November 1994

UILU-ENG-94-2241
CRHC 94-24

Center for Reliable and High Performance Computing

Compiler-Assisted Debugging and Multiple Instruction Retry

Shyh-Kweh Chen

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-94-2241 (CRHC 94-24)	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 Office of Naval Research and NASA
6c. ADDRESS *(City, State, and ZIP Code)* 1308 W. Main ST. Urbana, IL 61801	7b. ADDRESS *(City, State, and ZIP Code)* 800 N. Quincy St. Arlington, VA 22217 AMES RESEARCH CTR. MOFFETT FIELD, CA
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014- 90-J-1270 NASA NAG 1-613
8c. ADDRESS *(City, State, and ZIP Code)* 800 N. Quincy St. Arlington, VA 22217 7b:	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
COMPILER-ASSISTED DEBUGGING AND MULTIPLE INSTRUCTION RETRY

12. PERSONAL AUTHOR(S)
Chen, Shyh-Kweh

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	November 1994	83

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	instruction retry, reversible debugging, instruction level parallelism, instruction level fault-tolerance, compiler

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

Backward execution requires the saving of historic information concurrently with the normal execution in order for a program to roll back. There are several applications for which backward execution is useful. In an environment where reliability is a concern, it may be necessary to roll back the program to a previously executed state in case of system faults. Multiple instruction retry is an alternative to checkpointing for recovery from a transient processor failure, especially when the error detection latency is only a few cycles. Speculative execution can achieve significant speedup for superscalar architectures by utilizing instruction retry to roll back the computation above mispredicted branches. Debugging is another field that can benefit from backward execution. Allowing the user to undo several instructions at specific positions may facilitate program debugging.

This thesis describes schemes that have been implemented for multiple instruction retry for both RISC-type scalar processors, and very long instruction word (VLIW) architectures. The thesis also presents approaches that incorporate backward execution into a debugger, at both the compiler and the debugger levels.

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

COMPILER-ASSISTED DEBUGGING AND MULTIPLE INSTRUCTION RETRY

BY

SHYH-KWEI CHEN

B.S., National Taiwan University, 1983
M.S., University of Minnesota, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

COMPILER-ASSISTED DEBUGGING AND MULTIPLE INSTRUCTION RETRY

Shyh-Kwei Chen, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1994
W. Kent Fuchs, Advisor

Backward execution requires the saving of historic information concurrently with the normal execution in order for a program to roll back. There are several applications for which backward execution is useful. In an environment where reliability is a concern, it may be necessary to roll back the program to a previously executed state in case of system faults. Multiple instruction retry is an alternative to checkpointing for recovery from a transient processor failure, especially when the error detection latency is only a few cycles. Speculative execution can achieve significant speedup for superscalar architectures by utilizing instruction retry to roll back the computation above mispredicted branches. Debugging is another field that can benefit from backward execution. Allowing the user to undo several instructions at specific positions may facilitate program debugging.

This thesis describes schemes that have been implemented for multiple instruction retry for both RISC-type scalar processors, and very long instruction word (VLIW) architectures. The thesis also presents approaches that incorporate backward execution into a debugger, at both the compiler and the debugger levels.

Acknowledgements

I am greatly in debt to my thesis advisor, Professor W. Kent Fuchs, for his guidance, support and encouragement throughout the thesis research. Also, I wish to thank Professor Wen-Mei Hwu for his contribution to this thesis work. I would like to express my appreciation to Professor Geneva Belford, Professor Andrew Chien, and Professor David Padua for serving in my committee.

During the work on multiple instruction retry, I enjoyed discussion with Dr. Chung-Chi Jim Li and Dr. Neal Alewine. I would also like to thank all my friends in the Center for Reliable and High-Performance Computing at the Coordinated Science Laboratory, including Dr. Weiping Shi, Dr. Yi-Min Wang, Dr. Antoine Mourad, Dr. Junsheng Long, Dr. Paul Chen, Bob Janssens, Srikanth Venkataraman, Sujoy Basu, Ismed Hartanto, Kuo-Feng Ssu, Gaurav Suri, Ching-Han Tsai, and Vicki McDaniel, to name a few, who made my stay here a pleasant experience.

Finally I would like to thank my parents and brothers for their love, understanding, sacrifices and patience throughout my Ph.D. study.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Backward execution includes several useful applications. The capability of retrying a few instructions is desirable in situations requiring rapid recovery from transient processor failures. Checkpointing provides a means to rollback the computation for critical and long-running jobs in a multiprocessor environment. Speculative execution can achieve significant speedup for superscalar architectures if the recovery from mispredicted branches can be efficiently carried out. Exception repair finds the previous consistent system state using precise interrupt. Undo allows the user to perform a close examination around the error portion of the program.

In order for a program to roll back, the backward execution requires the saving of historic information. Both hardware and software implementations for some of these applications have been proposed. Hardware solutions for multiple instruction retry involve complex circuitry modifications, while software solutions suffer from large code growth and long compilation time. The thesis studies the efficient software-based approaches for multiple instruction retry and reversal debugging.

1.2 Thesis Contributions

The thesis studies the following backward execution applications: 1) compiler-based multiple instruction retry for scalar processors, 2) the application of multiple instruction retry for VLIW architectures, and 3) reversal debugging. The hardware redundancy for register file is eliminated by compile-time data dependency manipulation. Both the compilation time and the number of instructions that can be processed have been significantly improved. Multiple instruction retry for VLIW architectures has been developed, under a new code scheduling model. Compiler dataflow analysis has drastically reduced the data recording time associated with the reversal debugging.

Previous work on compiler-assisted multiple instruction retry has utilized a series of compiler transformations, *loop protection*, *node splitting*, and *loop expansion*, to eliminate anti-dependencies of length $\leq N$ in the *pseudo register*, *machine register*, and *the post-pass resolver* phases of compilation [1,2]. The results have provided a means of rapidly recovering from transient processor failures by rolling back N instructions. This thesis presents techniques for improving compilation and run-time performance in compiler-assisted multiple instruction retry. The compilation time has been reduced from more than two hours to within three minutes for source code of 1,000 instructions. The number of instructions that can be compiled increases from 800 to more than 15,000. Incremental updating enhances compilation time when new instructions are added to the program. Post-pass code rescheduling and spill register reassignment algorithms improve the run-time performance and decrease the code growth across the application programs studied. Branch hazards are also shown to be resolvable by simple modifications to the incremental updating schemes during the pseudo register phase and to the spill register reassignment algorithm during the post-pass phase.

Two compiler assisted multiple instruction word retry schemes for VLIW architectures have been developed. In the first scheme, compiler generated hazard-free code with different degrees of rollback capability for scalar processors [1] is compacted by a modified VLIW trace scheduling algorithm. Nops are then inserted in the scheduled code words to resolve data hazards for VLIW architectures. Performance is compared under three parameters : N, the rollback distance for scalar processors; P, the number of functional units; and n, the rollback distance for VLIW architectures. The second scheme employs a hardware read buffer [2] to resolve frequently occurring data hazards, and utilizes the compiler to resolve the remaining hazards. Performance penalty and code growth have been reduced to negligible amounts, with the hardware cost of a read buffer of $2nP$ entries. An analytical bookkeeping code scheduling algorithm is proposed, which performs local swappings to schedule code for VLIW architectures. Such a code scheduling algorithm alleviates the coding complexity of the previous trace scheduling algorithm, and reduces the code duplication of the superblock scheduling.

Two implementations for undo have been incorporated in a symbolic debugger, so that the reversible execution can be selectively enabled and disabled without waiting for a long time. The first implementation is easy to develop with no code growth ratio, but with a high normal execution time slow down, while the second employs compilers to insert buffering instructions into the user programs. The compiler-assisted implementation significantly reduces the run-time overhead with a moderate code growth. To increase the effective buffer usage for undo instructions, a twin-buffer scheme is proposed, where one buffer serves as the working buffer and the other buffer as the backup buffer.

1.3 Thesis Organization

Chapter 2 describes schemes that have been implemented for multiple instruction retry for RISC-type scalar processors. Incremental updating and post-pass code reordering are employed to improve the code run-time and reduce the compilation time.

Chapter 3 applies multiple instruction retry to VLIW architectures, using both compiler-based, and hardware/compiler combined approaches. To schedule code for VLIW architectures, an analytical bookkeeping code scheduling algorithm is proposed.

Chapter 4 incorporates the undo capability into debuggers. The merits of using a read buffer and a history write buffer are discussed. Two schemes were implemented based on a modified version of the GNU debugger, GDB. The first scheme traps every instruction and stores suitable values based on de-assembling the current instruction. while the second utilizes the compiler to insert suitable instructions to store values at the run-time.

Chapter 2

Compiler-Based Multiple Instruction Retry

2.1 Introduction

To achieve uninterrupted operation, fault-tolerant computer systems usually possess the ability of detecting errors and either correcting the errors immediately or recovering the systems to previous consistent states prior to the occurrences of the errors. There is evidence that hardware transient faults, due mainly to temporary changes of electric, electromagnetic and radioactive conditions, occur more often than permanent faults [3,4]. In an environment with a high transient fault rate, it is desirable for a system to recover rapidly without resorting to a global restart whenever a fault occurs.

Software based checkpointing provides for rollback recovery when transient system faults occur. In such schemes, a checkpoint of the system state is captured and recorded at regular intervals [5–7], or predetermined positions in the application program [8]. In the event of a fault, the system can be rolled back to one of the previously recorded checkpoints, returning the system to a consistent state [9]. Software checkpointing can accommodate long error detection latencies at the cost of potentially long recovery time.

In contrast to full software checkpointing, multiple instruction retry schemes aid in rollback of just a few instructions. Instruction retry schemes have traditionally been implemented in hardware, both in full checkpointing [10–14], and in incremental checkpointing (sliding window) [15–19] formats.

The micro-rollback scheme [17] employs a delayed write buffer to sustain rollback capability, maintaining the old data for N cycles, which is the maximum error detection latency (or exception latency). A write to the register file is written to the buffer instead. The delayed write buffer has N entries, where each entry corresponds to an instruction of the last N cycles and consists of two fields, the name of the destination register and the new value. The new value and the information regarding its destination register are held in the buffer for N cycles before updating the real register. A prioritized by-pass circuitry is needed to retrieve the most recent copy of the register. During recovery, the buffer contents are invalidated and the program counter(PC) and the program status word(PSW) are reloaded with the rollback values.

A compiler-based multiple instruction retry scheme has been developed, in which compiler-driven data flow manipulation is used to resolve data hazards associated with rollback recovery [1] by removing anti-dependencies of length $\leq N$ instructions. If an error is detected, the execution is recovered by loading the correct values of the PC and the PSW. The delayed write buffer for the register file is removed in this approach. However, the original implementation suffered from long compilation times. An alternative to the compiler-based technique is the combined compiler-hardware scheme [2], which can remove one type of hazard using a hardware read buffer, while allowing the compiler transformations to resolve the remaining hazards.

This chapter addresses the compile-time limitations of the original compiler-based hazard removal approach to multiple instruction retry [1]. The techniques described include incremental updating, post-pass code rescheduling, spill register reassignment and branch hazard resolution.

2.2 Error Model and Hazard Types

Targeted processor errors are described as follows [2]. Error detection latency is $\leq N$ instructions. Units external to the CPU, such as memory and I/O, have their own rollback capability (e.g., delayed write buffers of depth N and appropriate bypass logic). The PC and the PSW contents at each instruction are preserved by an external recording device or by shadow registers [17]. A restartable CPU state can be restored by loading the correct contents of the PC and the PSW.

Given the above assumptions, a permissible error is one which does not result in a path inconsistent with the control flow graph (CFG) of the target application program provided that the register file contents do not spontaneously change and data is not written to an incorrect register location. Errors targeted for recovery via multiple instruction retry are summarized

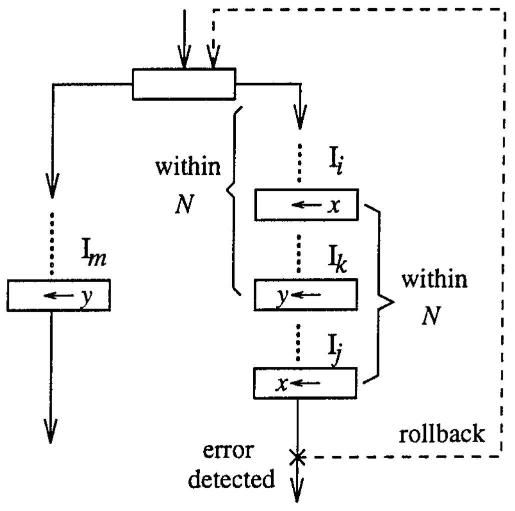


Figure 2.1 On-path and branch hazards

as follows: 1) CPU errors such as those caused by a faulty ALU; 2) incorrect values read from memory, the register file, or external functional units such as the floating point unit; 3) correct/incorrect operands read from incorrect locations within the I/O, memory, or register file; and 4) incorrect branch decisions resulting from errors 1 through 3.

The code can be represented as a CFG, $G(V,E)$, where V is the set of nodes denoting instructions and E the set of edges denoting flow information. If there is a direct control flow from instructions I_i to I_j, where $I_i \in V$ and $I_j \in V$, then there is an edge $(I_i, I_j) \in E$.

Within the general error model above, data hazards resulting from instruction retry are of two types [2]. *On-path hazards* are those encountered when the instruction path after rollback is the same as the initial instruction path and *branch hazards* are those encountered when the instruction path after rollback is different from the initial instruction path. On-path hazards can be described as anti-dependencies of length $\leq N$ in $G(V,E)$ [20]. As shown in Figure 2.1, register x of node I_j represents an on-path hazard and register y of node I_k represents a branch hazard.

2.3 Overview of Schemes Implemented

The implementation is based on the intermediate code generated by the original version of the IMPACT C compiler [21] after optimization but before register allocation. Data hazards are resolved in three different phases, the *pseudo register* phase, the *machine register* phase, and

Table 2.1 Implementation summary

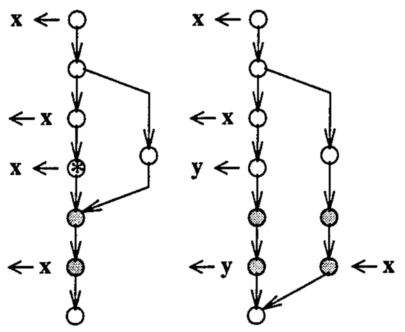
	Pseudo register	machine register	Nop insertion
Scheme L	on-path	on-path	on-path
Scheme A	on-path + branch[*]	on-path + branch	on-path + branch
Scheme 0	on-path[i]	on-path	on-path[cr]
Scheme 1	on-path[i]	on-path	on-path + branch[cr]
Scheme 2	on-path + branch[i]	on-path	on-path + branch[cr]
Scheme 3	on-path + branch[i]	on-path + branch	on-path + branch[cr]

the *nop insertion* phase. In order to compare compile time and run time efficiency, alternative schemes for each of the phases were implemented, as shown in Table 2.1.

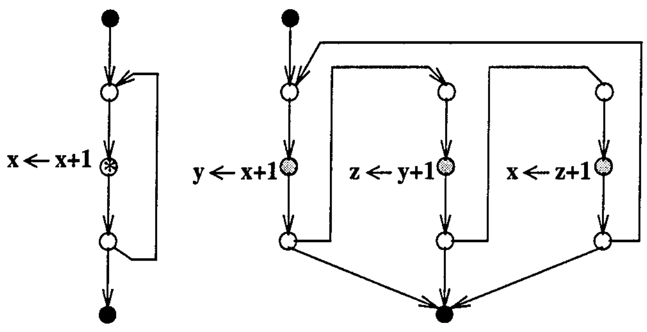
Scheme L [1] resolves on-path hazards only. Scheme A [2] resolves both on-path and branch hazards at the latter two phases, but does not resolve all pseudo register branch hazards at the first phase, as marked "[*]". The dominant fraction of compile time in the previous Schemes L and A is devoted to resolving pseudo register hazards. Both schemes implement a simple pseudo register phase, and the data structure updating is not incrementally maintained. Therefore, four alternative schemes that exploit incremental compilation techniques were implemented and compared. Scheme 0 uses incremental updating in the pseudo register phase for resolving on-path hazards. Compilation time has been enhanced with respect to Scheme L. Scheme 0 also employs post-pass code rescheduling and spill register reassignment algorithms to enhance the run-time performance and decrease the code growth across the application programs studied. The marker "[i]" denotes incremental updating, while "[cr]" denotes code rescheduling. Modifications to the post-pass algorithms can resolve both types of hazards during the nop insertion phase (Schemes 1, 2, and 3). We also show that a slightly modified incremental updating scheme can resolve branch hazards as well in the pseudo register phase (Schemes 2 and 3), though experimental results favor Scheme 1 in code run-time, code growth and compilation speed.

2.4 Review of the Pseudo Register Phase in Scheme L

The following notation is for on-path hazards, while those for branch hazards can be similarly defined. A node I_d is a *hazard node* if I_d defines a register x, another node I_u uses x, and there is a directed path of length less than or equal to N from I_u to I_d. Register x is called a *hazard*



(a) Node splitting, $N = 5$.



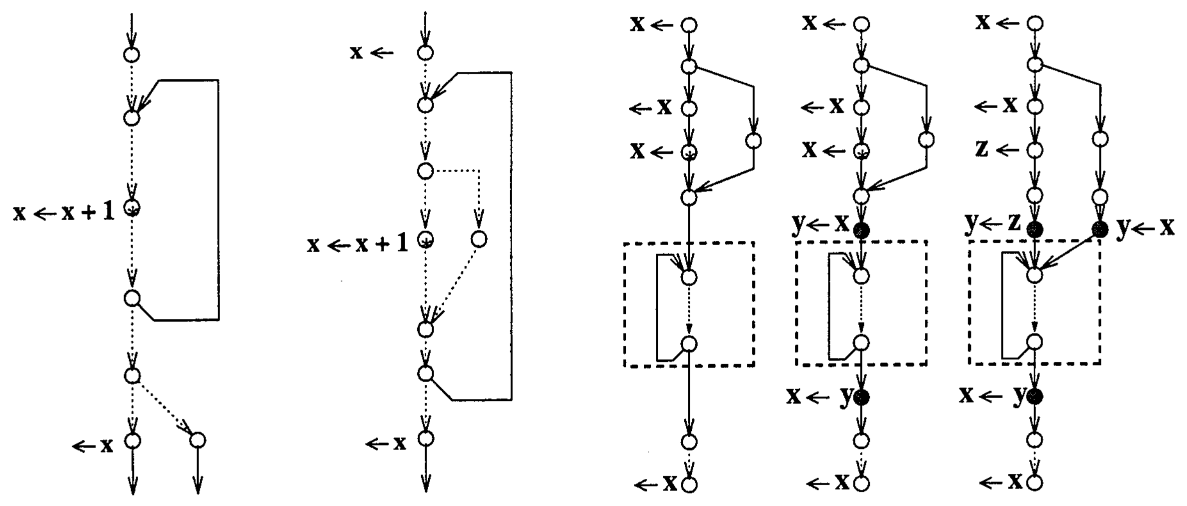
(b) Loop expansion, $N = 5$.

Figure 2.2 Node splitting, loop expansion, and renaming

register or a *hazard* that causes data inconsistency. A loop header is the beginning of the loop, and a loop tail is the node that is within the loop and has a directed connection to the loop header. *Live_in*(I) and *live_out*(I) are the sets of registers whose values have later uses immediately before and after node I respectively [22].

Scheme L resolves pseudo register hazards in three sequential stages, loop protection, node splitting, and loop expansion. All three stages may insert new nodes, which change the CFG, loop structure, and data flow information. Renaming is the primary technique for hazard resolution. Figure 2.2 illustrates how node splitting and loop expansion resolve hazards. A hazard node is denoted by a circle with a "*". In Scheme L, node I_j will be split due to hazard register x if $x \in$ *live_in*(I_j) and there is more than one definition of x that can reach I_j. Nodes are scanned sequentially in the node splitting process. Scheme L also derives the number of times a loop should be expanded to resolve hazards. To prevent some loop headers from being split, and to allow the targeted hazards to be renamed freely after loop expansion, save and restore nodes are inserted around loop headers, tails, and exit nodes. A loop can be protected either from outside or from inside. The following conditions are used to determine if a loop L should be protected for register x: C1. x is a hazard register which is live after the extended loop $\tilde{L}$ for register x; C2. L's header will be split due to its hazard register x; and C3. L's header will be split due to out of loop hazard register x.

The extended loop $\tilde{L}$ for register x consists of all nodes in L and all nodes I_1 satisfying the following rules: 1) $x \in$ *live_in*(I_1), 2) I_1 has only one successor, 3) I_1 has only one predecessor I_0, and 4) I_0 is in $\tilde{L}$. C2 may occur since some tail has more than one reaching definition and at least one is a hazard node defining x. If C1 or C2 is true, L is protected from inside. If



(a) Condition C1. (b) Condition C2. (c) Condition C3.

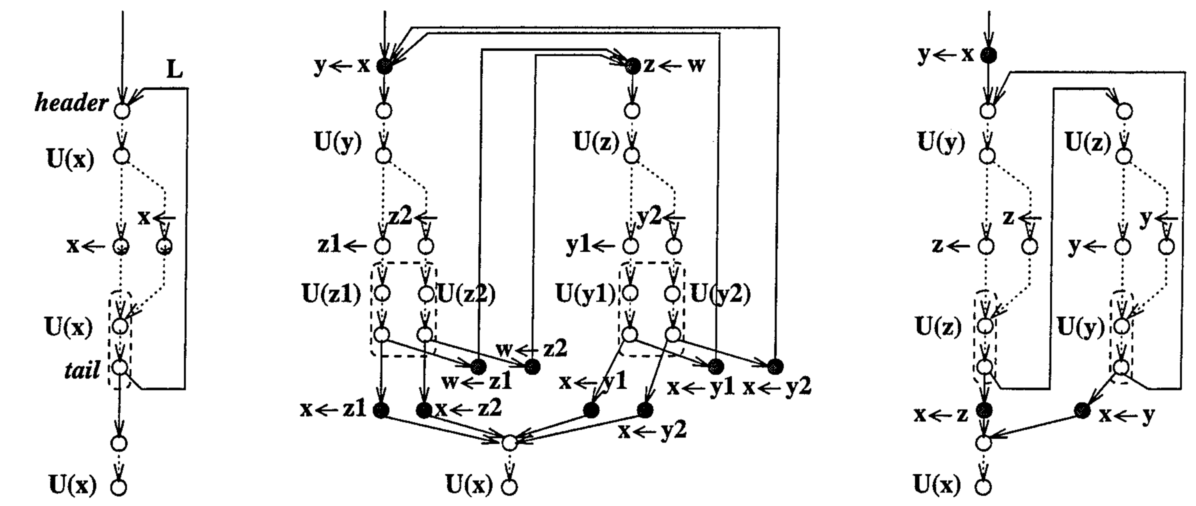
Figure 2.3 Conditions for loop protection

C3 is true, L is protected from outside. C1 is for $\tilde{L}$ instead of L since L may not have to be protected if all nodes in which x is live after L are in $\tilde{L} - L$. Checking C3 prevents L's header from being split, while observing C1 and C2 can confine x's live range to within each iteration of L, so that after loop expansion, x can be renamed correctly within each new loop copy.

Example Figure 2.3 illustrates the three conditions for loop protection. In Figure 2.3(c), to limit the code growth, the loop on the splitting path needs to be protected from outside for x. Dotted lines denote that there may be some nodes in between as long as they do not redefine register x.

2.5 Performance Enhancement Techniques – Pseudo Register Phase

Let $d(I_i, I_j)$ denote the minimum number of edges on any path from I_i to I_j, and $d_L(I_i, I_j)$ is similar to $d(I_i, I_j)$ except that all the nodes in the minimum length path must be within loop L. Let D_L denote the minimum number of edges from L's loop header to any of L's tail. $\{I_u, I_d\}$ is a *hazard pair* within loop L on register x if I_u uses x, I_d defines x, and $d_L(I_u, I_d) \leq N$. Register x is a *cut hazard register* in loop L if 1) there is a hazard node in L, defining x; and 2) any header to tail path within loop L has at least one node defining x.



(a) A loop segment. (b) Scheme L. (c) The new scheme.

Figure 2.4 Loop protection and cut hazard register

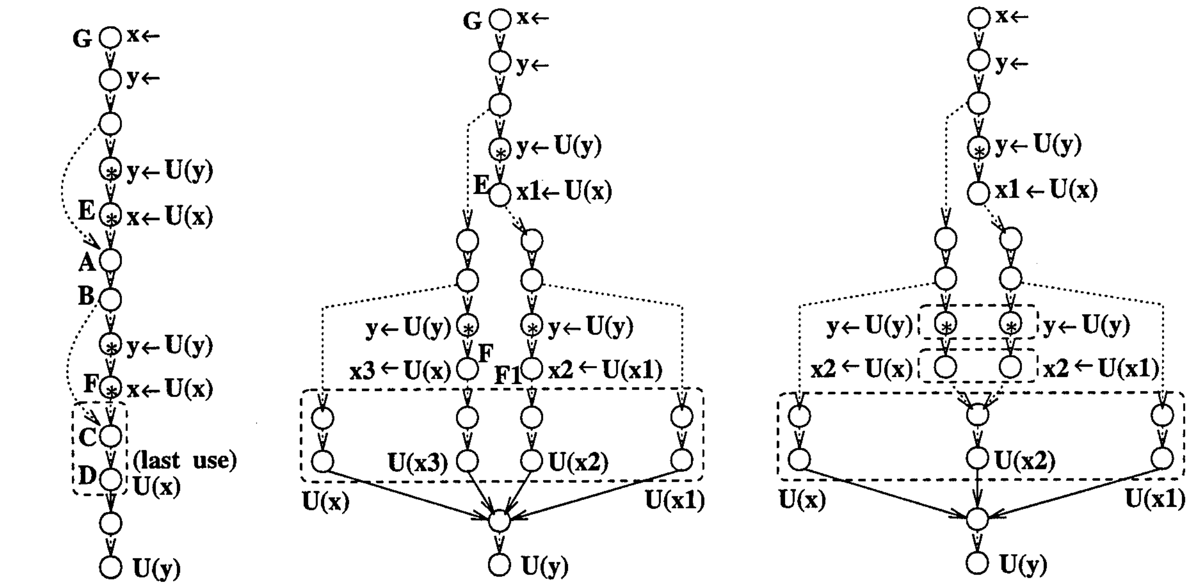
2.5.1 Loop protection

In Scheme L, if x is a hazard register inside loop L, and condition C1 or C2 is true, then L should be protected from inside for x. However, if x is also a cut hazard register, L can be renamed correctly after being protected from outside for x and expanded a sufficient number of times.

Example As shown in Figure 2.4(a), register x is such a cut hazard register. According to Scheme L, L is protected from inside since both conditions C1 and C2 are true. $U(x)$ represents using register x. Figure 2.4(b) illustrates the program segment after applying node splitting, loop expansion twice, and renaming. The shaded circles denote save and restore nodes for register x. Observing that every iteration of the loop redefines x, we can protect L from outside and still get the correct renaming after expanding the loop twice, as shown in Figure 2.4(c). The number of nodes is reduced, and the run time is improved since every iteration of loop L dose not execute save and restore nodes. Similarly register x within the loop in Figure 2.3(a) is also a cut hazard register.

2.5.2 Node splitting

A hazard node can be split if it is on the splitting path of some other hazards. Such new hazard nodes may cause redundant splittings in Scheme L. We have implemented a scheme

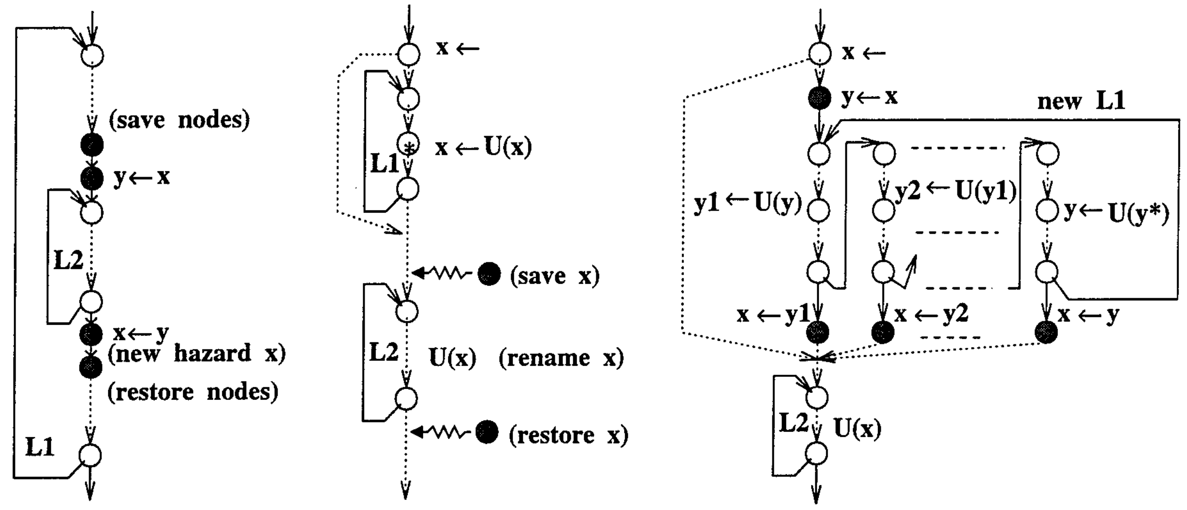


(a) A program segment. (b) Split and rename based on x. (c) The new node splitting strategy.

Figure 2.5 Node splitting enhancement

in which the number of copies for node I after splitting equals the number of original hazard reaching definitions (plus 1 if there is at least one non-hazard reaching definition). This can be done by using a stamp heap data structure [23], so that if a hazard node I is split into I, I_1, I_2, ..., I_{S-1}, then the stamp field of I_i, $i = 1, 2, \ldots, S-1$, points to I. The hazard nodes in the same heap will be assigned to the same new destination register if renaming is required. During the sequential scanning process, node I should be split due to hazard register x if 1) x is in *live_in(I)*, and 2) all reaching definitions of x that can reach I do not belong to the same stamp heap, assuming that all non-hazard nodes defining x belong to the same stamp heap.

Example Consider the program segment shown in Figure 2.5(a). We process hazard register x first. The nodes from A to B are split into two copies in Scheme L. However, the hazard node F, defining x, is also split since x is still live, resulting in two hazard nodes, F and F_1. Therefore, the nodes from C to D are split into four copies due to the hazard reaching definitions E, F, F_1, and the reaching definition G. By applying the stamp heap strategy, we can view F and F_1 as the same reaching definition. Only three copies are required from nodes C to D, as shown in Figure 2.5(c).



(a) The inner loop first rule. (b) Process L2 first. (c) Process L1 first.

Figure 2.6 Loop processing order

2.5.3 Loop and node processing order

Node splitting transforms all the hazards within the current loop across its backedges, while loop expansion resolves all such hazards. In this manner, when we process a given loop, there is no data hazard across the backedges of its inner loops. Therefore it is natural to process the loops from inside out so that the levels of data hazards can be successively reduced until all of them occur at the root level. The hazards at the root level then can be resolved by node splitting and renaming.

In addition to the inner loop first rule, we have to enforce the sequential order rule (top-down) to smoothly check condition C3 for parent hazard registers and to further eliminate extra save/restore nodes. Figure 2.6(a) illustrates the inner loop first rule that new hazards due to loop protection are propagated to the outer loop. The program segment in Figure 2.6(b) illustrates the sequential order rule. Suppose L_2 is processed first. Without enforcing the sequential order rule, L_2 may need to be protected from outside for register x. However, such protection is redundant if we process L_1 first and remove hazards that might affect L_2, as shown in Figure 2.6(c).

Breadth first search (BFS) is used to determine the processing order of nodes within loops or nodes of the entire program. The starting nodes may be the headers of loops or the root of the program. For some procedures, we have to modify the BFS algorithm by enforcing the following rules : 1) a node can be processed if and only if all of its parents have been processed, MBFS; and 2) reverse the direction of searching, RBFS.

2.5.4 Loop expansion

Our formula for the number of copies of L needed to resolve all on-path hazards within L is the same as the formula in Scheme L [1], with a slight modification. To simplify the analysis, we assume that loop L has a header I_h, and a single tail I_t. It can be easily extended to loops with multiple tails. Let $D_L = d(I_h, I_t)$. Assume that $\{I_u, I_d\}$ is a hazard pair within loop L for register x. The new formula includes the following cases: Case 1. The backedge (I_t, I_h) is not counted in $d_L(I_u, I_d)$; Case 2. The backedge (I_t, I_h) is counted in $d_L(I_u, I_d)$, and within L there exists a directed path that does not include (I_t, I_h) from I_d to I_u; and Case 3. The backedge (I_t, I_h) is counted in $d_L(I_u, I_d)$, and within L not considering (I_t, I_h), there is no directed path from I_d to I_u.

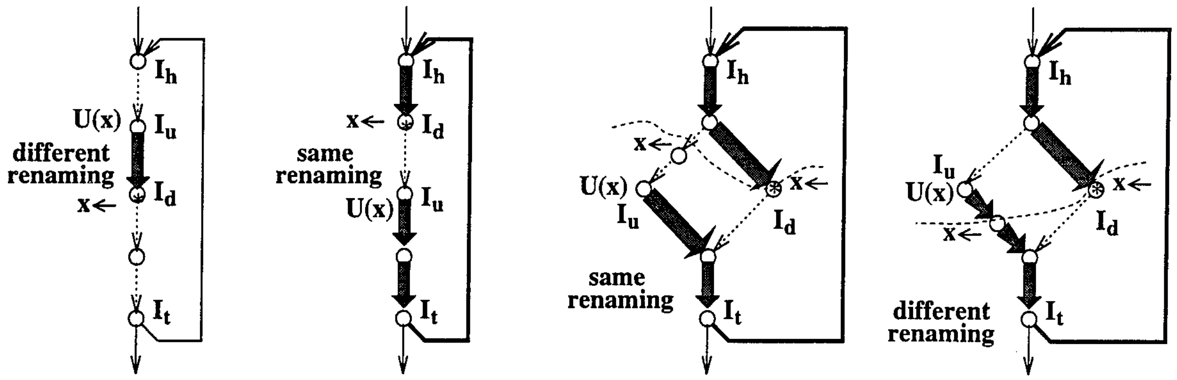
Suppose it takes K_1, K_2 and K_3 copies to resolve the hazard pair $\{I_u, I_d\}$ for each case respectively, where a value 1 denotes no replication. For case 1, since the use of x in I_u and the definition of x in I_d are renamed to different registers in the same loop iteration after expanding the loop K_1 times, the potential hazard distance would be from I_u through $(K_1 - 2)$ copies of L to the K_1th copy of I_d, which is $d(I_u, I_t) + (K_1 - 2)D_L + d(I_h, I_d) + K_1 - 1$, as shown in Figure 2.7(a). On the other hand, for case 2, both x's can be renamed to the same register, and the potential distance is from I_u through $(K_2 - 1)$ copies of L to the first copy of I_d, which is $d(I_u, I_t) + (K_2 - 1)D_L + d(I_h, I_d) + K_2$, as shown in Figure 2.7(b). These terms must be greater than N. Solving both inequalities, we have

$$K_2 = \left\lfloor \frac{N - d(I_u, I_t) - d(I_h, I_d) - 1}{D_L + 1} \right\rfloor + 2$$

and

$$K_1 = \begin{cases} 2 & \text{if } d(I_u, I_t) + d(I_h, I_d) + 1 > N \\ \left\lfloor \frac{N - d(I_u, I_t) - d(I_h, I_d) - 1}{D_L + 1} \right\rfloor + 3 & \text{otherwise} \end{cases}$$

For case 3, due to our observation concerning cut hazard registers, K_3 may be either K_1 or K_2, depending on if both x's in I_u and I_d can be renamed to different registers, as shown in Figure 2.7(c) and (d) respectively. For fixed $d(I_u, I_t)$ and $d(I_h, I_d)$, we have $K_1 = K_2 + 1$. Since case 3 rarely occurs, we choose $K_3 = K_1$ in our implementation. The number of copies of L needed to resolve all hazards within L is the maximum of all such K's.



(a) Case 1, K1. (b) Case 2, K2. (c) Case 3, K3 = K2. (d) Case 3, K3 = K1.

Figure 2.7 Cases for loop expansion

2.5.5 Self-anti-dependency

A node I is *self-anti-dependent* if I defines what it uses. For example, $x \leftarrow x + a$ is a self-anti-dependent node that uses and defines pseudo register x. This type of anti-dependency can be resolved by splitting I into two nodes : ($I_1 : y \leftarrow x + a$, $I_2 : x \leftarrow y$), and then inserting N nops between them [1,2]. However, using renaming with the aid of node splitting and loop protection, we can rename the definition of x to a new pseudo register without introducing a new node.

2.6 Incremental Updating

2.6.1 For on-path hazards – Scheme 0

Figure 2.8 shows the flowchart of the incremental scheme for resolving on-path hazards during the pseudo register phase. Three subroutines *loop-protection*, *node-splitting*, and *replicate-loop*, marked by "*", may insert new nodes to loops. Information associated with each node, including register live range, stamp heap and loop structure, is updated locally whenever a node is inserted.

Assume that L_i's immediate parent loop is L_j, which may be the entire program, and $\{I_u, I_d\}$ is a hazard pair for register x. The two cases in which we consider protecting L_i from outside for x are shown in Figure 2.9(a) and (b). In Figure 2.9(a), since the x in I_d will be renamed, we only need to check if there is any other definition of x, I_ω, that can reach I_h, and is not in the same stamp heap as I_d. The search for I_d is restricted to the shaded area, denoting

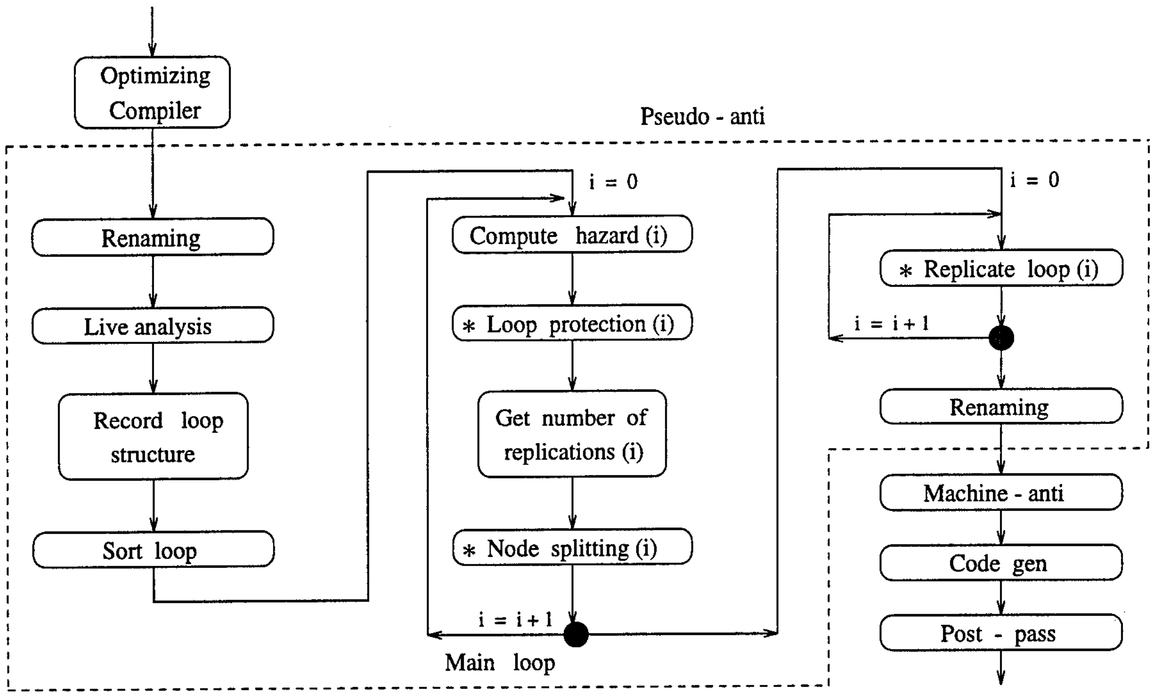
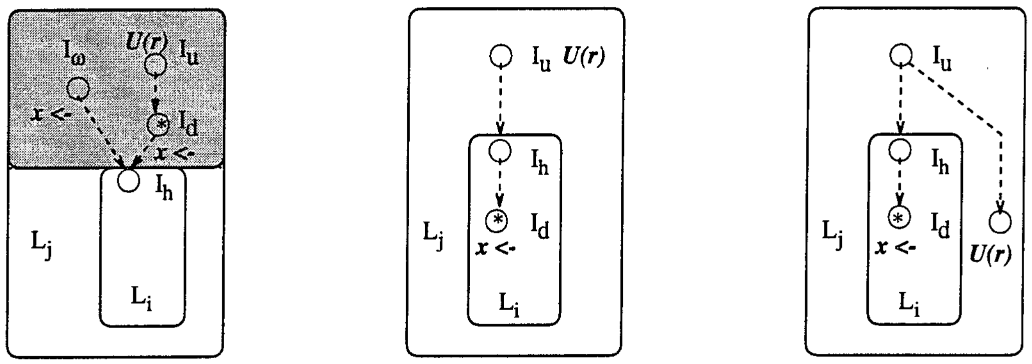


Figure 2.8 Incremental updating for resolving on-path hazards

the definitions within L_j that can reach I_h without going through backedges, but I_ω can be nodes in the upper levels that can reach I_h. The hazard in Figure 2.9(b) can also be resolved by expanding L_i a sufficient number of times and renaming registers within L_i. For simplicity, we protect L_i from outside for register x instead, so that the hazard is automatically resolved.

Subroutines *renaming*, *live-analysis*, *record-loop-structure*, and *sort-loop* are executed only once. The incremental scheme does not perform global DU-chain and global reaching definition analysis as Scheme L does, but rather performs a global live range analysis [22]. Loop structure and dataflow information *live_in* and *live_out* are maintained and updated locally throughout the computation.

Subroutine *compute-hazard* computes all hazard registers and hazard nodes within the current loop, bypassing inner loop hazards. It traverses nodes within L_i from the loop header in a BFS order. If node I defines x, it performs an RBFS traversal from node I up to distance N, but the search never leaves L_i. If there is a use of x within distance N, it records x a hazard register, and I a hazard node. Subroutine *loop-protection* protects loop L_i according to conditions C1, C2, and C3. Subroutine *get-number-of-replications* performs a BFS traversal to compute $d(I_h, I_\beta)$ and an RBFS traversal to compute $d(I_\alpha, I_t)$ for all nodes I_α, I_β in L_i. It



(a) Hazard splitting the loop header. (b) Across loop on-path hazard. (c) Across loop branch hazard.

Figure 2.9 The confinement of search nodes outside the loop

then computes K using the new formula, for every hazard pair $\{I_u, I_d\}$ in L_i. The maximum of all such values is the number of replications needed for L_i to resolve its hazards.

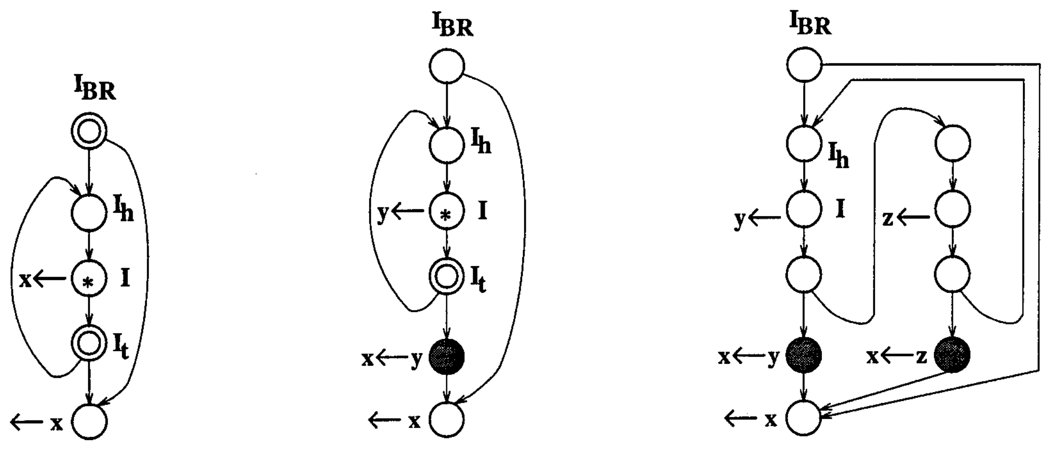
Subroutine *node-splitting* executes the criterion mentioned in the previous section, and scans the loop nodes in an MBFS order, bypassing the inner loops. Subroutine *replicate-loop* first marks the extended loop $\tilde{L_i}$ for all hazard registers, and then applies a BFS traversal to replicate $\tilde{L_i}$. The number of copies is obtained from *get-number-of-replications* subroutine.

As shown in Figure 2.8, each program loop is examined once. The actual code growth occurs after all loops have been inspected.

2.6.2 Incorporating branch hazards – Schemes 2 and 3

Branch hazards occur at branch boundaries when an error results in a wrong branch decision. The following criterion can be used to locate all branch hazards : Register x is a branch hazard if there exists a branch node I_{BR}, such that the distance from I_{BR} to a definition of x along one branch path of I_{BR} is within N, and x is *live* at the other branch paths of I_{BR}. By viewing x as if it is used at I_{BR}, renaming can resolve branch hazards as well as on-path hazards. Similar to the case shown in Figure 2.9(b), we modify the loop protection conditions. As shown in Figure 2.9(c), I_u is a branch node that does not use x, and x is live along one branch path of I_u. Loop L_i is protected from outside for register x, as if branch node I_u uses register x.

Example Consider the partial segment shown in Figure 2.10(a), and $N = 3$. Register x at node I is a branch hazard due to branch nodes I_{BR} and I_t, denoted by double circles. After loop protection as in Figure 2.9(c), and renaming x to y, the the register y at node I is a branch



(a) A program segment. (b) After loop protection on x. (c) After expanding twice and renaming.

Figure 2.10 The loop expansion for branch hazards, $N = 3$

hazard due to branch node I_t, as shown in Figure 2.10(b). Note that the save instruction $y \leftarrow x$ before the loop header I_h is removed since x is not live at I_h. In Figure 2.10(c), by expanding the loop twice and renaming, the branch hazard is resolved. The formula for the number of loop replications can also be modified by viewing the branch node as using the register x.

2.7 Post-Pass Code Rescheduling and Spill Register Reassignment

2.7.1 On-path hazards – Scheme 0

Although the pseudo register phase aims at removing on-path hazards within a function, new hazards may emerge after the machine register phase. First, the stack pointer adjustment instructions within the prologue segment and the epilogue segment create immediate self-anti-dependencies. Second, before calling a procedure, the registers used as parameters need to be saved before the new values can be loaded. Register spilling may also create on-path hazards. When a register is to be spilled, most likely it will be loaded with new values, thus creating a use-before-definition scenario. A straightforward post-pass nop insertion algorithm was employed in Scheme L to resolve these new hazards. Sufficient nops are inserted before the hazard definitions to force all anti-dependency distances exceeding N.

Code rescheduling is applied within the prologue and the epilogue segments, and a register reassignment algorithm for rearranging spill registers, so that the total number of nops inserted is greatly reduced. The post-pass algorithm includes the following steps : 1) reassign spill registers; 2) reschedule code and insert nops in the prologue segment; 3) reschedule code and insert nops in the epilogue segment; and 4) insert remaining nops.

The original version of the IMPACT C compiler [21] reserves three registers, \$3, \$24, and \$25, as spill registers. The spill registers perform two functions to access memory, *load* and *store*. The compiler generates instructions of the following groups for load and store functions respectively, where $\$r_1$ and $\$r_2$ are different spill registers, and are dead after the second (or the third) instruction :

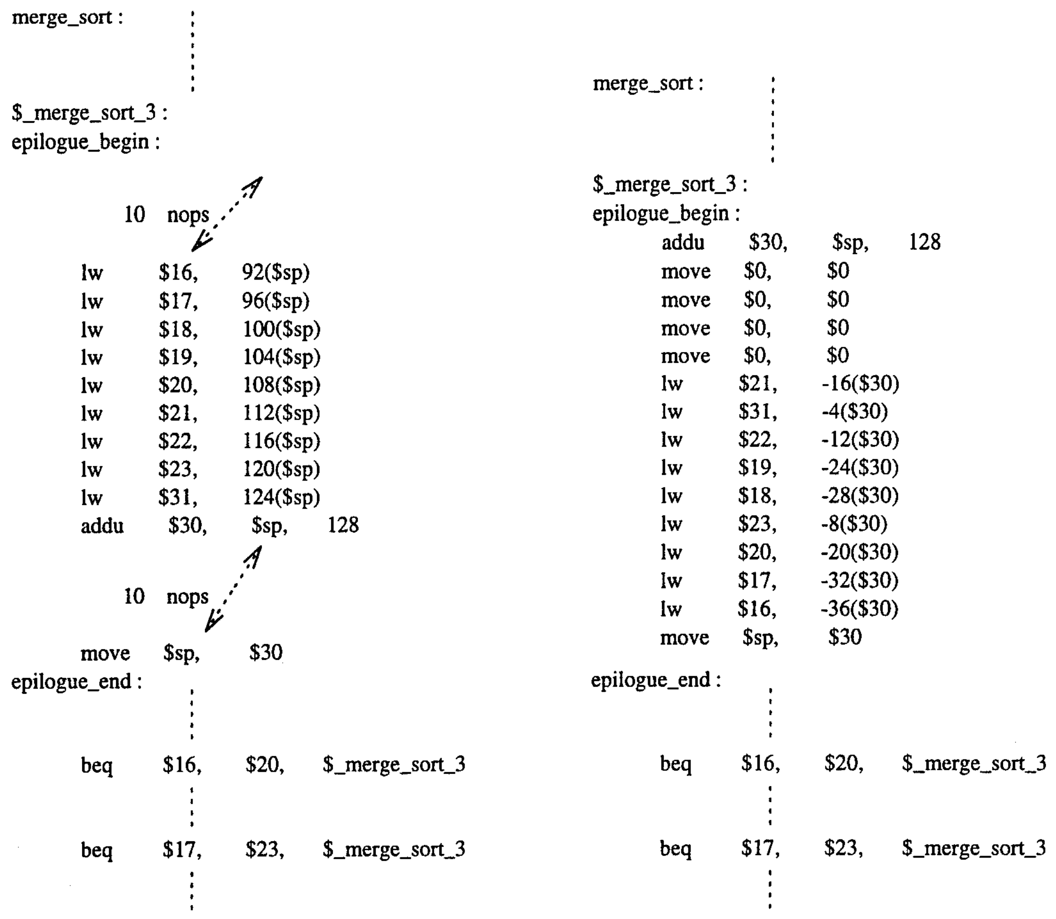
load $\$r_1$, memory;	load $\$r_1$, $memory_1$;	operation defining $\$r_1$;
use $\$r_1$;	load $\$r_2$, $memory_2$;	store $\$r_1$, memory;
	use $\$r_1$, $\$r_2$;	

Spill registers serve as temporaries and have very short live ranges, i.e., 2 or 3. On-path hazards occur when two groups of spill code use the same spill register and their distance, from the use of the first group to the definition of the second group, is less than or equal to N. The goal is to minimize the number of nops needed to resolve all hazards. Our approach is to utilize dead registers as substitutes within groups so that the sum of all the anti-dependency distances for spill registers and substitutes is maximized, considering the anti-dependency distance between groups of different spill registers and substitutes $N + 1$. In general, this problem is NP-hard, which includes as a special case the following NP-complete problem after determining that only spill registers are dead registers, and $N = 1$:

Given K colors, an undirected graph G and an integer n, is there a node coloring such that the number of edges with the same colors at both ends is at most n?

This can be proven by restricting n to 0, and it becomes the K-colorability problem [24]. However, we propose a simple heuristic algorithm to reassign spill registers within groups in a BFS traversal of the entire program. We always choose as a substitute the register which is dead before and after the group, and whose sum of the distance backward to the first use and the distance forward to the first definition is maximum.

The prologue segment includes code to adjust the stack pointer and to save the values of local registers to memory. The epilogue segment includes code to retrieve the original register



```
merge_sort :      :
                  :
                  :
$_merge_sort_3 :
epilogue_begin :

          10   nops

          lw      $16,    92($sp)
          lw      $17,    96($sp)
          lw      $18,    100($sp)
          lw      $19,    104($sp)
          lw      $20,    108($sp)
          lw      $21,    112($sp)
          lw      $22,    116($sp)
          lw      $23,    120($sp)
          lw      $31,    124($sp)
          addu    $30,    $sp,    128

          10   nops

          move    $sp,    $30
epilogue_end :    :
                  :
          beq     $16,    $20,    $_merge_sort_3
                  :
          beq     $17,    $23,    $_merge_sort_3
                  :
```

(a) Scheme L.

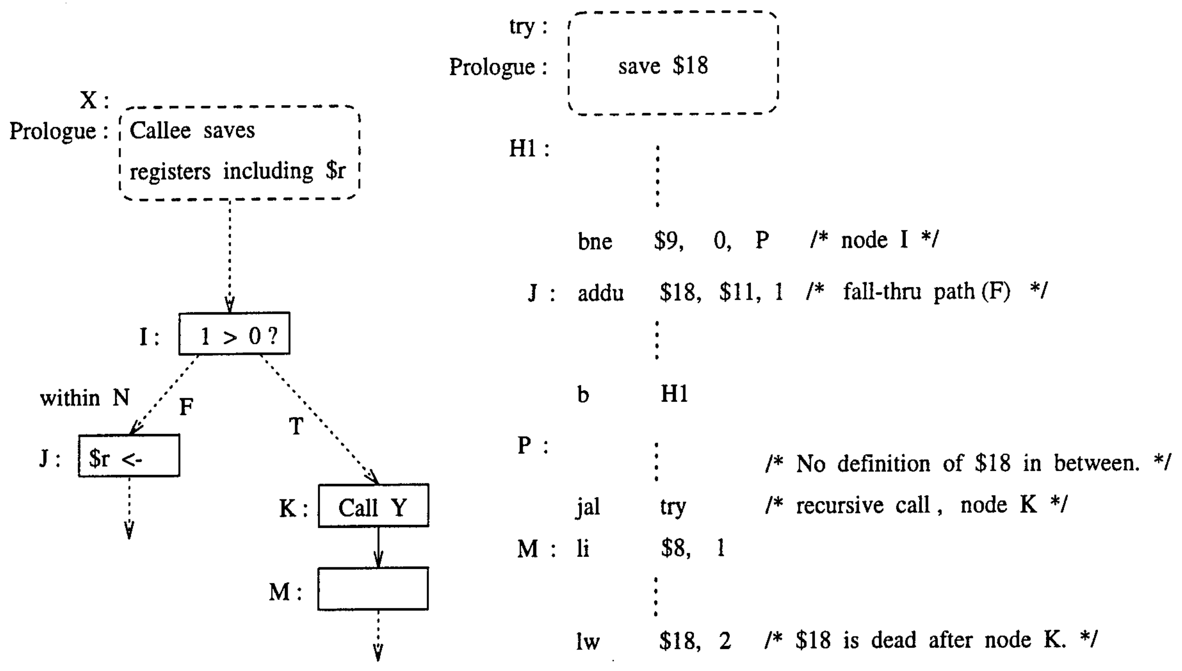
```
merge_sort :      :
                  :
                  :
$_merge_sort_3 :
epilogue_begin :
          addu    $30,    $sp,    128
          move    $0,     $0
          move    $0,     $0
          move    $0,     $0
          move    $0,     $0
          lw      $21,    -16($30)
          lw      $31,    -4($30)
          lw      $22,    -12($30)
          lw      $19,    -24($30)
          lw      $18,    -28($30)
          lw      $23,    -8($30)
          lw      $20,    -20($30)
          lw      $17,    -32($30)
          lw      $16,    -36($30)
          move    $sp,    $30
epilogue_end :    :
                  :
          beq     $16,    $20,    $_merge_sort_3
                  :
          beq     $17,    $23,    $_merge_sort_3
                  :
```

(b) The new scheme.

Figure 2.11 Post-pass code rescheduling for an epilogue segment, $N = 10$

values from memory and to adjust the stack pointer. The last step simply performs a BFS traversal, and inserts nops to resolve all remaining on-path hazards.

Example Figure 2.11(a) shows the epilogue segment processed by Scheme L in post pass, for $N = 10$. Figure 2.11(b) illustrates how the register assignment and code rescheduling are used to eliminate 16 nops in the epilogue segment. Instruction 'addu $30, $sp, 128' has been moved backward up to before all instructions of loading local registers, with the base register being replaced by $30. The instructions to load local registers are rescheduled according to their distances from the first uses of corresponding registers. The four instructions loading registers $16, $17, $20, and $23 are thus moved to the end of the load instructions. Four more nops are needed to resolve the hazard register $23.



```
try :
Prologue :  save $18

H1 :
         :
      bne   $9,  0,  P    /* node I */
J  :  addu  $18, $11, 1  /*  fall-thru path (F)  */
         :
      b     H1
P  :
         :        /* No definition of $18 in between. */
      jal   try   /* recursive call,  node K */
M  :  li    $8,  1
         :
      lw    $18, 2   /* $18 is dead after node K. */
```

(b)

Figure 2.12 Register live range across procedure boundaries

2.7.2 Both types of hazards – Schemes 1, 2, and 3

Post-pass nop insertion can also resolve extra branch hazards generated by the machine register allocator. The branch hazard check can be incorporated in the original on-path hazard check. The heuristic to reassign spill registers has to be modified as follows. The register we choose to replace the reserved spill register at a specific group G of spill instructions must be not only dead before and after G, but also requires as few nops as possible to resolve the new branch hazard induced by the substitute register.

The above schemes for incorporating branch hazard resolution do not create extra hazards across procedural boundaries. However, depending on implementations, the callee-saved registers may have a performance impact due to separate compilations. As shown in Figure 2.12(a), suppose at branch node I, a wrong decision is made. After rollback and a correct decision at I, register $\$r$ has a wrong value. If $\$r$ is in Y's callee-saved register set, then $\$r$ is live along I's target (T) branch. Several nops should be inserted between I and J to resolve such branch hazard. However, since Y's callee-saved register set are unknown at current procedure X, a conservative scheme may assume that the registers are all in the set, e.g., \$16, \$17, $\cdots$, \$23 in the original version of the IMPACT C compiler. By viewing K as a node that uses such set, we can incorporate it in the initial global live range analysis.

For library routines, a built-in table holding corresponding saved register sets can be attached to the compiler to relieve the situation described above. The following checking can determine $\$r$'s live range before the procedure call, regardless of whether $\$r$ belongs to the callee-saved register set. $\$r \in live\_in(M)$ *iff* $\$r$ is live at node K, where M is the next instruction following the subroutine call node K. Such live range checking starting from M should skip any subroutine call encountered.

Example Figure 2.12(b) is an assembly code segment for the recursive function *try* . Without checking the additional condition, N nops are inserted between node I and node J to eliminate the hazard \$18. None is required by observing \$18 is dead after node K. Code run time performance is improved since such N nops are within a loop.

2.8 Performance Evaluation

Implementation and performance benefits of the schemes are evaluated on a set of twelve programs cross-compiled on a SPARC server 490 by the IMPACT C compiler with the hazard removal schemes, and executed on a DEC station 3100. The benchmarks and descriptions are as follows: QUEEN(148), 8-queen program; QSORT(261), recursive quick sort algorithm; PUZZLE(877), a game; WC(181), CMP(251), GREP(926), COMPRESS(1828), UNIX utilities; EQN(6251), mathematics typesetting program; LEX(6873), lexical analyzer; YACC(8099), parser generator; CCCP(8775), preprocessor for gnu C compiler; and TBL(9191), table formatter. The number within the parentheses is the number of instructions generated by the original version of the IMPACT C compiler without removing hazards. The chosen benchmarks consist of a variety of typical program constructs including sequential single loops, highly nested loops, and recursive functions.

2.8.1 Resolving on-path hazards – Scheme 0 vs. Scheme L

The incremental updating scheme and the post-pass code rescheduler improve application compile time, run-time performance, and reduce code growth for most applications studied. In this section we compare the performance impact of Scheme 0 and Scheme L with respect to the

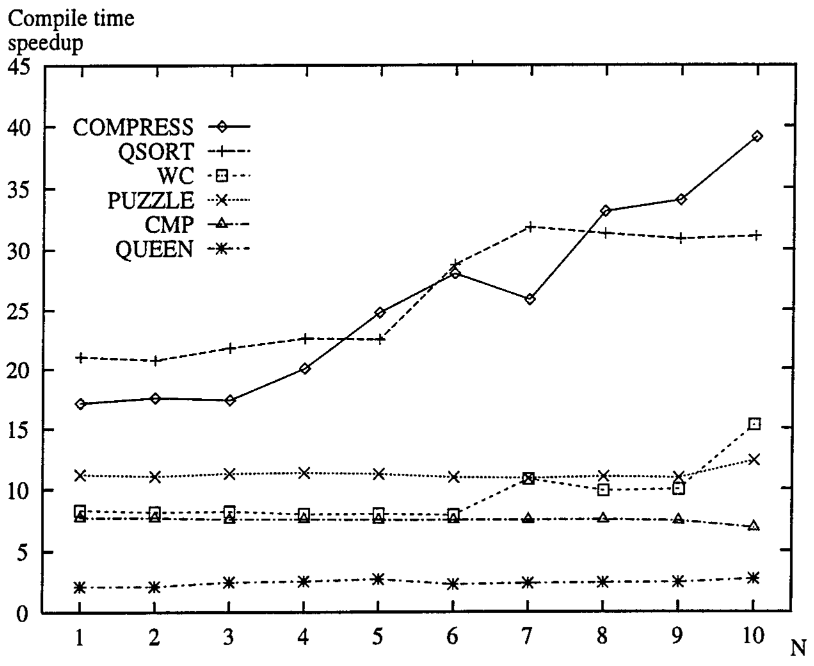


Figure 2.13 Compile time speedup

compile time, code run time and code size. We investigate the same set of benchmarks used in [1]: CMP, COMPRESS, PUZZLE, QSORT, QUEEN, and WC.

For $N = 10$, Scheme L requires more than 8 minutes, 15 seconds, 1.5 minutes, 3.5 minutes, and 9.5 minutes compiling QSORT, QUEEN, CMP, WC, and PUZZLE respectively, while Scheme 0 takes compile time less than 16 seconds, 8 seconds, 15 seconds, 15 seconds and 50 seconds respectively. COMPRESS has the best compile time improvement. Scheme L spends more than an hour for $N = 7$, 8, and 9, and almost two hours for $N = 10$ on compilation, while Scheme 0 compiles in less than 3 minutes. Figure 2.13 shows the compile time speedup which is the compile time ratio between Scheme L and Scheme 0.

Table 2.2 lists code run time overhead for both Schemes L and 0 respectively. The base of comparison is the original code run time. Some benchmarks, e.g., CMP and PUZZLE, have improved performance, as shown by negative numbers. The register allocator, nop inserter, and spill register reassignment involve heuristic algorithms that in some cases provide improved performance under loop expansion. The MIPS post-pass code reorganizer also sometimes changes the execution order for different N. Two benchmarks, QSORT, and QUEEN, include recursive functions and have among the largest run-time enhancements, for $N > 5$. Post-pass code rescheduling is a significant contributor to these two benchmarks.

Table 2.2 Code run time overhead for Schemes L and 0

N		1	2	3	4	5	6	7	8	9	10
QSORT	L	6.2%	8.3%	8.3%	10.4%	11.5%	13.5%	14.6%	26.0%	22.9%	30.2%
	0	5.2%	6.2%	6.2%	8.3%	8.3%	10.4%	10.4%	13.5%	15.6%	16.7%
QUEEN	L	3.0%	5.3%	7.2%	7.2%	9.0%	9.8%	11.5%	15.8%	16.3%	20.9%
	0	2.9%	3.5%	3.9%	4.9%	5.1%	5.5%	6.0%	8.0%	10.2%	16.3%
CMP	L	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%	-1.8%
	0	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%	-2.4%
WC	L	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	3.8%	4.4%
	0	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.3%	1.3%
PUZZLE	L	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%
	0	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	-0.7%	0.0%	0.0%
COMPRESS	L	-0.6%	0.0%	0.0%	0.0%	1.2%	2.5%	5.6%	6.2%	11.2%	18.8%
	0	-0.6%	-0.6%	-0.6%	-0.6%	1.2%	1.2%	5.0%	5.6%	10.6%	16.9%

Table 2.3 Code size overhead for Schemes L and 0

N		1	2	3	4	5	6	7	8	9	10
QSORT	L	63%	70%	105%	115%	123%	136%	154%	199%	219%	274%
	0	101%	104%	105%	110%	118%	130%	138%	146%	169%	191%
QUEEN	L	57%	69%	124%	134%	152%	164%	176%	208%	219%	310%
	0	48%	53%	58%	68%	78%	127%	132%	147%	151%	179%
CMP	L	75%	80%	92%	107%	120%	141%	158%	179%	200%	228%
	0	60%	63%	67%	76%	82%	84%	88%	90%	94%	122%
WC	L	133%	138%	160%	167%	179%	216%	245%	249%	257%	290%
	0	153%	155%	160%	163%	164%	165%	187%	205%	209%	244%
PUZZLE	L	80%	80%	87%	89%	91%	94%	96%	101%	106%	126%
	0	79%	79%	81%	84%	85%	87%	96%	99%	101%	111%
COMPRESS	L	28%	32%	38%	52%	60%	69%	80%	94%	107%	129%
	0	70%	73%	74%	78%	82%	87%	108%	122%	152%	156%

Table 2.3 lists the code size overhead for Schemes L and 0. The base of comparison is the number of instructions in the original code. COMPRESS has larger code growth in Scheme 0 due to the removal of the 800 instruction threshold [1] and the change in the number of functions compiled in simplified mode which bypasses the rest of pseudo register hazard resolution except the breaking of self-anti-dependent instructions, after exceeding the threshold. In Scheme 0, QSORT and WC have larger code growth when $N = 1$ and 2. Loop expansion is the major stage that results in most of the code growth. In Scheme L proper renaming after protecting the loop from inside and node splitting for small N may prevent the loop from being expanded. Also, if the loop is protected for several registers from inside, the hazards can be removed after arranging the order of the save/restore nodes, and renaming without actually expanding the loop. However, using the cut hazard register technique, as in Scheme 0, to move save/restore nodes out of the loop L requires L to be expanded at least once.

2.8.2 Resolving on-path and branch hazards – Schemes 1, 2, and 3

Schemes 1, 2, and 3 deal with removing both types of hazards during three separate phases. Scheme 1 has the fastest compilation speed since it postpones the branch hazard resolution to the last phase.

All three schemes perform relatively the same for the twelve benchmarks studied. Reasons for this behavior include 1) the occurrences of branch hazards are infrequent; 2) both machine register and nop insertion phases employ heuristics, and the spill register reassignment heuristic may be efficient enough to resolve branch hazards in the post-pass; and 3) resolving branch hazards at the pseudo register phase or the machine register phase is likely to have larger code growth, due to the extra node splitting and loop expansion. In most benchmarks, Scheme 1 even outperforms the other two schemes in both code run-time and code growth.

The performance overhead of Scheme 1 is tabulated in Table 2.4. Due to the heuristic algorithm employed in the post-pass phase, the performance overhead observed is not monotonically increasing according to N. However, the code generated to allow N instruction rollback is correct for $N - 1$ instruction rollback as well. Therefore, the overhead can be recorded as non-decreasing. Several functions generate more than 15,000 nodes, which increases the computation time for the machine register assignment phase, when $N > 6$. YACC has two such functions, and CCCP has one. For these three functions, we resolve the rollback hazards of

Table 2.4 Run time overhead for Scheme 1

N	1	2	3	4	5	6	7	8	9	10
QSORT	6.2%	6.2%	7.3%	9.4%	9.4%	12.5%	12.5%	16.7%	18.7%	18.7%
QUEEN	2.8%	3.1%	4.1%	5.7%	6.3%	6.7%	7.4%	11.1%	11.2%	18.0%
CMP	-3.0%	-3.0%	-3.0%	-3.0%	-3.0%	-3.0%	-2.4%	-1.8%	-1.2%	-1.2%
WC	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%	1.3%
PUZZLE	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.7%	0.7%	0.7%
COMPRESS	1.3%	2.0%	2.6%	4.0%	7.3%	9.3%	9.9%	11.3%	13.9%	17.9%
GREP	11.1%	11.1%	11.1%	11.1%	11.1%	13.0%	13.0%	13.0%	14.8%	24.1%
LEX	10.5%	11.6%	11.6%	11.6%	11.6%	11.6%	12.8%	14.0%	14.0%	18.6%
EQN	7.8%	11.3%	12.2%	12.2%	12.2%	12.2%	13.9%	13.9%	13.9%	13.9%
YACC	0.0%	0.0%	2.4%	2.4%	2.4%	*7.1%	*11.9%	*16.7%	*23.8%	*28.6%
CCCP	8.5%	9.3%	10.1%	11.6%	11.6%	*17.1%	*17.1%	*19.4%	*20.9%	*26.4%
TBL	5.3%	7.9%	7.9%	7.9%	7.9%	7.9%	14.5%	14.5%	14.5%	15.8%

distance 5 in the pseudo register phase, and then resolve the rollback hazards of distance $N > 5$ in the post-pass phase, as marked by "*" in Table 2.4.

Figure 2.14 depicts the percentage of hazard nodes that are branch hazard nodes but are not on-path hazard nodes, for various rollback distances. Benchmarks QUEEN and QSORT have 0 percentage for N within 10 because either they have no branch hazards, or all of their branch hazards are also on-path hazards. PUZZLE has the highest percentage of branch hazard nodes, 42.42% when $N = 3$. There is a significant rise from $N = 2$ to $N = 3$ due to the relative distances between branch nodes and hazard nodes. This can explain why in Scheme A, PUZZLE has the highest run-time overhead 10% when $N = 10$ [2]. The post-pass algorithms apparently reduce the overhead to 0.7%, as shown in Table 2.4. All the other benchmarks have less than a quarter of the hazard nodes that are branch hazard nodes but not on-path hazard nodes.

2.9 Summary

The previous compiler-based multiple instruction retry resolves data hazards associated with rollback recovery [1]. The proposed approach eliminates the delayed write buffer for the register file, but suffers from long compilation times. Incremental updating has been incorporated in the compiler-based approach, resulting in significantly reduced compile times. The code in the prologue and the epilogue segments was rescheduled, and the spill registers were reassigned to reduce the total number of nops inserted. The threshold for the number of nodes increased from 800 to 15,000. Branch hazards were shown to be resolvable by simple modifications to

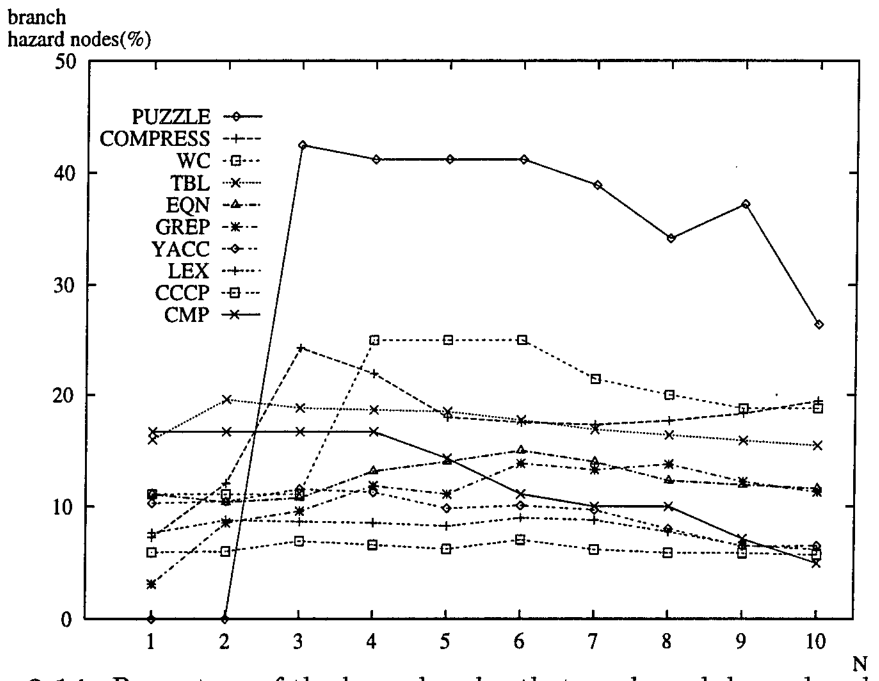


Figure 2.14 Percentage of the hazard nodes that are branch hazard nodes

the proposed approaches. Three schemes were implemented and compared. Postponing the resolution of branch hazards to the last phase was shown to provide the fastest compilation. This approach also typically generated code as good as the alternatives in both code run time and code growth.

The hardware/compiler combined approach [2] is another alternative, resolving on-path hazards with a read buffer, and branch hazards with a compiler. The combined approach requires less hardware than the micro rollback scheme [17], and results in less performance penalty than using the compiler alone.

Chapter 3

Application of Multiple Instruction Retry to VLIW Architectures

3.1 Introduction

VLIW machines can simultaneously execute multiple instructions which are grouped as an instruction word [25–27]. General VLIW architectures consist of multiple functional units. Instruction words include several operation fields, each of which controls one specific functional unit. The functional units typically operate in a single instruction stream with lock-step timing control. Significant speedup can be achieved if the machine can execute more than one operation within each machine cycle. However, due to data dependencies and resource constraints, the parallelism may not be fully exploited. Scheduling techniques such as trace scheduling [28,29], superblock scheduling [30], and software pipelining [31] can effectively increase performance, especially for scientific applications where conditional branches are highly predictable.

The application of concurrent error detection, signature monitoring, and TMR to VLIW has recently been described by several researchers [32–34]. We describe two compiler assisted multiple instruction word retry schemes for VLIW architectures. The first scheme is a compiler-based approach which schedules the compiler-generated hazard-free code with various rollback distances for scalar processors, using a modified trace scheduling algorithm [28]. Nops are then inserted in the compacted code to resolve the remaining data hazards. The performance impact is measured by N, the rollback distance for a scalar processor; P, the number of functional units; and n, the rollback distance for a VLIW architecture. The second scheme uses a read buffer (not a write buffer) of n deep and $2 \times P$ wide to hold all reads within the last n instruction words executed. Such a mechanism can resolve frequent data hazards, while the remaining class of data hazards is resolved by the compiler [2].

3.2 Machine Model

The machine model consists of several functional units, each of which has two read ports, and one write port connecting to a general register file. Memory is accessed by loading and storing from the register file. The functional units operate simultaneously accessing the register file, but do not support pipelining.

Transient errors may occur in any of the functional units due to an incorrect read from the register file, incorrect arithmetic operations performed by the functional units, or incorrect branch decisions. An error detection mechanism triggers a rollback within n instruction words (cycles) from the inception of the error. Register file contents do not spontaneously change, and data writes can not be written into incorrect registers. Up to n instruction word write buffers are associated with the memory and I/O device [17], so that they can have their own rollback capability. To facilitate instruction word retry, a history file of size n serves as a shadow file for the program counter [27].

Data hazards are those that cause inconsistencies during several retries of the same instruction word sequence. Similar to scalar processors, two types of hazards are classified for the machine model. *On-path hazards* [1] appear in the form of anti-dependencies [20] where the retry of the same read-write path is inconsistent with the previous run due to the possible incorrect write destroying the value needed by the read during retry. *Branch hazards* [2] occur at branch instructions where an incorrect decision causes a register to be defined at a wrong branch path while it is *live* [22] at the other branch path.

3.3 Approach

3.3.1 Compiler-based scheme for VLIW architectures

The instruction retry scheme for scalar processors can be naturally extended to VLIW architectures. A modified trace scheduling algorithm has been implemented for trace-based simulation. Profiling is implemented to guide the trace selection [28, 29]. The most frequently

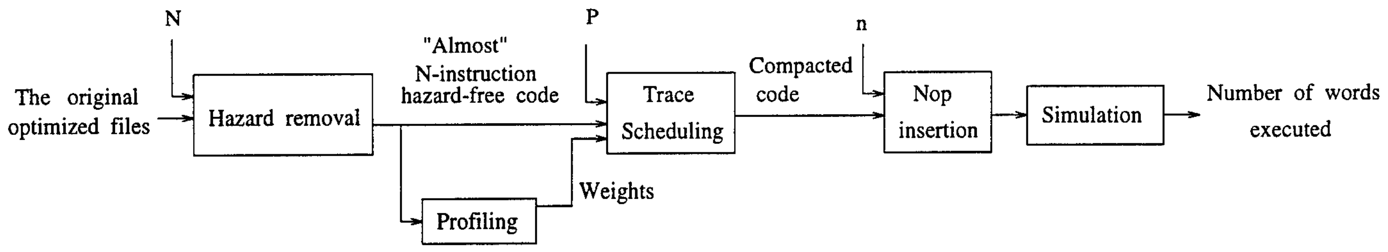
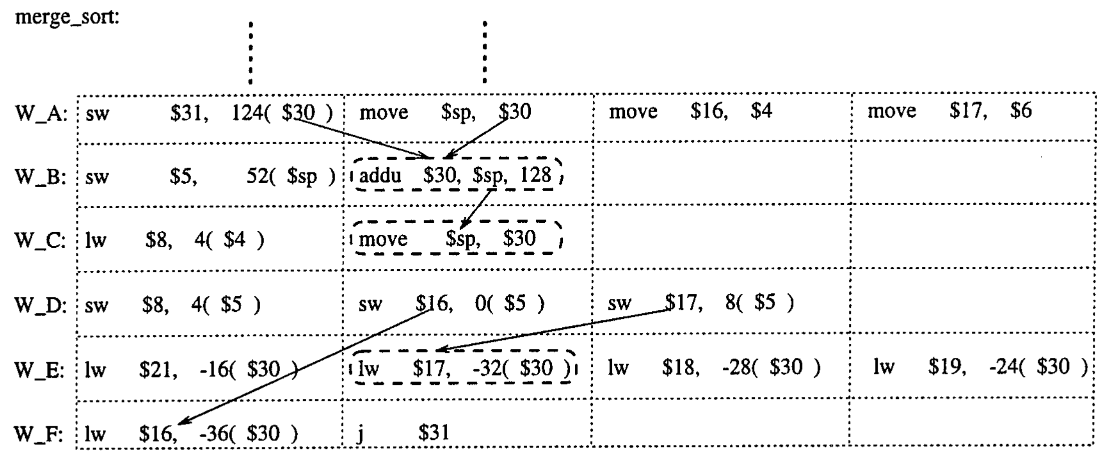


Figure 3.1 Hazard removal, scheduling and simulation for VLIW architectures

executed path is scheduled first. List scheduling operates on the selected trace by building a data dependence graph and successively scheduling the instructions whose predecessors in the dependence graph have all been scheduled. To maintain the correct program semantics, redundant code will have to be inserted in the unscheduled program segments. In our current implementation, we do not implement multiple jump instructions within a word.

Given a rollback distance N for scalar processors, we can apply our previous approach to generate N-instruction hazard-free code. One direct extension would schedule such hazard-free code by bounding groups of nops together so that they still serve as delimiters for hazards in the compacted code. Although an easy modification to the data dependence graph building process in the list scheduling can achieve this, such groups of nops will tend to block the code motion around them, and as a result reduce the available parallelism. We thus choose an alternative approach. The code generated after the code reordering but before the final nop insertion is scheduled. Nop instruction words are then inserted to resolve both types of hazards in the compacted code. Figure 3.1 outlines the entire process for hazard removals, scheduling and simulation.

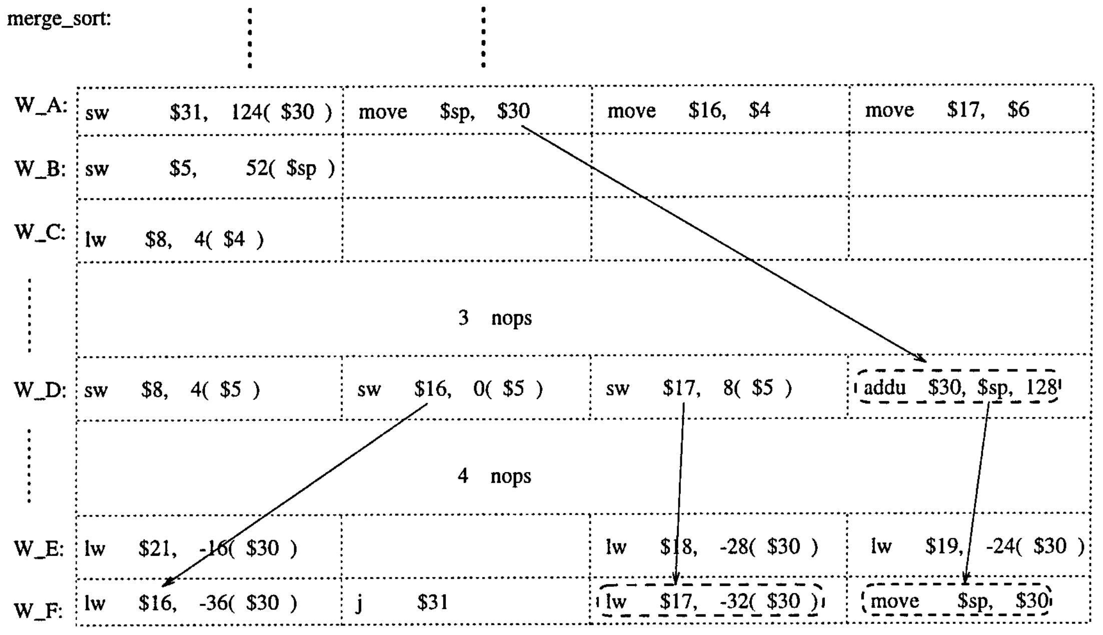
Further enhancement can be made by introducing priorities for selecting feasible instructions during list scheduling. For example, consider the instruction word segment of QSORT generated by trace scheduling, as shown in Figure 3.2(a). Arrows denote anti-dependencies which should be separated by n instruction words (in this example $n = 5$). A total of 15 nops are required to resolve all on-path hazards, where 5 nops for each gap between words W_A and W_B, between W_B and W_C, and between W_D and W_E respectively. By employing the prioritized list scheduling, the number of nops can be reduced. When there are more than one instruction whose predecessors in the dependency graph have all been scheduled, we schedule those instructions with the longest dependence chain first. In computation of such a chain, flow dependency



merge_sort:

W_A:	sw $31, 124($30)	move $sp, $30	move $16, $4	move $17, $6
W_B:	sw $5, 52($sp)	addu $30, $sp, 128		
W_C:	lw $8, 4($4)	move $sp, $30		
W_D:	sw $8, 4($5)	sw $16, 0($5)	sw $17, 8($5)	
W_E:	lw $21, -16($30)	lw $17, -32($30)	lw $18, -28($30)	lw $19, -24($30)
W_F:	lw $16, -36($30)	j $31		

(a) The compacted code segment for QSORT



merge_sort:

W_A:	sw $31, 124($30)	move $sp, $30	move $16, $4	move $17, $6
W_B:	sw $5, 52($sp)			
W_C:	lw $8, 4($4)			
	3 nops			
W_D:	sw $8, 4($5)	sw $16, 0($5)	sw $17, 8($5)	addu $30, $sp, 128
	4 nops			
W_E:	lw $21, -16($30)		lw $18, -28($30)	lw $19, -24($30)
W_F:	lw $16, -36($30)	j $31	lw $17, -32($30)	move $sp, $30

(b) The enhanced word schedule

Figure 3.2 Enhanced list scheduling algorithm

and output dependency are counted one, and anti-dependency contributes $n + 1$. Also every instruction has an assumed schedule time. An instruction can be scheduled at the current word if its assumed schedule time is not greater than the current time. The schedule time is updated when any of its predecessors is scheduled. Nops are inserted on the fly if there is no available instruction that can be scheduled at the current time. As illustrated in Figure 3.2(b), the longest chain has length 12, and the schedule for instructions 'addu \$30, \$sp, 128' and 'move \$sp, \$30' can be postponed to words *W_D* and *W_F* respectively. The number of nops needed is now 7.

3.3.2 Hardware-software combined scheme for VLIW architectures

The second scheme employs a read buffer [2] to backup all register values read within the last n instruction words executed. The depth of the read buffer is n, while the width is $2 \times P$, since each instruction can read at most two registers, and there are at most P instructions in a word. Such a hardware scheme can capture all on-path hazards. By inserting dummy instructions of the form 'move \$r, \$r', after the branch node along the path that defines register $\$r$ within distance n, such branch hazards can be resolved, since the old value of register $\$r$ is now in the read buffer due to the read in the dummy instructions.

The same strategy can be applied in the compacted code word so that all branch hazards can be treated as on-path hazards, and subsequently resolved by the read buffer. $\lceil \frac{B}{P} \rceil$ dummy words can be inserted after the instruction word containing a branch instruction I_{BR}, where B is the number of hazard registers for I_{BR} along one branch. Actually, by utilizing dead registers at I_{BR}, and some 2-operand instructions, e.g., *add*, only $\lceil \frac{B}{2P} \rceil$ dummy words are required.

3.4 Analytical Bookkeeping Code Scheduling

Superscalar and Very Long Instruction Word (VLIW) architectures exploit fine-grain parallelism to achieve better performance. Static scheduling techniques, such as trace scheduling [28] and superblock scheduling [30], can effectively produce compact code for these architectures. In this section, we present an analytical approach for bookkeeping in code scheduling that alleviates the coding complexity and instruction duplication limitations of the previous approaches. Performance is compared with respect to the speed-up, the code size and the scheduling time.

Although our main concern is to develop a scheduling algorithm for VLIW machines, the algorithm can be well applied to superscalar architectures. We thus present it for both architectures.

3.4.1 Bookkeeping in trace scheduling

Superscalar [35] and VLIW [25–27] machines can execute multiple instructions simultaneously, exploiting instruction level parallelism. General superscalar and VLIW architectures consist of multiple functional units. In a VLIW machine, instruction words include several operation fields, each of which controls one specific functional unit, while in a superscalar architecture, functional units are fed with several instructions forwarded by the instruction issuing logic. Due to data dependencies and resource constraints, the parallelism may not be fully exploited. Profiling-based scheduling algorithm, such as trace scheduling [28,29], and superblock scheduling [30], can effectively increase performance, if the predictions of the conditional branches are quite accurate. It is especially true for scientific applications where conditional branches are highly predictable.

Trace scheduling utilizes profiling information to select the most frequently executed trace. List scheduling then operates on the selected trace by building a data dependence graph and successively scheduling the instructions whose predecessors in the dependence graph have all been scheduled. Scheduling is beyond basic blocks because instructions can be moved up above or down below branch instructions. If the instruction defines a register whose value is never used along the branch out-of-trace path, i.e., dead, it can be scheduled before the branch instruction. To maintain the correct program semantics, redundant code has to be inserted in the unscheduled program segments. Bookkeeping procedures can be complicated to implement [28]. Superblock scheduling [30] is an alternative that employs tail duplication to eliminate all the join edges by copying the instructions from the join to the end of trace into a new superblock, and making the jump direct to the new superblock. Scheduling is then performed on the trimmed trace, thus reducing the bookkeeping complexity. The disadvantage is the code growth due to redundant duplication.

An analytical bookkeeping algorithm is presented, which is precise and without excessive duplication. Out-of-order instructions are sorted via successively local swappings sweeping over the trace. The transformed program is equivalent to the original program, and is the union of many in-order traces, each of which includes adjacent instructions either in the same schedule

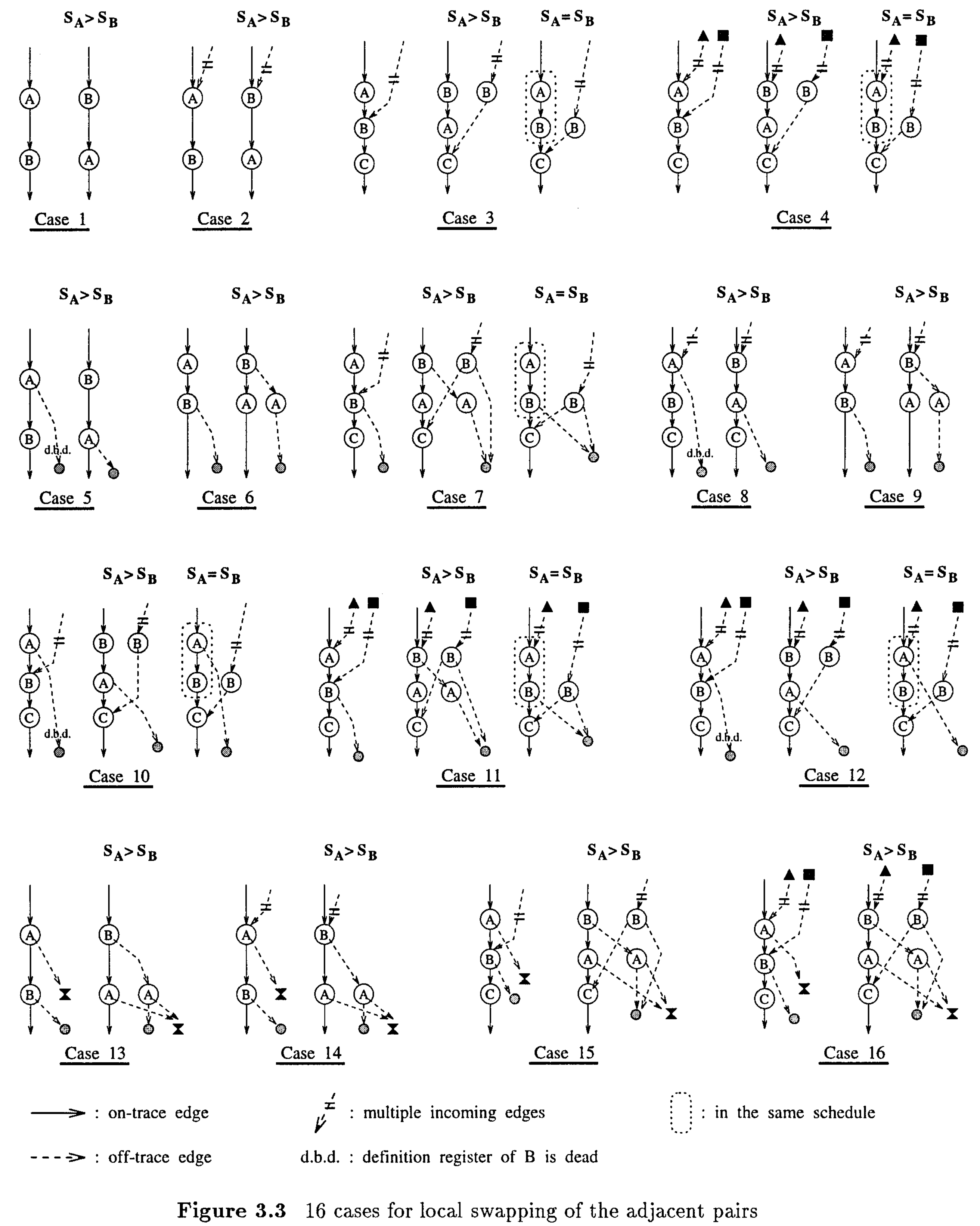


Figure 3.3 16 cases for local swapping of the adjacent pairs

or in the neighboring schedules. Branch instructions are then inserted to connect individual traces. The limitations of our approach are the increased scheduling and compilation time, and the necessity of structured programs.

3.4.2 Analytical bookkeeping

The technique aims at alleviating the coding complexity of the bookkeeping procedure. The design should be able to fit well into the improved trace scheduling, superblock scheduling, and the scheduling algorithms that support speculative executions [35]. For purposes of clarity, pipelining in functional units, and multiple branch instructions within the same clock cycle are not considered.

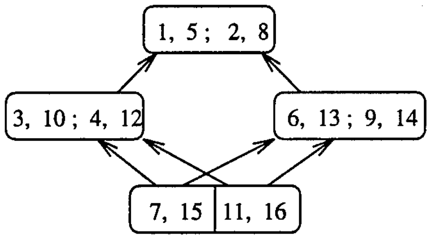
In the implementation, loops are scheduled in top-down and inner-loop-first order, because loops tend to iterate many times and the loop constructs can be maintained throughout scheduling. In order to maintain the correct program semantics, several sweeps of swapping adjacent pairs over the trace will be performed until the code is in order. The program is represented by directed graphs, with nodes and directed edges denoting instructions and control transfers respectively. Since a node can be either a join or not a join, and either a branch or not a branch, any pair of adjacent nodes can have 16 combinations, as shown in Figure 3.3. The leftmost column in each case represents the original instruction sequence in the trace. The labels $S_A > S_B$ and $S_A = S_B$ denote the conditions that A is scheduled after B, and A is in the same schedule as B, respectively. The sequence will be transformed to its equivalent form if the condition specified on top of the figure is satisfied. The sequence remains the same for conditions that are missing.

Figure 3.4 illustrates that a total of five groups are formed. The second row denotes that the join node is changed if a swapping is needed, i.e., $S_A > S_B$, and cases 2, 8, 4, 12, 9, 14, 11, and 16 are involved. The third row outlines the actions that are required for each group of four or two cases, under conditions $S_A > S_B$ and $S_A = S_B$. Group (1, 5; 2, 8) is the simplest, involving only swapping A and B, if $S_A > S_B$. Group (7, 11) and group (15, 16) can be combined, should multiple jumps be allowed in the same word. Both groups can be implemented as the additions of two other groups, (3, 10; 4, 12) and (6, 13; 9, 14).

The bookkeeping algorithm, listed in Figure 3.5, resembles dataflow analysis [22], acts similar to the bubble sort algorithm [36], and is highly suitable for parallel implementation. Loop

	1, 5 ; 2, 8	3, 10 ; 4, 12	6, 13 ; 9, 14	7, 11	15, 16
Change join A $\rightarrow$ B	$S_A > S_B$	$S_A > S_B$	$S_A > S_B$	$S_A > S_B$	$S_A > S_B$
Swap and Copy	$S_A > S_B$: Swap A, B	$S_A > S_B$: Swap A, B Copy B $S_A = S_B$: Copy B	$S_A > S_B$: Swap A, B Copy B	$S_A > S_B$: Swap A, B Copy A Copy B $S_A = S_B$: Copy B	$S_A > S_B$: Swap A, B Copy A Copy B

(a) Action table.



(b) Group implementation.

Figure 3.4 Group hierarchical implementation

```
change = 1;
while (change) {
    change = 0;
    for ($i = 0$; $i <$ length of trace $-1$; $i++$) {
        determine case number for the node pair ($i$, $i+1$);
        according to the case number, handle ($i$, $i+1$)
        . swap, duplicate, change join
        change = 1 if any of the above occurs;
        incremental loop maintenance for new nodes; } }
```

Figure 3.5 The bubble sort implementation

Table 3.1 Code growth, the analytical bookkeeping

	single issue $P = 1$	VLIW		Superscalar	
		2	4	2	4
QUEEN	148	36%	145%	3%	16%
WC	181	51%	195%	9%	27%
CMP	251	94%	269%	28%	30%
QSORT	261	23%	101%	3%	19%
PUZZLE	877	46%	166%	12%	12%
COMPRESS	1828	45%	169%	9%	14%

information is incrementally updated whenever a new node is inserted. The algorithm is guaranteed to halt, since the condition $S_A > S_B$ is non-reversible and the joins are either unchanged or shifting towards the end of the trace. In the worst case, $n-1$ sweeps over the trace are needed, where n is the number of instructions in the trace. Although a merge-sort-like algorithm can be designed to reduce the number of sweeps to $\log n$, we still favor the current implementation due to its clarity, and since the number of instructions within a trace is usually small. By changing the actions taken for all cases, e.g., pushing join down to node C, and making extra copies of A and/or B, we can realize superblock scheduling naturally, without explicit tail duplication.

3.4.3 Speedup, code run-time, and code growth

The simulation is conducted on a DEC station 3100, a MIPS processor with a reduced instruction set. Assuming that each instruction takes unit time to execute. The speedup is measured as the ratio between the number of instructions executed for single issue machines and the number of code words executed for VLIW machines (or the number of instruction cycles for superscalar architectures). The code size for a superscalar architecture is the total number of MIPS instructions in the schedule, while the code size for a VLIW machine is the number of code words times the number of functional units.

The original optimized code generated by the IMPACT C compiler [21] for single issue machines serves as the base, which is first profiled and scheduled under various numbers of functional units (P) 2 and 4. The scheduling and compilation time, the number of instruction words (or cycles) executed, and the code size are then collected. We obtain the performance data for the analytical approach on the following benchmarks : QUEEN, QSORT, WC, CMP, COMPRESS, and PUZZLE. Both QUEEN and QSORT include recursive subroutines. CMP,

Table 3.2 Speedup and scheduling time overhead

	Speedup		Scheduling time overhead	
P	2	4	2	4
QUEEN	1.51	1.66	10%	30%
WC	1.30	1.30	30%	30%
CMP	1.24	1.24	42%	42%
QSORT	1.80	2.62	20%	20%
PUZZLE	1.35	1.40	39%	44%
COMPRESS	1.49	1.53	53%	57%

COMPRESS and WC are UNIX utilities which compare two files, compress files, and count the number of words in a file, respectively. PUZZLE is a game program which includes several consecutive single loops.

Table 3.1 and Table 3.2 tabulate the code growth, the speedup and the additional scheduling overhead for the analytical approach. As indicated in Table 3.1, VLIW machines have a larger code growth, mainly because no-op instructions due to unscheduled slots in the code word also take up space. Superscalar architectures have code growth all within 30% for all benchmarks under $P = 2$ and 4. $P = 1$ denotes issuing single instruction per cycle, and its corresponding column represents the number of MIPS instructions in each benchmark program. Because we do not consider functional unit pipelining within different schedules, and no multiple branch instructions can be in the same schedule, the speedup is flat for most benchmarks. The speedup can be even higher if we employ the more advanced software pipelining [31] and loop unrolling [37] for the inner loops. A postpass percolation scheduling can be applied to fine-tune the performance, since a loop invariant instruction can not be moved out of the loop under our current hierarchical loop scheduling order. QSORT includes a recursive merge sort algorithm and has the best speedup for all P's. Having a larger code growth does not imply a larger speedup, as witnessed by CMP, a UNIX file comparison utility. The scheduling time overhead is obtained by comparing the additional scheduling time plus the compilation time for the scheduled code to the base compiler time for the optimized code. As illustrated in Table 3.2, the scheduling time overhead increases as the code size increases. The scheduling and compilation time is less than double the original compilation time for the benchmarks.

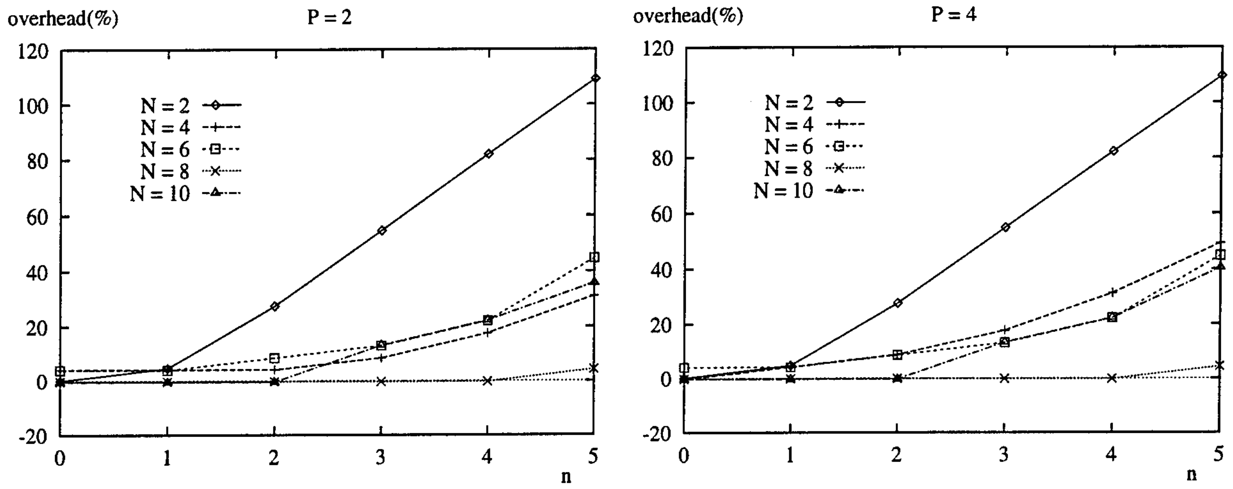


Figure 3.6 Performance overhead for CMP

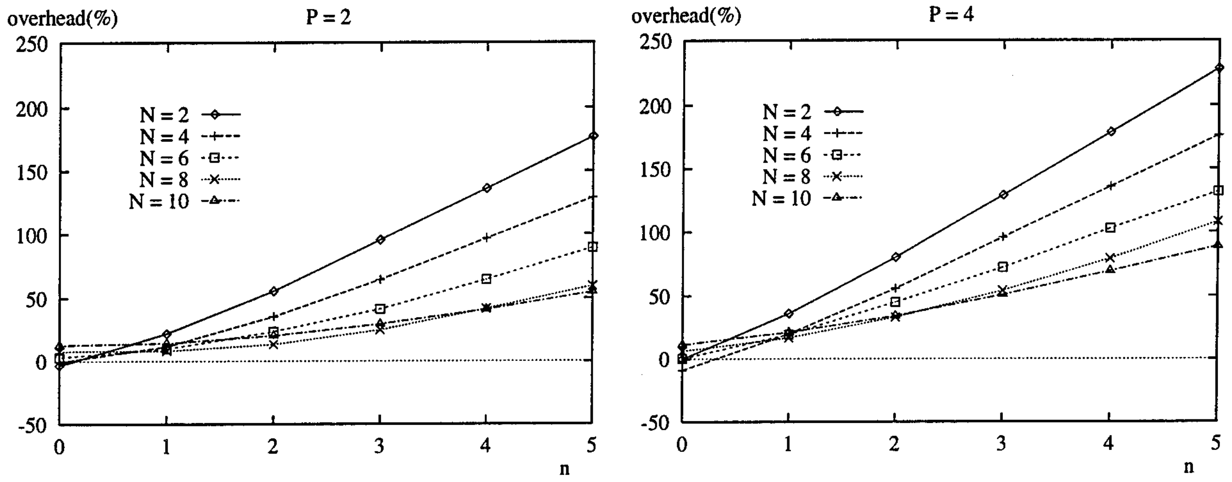


Figure 3.7 Performance overhead for COMPRESS

3.5 Simulation Results

The run-time performance and the code growth are measured for both compiler-based approach and compiler/read buffer combined approach, using the proposed scheduling algorithm. For simplicity, we assume each instruction word takes a unit time to complete. Performance is measured by counting the number of instruction words executed. This is done by allocating a counter and inserting increment instructions in every instruction word. As the compacted code is executed on a scalar processor DEC station 3100 (MIPS processor), the counter contains the number of instruction words executed at the end of execution. We investigate the relative performance impact for various rollback capability for scalar processors (N) and the desired rollback distance for VLIW architectures (n) with a varying number of functional units P.

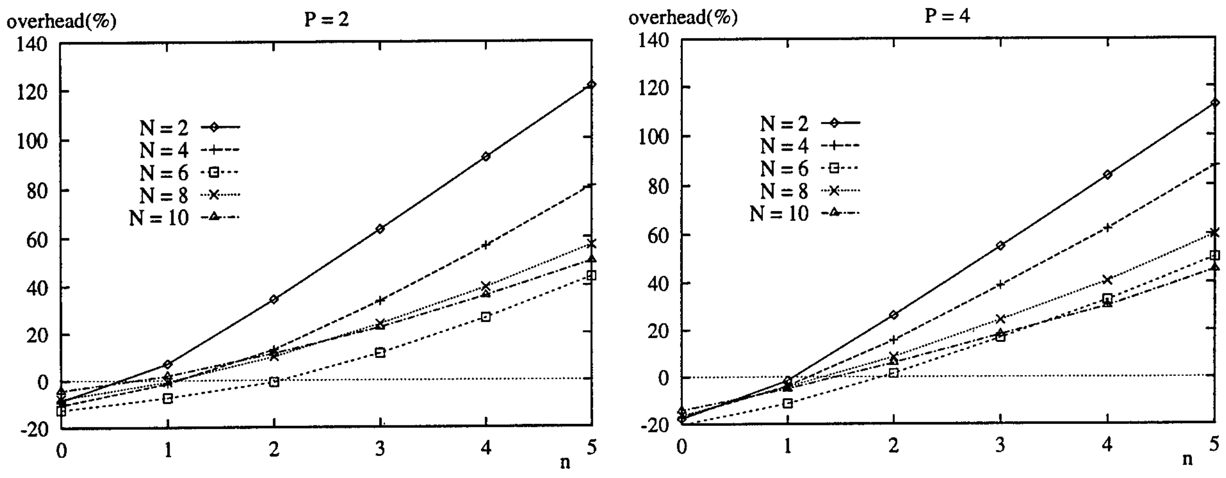


Figure 3.8 Performance overhead for PUZZLE

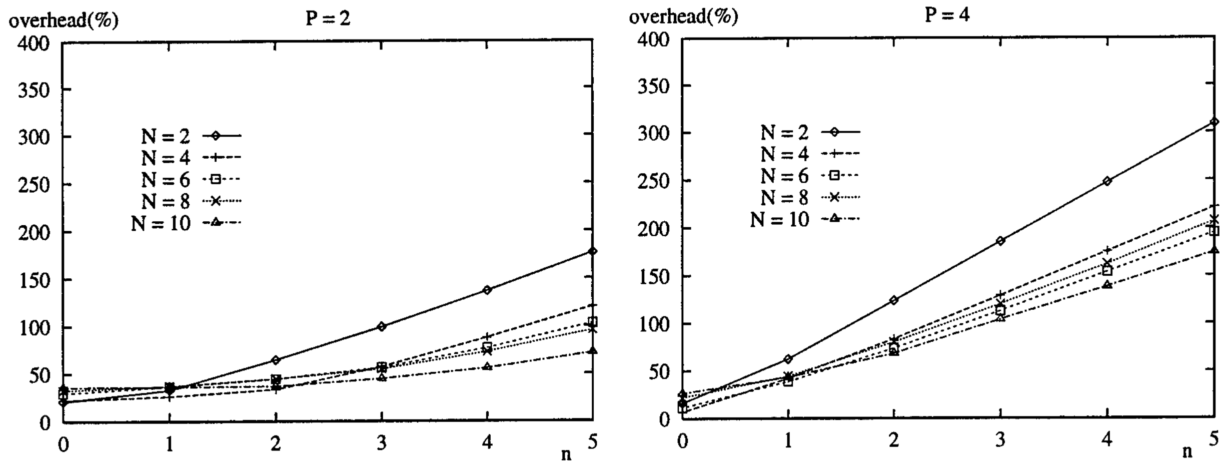


Figure 3.9 Performance overhead for QSORT

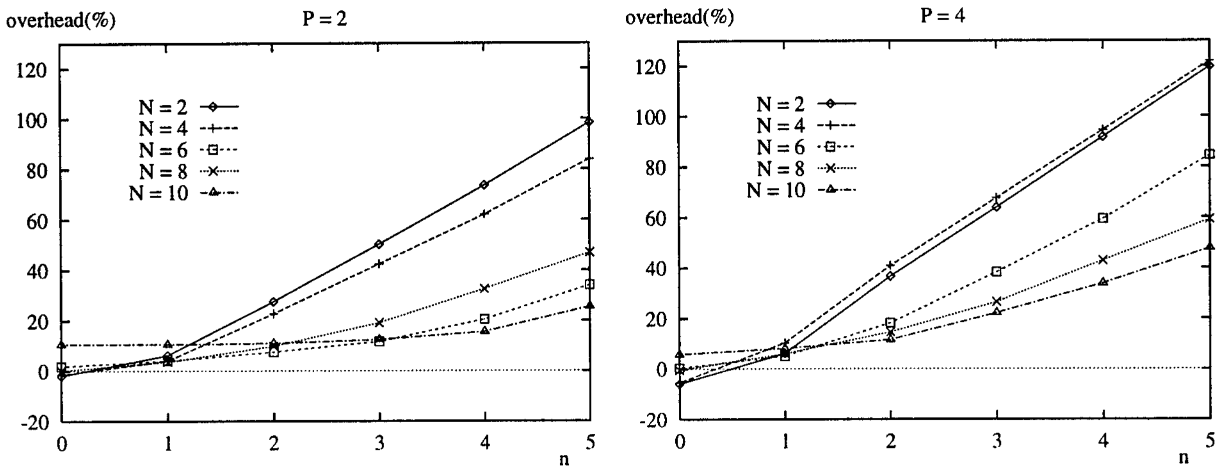


Figure 3.10 Performance overhead for QUEEN

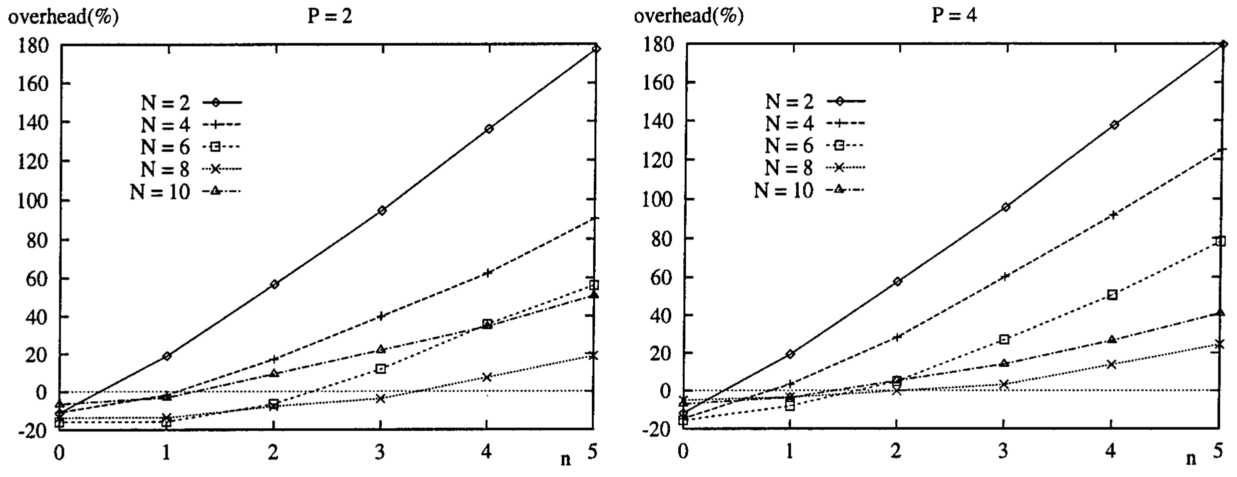


Figure 3.11 Performance overhead for WC

The optimized code generated by the original version of the IMPACT C compiler serves as the base, which is first profiled and scheduled under different P. Simulations are then performed to collect the number of instruction words executed. Figures 3.6–3.11 illustrate the performance overhead on the benchmarks for $P = 2, 4$, $N = 2, 4, 6, 8, 10$, and $n = 0, 1, 2, 3, 4, 5$. $n = 0$ means that only hazard removal is performed without inserting any nops. This case is used to demonstrate the performance improvement attainable as a result of loop expansion.

As the program results indicate, in most cases, for fixed P and n, the larger N tends to generate compacted code with better performance. This is because a larger N may require expanding loops more times to resolve data hazards within loops, consequently making register live ranges shorter, which is beneficial to the scheduling algorithm. For example, under $P = 4$ and $n = 5$, all benchmarks have minimal overheads when $N = 8$ or 10.

Without inserting any nops, i.e., $n = 0$, all benchmarks except QSORT have improved code schedulings for $N = 2$ and 4. However, for both cases all benchmarks have the worst performance under the same P and $n = 5$. That is because when fewer loops are expanded, fewer registers are used. Also, the scheduled code is so compact that more nops are needed to resolve hazards between code words. For smaller n's, e.g., $n = 1$ and 2, PUZZLE and WC have better compacted schedulings over those of the original optimized programs. Both programs have very simple loop structures, and the loops are executed frequently. The instruction retry scheme functions well in scientific applications, where branch predictions are highly accurate, and the loops are iterated many times. QSORT has the worst performance overhead because

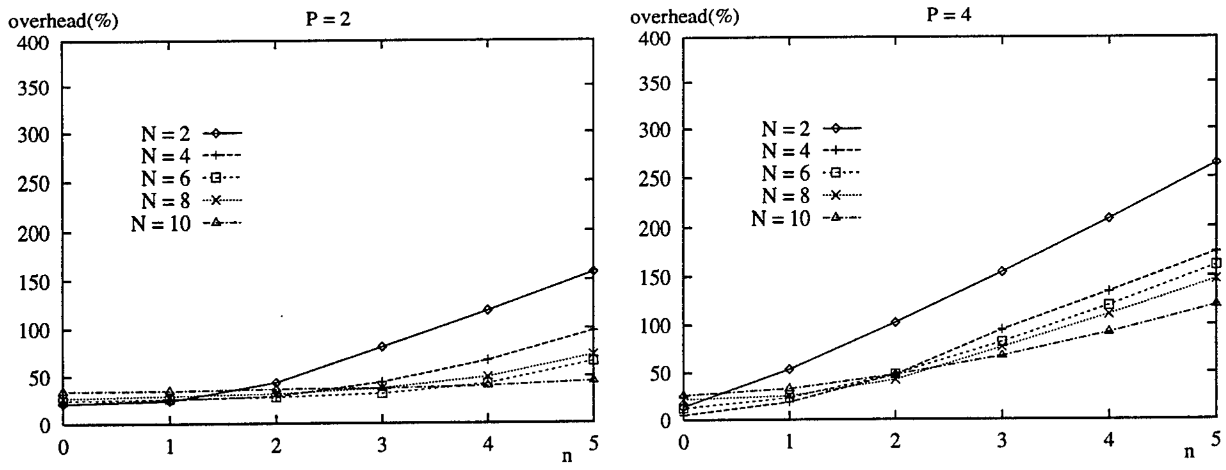


Figure 3.12 Performance overhead for QSORT under the prioritized list scheduling algorithm

it has a frequently called recursive subroutine, *merge_sort*. The prioritized list scheduling algorithm helps to improve its performance, as shown in Figure 3.12.

In the original programs, the performance of the compacted code for a larger P is no worse than the performance of the code for a smaller P. However, the situation is usually reversed for instruction retry schemes. For fixed N and n, since larger P tends to generate more compacted code, the distances between hazard words are closer, resulting in more nops.

Figures 3.13–3.18 show the code growth ratio for the six benchmarks, which may have an impact on the instruction cache miss ratio. The ratio is relative to the code growth of the original compacted scheme, for $P = 2$ and 4, without inserting nops. The results indicate that for $n = 5$, most benchmarks have a minimum code growth ratio when $N = 6$, and a maximum code growth ratio when $N = 2$. The code growth ratios are within 450%, and 700% for $P = 2$ and $P = 4$ respectively. If only loop expansion and node splitting are performed without nop insertion, as in the case $n = 0$, smaller N tends to have a smaller code growth ratio.

Figure 3.19 illustrates the performance overhead when employing a read buffer. For $P = 2$ and $P = 4$, both figures almost have the same shape, e.g., CMP and WC. Their identical speedups for $P = 2$ and 4 can explain this situation. Also some benchmarks have very flat overheads. This case is mainly due to the stepwise branch hazard distance distribution. PUZZLE has a near 0 performance impact, since all branch hazards can be resolved by the read buffer. CMP has the highest overhead, 4.52% for $P = 2$, $n = 5$, and 4.53% for $P = 4$, $n = 5$. Most of the branch hazards in CMP have distance 2, resulting in the sudden rise from $n = 1$ to 2, and flat for $n > 2$. In average, when $n = 5$, the performance overhead for $P = 2$ is 2.7%

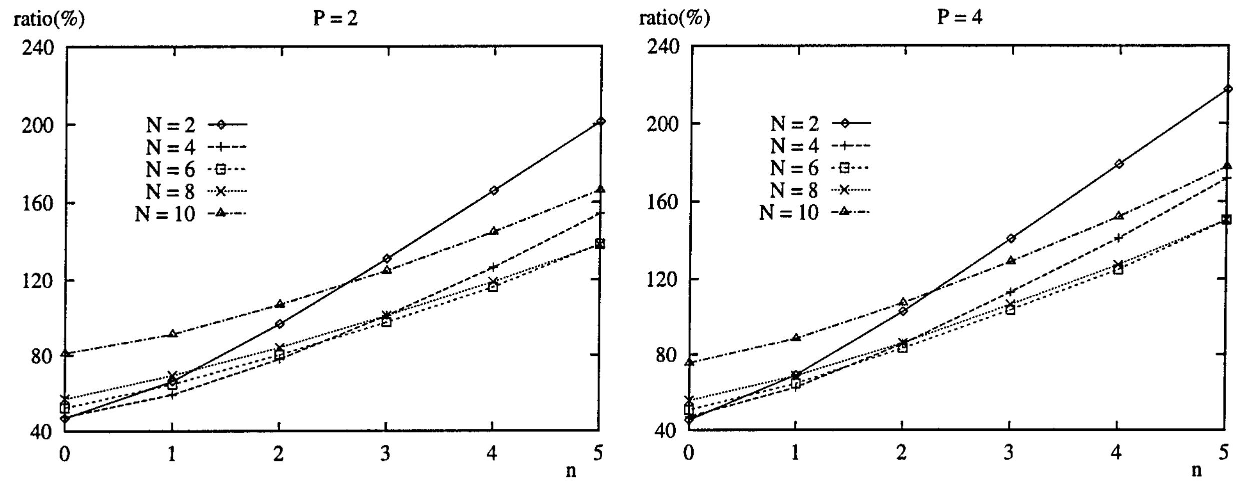


Figure 3.13 Code growth ratio for CMP

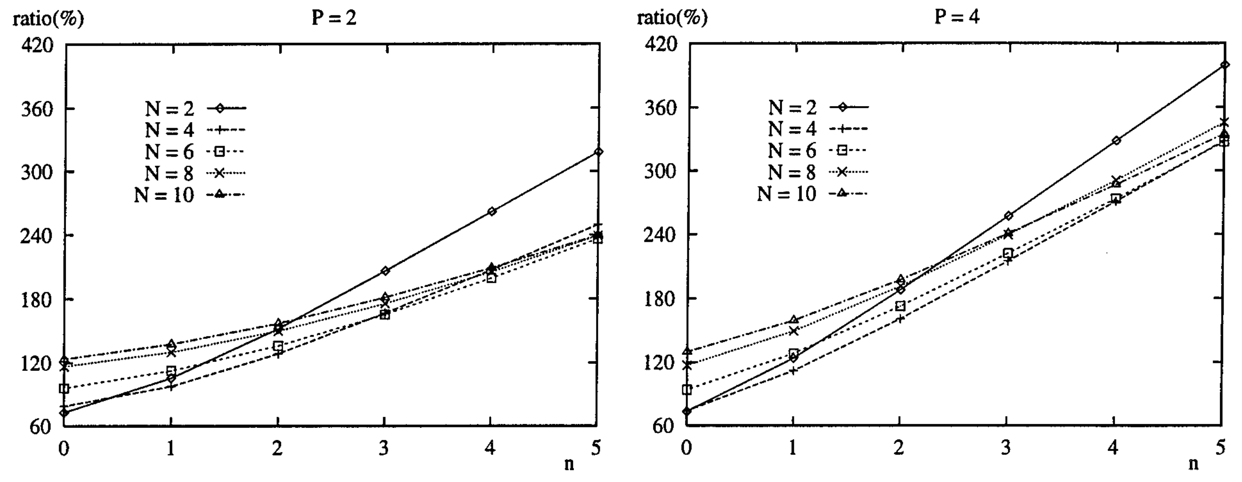


Figure 3.14 Code growth ratio for COMPRESS

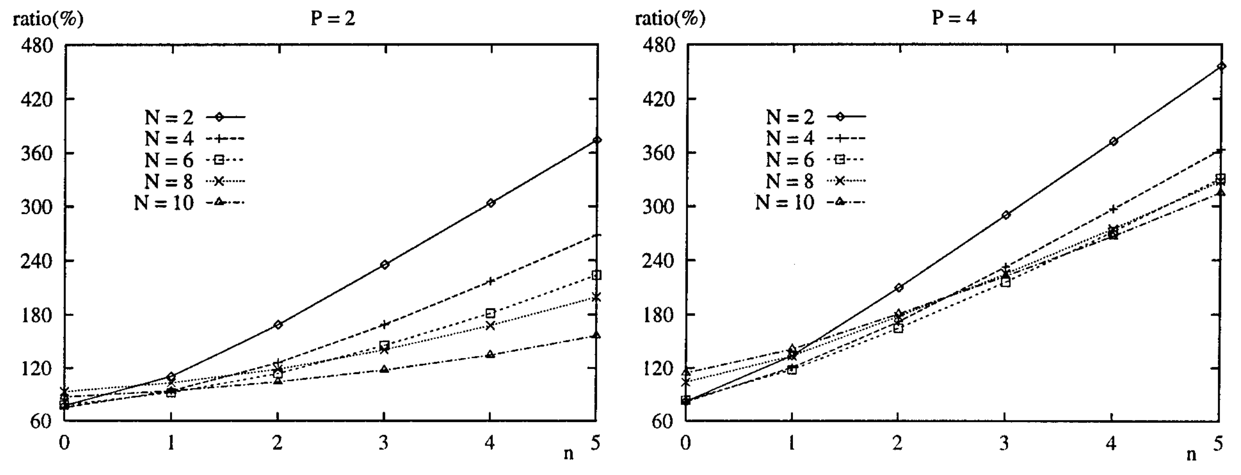


Figure 3.15 Code growth ratio for PUZZLE

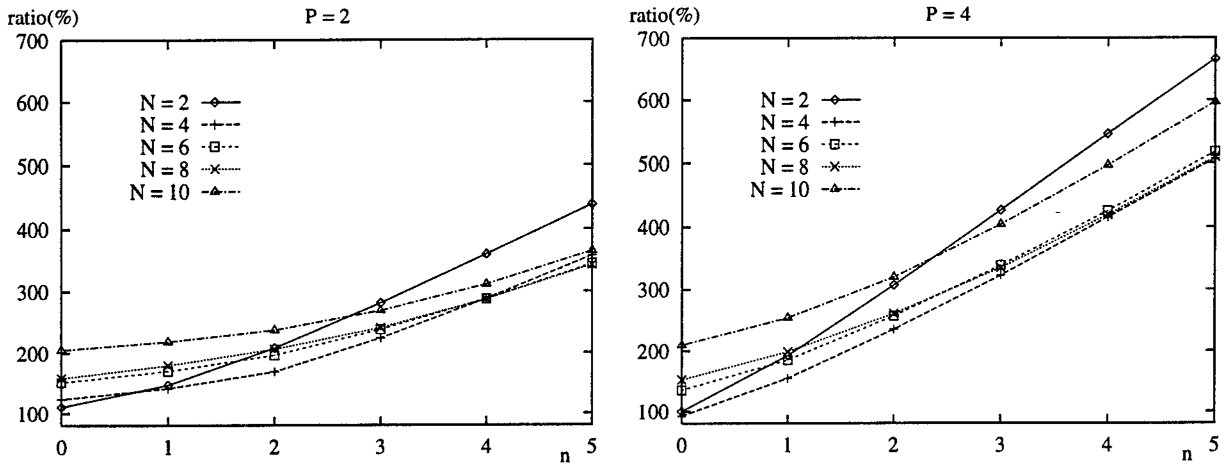


Figure 3.16 Code growth ratio for QSORT

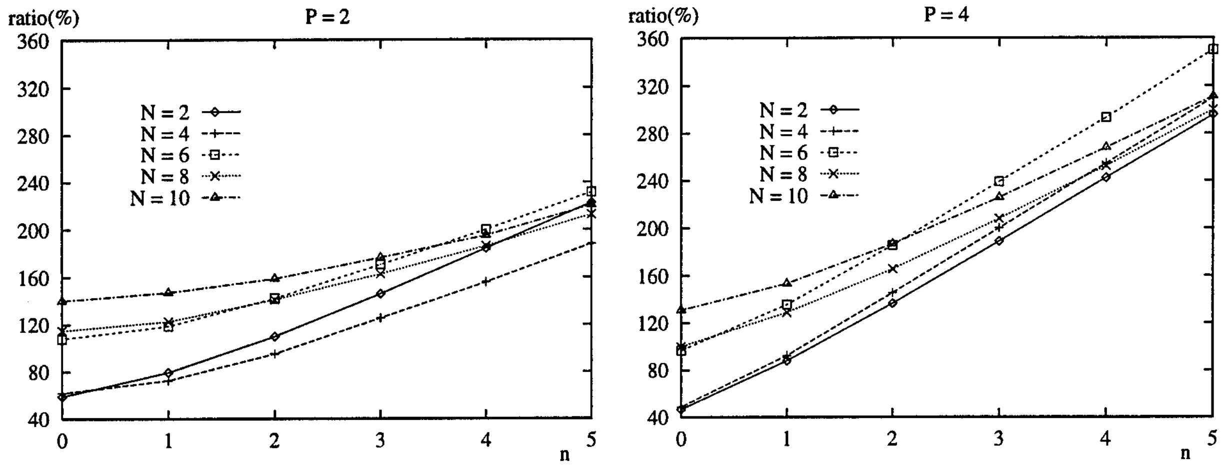


Figure 3.17 Code growth ratio for QUEEN

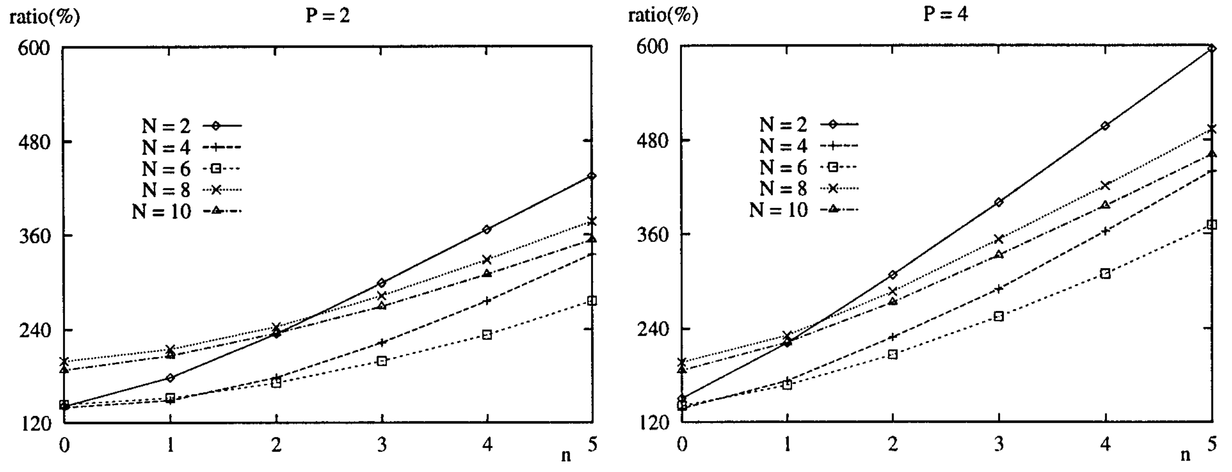


Figure 3.18 Code growth ratio for WC

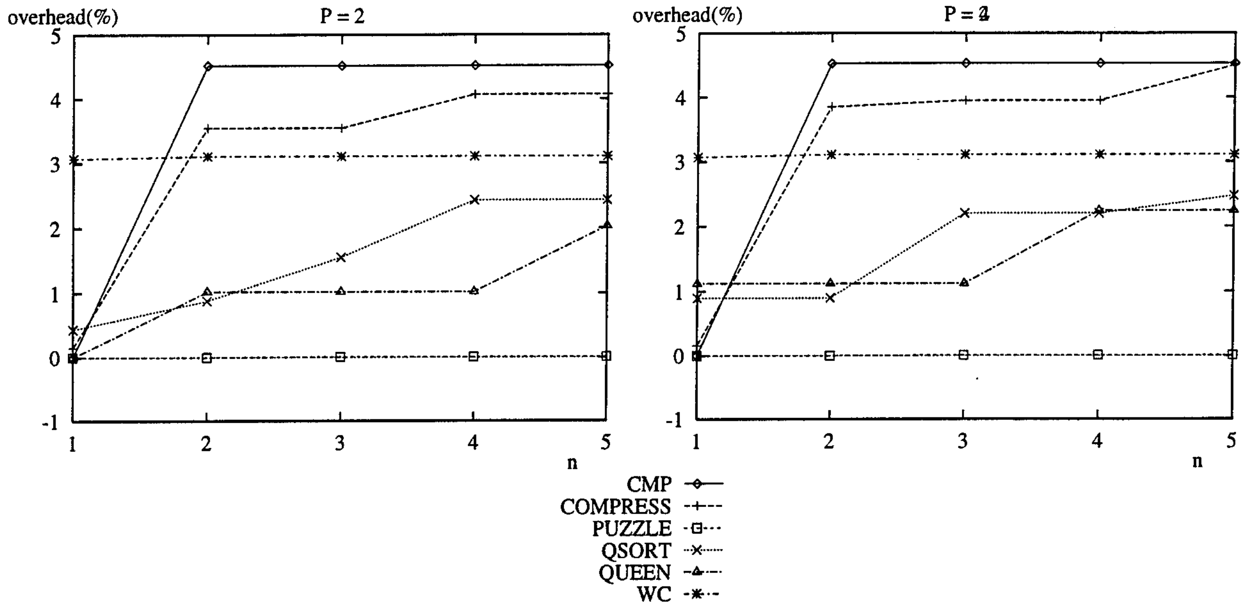


Figure 3.19 Performance overhead with a read buffer

and for $P = 4$ is 2.81%. Figure 3.20 illustrates the code growth ratio of the read buffer scheme. When $n = 5$, the average code growth ratio is 9.37% for $P = 2$ and 10.97% for $P = 4$.

3.6 Summary

Two approaches to multiple instruction word retry for VLIW architectures were described and evaluated. The software-based approach employs compiler technology to resolve all hazards. The performance costs range from 20% to 70% for two functional units, and from 5% to 170% for four functional units. The hardware read buffer with compiler-assisted approach retains reads within the last n instruction words. The mechanism resolves the frequently occurring on-path hazards, while the compiler resolves the branch hazards. The hardware cost is a read buffer of $2n \times P$ entries for register backup. Over the six benchmarks, the experimental results show less than 5% performance degradation for a rollback distance of $n = 5$ and the number of functional units $P = 2$ and 4.

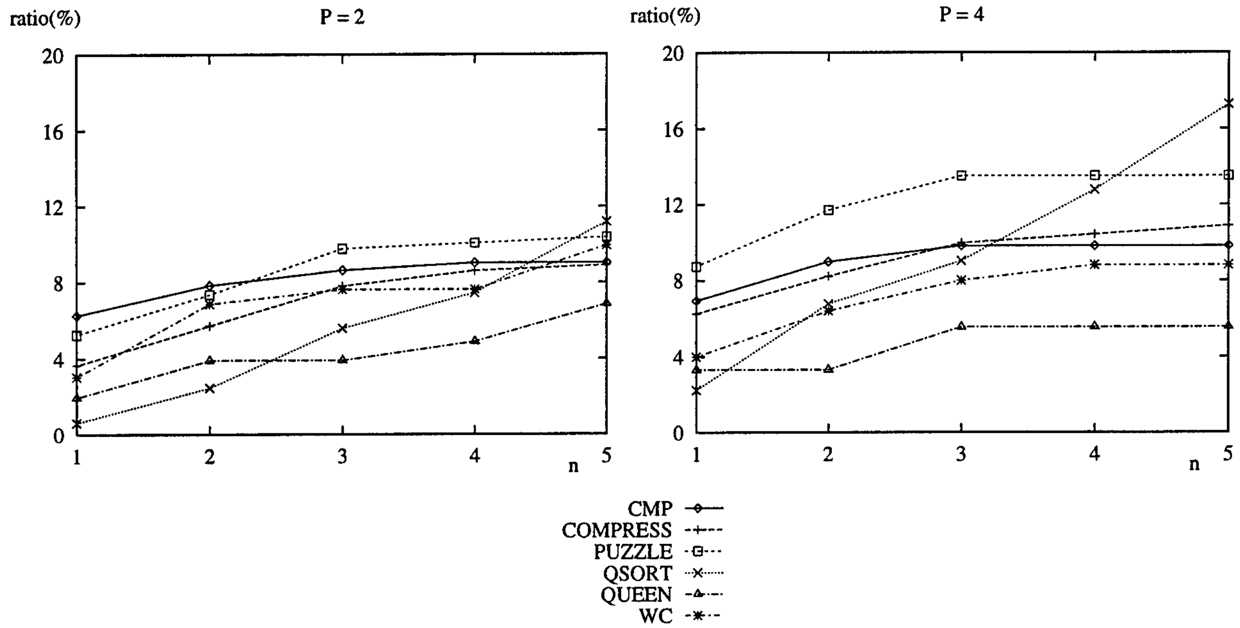


Figure 3.20 Code growth ratio with a read buffer

Chapter 4

Compiler-Supported Reversible Debugging

4.1 Introduction

Symbolic debuggers are useful tools during the program development stage. Debugging an optimized code is a classic problem that has received much attention [38–40]. Optimization may cause breakpoint mismatch between the source code and the executable code. The actual breakpoint location may not be the same as the user expected. Also some variable values may be misleading after optimization because they do not equal the values observed by a close checking of the source code, i.e., *non-current* [38,41].

Traditional debuggers usually turn off the program optimization mode so that every variable is *current*, and the user can clearly identify instruction boundaries due to breakpoint setting [42–44]. Their primitive features include breakpoint setting, single stepping, data displaying, value updating, and control flow altering etc.. Backward execution is one feature that is helpful but is not fully exploited. Most debuggers have the ability to log run-time information in the history tapes [42], and already possess high potential to undo several instructions. The history tape approach suffers from a long execution time because of the additional I/O operations.

The applications of *undo* include programming languages, editing and formatting, and the development of programming environments [45]. Numerous schemes modifies the programming languages to include backward execution commands, such as *undo*, *redo*, *skip* and *retrace* [46–48]. The program development system COPE [49] implemented reversible execution by allowing the user to explicitly generate files for procedures, input data, and results. Each file includes a sequence of fixed-size blocks, and the modifications to a block result in a complete new copy of the block with the changes. Storage requirement for maintaining the list of updated blocks is a main concern in this approach. IGOR project [50] implemented a debugger that can perform backward execution, using a checkpointing scheme. Checkpointing periodically saves the system state during program execution. When stopped at certain breakpoint, the debugger can backtrack a few statements or instructions by starting forward emulation (using an interpreter) from the last checkpoint until reaching the desired statement or instruction. The checkpointing scheme relies heavily on locating the nearest checkpoint. Also the forward emulation from the nearest checkpoint towards the desired instruction slows down the execution for more than 150 times.

This chapter describes the implementation requirements for backward execution. To reduce the potential large execution overhead associated with the backward execution, the user can select critical regions to inspect, and the function can be easily enabled and disabled [51]. A twin-buffer is proposed to store the old data to increase the expected number of instructions that can be undone. Two approaches to incorporating the undo capability directly into a debugger are presented. The first approach modifies the debugger without changing the compiler. Storage overhead is zero for the user executable code, but with a slow program debugging time due to trapping every instruction and context switching. The second approach employs the compiler to insert buffering instructions to the executable code, so that useful data can be stored to buffers at the run-time. The normal execution time in recording mode is significantly improved, with a moderate code growth. The implementation is based on the GNU debugger, *GDB* [52].

4.2 GNU Debugger – GDB

Before incorporating the backward execution into the *GDB* , we describe its basic methodology upon which our implementation will be based in this section. GDB is a product of Free Software Foundation, suitable for machines running the UNIX system [53]. The main routine of GDB can accept user commands from the terminal. The frequently used commands include breakpoint setting, source code listing, data displaying, value updating, single stepping, program restarting, and execution continuation etc.. Additional features like command history, data history and remote debugging, can be invoked if requested.

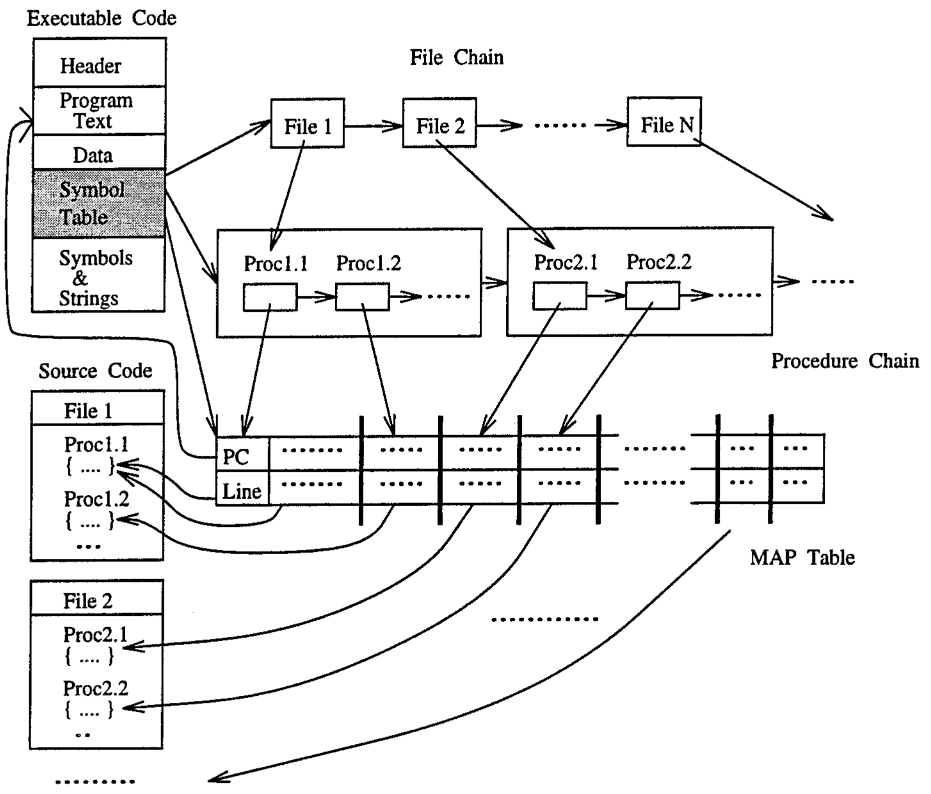


Figure 4.1 Basic data structures for GDB

The GNU C compiler (*GCC*) [54] attaches symbol tables and the mappings between source code statements and object code addresses to the generated executable code. The line increments in both the source code and the executable code are maintained in the mappings. When the object file being debugged is first loaded, only partial information is read in. It will be expanded to a whole symbol table only when it is necessary. Figure 4.1 illustrates the basic data structures and mapping connections required for the GDB to function correctly. The executable code may contain procedures from different files, and system library routines. The file chain records source file names, pointers to internal and external symbols, and pointers to corresponding procedures. The procedure chain includes procedure names, pointers to symbols, and pointers to corresponding PC ranges and source file line ranges. The implementation is actually a modified GDB, based on the structures and connections in Figure 4.1. We refer to the modified GDB as GDB thereafter. For the clarity of presentation, we omit the links that are used mainly for bookkeeping.

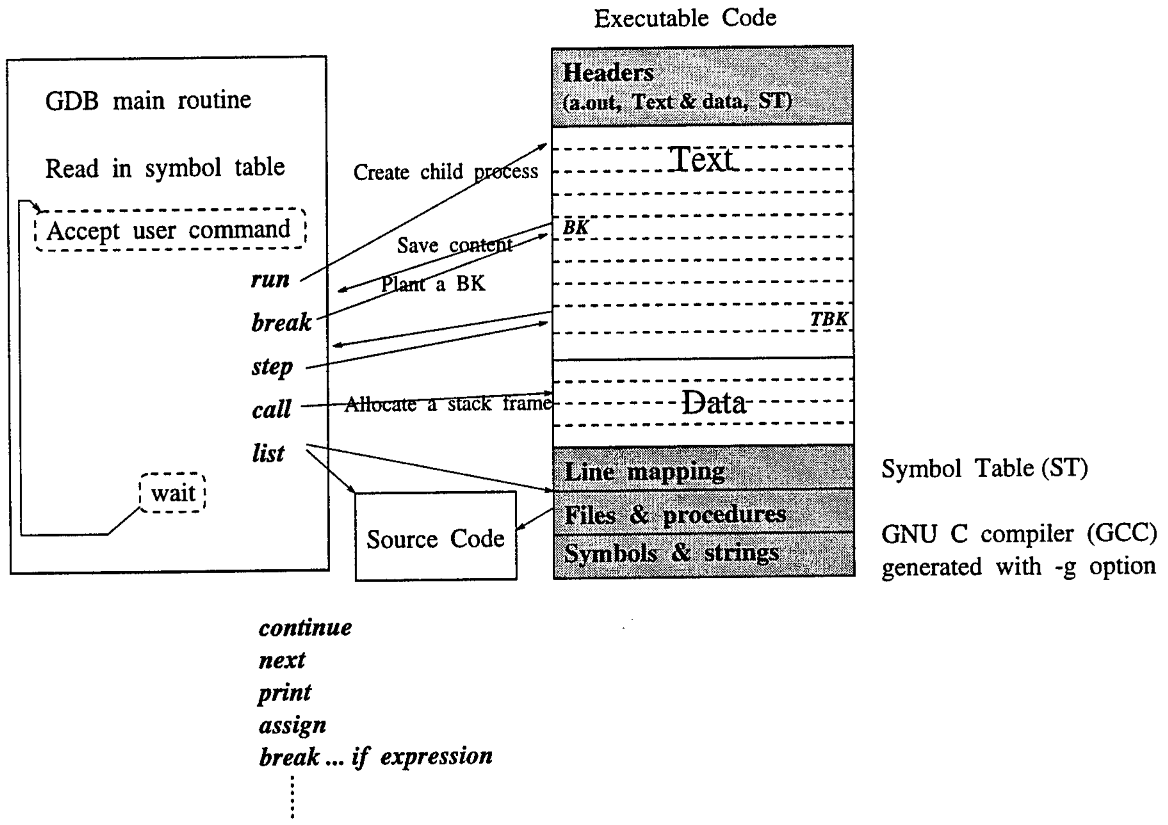


Figure 4.2 The GDB frame work

Figure 4.2 outlines the frame work of GDB. The breakpoint setting command involves checking the MAP table, converting the line number specified by the user to a PC within range, and attaching the information to a breakpoint list. The step command plants temporary breakpoints in order to perform single stepping. In response to the *run* command given by the user, GDB retrieves and saves the instructions in the program text specified by the breakpoint list, plants break instructions accordingly, and then creates a child process executing the object file. The parent process then waits for its child's halting. The causes for halting may be normal exit, error abort, stopping at a user-preset breakpoint, or stopping due to single stepping. The parent process can distinguish them by the type of signals received, and take corresponding actions, e.g., print proper messages, ignore the signal and continue the child process, or accept further user commands from the terminal.

The special UNIX trapping utility, *ptrace*, is the main tool for communication between the parent and child processes. The parent process can peek the data (registers, program counter, and memory contents) from the child process's address space, and monitor the child process's control flow (restarting, resuming, single stepping). The command *break ... if expression*, denoting 'stop if the expression is true', is implemented by trapping every statement, and checking the condition specified in *expression*. Although the execution is slowed down dramatically, the users can choose to utilize such a command at their own discretion. This is sometimes useful to catch deeply hiding bugs.

4.3 Recording Mode and Undo Command

Software implemented data recording induces performance degradation during normal execution. To run the entire program in a data recording mode would make the cost of debugging process extremely high. Therefore, we introduces several new commands, *record-on*, *record-off*, and *undo*, for GDB. The user can turn on the recording mode at certain critical regions, and turn off the mode for a faster program execution. Such critical regions may start at the beginning of the program, the beginning of a subroutine, or some user-defined breakpoints. Undo command is valid only when the debugger is in the recording mode. These additional commands satisfy the backward execution requirements, including selective use and easy enablement/disablement [51].

4.4 Buffering Schemes

This section describes the two buffering schemes, *read buffer* and *history write buffer*, which can be employed to record useful values. We also discuss the buffer size limitation and propose a twin-buffer to increase the expected number of valid buffer entries.

4.4.1 Read buffers and history write buffers

Read buffer scheme stores to the buffer every value read, while *history write buffer* scheme stores the old value before it is overwritten. Table 4.1 summarizes the values that are saved in various instructions for both buffering schemes, assuming that the current instruction has label $L1$. For example, the memory store instruction, 'sw \$31, 20(\$sp)', stores the value in register \$31 to a memory location addressed by $20 + value\ in\ \$sp$. We need to save the contents of registers \$31 and $\$sp$ for the read buffer scheme, and save the old value in the memory

Table 4.1 The stored values vs. instruction types

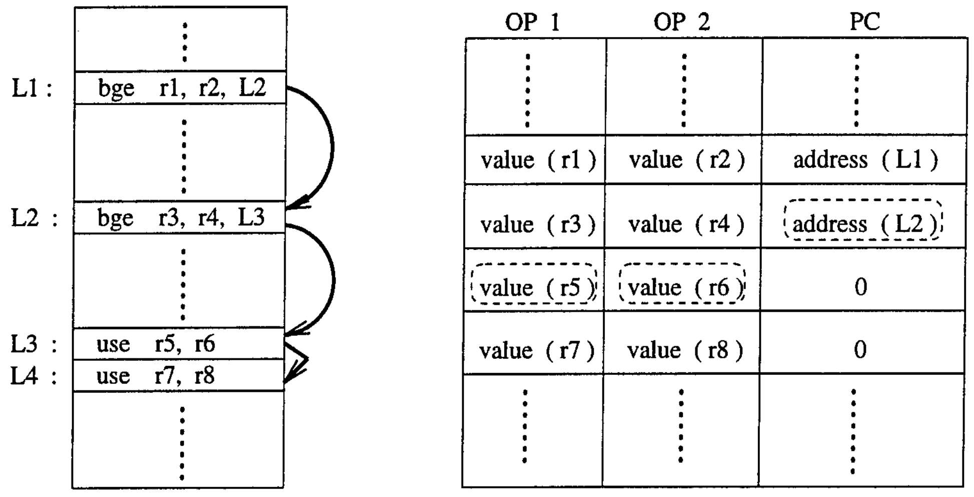
Instruction Type	Example	save to read buffer			save to write buffer		
		register	memory	PC	register	memory	PC
Load	*lw $25, 36($sp)*	*$sp*	*36($sp)*		$25		
Store	*sw $31, 20($sp)*	*$31* *$sp*				*20($sp)*	
Move	*move $10, $11*	*$11*			*$10*		
2-Operand (or 1-) Operations	*addu $8, $10, $11*	*$10* *$11*			*$8*		
Branch	*b Label_i*			*L*1			*L*1
Conditional Branch	*bge $10, $11, Label_i*	*$10* $11		*L*1			*L*1
Subroutine Call	*jal test*			*L*1	$31		*L*1
Return	*j $31*	*$31*		*L*1			*L*1

location with address $20 + value\ in$ $sp for the history write buffer scheme. Note that both schemes do not store the register numbers and memory addresses to the buffer, as the hardware implemented buffers usually do [2,17]. During the backward execution, the values in the buffer can be restored to correct destinations by decoding the current instruction under investigation. We can have faster normal execution while shifting the burden to the recovery process.

Branch instructions, subroutine calls and returns can alter the control flow. Therefore, the current program counter (PC) needs to be saved for potential later recovery. For sequential instructions, however, their PC values are not saved, since their addresses can be easily recovered. Figure 4.3 depicts the recovery procedure using the read buffer scheme. The PC column keeps track of the flow control, so that the previous instruction can be recovered. A "0" denotes that the immediate next instruction will be executed. For example, consider a recovery procedure stopping at instruction $L3$. After the procedure restores the old values of $r5$ and $r6$, the PC value in the previous read buffer entry can be used to recover the previous instruction $L2$. Similarly, $L1$ is the next instruction to be recovered.

4.4.2 Buffer size

Table 4.1 indicates that the read buffer scheme needs three buffers, two for operands and one for PC, and the write buffer scheme needs two. However, by a proper interleaving, we only need one buffer. For example, we can maintain a variable *buf_index*, serving as the buffer index. Assuming that each buffer entry takes four bytes. For the read buffer scheme, the first operand



(a) The flow control. (b) The read buffer.

Figure 4.3 Recovery using a read buffer

(a register value) is saved to location *buf_index + 8*, the second operand (either a register value or a memory content) is saved to location *buf_index + 4*, and the PC is stored to location *buf_index*, while for the history write buffer scheme, the old operand (either a register value or a memory content) is saved to location *buf_index + 4*, and the PC is stored to location *buf_index*. The index decrements by 12 and 8 for both schemes respectively, until it reaches the buffer boundary pointed by *buf_limit*, in which case the index will be reset to the beginning of the buffer to form a circular queue.

When the buffer is full, the buffer contents may need to be stored to secondary storages. Depending on the applications, the flush buffer process can take a long time to complete, since it may involve external I/O operations. Also the storage that is required to save old values can be extremely large. If the user can carefully choose critical regions to examine, the buffer can serve as a sliding window, such that the undo can only function within the window. There is no need to flush the buffer in this case, which greatly reduces the program execution time with a recording mode.

4.4.3 Twin-buffers

Although the buffer is implemented as a circular queue, not all buffer entries are useful all the time. For example, Figure 4.4 illustrates that when the normal execution stops with a

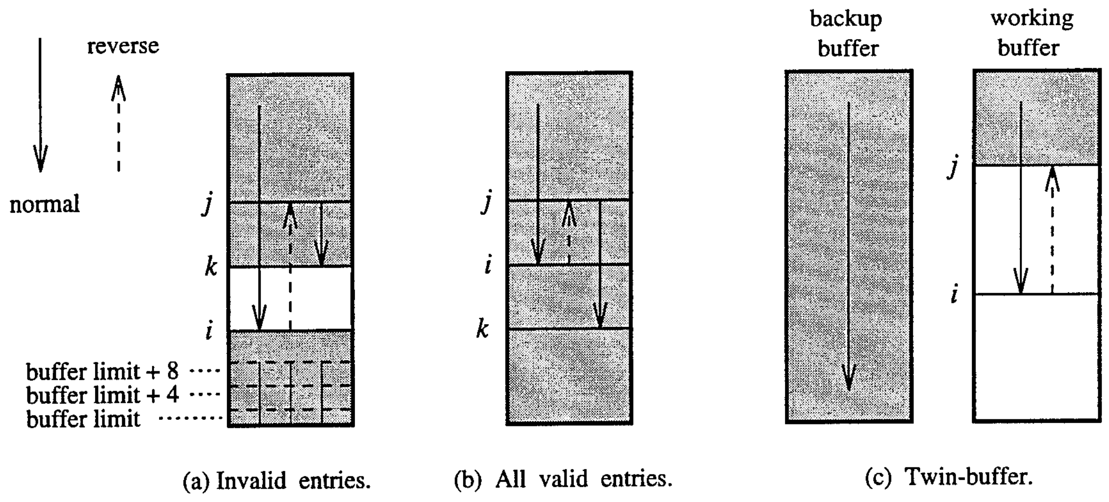
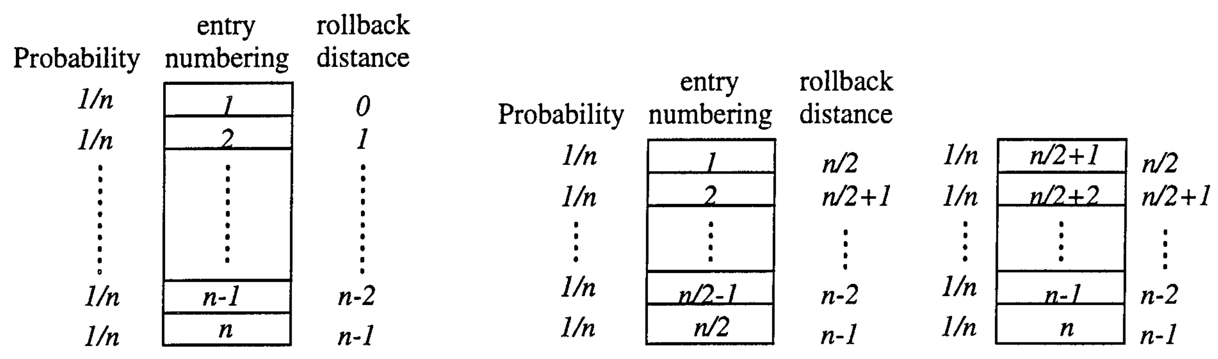


Figure 4.4 The twin-buffer approach

buffer entry i, the user undoes several instructions, thus moving buffer index back to entry j. Solid boxes denote valid entries. The normal execution continues until it halts with a buffer entry k. The entries between k and i may not be valid historical information, as shown in Figure 4.4(a). By marking i as the smallest index that the buffer has used after the previous buffer flushing, we can determine if the buffer entries are valid. However, it requires extra attention during normal execution, to check if the current buffer entry k passes through entry i, as shown in Figure 4.4(b). Such a checking would further increase the program execution time. Therefore, to obtain a fast debugging time, the undo wraparound capability should be disabled, reducing the effective buffer usage. To increase the expected number of valid buffer entries without slowing down the program execution, we employ a twin-buffer where one buffer serves as the current working buffer and the other as the backup, as shown in Figure 4.4(c). When the current buffer is full, the backup buffer contents are void (or flushed). The roles of both buffers are then interchanged. The buffer index is reset to the beginning of the new working buffer. Also, when a sequence of undo commands exhausts the working buffer, the backup buffer becomes the working buffer and there is no backup buffer in this case.

4.4.4 Queueing models

Queueing models can be established to measure the effective buffer usages for different implementations, i.e., wraparound buffer, single buffer without undo wraparound, and twin-buffer without full undo wraparound. As discussed in the previous section, wraparound buffer



The buffer usage = (n-1) / 2

The buffer usage = (3/4)n - 1/2

(a) single buffer without undo wraparound.

(b) twin-buffer without full undo wraparound.

Figure 4.5 The buffer usages for two without undo wraparound implementations

implementation needs a special marker to represent the smallest buffer index that has been used, thus slowing down the normal program execution. The other two implementations improve the debugging time by eliminating the special marker maintenance, while disable the full undo wraparound capability. Using two different models, the wraparound buffer implementation is shown to have the best buffer usage. Also by disabling full undo wraparound, the twin-buffer implementation has a better buffer usage than the single buffer implementation.

Let n be an even number and denote the number of buffer entries. Without loss of generality, we assume that there is no system routine call to interrupt the filling of the buffers. The buffer usage is defined as the expected number of valid entries that can be used to undo instructions. The first model deals with the steady state, assuming that it is equally likely to hit any buffer entry, i.e., with a probability $1/n$. The buffer usage for a wraparound buffer is n. The buffer usage equals $\sum_{i=0}^{n}(i \cdot \frac{1}{n}) = \frac{n-1}{2}$ for the single buffer without undo wraparound implementation, and $2\sum_{i=0}^{n/2-1}[(\frac{n}{2}+i) \cdot \frac{1}{n}] = \frac{3}{4}n - \frac{1}{2}$ for the twin-buffer without full undo wraparound implementation, as shown in Figures 4.5(a) and (b). Without considering undo wraparound, the twin-buffer scheme performs better than the single buffer scheme in terms of buffer usage.

The second model employs the discrete-time birth-death process [55], where state i represents that the number of instructions undo-able is i, and P_i is its stationary probability, for $i = 0, 1, 2, \ldots, n$. The transitions between states denote the possibilities of forward and undo commands, with values p and $1-p$ respectively. Because the computation can only execute one instruction at a time, either forward or backward, the discrete-time model is chosen throughout the examination. Assuming that both forward and undo commands can execute only one instruction, and there are no system library calls interrupting the single birth (advance) and single death (undo) sequence. Since a forward command occurs more frequently than an undo, we confine p's range to $1/2 \leq p \leq 1$. The buffer usage is $\sum_{i=0}^{n} iP_i$, the expected number of instructions that can be undone. Figure 4.6(a) illustrates the state transition diagram for the wraparound buffer implementation. A forward command at state n stays at state n due to wraparound, while an undo command at state 0 stays at state 0 due to no valid buffer entries at this state. The wraparound implementation can apply to both single buffer and twin-buffer. We can obtain the following equations during steady state.

$$\begin{aligned} P_n &= pP_n + pP_{n-1} \\ P_{n-1} &= (1-p)P_n + pP_{n-2} \\ &\vdots \\ P_1 &= (1-p)P_2 + pP_0 \\ P_0 &= (1-p)P_1 + (1-p)P_0 \end{aligned}$$

Since $\sum_{i=0}^{n} P_i = 1$, we can solve the equations and obtain $P_i = r^{n-i} \cdot \frac{1-r}{1-r^{n+1}}$, where $r = (1-p)/p$ and $i = 0, 1, \ldots, n$. Therefore, the buffer usage equals

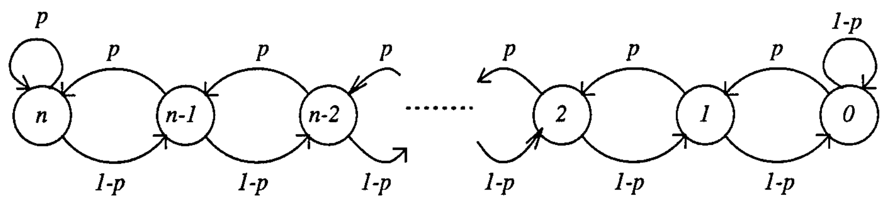
$$\sum_{i=0}^{n} iP_i = \frac{n - nr - r + r^{n+1}}{(1-r)(1-r^{n+1})} = \frac{n+1}{1-r^{n+1}} - \frac{1}{1-r}.$$

The buffer usage approaches n when p approaches 1 and r approaches 0. If the user chooses the forward command most of the time, the entire buffer entries are always valid. When the forward command and the undo command occur equally likely, i.e., $p = 1/2$ and $r = 1$, we can have the buffer usage

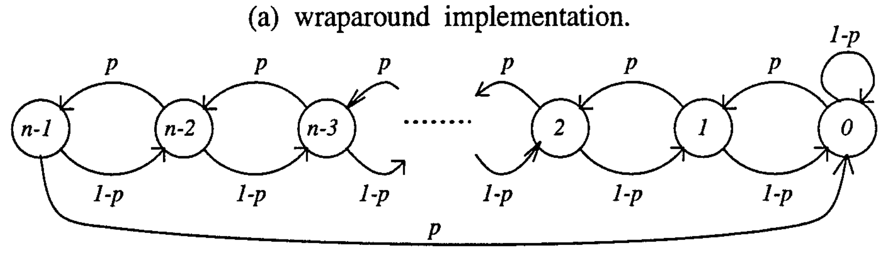
$$\lim_{r \to 1} \frac{n - nr - r + r^{n+1}}{(1-r)(1-r^{n+1})} = \frac{n}{2}.$$

Figure 4.6(b) illustrates the state transition diagram for the single buffer without undo wraparound implementation. Because the undo wraparound capability is disabled, all the buffer entries are invalid when we detect a full buffer. Therefore, a forward command at state $n-1$ transfers to state 0. We can obtain the following steady state equations and $\sum_{i=0}^{n-1} P_i = 1$.

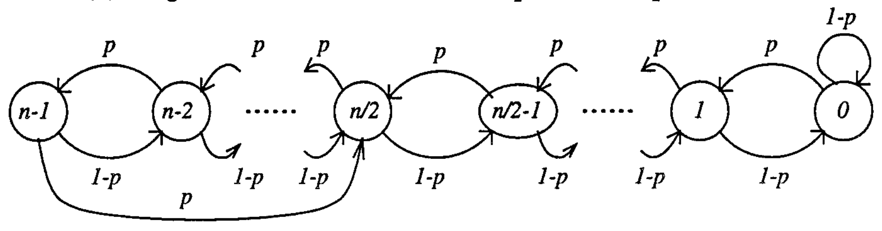
$$\begin{aligned} P_{n-1} &= pP_{n-2} \\ P_{n-2} &= (1-p)P_{n-1} + pP_{n-3} \end{aligned}$$



(a) wraparound implementation.



(b) single buffer without undo wraparound implementation.



(c) twin-buffer without full undo wraparound implementation.

Figure 4.6 Queueing models for three implementations

$$\vdots$$

$$\begin{aligned} P_1 &= (1-p)P_2 + pP_0 \\ P_0 &= (1-p)P_1 + (1-p)P_0 + pP_{n-1} \end{aligned}$$

As p approaches 1 and r approaches 0, we can have $P_0 = P_1 = \cdots = P_{n-1} = 1/n$. The buffer usage is $\sum_{i=0}^{n-1} i \cdot \frac{1}{n} = \frac{n-1}{2}$. When $p = 1/2$ and $r = 1$, we can have $P_i = (n-i)P_{n-1}$, for $i = 0, 1, \ldots, n-1$, and $P_{n-1} = \frac{2}{n(n+1)}$. Therefore, the buffer usage is

$$\sum_{i=0}^{n-1} iP_i = \sum_{i=0}^{n-1} i(n-i)P_{n-1} = \frac{n-1}{3}.$$

Figure 4.6(c) illustrates the state transition diagram for the twin-buffer without full undo wraparound implementation. When we detect a full working buffer, the roles of the working buffer and the backup buffer are interchanged. Although the entries in the original backup buffer are invalid due to no undo wraparound, all the entries in the original working buffer are valid, and can be used to undo instructions. State $n-1$ thus leads to state $n/2$ for a forward command. We can obtain the following steady state equations and $\sum_{i=0}^{n-1} P_i = 1$.

$$\begin{aligned}
P_{n-1} &= pP_{n-2} \\
P_{n-2} &= (1-p)P_{n-1} + pP_{n-3}; \quad \cdots\cdots \quad ; P_{n/2+1} = (1-p)P_{n/2+2} + pP_{n/2} \\
P_{n/2} &= (1-p)P_{n/2+1} + pP_{n/2-1} + pP_{n-1} \\
P_{n/2-1} &= (1-p)P_{n/2} + pP_{n/2-2}; \quad \cdots\cdots \quad ; P_1 = (1-p)P_2 + pP_0 \\
P_0 &= (1-p)P_0 + (1-p)P_1
\end{aligned}$$

As p approaches 1 and r approaches 0, we can have

$$\begin{aligned}
P_{n-1} &= P_{n-2} = \cdots = P_{n/2+1} = P_{n/2} \\
P_{n/2} &= P_{n-1} + P_{n/2-1} \\
P_{n/2-1} &= P_{n/2-2} = \cdots = P_1 = P_0
\end{aligned}$$

Therefore,

$$P_i = \begin{cases} \frac{2}{n} & \text{for } i = n/2, n/2+1, \ldots, n-1 \\ 0 & \text{for } i = 0, 1, \ldots, n/2-1 \end{cases}$$

The buffer usage is $\sum_{i=0}^{n-1} iP_i = \sum_{i=n/2}^{n-1} i \cdot (\frac{2}{n}) = \frac{3}{4}n - \frac{1}{2}$. When $p = 1/2$ and $r = 1$, we can have

$$P_{n-1} = \frac{8}{n(3n+2)}$$

and

$$P_i = \begin{cases} (n-i)P_{n-1} & \text{for } i = n/2, n/2+1, \ldots, n-2 \\ \frac{n}{2}P_{n-1} & \text{for } i = 0, 1, \ldots, n/2-1 \end{cases}$$

Therefore, the buffer usage is $\sum_{i=0}^{n-1} iP_i = \frac{7n^2-4}{18n+12}$.

Table 4.2 summarizes the buffer usages for a large n under different models, implementations, and forward probability p. When $p = 1$, the buffer usages of the second model match those of the first model. Among the implementations listed, the twin-buffer performs better than the single buffer in terms of the buffer usages. It is very unlikely that a user will issue more than two consecutive undo commands when one realizes that the buffer is empty. To incorporate this condition, we can adjust the stationary probability to 1 (or 0) from state 0 to state 1 (or from state 0 to state 0). We can further divide the buffers into more smaller buffers, so that the buffer usage increases. The buffer usage for a k-buffer, $k > 2$, is $\frac{2k-1}{2k}n - \frac{1}{2}$ under

Table 4.2 The buffer usages for a large n

	First model	Second model	
		$p = 1$	$p = 1/2$
Wraparound	n	n	$n/2$
Single buffer no undo wraparound	n/2	n/2	n/3
Twin-buffer no full undo wraparound	3n/4	3n/4	7n/18

the first model. For example, we can implement a four-buffer without full undo wraparound by adjusting the working buffer whenever the buffer is full. One buffer serves as the working buffer and the other three buffers as the backup buffers. State $n - 1$ leads to state $3n/4$ for a forward command. In fact, we can divide the buffer into n small buffers, where each small buffer include only one entry. There is one working buffer and $n - 1$ backup buffers. The n-buffer without full undo wraparound implementation switches the working buffer for every instruction. The actual debugging time does not improve because the frequent working buffer switching is no better than the marker checking for every instruction in the single buffer wraparound implementation. We choose the twin-buffer due to its ease of implementation, better buffer usage than single buffer, and less frequent working buffer switching.

4.5 Implementation

Two approaches have been implemented for incorporating the undo capability in GDB. The first approach performs a direct modification to GDB, using the standard *ptrace* and trapping system utilities. The implementation is straightforward, but the cost of trapping every instruction is high. The second approach employs a compiler to insert buffering instructions around each instruction. The normal execution time when the recording mode is on is significantly improved, with a code expansion penalty. We introduce several new commands which perform backward execution, and can enhance the user's debugging power. As mentioned in Section 4.3, *record-on*, *record-off*, and *undo* are the basic commands for the user to choose critical regions to examine. *Record-run* executes the program in a recording mode from the beginning. It is useful in measuring the run-time overhead induced by backward execution. *Print-buffer* dumps the twin-buffer contents to allow a close investigation over the program execution history.

The twin-buffer, proposed in Section 4.4.3, serves as the temporary storage, so that the effective buffer usage is increased without slowing down the debugging time. System library routines are considered well-developed programs. Both buffers are flushed when library routines are executed, to save the program debugging time. As a consequence, the debugger does not allow undoing system library routines. The strategy can apply to user-defined routines, such that the user can choose to skip recording certain well-debugged routines.

4.5.1 De-assembly and context switching

Without changing the compilers, the GDB can be modified directly to incorporate the undo command. The twin-buffer is implemented at the debugger's working space. When the recording mode is on, the debugger traps every instruction during continuous program execution. The current instruction is analyzed (or de-assembled) before its execution. The debugger can decide which values should be saved, fetch values from the corresponding locations, and store them to the working buffer. The debugger then switches the control back to its child process to execute the current instruction. Undo command involves decoding the previous instruction, fetching the values from the working buffer, restoring them to the corresponding locations, and adjusting the buffer index. This implementation is a direct extension of the *break ... if expression* command.

4.5.2 Compiler-assisted

Both the compiler and the debugger can be modified. The twin-buffer is implemented at the user's address space. Suitable instructions are inserted around each instruction by the compiler. Such instructions can be used to store register or memory values to the buffer, to adjust the buffer index, and to check if the buffer index is still within range. To better illustrate our idea, we insert buffering instructions at the assembly code level. Figure 4.7 illustrates the standard instructions inserted around the current instruction. Depending on the type of the current instruction, the values that need to be saved are different. *Flush_buffer* routine interchanges the roles of the backup buffer and the working buffer by reassigning *buf_index* and *buf_limit* to the beginning and the end of the backup buffer respectively. The buffer contents

```
store $r1;
load $r1 with buf_index;
store values to buffer using index $r1;
store PC;
decrease $r1;
store $r2;
load $r2 with buf_limit;
if ($r1 < $r2)
        flush_buffer();
restore $r2;
store $r1 to buf_index;
restore $r1;
*current instruction;
```

Figure 4.7 Standard buffering instructions

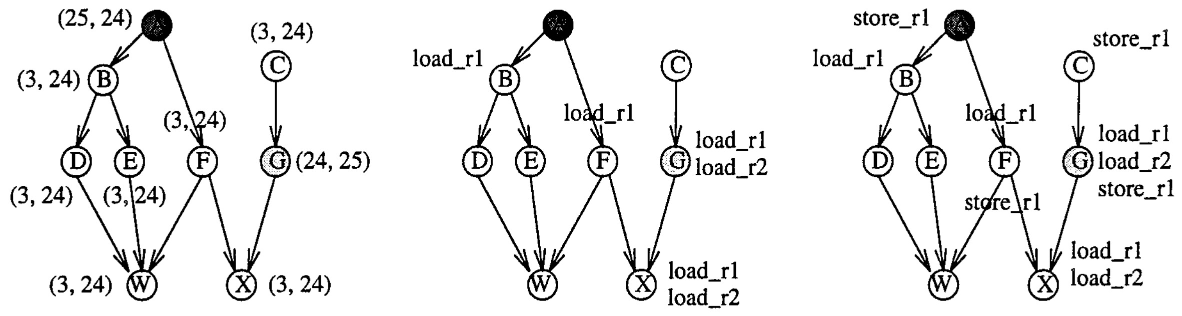
can also be flushed into secondary storages using this routine. Such capability is disabled for a faster program execution time in the recording mode. Figure 4.7 indicates that at least twelve instructions are inserted for bookkeeping and executing the current instruction. Furthermore, the third statement 'store values to buffer using index $\$r1$' may involve memory access, which requires saving a new register $\$r3$, loading the memory content to $\$r3$, and restoring $\$r3$. The extra instructions result in both code expansion and a slower program execution time.

The strategy is to keep the registers holding *buf_index* and *buf_limit* as long as possible without restoring their values back to the corresponding memory locations. If two specific registers are designated to hold *buf_index* and *buf_limit* respectively, all of the loads and stores of $\$r1$ and $\$r2$ can be eliminated, reducing the number of instructions inserted by 7. However, such assignment will impact the register usage since the number of available registers are two less. A more flexible strategy identifies and utilizes the set of registers whose values have no later uses at each instruction. A global dataflow analysis first computes the set of registers that do not have later uses, i.e., the dead register set [22], for each instruction. The compiler assigns priorities to the registers in the set, to serve as the guideline if a register should be chosen to hold the values of *buf_index* or *buf_limit*. For example, spill registers have the highest priorities since their live ranges are within 3.

The implementation includes three BFS traversals over the original instructions in the entire program. Actually the first BFS traversal builds up links so that the second and the third traversals only perform sequential searches along the links. The first traversal assigns each original instruction three registers, to hold *buf_index*, *buf_limit*, and the memory content if the current instruction is a store-to-memory instruction. The assignment is based on if the registers are dead at the current instruction and their priorities. Dead registers are considered first over the priority. If the number of the dead registers is less than 3, we choose the registers with the highest priorities, which are not used nor defined at the current instruction. There are three markers, *load_r1*, *load_r2*, and *store_r1*, initially 0's associated with each original instruction, representing if the current instruction should include 'load *buf_index* to $\$r1$', 'load *buf_limit* to $\$r2$', and 'store $\$r1$ to *buf_index*' respectively. The second traversal sets *load_r1* (or *load_r2*) for the original instruction I_1 if there exists a parent instruction whose $\$r1$ (or $\$r2$) does not match I_1's $\$r1$ (or $\$r2$). In other words, there is no need to load the buffer index (or buffer limit) from memory if the current instruction shares with all of its parents the same register that holds the buffer index (or buffer limit). The third traversal sets *store_r1* for the original instruction I_1 if there is a child I_j whose *load_r1* marker is set. Since I_j will perform a load $\$r1$ with *buf_index* instruction, we must assign *buf_index* a correct value. Therefore, I_i should save its $\$r1$ value to *buf_index*.

Example Consider the flow graph as shown in Figure 4.8. The pair of numbers associated with each instruction denote the assigned values for $\$r1$ and $\$r2$ respectively, in Figure 4.8(a). There are three registers, 3, 24, and 25, chosen most of the time, since they are the spill registers used in the old version of the IMPACT C compiler. Solid shadow, light shadow, and white represent the colors of the nodes, i.e., the different assignments of $\$r1$. In Figure 4.8(b), since nodes B, F, G and X have at least a parent with different $\$r1$'s, i.e., in different colors, their corresponding *load_r1* values are set to 1. Similar argument applies to nodes G and X for $\$r2$. In Figure 4.8(c), since nodes A, C, F, and G have at least one child with a *load_r1* value set to 1, their corresponding *store_r1* values are set to 1.

A post-pass compiler have been implemented, which performs a global dataflow analysis, inserts buffering instructions, adjusts procedure names, and generates the additional map table. Both the original source code and the new expanded code in assembly format are compiled into the new executable code. Procedure calls except system routines have two versions, the original and the expanded one with a different name. Procedure call instructions in an expanded copy are modified to call the corresponding procedures of the expanded versions instead.



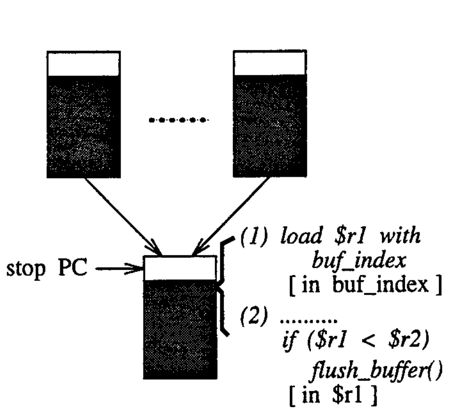
(a) Assign temporary registers. (b) After the 2nd traversal. (c) After the 3rd traversal.

Figure 4.8 The three BFS traversals for buffering instruction insertion

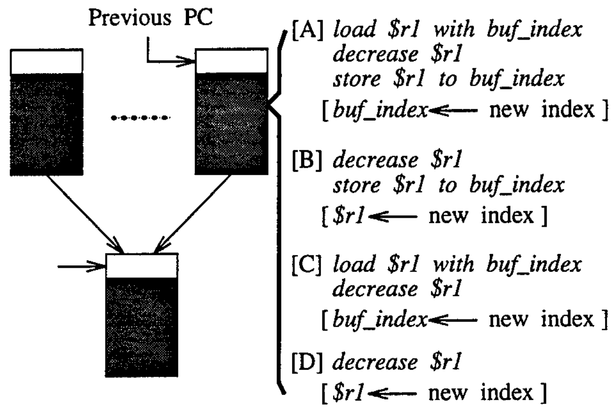
The compiler can generate an additional map table, $MAP2$, between the original executable code and the new expanded executable code by incrementally maintaining the program counter. When buffering instructions are inserted, their corresponding numbers of bytes are calculated to update the beginning of the next instruction. In this way, the modified GDB can have correct operations for several user commands.

When the GDB is in the record-off mode, the user commands are operated on the original executable code, except the *record-on* and *record-run* commands. The *record-run* starts the execution from the beginning of the expanded *main* routine, while the *record-on* sets the program counter of the user program to the corresponding procedure of the expanded version, both involving $MAP2$ table lookup. During the recording mode, the user can type in usual commands like *step*, *next*, *continue*, *breakpoint* etc., with the modification to GDB to include checking $MAP2$ table. *Print-buffer* command provides a way to examine the program recent history. The *record-off* command in the recording mode switches the program counter back to its corresponding original procedure, using the $MAP2$ table.

The *undo* command, however, is more complicated, since we need to recover the value of *buf_index*, and possibly, the value of *buf_limit*. Because of the elimination of certain load and store instructions, the most recent value of *buf_index* may be in its corresponding memory location, or in register $\$r1$. We need to find the buffer index first, and then adjust the content in either the memory location, i.e., *buf_index*, or $\$r1$, for the previous instruction. Assuming that the user program stops at a location in the expanded code, corresponding to the beginning of a specific instruction in the original executable code. The 'load $\$r1$ with buf_index' for the current instructions may be present or absent. The cases can be determined by de-assembling and



(a) Locate the buffer index.



(b) Adjust the variables holding buffer index.

Figure 4.9 Buffer index and buffer limit recovery

analyzing several instructions. If such an instruction is present, the correct buffer index value is in *buf_index*. Otherwise, we look for instructions in the format 'if ($r1 < $r2) flush_buffer()', and get the correct buffer index value from $r1, as shown in Figure 4.9(a). The buffer index value is then increased by 8 for the history write buffer scheme, or 12 for the read buffer scheme, which can be used to find the buffer entry for the previous instruction, and locate its program counter from the buffer. The user program can then be rolled back using the corresponding program counter, and the old values are recovered. However, the user program can not continue without suitable modifications due to the fact that the memory content in *buf_index* or the register value in $r1 does not have the most recent value.

Both 'load $r1 with *buf_index*' and 'store $r1 to *buf_index*' instructions for the previous instruction may be present or absent in the expanded code, resulting in four cases as shown in Figure 4.9(b). For cases [A] and [C], the memory location *buf_index* must be reloaded with the current buffer index, while for cases [B] and [D], $r1's value must be adjusted. In other words, de-assembly and analysis should be applied to both the stopped instruction and the previous instruction.

Similarly, *buf_limit* or $r2 can be recovered during the *undo*. Whenever an undo exhausts the working buffer, and switches to the backup buffer, if any, the new buffer limit should be assigned to *buf_limit*. Also if the instruction to be recovered does not include 'load $r2, buf_limit', $r2 should be assigned the *buf_limit* value.

4.6 Experimental Results

The same set of benchmarks as in the previous chapters are evaluated for read buffer and history write buffer schemes respectively. The experiments are conducted on a DEC-station 3100. The size overhead is measured by calculating the ratio of the numbers of assembly instructions between the expanded code and the original code, while the execution time overhead is the ratio of program execution times, between the expanded version of the program in a recording mode and the original code.

For the first approach, i.e., the context switching approach, there is no code growth, since the GDB is modified to run on the original executable code. However, the execution time overhead increases drastically as the size of the twin-buffer increases. For example, it runs more than 2,000 times slower on the recording mode for QUEEN benchmark, using a twin-buffer of size 1M bytes.

The compiler-assisted approach improves the normal execution time significantly when the recording mode is on, but with a moderate code growth. The code growth ratio ranges from 6.5 and 8.2, with an average 7.1 for the history write buffer scheme, and from 6.5 and 8.6, with an average 7.3 for the read buffer scheme. In general, the history write buffer scheme generates smaller code growth ratio than the read buffer scheme does. It may be because the history write buffer scheme saves one operand, and the read buffer scheme saves two. Among the twelve benchmarks, only PUZZLE and COMPRESS have a slightly better code growth ratio for the read buffer scheme. We thus choose the history write buffer scheme in our implementation, and measure its run-time overhead. The run-time overhead when the recording mode is on ranges from 4.5 to 6.4, with an average 5.6, for a twin-buffer of size ranges from 4K bytes to 8M bytes. The overhead increases as the twin-buffer size decreases within 4K bytes, since the effect of swapping buffers between the working buffer and the backup buffer impacts the normal execution. For each benchmark, the overhead stabilizes when the size of the twin-buffer exceeds 4K bytes. The compiler aborts when we try to compile programs with a twin-buffer of size more than 8M bytes for either schemes. Since the history write buffer scheme stores two data item, accounting to eight bytes for MIPS machines, for each buffer entry, up to 1M instructions can be undone. The read buffer scheme saves three data item, accounting to twelve bytes per buffer entry, and up to 666K instructions can be undone.

Although the undo command involves de-assembling and analyzing the expanded current instruction and the expanded previous instruction, the time for an undo command to finish is within 0.5 seconds, which is a reasonable number in an interactive environment. The normal execution time in a recording mode will increase if the buffer contents are flushed to secondary storages. The following formula is used to measure its effect. Let N be the total number of instructions executed, $2 \times s$ the twin-buffer size, avg the average number of instructions that each original instruction is expanded, and $B(s)$ the function of time needed to flush a buffer of size s to a secondary storage. Assuming that each instruction takes a unit time to complete. The total running time is $avg \times N + \frac{N}{s} \times B(s)$. In other words, the overhead is $(avg - 1) + \frac{B(s)}{s}$. For the best case, it takes one unit time to flush each buffer entry, resulting in a total execution time $(avg + 1) \times N$. However, the secondary storage access time is much larger than the functional unit cycle time. The overhead $\frac{B(s)}{s}$ can be quite a large number. Because of this factor, the larger the twin-buffer is, the longer time it takes to flush the buffer.

4.7 Summary

Two approaches that incorporate the undo capability into GDB were described. The context-switching is easy to program, and has no code growth on the user executable program, with a high normal execution time slow down. The compiler can significantly reduce the run-time overhead, with a moderate code growth. A twin-buffer is proposed to allow half of the buffer entries useful to undo instructions when the working buffer is full. The twin-buffer without full undo wraparound implementation has a better buffer usage than the single buffer without undo wraparound.

The data recording in the implementations is at the assembly code level due to its ease of presentation [44]. The performance measurement can serve as the worst case that is required to incorporate the undo capability into a debugger. A smaller code growth, and a faster program debugging time can be achieved if the recording is at a higher level, e.g., each statement boundary in C language [56]. Also if a page protection mechanism is employed, the buffer out of bound checking can be done by protecting the last buffer entry. Therefore all the buffering instructions that include $\$r2$ can be eliminated.

System routines can not be undone. The strategy can apply to well-developed procedures. Checkpoint setting can be easily incorporated in the implementation, so that the execution can be rolled back to a previously saved checkpoints. Procedure and loop boundaries are the potential locations to set checkpoints. With this extension, the user can have more power in program debugging.

Chapter 5

Conclusions

5.1 Summary

This thesis has described schemes that have been implemented for multiple instruction retry for RISC-type scalar processors. Incremental updating and post-pass code reordering were employed to improve the code run-time and reduce the compilation time. The threshold for the number of instructions increased from 800 to 15,000. The compilation time has been reduced from more than two hours to within three minutes for a source program of 1,000 lines.

The multiple instruction retry has also been applied to VLIW architectures, using both compiler-based, and hardware/compiler combined approaches. An analytical bookkeeping code scheduling algorithm was proposed to schedule code for VLIW architectures. Such a code scheduling algorithm alleviates the coding complexity of the previous trace scheduling algorithm, and reduces the code duplication of the superblock scheduling. Experimental results have shown that with a read buffer of $2n \times P$ entries, both code growth and execution time overhead due to multiple instruction word retry can be reduced to negligible amounts.

Approaches to incorporating the undo capability into debuggers were presented, using both history write buffers and read buffers. A twin-buffer has been proposed to allow an efficient buffer usage for undoing up to 1M instructions. Two approaches were implemented based on a modified version of the GDB. Context-switching traps every instruction and stores suitable values based on de-assembling the current instruction. Storage overhead is zero for the user executable code, but with a slow program debugging time. Compilers can be utilized to insert suitable instructions to store values at the run-time. The normal execution time in recording mode is significantly improved, with a moderate code growth.

5.2 Limitations

The compiler-based multiple instruction retry for both scalar processors and VLIW architectures resolves the data hazards associated with instruction rollback. The hardware delayed write buffers are removed for general purpose register files. However, since memory disambiguation problem is pretty hard to handle during the compile time, a delayed write buffer for memory is still needed. The I/O units should also maintain delayed write buffers. To perform instruction retry using compiler approach, the system routines will have to be re-compiled.

Both the history write buffer and the read buffer have been employed to store the past instruction history. The two buffering schemes can not replay output commit instructions, which is the main reason that system libraries can not be undone, and the checkpoint setting before each system library call is observed. Although the secondary storage may be present, it may still be impossible to rollback to the beginning for long running programs. The variables in the read buffer scheme may not be current [38], since the last read of the variable can be out of the buffer range, and the buffer does not have its old value. Such a variable can not be recovered if there is a write to that variable before undoing occurs. Also, the size of the buffer is a main factor in the performance of the secondary storage.

5.3 Future Research

The code scheduling algorithm for VLIW machines does not include the advanced loop scheduling, such as software pipelining [31] and loop unrolling [37]. Code motion around various program constructs, including loops, if-then-elses, and subroutine calls [57] can also be incorporated in the scheduling algorithm. The assumption that each instruction takes a unit time to complete can be replaced by the actual cycle time of the corresponding instruction, in order to get a more accurate estimation for the VLIW performance.

Checkpoint setting can be incorporated into the current implementation of GDB, to bypass loops, system libraries, and well-debugged procedures. In UNIX systems, the system library *gcore* can be used to dump the child's code space to a file. However, to rollback to a specific checkpoint requires the debugger to look into the checkpointed core file and to restore suitable information from the file, including the stack frames, the heap data generated by memory allocation libraries, local and global variables, the twin-buffer, the buffer index, the register file, and the PC etc.. The buffering schemes can be applied in building a symbolic debugger for VLIW and superscalar architectures. Also the modified GDB can serve as a frame work useful for developing a symbolic debugger in a distributed environment.

References

[1] C.-C. J. Li, S.-K. Chen, W. K. Fuchs, and W.-M. W. Hwu, "Compiler-assisted multiple instruction retry," Tech. Rep. CRHC-91-31, Coordinated Science Laboratory, University of Illinois, May 1991. to appear in *IEEE Trans. on Computers*.

[2] 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 *The Twenty-Second International Symposium on Fault-Tolerant Computing*, pp. 66–73, July 1992.

[3] X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and calibration of a transient error reliability model," *IEEE Transactions on Computers*, vol. C-31, pp. 658-671, July 1982.

[4] R. Iyer and D. Rossetti, "A measurement-based model for workload dependence of CPU errors," *IEEE Transactions on Computers*, vol. C-35, pp. 511-519, June 1986.

[5] L. Svobodova, "Resilient distributed computing," *IEEE Transactions on Software Engineering*, vol. SE-10, No. 3, pp. 257–268, May 1984.

[6] L. Lin and M. Ahamad, "Checkpointing and rollback-recovery in distributed object based systems," in *The Twentieth International Symposium on Fault-Tolerant Computing*, pp. 97–104, 1990.

[7] K. Tsuruoka, A. Kaneko, and Y. Nishihara, "Dynamic recovery schemes for distributed processes," in *IEEE 2nd Symp. on Reliability in Distributed Software and Database Systems*, pp. 124–130, 1981.

[8] C.-C. J. Li and W. K. Fuchs, "CATCH - Compiler-assisted techniques for checkpointing," in *The Twentieth International Symposium on Fault-Tolerant Computing*, pp. 74–81, June 1990.

[9] W.-M. W. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Transactions on Computers*, vol. C-36, pp. 1496 -1514, Dec. 1987.

[10] M. L. Ciacelli, "Fault handling on the IBM 4341 processor," in *The Eleventh International Symposium on Fault-Tolerant Computing*, pp. 9–12, June 1981.

[11] M. S. Pittler, D. M. Powers, and D. L. Schnabel, "System development and technology aspects of the IBM 3081 processor complex," *IBM Journal of Research and Development*, vol. 26, pp. 2-11, Jan. 1982.

[12] W. F. Bruckert and R. E. Josephson, "Designing reliability into the VAX 8600 System," *Digital Technical Journal of Digital Equipment Corporation*, vol. pp. 71-77, Aug. 1985.

[13] D. B. Fite, T. Fossum, and D. Manley, "Design strategy for the VAX 9000 system," *Digital Technical Journal of Digital Equipment Corporation*, vol. pp. 13-24, Fall 1990.

[14] 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.

[15] G. L. Hicks, D. Howe Jr., and A. Zurla Jr., "Instruction retry mechanism for a data processing system." United States Patent, no. 4044337, Aug. 1977, Assignee: International Business Machines Corporation, Armonk, N.Y.

[16] J. E. Smith and A. R. Pleszkun, "Implementing precise interupts in pipelined processors," *IEEE Transactions on Computers*, vol. C-37, pp. 562-573, May 1988.

[17] Y. Tamir and M. Tremblay, "High-performance fault-tolerant VLSI systems using micro rollback," *IEEE Transactions on Computers*, vol. C-39, pp. 548-554, Apr. 1990.

[18] Y. Tamir, M. Liang, T. Lai, and M. Tremblay, "The UCLA mirror processor: A building block for self-checking self-repairing computing nodes," in *The Twenty-First International Symposium on Fault-Tolerant Computing*, pp. 178–185, June 1991.

[19] L. Spainhower, J. Isenberg, R. Chillarege, and J. Berding, "Design for fault-tolerance in system ES/9000 model 900," in *The Twenty-Second International Symposium on Fault-Tolerant Computing*, pp. 38–47, July 1992.

[20] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, pp 1184-1201, Dec. 1986.

[21] P. Chang, W. Chen, N. Warter, and W.-M. W. Hwu, "IMPACT: An architecture framework for multiple-instruction-issue processors," in *The 18th Annual International Symposium on Computer Architecture*, pp. 266–275, May 1991.

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

[23] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* The MIT Press, 1990.

[24] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, 1979.

[25] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *The 10th Annual International Symposium on Computer Architecture*, pp. 140–150, 1983.

[26] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180–192, 1987.

[27] B. R. Rau, D. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *Computer*, pp. 12–35, Jan. 1989.

[28] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478 -490, July 1981.

[29] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, 1986.

[30] W.-M. W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery, "The Superblock: an effective technique for VLIW and superscalar compilation," *the Journal of Supercomputing*, pp. 229–248, July 1993.

[31] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, 1988.

[32] J. G. Holm and P. Banerjee, "Low cost concurrent error detection in a VLIW architecture using replicated instructions," in *The Proceedings of the International Conference on Parallel Processing*, pp. 192–195, 1992.

[33] M. A. Schuette and J. P. Shen, "Exploiting instruction-level resource parallelism for transparent integrated control-flow monitoring," in *The Twenty-First International Symposium on Fault-Tolerant Computing*, pp. 318–325, 1991.

[34] D. M. Blough and A. Nicolau, "Fault tolerance in super-scalar and VLIW processors," in *1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 193–200, 1992.

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

[36] D. E. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching.* Addison-Wesley, 1973.

[37] S. Weiss and J. E. Smith, "A study of scalar compilation techniques for pipelined supercomputers," in *Proceedings of the Second International Conference on Architectural Support for Programming*, pp. 105–111, Oct. 1987.

[38] J. L. Hennessy, "Symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems*, vol. 4, No. 3, pp. 323–344, July 1982.

[39] D. Wall, A. Srivastava, and R. Templin, "A note on Hennessy's symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems*, vol. 7, No. 1, pp. 176–181, Jan. 1985.

[40] M. Copperman and C. E. McDowell, "A further note on Hennessy's symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems*, vol. 15, No. 2, pp. 357–365, Apr. 1993.

[41] M. Copperman, "Debugging optimized code without being misled," *ACM Transactions on Programming Languages and Systems*, vol. 16, No. 3, pp. 387–427, May 1994.

[42] R. M. Balzer, "EXDAMS – Extendable debugging and monitoring system," in *Proceedings of the Spring Joint Computer Conference*, pp. 567–580, 1969.

[43] E. H. Satterthwaite Jr., "Debugging tools for high level languages," *Software–Practice and Experience*, vol. 2:3, pp. 197–217, July 1972.

[44] R. E. Fairley, "ALADDIN: Assembly language assertion driven debugging interpreter," *IEEE Transactions on Software Engineering*, vol. SE-5, No. 4, pp. 426–428, July 1979.

[45] G. B. Leeman Jr., "A formal approach to undo operations in programming languages," *ACM Transactions on Programming Languages and Systems*, vol. 8, No. 1, pp. 50–87, Jan. 1986.

[46] M. V. Zelkowitz, "Reversible execution," *Communications of the ACM*, vol. 16:9, pp. 566, Sept. 1973.

[47] J. S. Vitter, "US&R: A new framework for redoing," in *SIGPLAN Not. Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments 19*, pp. 168–176, Apr. 1984.

[48] G. B. Leeman Jr, "Building undo/redo operations into the C programming language," in *The 15th International Symposium on Fault-Tolerant Computing*, pp. 410–415, 1985.

[49] J. E. Archer Jr., R. Conway, and F. B. Schneider, "User recovery and reversal in interactive systems," *ACM Transactions on Programming Languages and Systems*, vol. 6, No. 1, pp. 1–19, Jan. 1984.

[50] S. I. Feldman and C. B. Brown, "IGOR : A system for program debugging via reversible execution," in *SIGPLAN Notice*, pp. 112–123, Jan. 1989.

[51] M. S. Johnson, "Some requirements for architectural support of software debugging," in *SIGPLAN Notice*, pp. 140–148, Mar. 1982.

[52] *GNU debugger.* Free Software Foundation, Inc.

[53] B. W. Kernighan and R. Pike, *The UNIX Programming Environment.* Prentice-Hall, Inc., Englewood Cliffs, 1984.

[54] R. M. Stallman, *Using and Porting GNU CC.* Free Software Foundation, Inc., 1989.

[55] L. Kleinrock, *Queusing Systems, Volumn I : Theory.* Wiley-Interscience (New York), 1975.

[56] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, 1978.

[57] S.-K. Chen, W. K. Fuchs, and W.-M. W. Hwu, "An analytical approach to scheduling code for superscalar and VLIW architectures," in *The Proceedings of the International Conference on Parallel Processing*, vol. I, pp. 285–292, Aug. 1994.

Vita

Shyh-Kwei Chen was born in Kaohsiung, Taiwan, on August 9, 1961. He received the B.S. degree in Computer Science and Information Engineering from the National Taiwan University, Taipei, Taiwan, in 1983, and the M.S. degree in Computer Science from the University of Minnesota, Minneapolis, MN, in 1987. While pursuing his Ph.D. degree at the University of Illinois, he was a research assistant in the Center for Reliable and High-Performance Computing at the Coordinated Science Laboratory from 1990 to 1994. He is a student member of the IEEE Computer Society. After getting his Ph.D., he will be joining IBM T. J. Watson Research Center, Yorktown Heights, New York, as a postdoctor, working on a parallelizing compiler and architecture for VLIW and superscalar machines. His research interests include parallel processing, compilers, debuggers, and fault-tolerant computing.