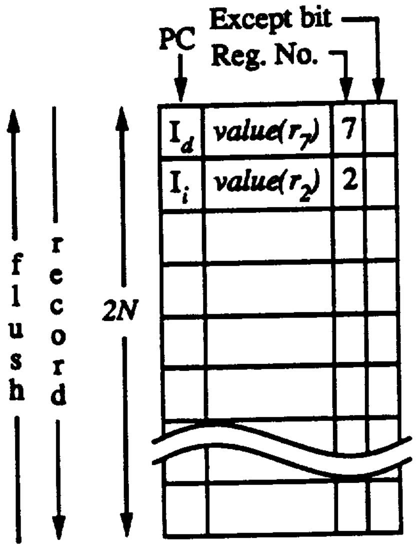
The *flush(k)* command selectively flushes the appropriate register values given a branch misprediction. For example, in schedule 2 of Figure 6.5, if I_c is predicted correctly and I_g is mispredicted, the SRB is flushed in reverse order up through I_i, restoring $value(r_2)$ from I_i but not restoring $value(r_7)$ 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 either be resolved with compiler transformations or by inserting MOV r_0, r_x instructions as described in Section 3.3. It would be necessary to mark the MOV instruction *speculated* to ensure that r_x is loaded into the SRB.

6.5 SRB Flush Penalty

The examples of Section 6.4 demonstrate that the compiler-assisted multiple instruction rollback scheme presented in Chapter 3 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% [34]. 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.

In Chapter 5, several read buffer configurations were studied. A technique was proposed to reduce the amount of redundant data in the read buffer so that the read buffer size could be reduced. 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 6.4, 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 6.6 shows the reduced contents of the SRB given schedule 1 of Figure 6.5. 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 6.5 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



SRB Contents

Figure 6.6: SRB with reduced content.

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 the opcode field which marks speculated trapping instructions. A better solution is to code all speculated trapping instructions as 01. This will indicate that exception handling is to be delayed and cause a reservation of an entry in the SRB. This latter approach will slightly increase the flush penalty during branch repairs. Separate SRBs could be maintained for branch and exception repairs.

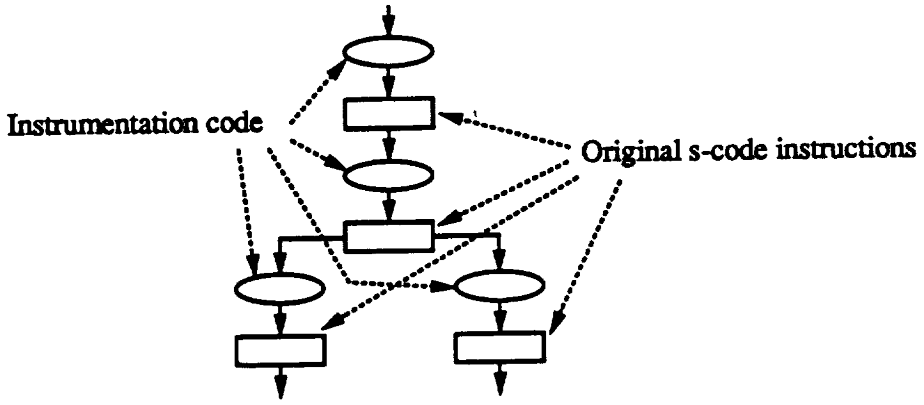


Figure 6.7: Instrumentation code placement.

6.6 Performance Evaluation

6.6.1 Evaluation methodology

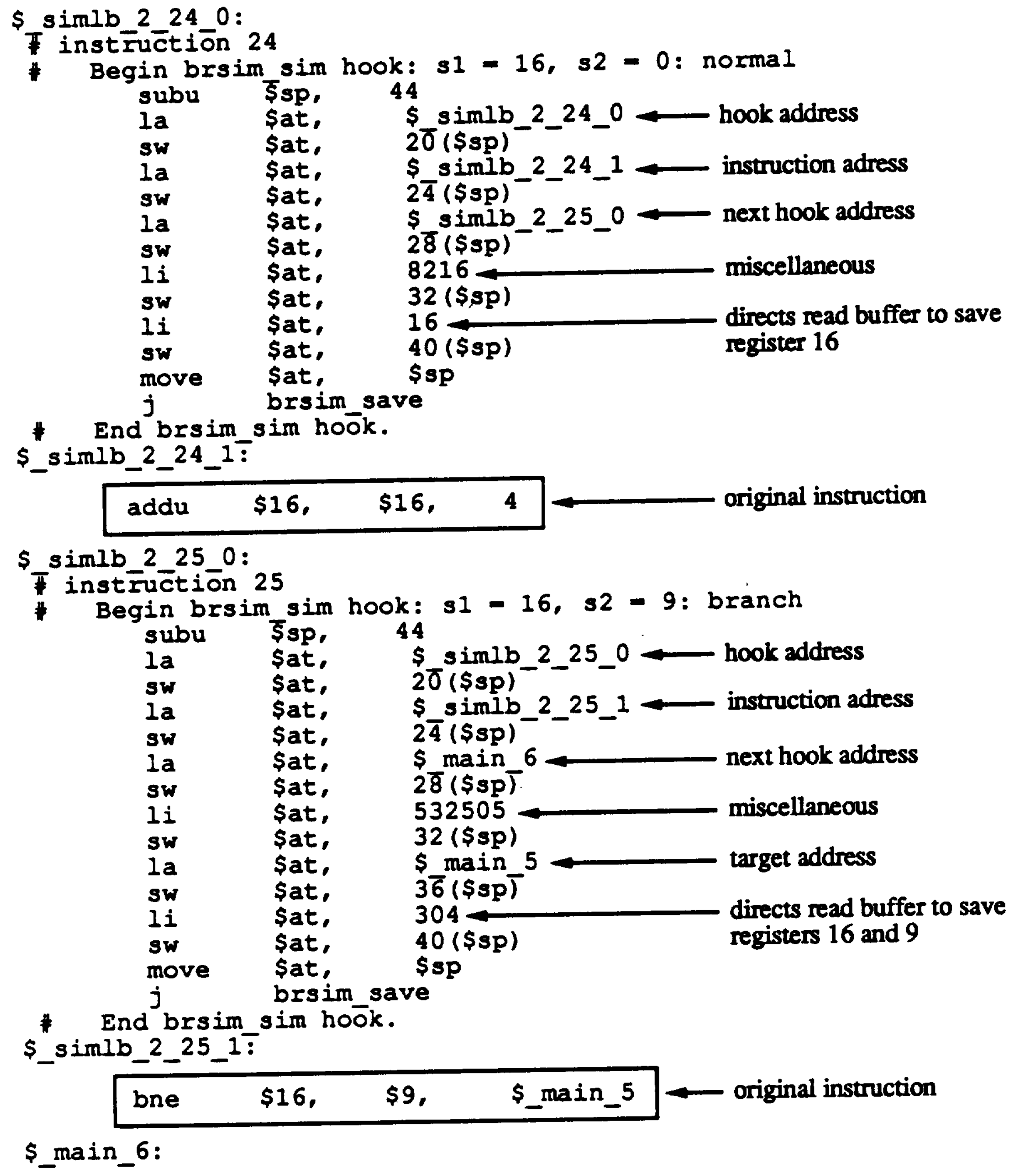
In this section, the read buffer flush penalty is evaluated using a similar strategy to the one presented in Chapter 5. The instrumentation code segments of Figure 6.7 call a branch error procedure which performs the following functions:

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 6.8. 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 6.9 gives a high level flow of operation for 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 [45] 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 by

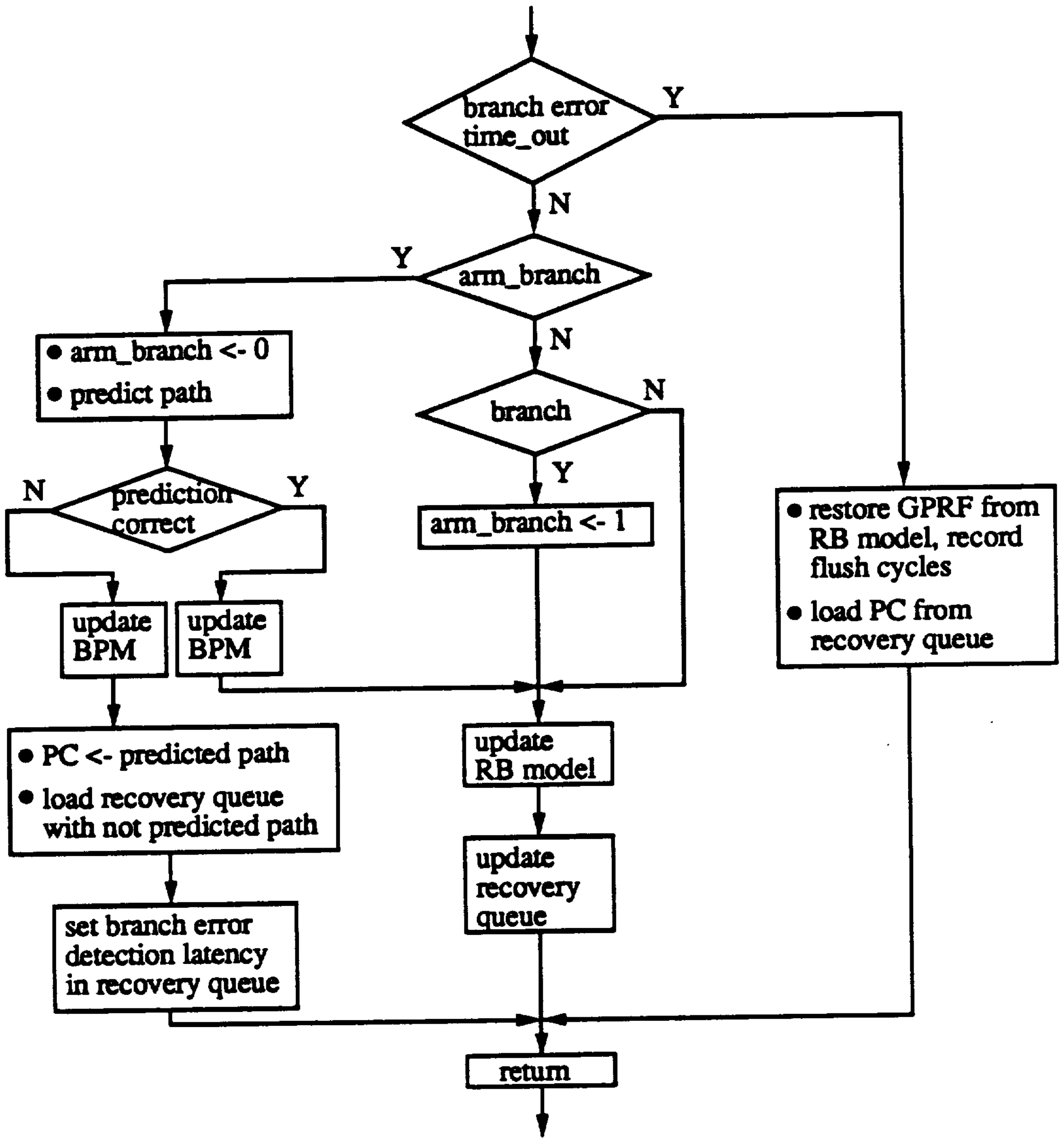


```
$_simlb_2_24_0:
 # instruction 24
 #    Begin brsim_sim hook: s1 = 16, s2 = 0: normal
        subu    $sp,    44
        la      $at,    $_simlb_2_24_0
        sw      $at,    20($sp)
        la      $at,    $_simlb_2_24_1
        sw      $at,    24($sp)
        la      $at,    $_simlb_2_25_0
        sw      $at,    28($sp)
        li      $at,    8216
        sw      $at,    32($sp)
        li      $at,    16
        sw      $at,    40($sp)
        move    $at,    $sp
        j       brsim_save
 #    End brsim_sim hook.
$_simlb_2_24_1:
        addu    $16,    $16,    4

$_simlb_2_25_0:
 # instruction 25
 #    Begin brsim_sim hook: s1 = 16, s2 = 9: branch
        subu    $sp,    44
        la      $at,    $_simlb_2_25_0
        sw      $at,    20($sp)
        la      $at,    $_simlb_2_25_1
        sw      $at,    24($sp)
        la      $at,    $_main_6
        sw      $at,    28($sp)
        li      $at,    532505
        sw      $at,    32($sp)
        la      $at,    $_main_5
        sw      $at,    36($sp)
        li      $at,    304
        sw      $at,    40($sp)
        move    $at,    $sp
        j       brsim_save
 #    End brsim_sim hook.
$_simlb_2_25_1:
        bne     $16,    $9,     $_main_5

$_main_6:
```

Figure 6.8: Instrumentation code sequences.



PC	- program counter
GPRF	- general purpose register file
RB	- read buffer
BPM	- branch prediction model

Figure 6.9: Branch error procedure operation.

the read buffer. The PC value of the correct branch path is obtained from the recovery queue. The number of cycles required to flush the read buffer during branch repair is also recorded.

It is assumed for this evaluation that two read buffer entries can be flushed in a single cycle. This corresponds to the split-cycle-save assumption of Chapter 5. 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 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 [32]. The transformation determines which operands require saving in the read buffer and inserts calls to the *initialization*, *branch error*, and *summary* procedures. Initialization and summary calculations are handled as in Chapter 5. 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 5.1 (p. 87) lists the ten application programs evaluated.

6.6.2 Evaluation results

Experimental measurements of read buffer flush overhead (*Flush OH*) for various BED latencies are shown in Figures 6.10 through 6.14 (pp. 122 through 124). The four branch prediction strategies used for the evaluation are referred to as: 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*).

As expected, 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 6.10 through 6.14.

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 6.10 through 6.14 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 14.81% and an average flush cost of 8.12% across the ten applications studied. This performance overhead is comparable to the overhead expected from a delayed write buffer scheme with a maximum allowable BED of ten [12]. However, 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, because 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 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.

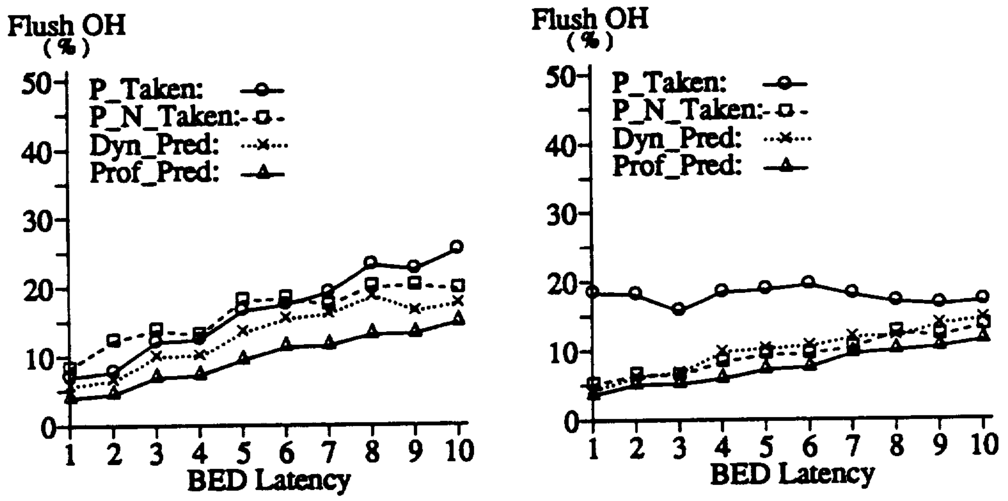


Figure 6.10: Flush penalty: QUEEN, WC.

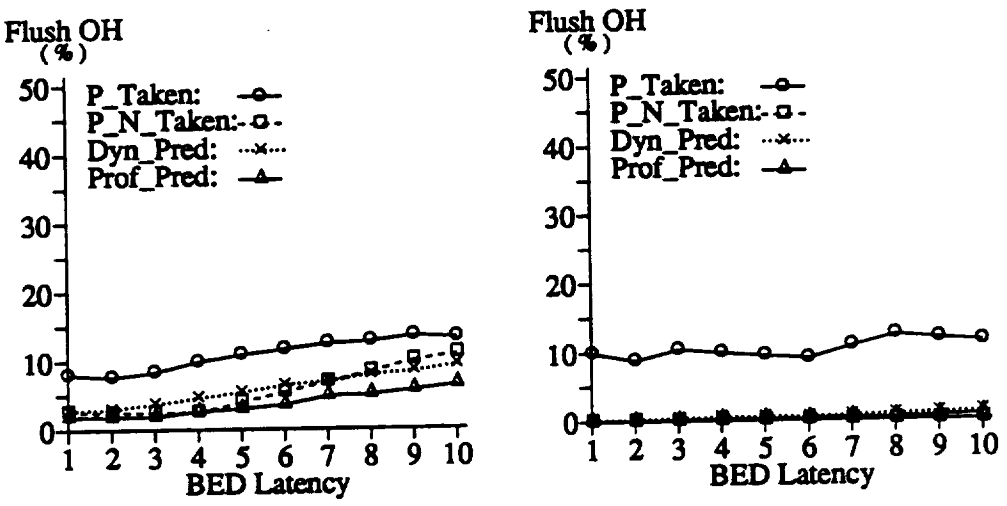


Figure 6.11: Flush penalty: COMPRESS, CMP.

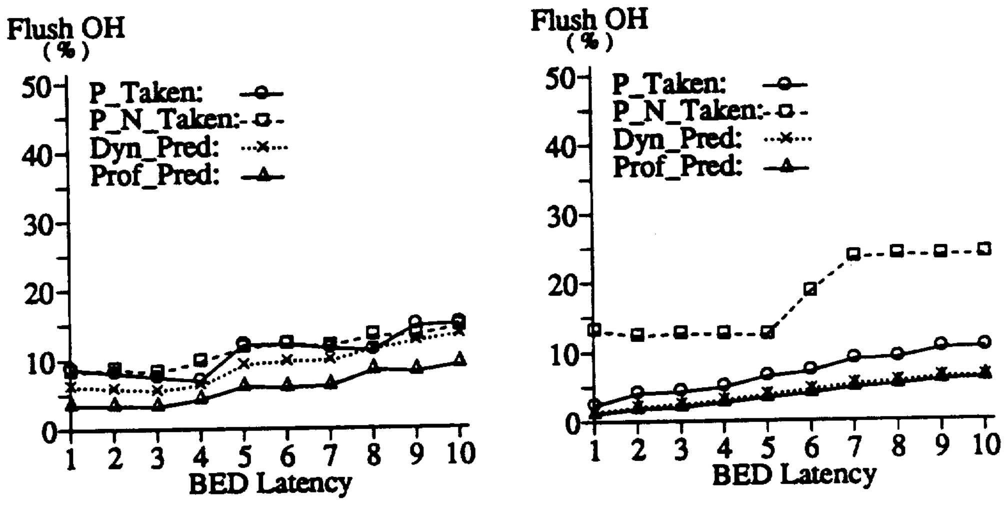
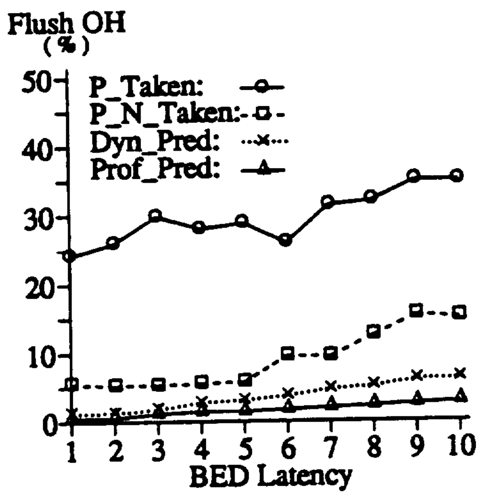


Figure 6.12: Flush penalty: PUZZLE, QSORT.



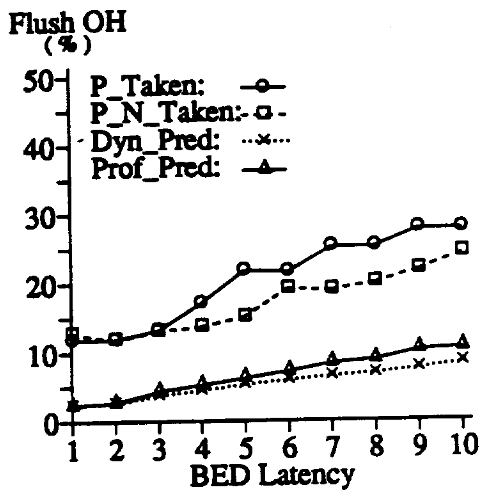
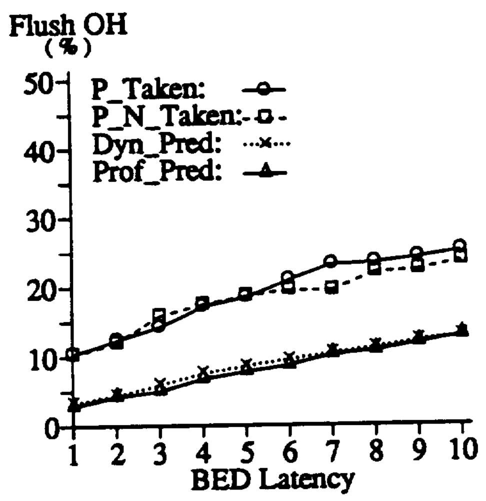


Figure 6.13: Flush penalty: GREP, LEX.



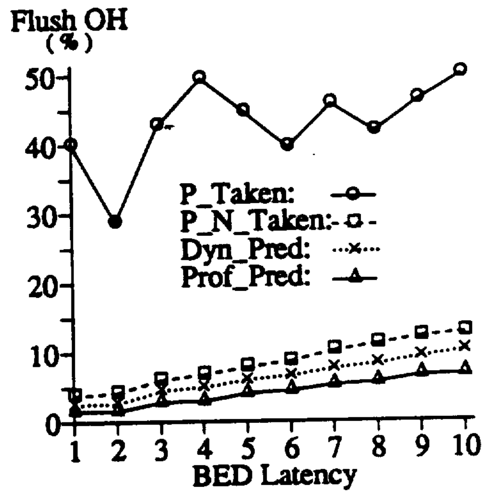


Figure 6.14: Flush penalty: YACC, CCCP.

6.7 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 chapter showed that branch hazards resulting from branch mispredictions are similar to branch hazards in multiple instruction rollback recovery. It was shown that compiler techniques similar to those presented in Chapter 3 can be used as an effective branch repair scheme in a speculative execution architecture. It was also shown that data hazards as a result of rollback due to exception repair are similar to on-path hazards described in Chapter 3, indicating that a read buffer approach to exception repair was viable.

Implicit index scheduling was introduced to exploit the unique characteristics of rollback recovery using a read buffer approach. The read buffer design was expanded to include PC values to aid in rollback from excepting speculated instructions. Similar to "covering" on-path hazards discussed in Chapter 3, the read buffer was shown to resolve some branch hazards without the need for compiler transformations.

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 the compiler-assisted multiple instruction rollback scheme presented in Chapter 3 is viable in application to branch and exception repair in a speculative execution architecture.

7. CONCLUSIONS

7.1 Summary

This thesis has presented a compiler-assisted multiple instruction rollback scheme which combines compiler-driven data-flow manipulations with dedicated data redundancy hardware to remove all data hazards that result from multiple instruction rollback. Experimental evaluation of the proposed compiler-assisted scheme with a maximum rollback distance of ten showed performance impacts of no more than 6.57% and an average impact of 1.80%, over the ten application programs studied. The performance evaluation indicates lower performance penalties than for previous compiler-only approaches or comparable hardware-only approaches. Compiler transformations used for hazard removal have been enhanced reducing application code growth and compile times, and in some cases improving application execution performance. Ten read buffer configurations were studied to determine the minimum size requirement for general applications. It was found that a 55% read buffer size reduction is achievable with an average reduction of

39.5%, but that additional control logic to handle read buffer overflows may limit the overall hardware savings. It was also shown that the proposed compiler-assisted multiple instruction rollback technique can be applied to speculative execution repair. The problems associated with recovery from mispredicted branches and excepting speculated instructions were shown to be similar to problems encountered with multiple instruction rollback recovery. A speculative scheduling scheme was proposed which utilizes compiler-driven hazard removal transformations and the read buffer to aid in hazard removal during exception handling and mispredicted branch handling.

7.2 Limitations

The compiler-assisted rollback recovery scheme presented limits system state space restoration to the register file. Other methods, such as history buffers, would be required to maintain an N cycle history of the program counter and program status word. Cache memory and main memory would require an N cycle rollback capability which could be implemented with an N depth delayed write buffer. Functional units such as the floating point unit and I/O units would require a rollback capability or a capability to be flushed and restarted. A spontaneous change to the contents of the register file is not recoverable by the compiler-assisted recovery scheme, although the propagation of such errors is recoverable if the errors are detected within N cycles. For enhanced fault tolerance, error detection/correction codes could be used in the register file. A similar limitation exists for cache and main memories. Unlike the previously mentioned limitations, the requirement that an error does not cause an illegal path in the control-flow graph of the program is unique to compiler-assisted rollback recovery. For enhanced fault tolerance, a control-flow error detection mechanism with a latency no greater than one cycle would be required. Finally, the compiler-assisted rollback recovery scheme requires recompilation of application programs and libraries.

7.3 Future Research

The use of profile data can be extended to the register allocation phase and is expected to result in further reduction in performance overhead. Application of compiler-assisted multiple instruction rollback recovery to super-scalar, VLIW, and parallel processing architectures is an area with great potential. Given the flexibility of the IMPACT compiler platform used for current hazard removal transformations, studies of rollback recovery schemes for the three architectures are feasible and should produce near-term results. Further evaluations of compiler-assisted rollback recovery applied to speculative execution repair would include modifying compiler transformations to operate in a super-scalar and VLIW environment. Again, the flexibility of the IMPACT compiler platform should simplify this investigation. An additional extension would be to develop and evaluate a scheme which handles both instruction rollback recovery and speculative execution repair.

REFERENCES

[1] C. L. Chen and M. Y. Hsiao, "Error Correcting Codes for Semiconductor Memory Applications: A State-of-the-art Review," *IBM J. Res. Dev.*, vol. 28, no. 2, pp. 124-134, Mar. 1984.

[2] R. M. Sedmak and H. L. Liebergot, "Fault Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration," *IEEE Trans. Comput.*, vol. 39, pp. 548-554, Apr. 1990.

[3] J. H. Patel and L. Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Trans. Comput.*, vol. C-31, no. 7, pp. 589-591, July 1982.

[4] J. G. Holm and P. Banerjee, "Low Cost Concurrent Error Detection in a VLIW Archtecture using Replicated Instructions," in *Proc. 1992 Int. Conf. Parallel Processing*, pp. 192–195, Aug. 1992.

[5] Y. Tamir, M. Liang, T. Lai, and M. Tremblay, "The UCLA Mirror Processor: A Building Block for Self-Checking Self-Repairing Computing Nodes," in *Proc. 21th Int. Symp. Fault-Tolerant Comput.*, pp. 178–185, June 1991.

[6] M. Schuette and P. J. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Trans. Comput.*, vol. C-36, no. 3, pp. 264-276, Mar. 1984.

[7] T. Sridhar and S. M. Thatte, "Concurrent Checking of Program Flow in VLSI Processors," in *Proc. 1982 IEEE Int. Test Conf.*, pp. 191–199, 1982.

[8] L. Svobodova, "Resilient Distributed Computing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 3, May 1984.

[9] L. Lin and M. Ahamad, "Checkpointing and Rollback-Recovery in Distributed Object Based Systems," in *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, pp. 97–104, 1990.

[10] K. Tsuruoka, A. Kaneko, and Y. Nishihara, "Dynamic Recovery Schemes for Distributed Processes," in *IEEE 2nd Symp. Reliability Distributed Softw. Database Syst.*, pp. 124–130, 1981.

[11] W.-M. W. Hwu and Y. N. Patt, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines," *IEEE Trans. Comput.*, vol. C-36, pp. 1496 -1514, Dec. 1987.

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

[13] M. S. Pittler, D. M. Powers, and D. L. Schnabel, "System Development and Technology Aspects of the IBM 3081 Processor Complex," *IBM J. Res. Dev.*, vol. 26, pp. 2-11, Jan. 1982.

[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] E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," in *Proc. 14th Design Autom. Conf.*, pp. 462–468, 1977.

[16] E. J. McClusky, *Logic Design Principles.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986.

[17] M. L. Ciacelli, "Fault Handling on the IBM 4341 Processor," in *Proc. 11th Int. Symp. Fault-Tolerant Comput.*, pp. 9–12, June 1981.

[18] R. N. Gustafson and F. J. Sparacio, "IBM 3081 Processor Unit: Design Considerations and Design Process," *IBM J. Res. Dev.*, vol. 26, pp. 12-21, Jan. 1982.

[19] W. F. Bruckert and R. E. Josephson, "Designing Reliability into the VAX 8600 System," *Digital Tech. J. Digital Equip. Corp.*, vol. 1, no. 1, pp. 71-77, Aug. 1985.

[20] D. B. Fite, T. Fossum, and D. Manley, "Design Strategy for the VAX 9000 System," *Digital Tech. J. Digital Equip. Corp.*, vol. 2, no. 4, pp. 13-24, Fall 1990.

[21] P. M. Kogge, K. T. Truong, D. A. Richard, and R. L. Schoenike, "Checkpoint Retry Mechanism." United States Patent, no. 4912707, Mar. 1990. Assignee: International Business Machines Corporation, Armonk, N.Y.

[22] G. L. Hicks, D. Howe, Jr., and A. Zurla, Jr., "Insruction Retry Mechanism for a Data Processing System." United States Patent, no. 4044337, Aug. 1977. Assignee: International Business Machines Corporation, Armonk, N.Y.

[23] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. Dev.*, vol. 11, pp. 25-33, Jan. 1967.

[24] L. Spainhower, J. Isenberg, R. Chillarege, and J. Berding, "Design for Fault-Tolerance in System ES/9000 Model 900," in *Proc. 22th Int. Symp. Fault-Tolerant Comput.*, pp. 38–47, July 1992.

[25] J. S. Liptay, "Computer System with Logic for Writing Instruction Indentifying Data into Array Control Lists for Precise Post-Branch Recoveries." United States Patent, no. 4901233, Feb. 1990. Assignee: International Business Machines Corporation, Armonk, N.Y.

[26] J. S. Liptay, "The ES/9000 High End Processor Design," *IBM J. Res. Dev.*, vol. 36, no. 3, May 1992.

[27] C.-C. J. Li and W. K. Fuchs, "CATCH - Compiler-Assisted Techniques for CHeck-pointing," in *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, pp. 74–81, June 1990.

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

[29] C.-C. J. Li, S.-K. Chen, W. K. Fuchs, and W.-M. W. Hwu, "Compiler-Assisted Multiple Instruction Retry." Manuscript, May 1991.

[30] 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.

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

[32] 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.

[33] S. Weiss and J. E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers," in *Proc. 2nd Int. Conf. Architecture Support Programming Languages and Operating Syst.*, pp. 105–111, Oct. 1987.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

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

[41] 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.

[42] 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.

[43] 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.

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

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

VITA

Neal Jon Alewine was born in [redacted], on [redacted]. He received his B.S. degree in Electrical Engineering from Florida Atlantic University, Boca Raton, Florida in March of 1980. He was employed by the International Business Machines Corporation, Boca Raton, Florida, and held several technical and management positions including design engineer, lead designer, first-level manager, technical assistant to the General Manager, program manager, and second-level manager. He received his M.S. degree in Electrical Engineering from Florida Atlantic University in December of 1988 and was selected to participate in the IBM Resident Study Program to pursue doctoral studies at the University of Illinois at Urbana-Champaign.

After completing his doctoral dissertation, Mr. Alewine will return to IBM at the Boca Raton facility. His research interests include high-performance microarchitecture, fault-tolerant computing, and performance evaluation.