SUBMITTED TO: IEEE TRANSACTIONS ON COMPUTERS

COMPILER-ASSISTED MULTIPLE INSTRUCTION ROLLBACK RECOVERY USING A READ BUFFER

N. J. Alewine[1], *S.-K. Chen, W. K. Fuchs, W.-M. Hwu*

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
1308 West Main Street
University of Illinois
Urbana, IL 61801

Primary contact: W. Kent Fuchs
Phone: (217) 333-8294
FAX: (217) 244-5686
e-mail to fuchs@crhc.uiuc.edu

May, 1993

ABSTRACT

Multiple instruction rollback (MIR) is a technique that has been implemented in mainframe computers to provide rapid recovery from transient processor failures. Hardware-based MIR designs eliminate rollback data hazards by providing data redundancy implemented in hardware. Compiler-based MIR designs have also been developed which remove rollback data hazards directly with data-flow transformations.

This paper focuses on compiler-assisted techniques to achieve multiple instruction rollback recovery. We observe that some data hazards resulting from instruction rollback can be resolved efficiently by providing an operand read buffer while others are resolved more efficiently with compiler transformations. A compiler-assisted multiple instruction rollback scheme is developed which combines hardware-implemented data redundancy with compiler-driven hazard removal transformations. Experimental performance evaluations indicate improved efficiency over previous hardware-based and compiler-based schemes.

Index terms: fault-tolerance, error recovery, instruction retry, compilers, hardware assisted retry.

[1]International Business Machines Corporation, Boca Raton, Fl.

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

1 Introduction

Instruction retry is a technique for rapid recovery from transient faults in a processing system. Multiple instruction rollback recovery is particularly appropriate when error detection latencies or when error reporting latencies are greater than a single instruction cycle.

When transient processor errors occur, multiple instruction rollback (also referred to as multiple instruction retry or simply instruction retry) can be an effective alternative to system-level checkpointing and rollback recovery [1–6]. Multiple instruction retry within a sliding window of a few instructions [2–5], or re-execution of a few cycles [7], can be implemented in parallel with concurrent, algorithm-based, or control-flow error detection methods for recovery from transient processor errors.

1.1 Hardware-Based Instruction Rollback

Hardware implemented instruction retry schemes belong to one of two groups: 1) full checkpointing and 2) incremental checkpointing. Full checkpointing maintains "snapshots" of the required system state space at regular, or predetermined, intervals. Upon error detection, the system can be rolled back to the appropriate checkpointed system state. Incremental checkpointing maintains changes to the system state in a "sliding window". Upon error detection the system state is restored by undoing, or "backing-out" the system state changes up to the instruction in which the error occurred.

The issues associated with instruction retry are similar to the issues encountered with exception handling in an out-of-order instruction execution architecture. If an instruction is to write to a register and N is the maximum error detection latency (or exception latency), two copies of the data must be maintained for N cycles. Hardware schemes such as reorder buffers, history buffers, future files [8], and micro-rollback [2] differ in where the updated and old values reside, circuit complexity, CPU cycle times, and rollback efficiency.

Table 1 gives a description of various hardware-based methods to restore the general purpose register file contents during single or multiple instruction rollback. In the VAX 8600 and VAX 9000, errors are detected prior to the completion of a faulty instruction. For most VAX instructions, updates to the system state occur at the end of the instruction. If the error is detected prior to the updating of the system state, the instruction can be rolled back and re-executed. If the system

Table 1: Hardware-based single and multiple instruction rollback schemes.

Rollback Scheme	Checkpoint Type	Rollback Distance	Location of Data	
			Primary	Redundant
IBM 4341 [9]	full	single instr.	register file	shadow file
IBM 3081 [1]	full	10-20 instr.	register file	shadow file
VAX 8600 [10]	full	single instr.	register file	not required
IBM patent 4,912,707 [6]	full	variable	register file	shadow file
IBM patent 4,044,337 [11]	incremental	single instr.	register file	shadow files
micro-rollback [2]	incremental	variable	write buffer	register file
history buffer [8]	incremental	variable	register file	history buffer
history file [8]	incremental	variable	register file	shadow file
VAX 9000 [12]	full	single instr.	register file	not required
IBM E/S 9000 [5]	incremental	variable	virtual file	physical file

state has changed prior to detection of the error, a flag is set to indicate that instruction rollback cannot be accomplished. Redundant data storage is not required for the VAX 8600 and VAX 9000.

The IBM 4341, IBM 3081, IBM patent 4,912,707, IBM patent 4,044,337, and history file all require shadow file structures to maintain redundant data. This data is used to restore the system state during rollback recovery. Shadow file structures can add significant circuit overhead, although the level sensitive scan design [13] of the IBM 4341 and IBM 3081 provides this feature without additional cost over that incurred to obtain testability.[2] The VAX 8600 and VAX 9000 schemes avoid shadow files, however, require an error detection latency of only one instruction.

The micro-rollback scheme also avoids shadow files by using a delayed write buffer to prevent old data from being overwritten until the error detection latency has expired; ensuring that the new data is fault-free. In a delayed write scheme, the most recent write values are contained in the delayed write buffer, and bypass circuitry is required to forward this data on subsequent reads. The performance impact introduced by the bypass circuitry is a function of the register file size and the maximum rollback distance [2].

The history buffer scheme maintains redundant data in a separate push-down array and therefore does not require bypass circuitry [8]. The history buffer does however require an extra register file port which complicates the file design and can impact performance by increasing file access

[2]The 126 scan rings of the IBM 3081 contains 35,000 bits of data.

times.

In an effort to increase the register file size while maintaining down-level code compatibility relative to the 16 architectural registers, the IBM E/S 9000 has introduced a virtual register management (VRM) system [14]. The VRM circuitry dynamically maps the eight architectural registers into 32 physical registers. When the data in a physical register becomes obsolete, the physical register is released for reassignment as a new virtual register. Although the VRM system was primarily intended to reduce register pressure and therefore improve system performance, it has been extended to provide data redundancy to assist in rollback recovery. In the VRM extension, remapping of a physical register to a new virtual register is postponed until the error detection latency has been exceeded for the data contained in the physical register.

1.2 Compiler-Based Instruction Rollback

Recently, compiler-based approaches to multiple instruction rollback recovery have been investigated [3,4]. Compiler-based MIR uses data-flow manipulations to remove data hazards that result from multiple instruction rollback. Rollback data hazards (or just hazards) are identified by *antidependencies*[3] of length $\leq N$, where N represents the maximum rollback distance. Antidependencies are removed at three levels: 1) pseudo-code level, or the code level prior to variables being assigned to physical registers, 2) machine-code level, or the code level in which variables are assigned to physical registers, and 3) post-pass level, which represents assembler-level code emitted by the compiler. Compiler-based multiple instruction rollback reduces the requirement for data redundancy logic present in hardware-based instruction rollback approaches.

1.3 Compiler-Assisted Instruction Rollback

Compiler-based multiple instruction rollback resolves all data hazards using compiler transformations. This paper introduces a compiler-assisted instruction rollback scheme which uses dedicated data redundancy hardware to resolve one type of rollback data hazard while relying on compiler assistance to resolve the remaining hazards. Experimental results indicate that by exploiting the unique characteristics of differing hazard types, the new compiler-assisted MIR design can achieve superior performance to either a hardware-only or compiler-based instruction rollback scheme.

[3]For a complete presentation of data-flow properties and manipulation methods, see [15].

2 Error Model and Hazard Classification

2.1 Rollback Data Hazard Model

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

Given the above assumptions, any error which does not manifest itself as an illegal path in the control-flow graph (CFG) of the program is allowed provided that the following two conditions are satisfied: 1) register file contents do not spontaneously change, and 2) data can not be written to an incorrect register location. There are four targeted error types: 1) CPU errors such as those caused by an ALU failure, 2) incorrect values being read from I/O, memory, the register file, or external functional units such as the floating point unit, 3) correct/incorrect values being read from incorrect locations within the I/O, memory, or register file, and 4) incorrect branch decisions resulting from error types 1, 2, or 3.

2.2 Hazard Classification

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

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

Property 1: $x \in H_{regs}$ *iff* there exists a sequence of instructions $I_1, I_2, \ldots, I_N$ which form a legal walk[4] in G such that x is *live* at I_1, and x is defined during the walk.

Proof: For the *if* case, an error occurring in I_1 will be detected by I_N. During the retry of I_1, x will be in an inconsistent state since it was defined during the walk. Since x is *live* at I_1, there

[4]A *walk* is a sequence of edge traversals in a graph where the edges visited can be repeated [16].

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

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

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

An on-path or branch data hazard occurs when I_i defines variable x, and after rollback, I_j uses the corrupted x value prior to its being redefined. To simplify subsequent discussion, such on-path and branch hazards will be denoted $h_o(i, j, x)$ and $h_b(i, j, x)$ respectively. Figure 1 illustrates this hazard notation.

3 Compiler-Assisted Instruction Rollback

As shown in Section 2, rollback data hazards are of two types: 1) on-path hazards, and 2) branch hazards. Previous work has shown that compiler-driven data-flow manipulations can be used to resolve both on-path [3] and branch [4] hazards. Compiler-assisted multiple instruction rollback described in this section uses hardware to resolve on-path hazards and relies on compiler assistance to resolve the remaining branch hazards.

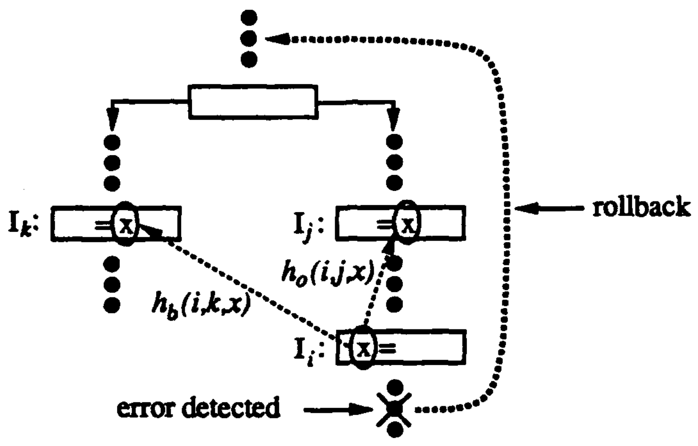


Figure 1: On-path and branch hazards.

3.1 On-path Hazard Resolution Using a Read Buffer

Figure 2 shows a hardware scheme to resolve on-path hazards. A *read buffer* is attached to the output ports of the register file. Each time a register is used it appears on the read port and is saved in the read buffer. If a register r_k is defined in I_i and it is an on-path hazard, then r_k must have been read within the last N cycles. In this case, the read buffer will contain the old value and it is permissible to write the new value into the register file. In the event of a rollback of N instructions, the contents of the read buffer are flushed in reverse order and stored back to the register file. For an on-path hazard, the path taken after the rollback will be the same as the path taken prior to rollback and each read of r_k will produce the same value as before. It is assumed that the read buffer is an integral part of the register file and any error in the system does not corrupt the transfer to the read buffer or its contents.

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

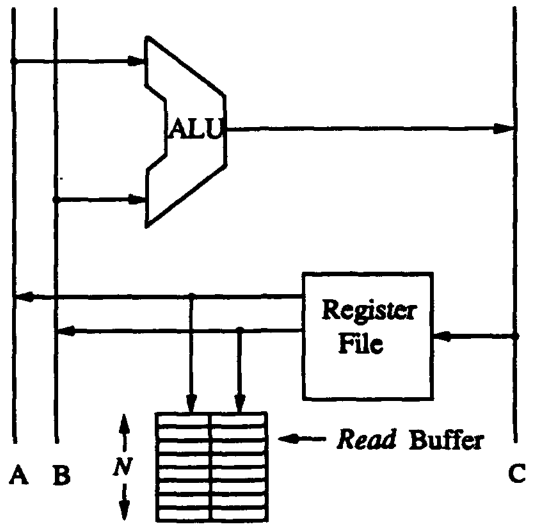


Figure 2: *Read* buffer.

3.1.1 Covering on-path hazards

In addition to resolving all on-path hazards, the read buffer will resolve some branch hazards. Figure 3 shows an on-path hazard and a branch hazard both with definitions of x in I_i and uses of x, after rollback, in instructions I_j and $I_{j'}$, respectively. Note that if path l is initially taken, the read buffer will contain the old value of x and rollback would be successful. However if path m is taken, the read buffer will not contain the old value of x and rollback would be unsuccessful. If only paths such as l exist, the presence of the on-path hazard assures successful rollback or "covers" the branch hazard. In this case, resolution of the branch hazard using compiler techniques is not necessary.

3.1.2 Post-pass transformation

Given the efficiency of the read buffer in resolving on-path hazards, a post-pass transformation on assembler-level code becomes possible as an alternative to nop insertion transformations [3]. The post-pass transformation creates on-path hazards when necessary to assure that all branch hazards are resolved by the read buffer. Given one such branch hazard which defines physical register r_k at instruction I_i, the transformation inserts an MOV r_k, r_k instruction immediately before I_i. This guarantees that all paths leading to I_i are like path l in Figure 3.

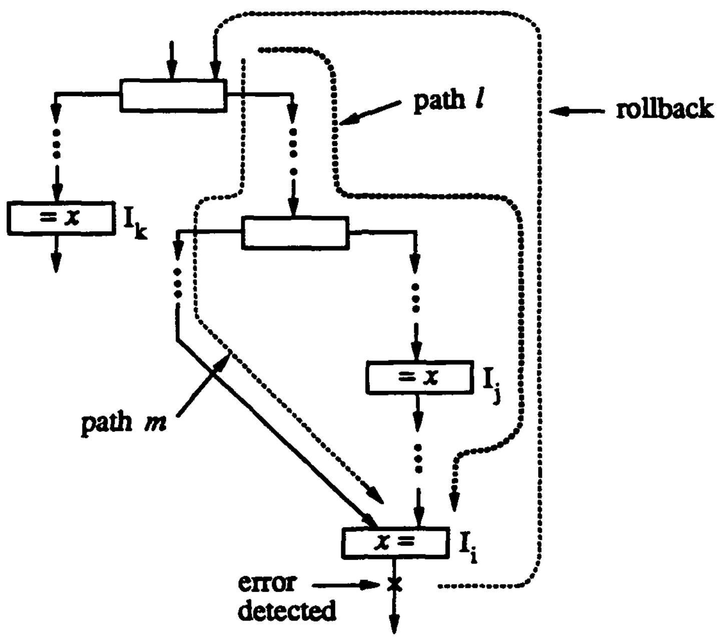


Figure 3: Covering on-path hazard.

3.2 Branch Hazard Resolution

Compiler transformations have been shown to be effective in resolving branch hazards [4]. Branch hazards are resolved at three levels: 1) pseudo-level, 2) machine-level, and 3) post-pass level. Pseudo-level hazards are removed by variable renaming, for example, renaming variable x to y in instruction I_i of Figure 1. Machine-level branch hazards occur when register assignments result in branch hazards that were not present at the pseudo-level. Machine-level hazards are resolved by adding hazard constraints to live range constraints prior to register assignment. Branch hazards which remain after pseudo-level and machine-level transformations are resolved at the post-pass level with read insertions as described in Section 3.1.2.

The primary pseudo-level renaming transformation for the removal of branch hazards, involves node splitting [4]. This section presents a new one-pass node splitting algorithm which results in marginally reduced code growths and dramatically reduced compile-times relative to previous node splitting algorithms.

3.2.1 Iterative node splitting a'gorithm

Node splitting breaks equivalence relationships which would prevent pseudo register renaming [3, 15]. When two definitions of a hazard variable reach a node in which the hazard variable is live, the node is split. Node splitting to resolve one hazard variable often resolves other unrelated hazard variables. This implies that the hazard set should be recalculated after splitting is performed for each hazard variable. Previous node splitting algorithms use this iterative algorithm to avoid unnecessary node splitting [3].

Figure 4 demonstrates the effect of the iterative node splitting algorithm on an example subgraph. Node splitting relative to hazard variable x ensures that the definition of x in node n_1 and the definition of x in node n_2 do not both reach the same use of x in node n_5. Node splitting relative to y ensures that the definition of y in node n_3 and the definition of y in node n_4 do not both reach the same use of y in node n_6. Figure 4 also shows an optimal subgraph which resolves both ..zards with less splitting than produced by the iterative algorithm, indicating that excessive node splitting is possible with the iterative algorithm.

3.2.2 Node splitting using graph coloring

To ensure minimal splitting, a new node splitting algorithm is developed using the concept of conflicting parents [17]. Ensuring that node n does not have conflicting parents enables resolution of the hazard using variable renaming. The node splitting strategy for a particular node is to group the parents of that node such that elements within a group do not conflict. Each group becomes parent nodes for a duplicate of the original node. For example, if node n has six parent nodes and these nodes can be organized into three nonconflicting groups, then only three total copies of n are required.

Figure 5 illustrates the use of conflicting parents and graph coloring in node splitting for the QSORT application described in Table 3 of Section 4.1. Node splitting is performed on pseudo-level code, which for this example is represented by *Lcode* from the IMPACT C compiler [18]. Figure 5 shows node 48 from the QSORT application. Node 48 has six parent nodes prior to splitting. These nodes can be arranged in a parent conflict graph, where each arc of the graph represents two nodes which conflict. Establishing groups can be achieved by finding the minimum coloring of the parent conflict graph, i.e., coloring the nodes such that no two nodes connected by an arc

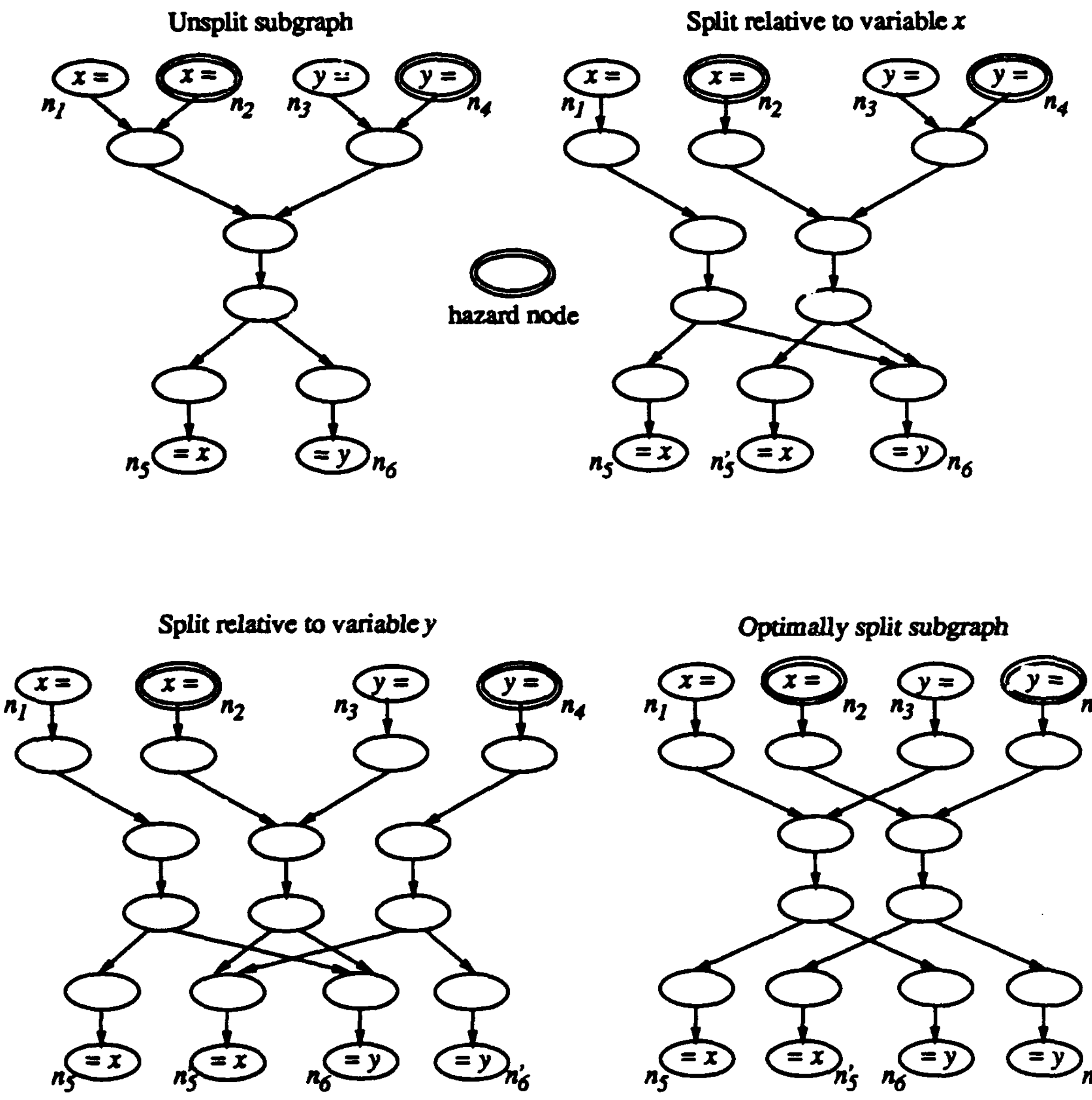


Figure 4: Iterative node splitting relative to hazard variables x and y.

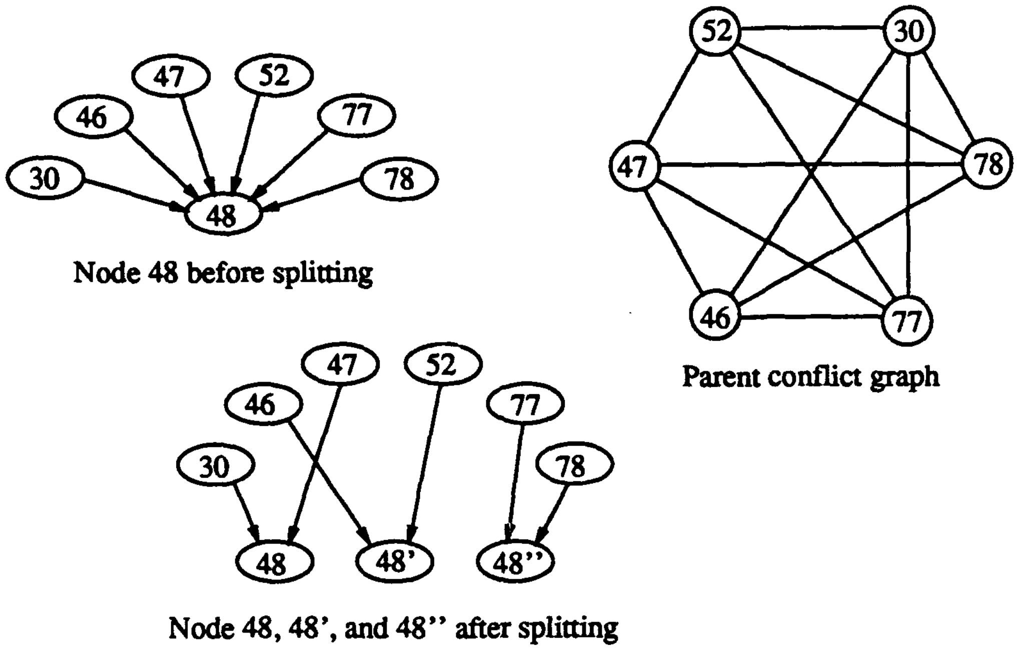


Figure 5: Node splitting using graph coloring; QSORT.

have the same color. For the example shown in Figure 5, three colors are sufficient to color the parent conflict graph, resulting in the splitting of node 48 into nodes 48, 48' and 48". Determining whether a graph is k-colorable is NP-complete in general. The graph coloring heuristic used for our one-pass node splitting algorithm is a modified version of an algorithm used for register allocation [15].

3.2.3 One-pass node splitting algorithm

Both *live_in(n)* and *reaching_out(n)*[5] analyses are required to identify conflicting parent nodes. A one-pass node splitting algorithm becomes possible by precalculating *live_in* and the hazard node set, and then, beginning with the root node, splitting in a topological traversal of the CFG. A topological traversal ensures than when processing node n, all ancestors of n have been processed and no descendants of n have been processed. This latter case ensures that the presplit calculation of *live_in(n)* can be used for parent conflict identification when processing a given node. Unlike *live_in(n)*, *reaching_out(n)* is affected by the splitting of ancestor nodes. Since *reaching_out(n)*

[5]A complete description of data-flow terminology can be found in *"Compilers: Principles, Techniques, and Tools"*, Aho et al., [15].

Table 2: Node splitting algorithm comparisons: COMPRESS.

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

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

is based solely on node n and its ancestors, *reaching_out(n)* can be calculated as node splitting proceeds. If a hazard node is split, each duplicate of the node must be added to the hazard node set. Since the root node does not have conflicting parents, a topological traversal of the CFG using the graph coloring node splitting technique ensures that no node in the resulting graph has conflicting parents.

Table 2 illustrates the improvement of the one-pass node splitting algorithm over the iterative algorithm for the COMPRESS application described in Table 3 of Section 4.1. The COMPRESS application was compiled on a SPARCserver 490 using the IMPACT C compiler [18] with a rollback distance of 10. Node count values represent pseudo instructions (Lcode) created by the IMPACT C compiler before and after splitting. Seven of the 14 COMPRESS functions which required splitting are listed. Algorithm run times represent the overall compile times given each of the two node splitting algorithms.

Table 2 shows a marginal overall code growth reduction for the one-pass algorithm. Although one function demonstrated a significant code growth reduction (6.7% down from 75.6%), the function is small and has minimal effect on the overall code size. The improvement in compile-time of the one-pass algorithm is more dramatic, resulting in a speedup of 30.2. The compile-time improvement can be explained as follows. If 60 hazard variables are present in a given function, the iterative algorithm may require up to 60 passes through the CFG of that function, including

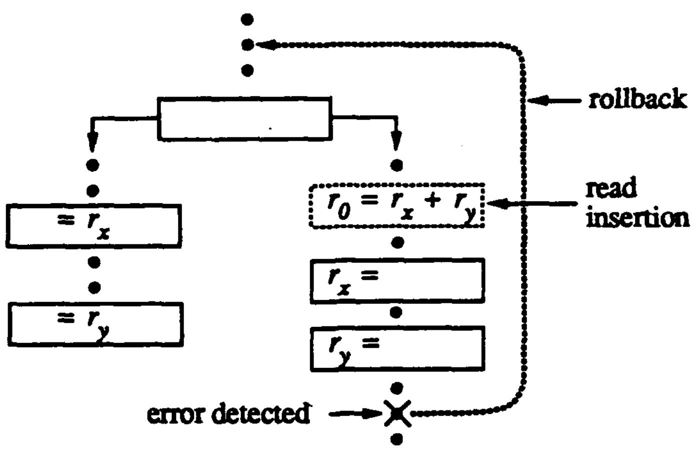


Figure 6: Post-pass hazard removal using read insertion.

60 data-flow analysis and hazard calculations. Although processing a given node in the one-pass algorithm is slightly more complex, a single data-flow analysis calculation and a single pass through the CFG are sufficient.

3.3 Performance Enhancement Through Profiling

3.3.1 Post-pass transformation versus loop protection

After hazards are removed by the compiler, some hazards remain and must be removed using a post-pass transformation. Previous post-pass transformations used nop insertions to increase all antidependency distances to $> N$ [3]. Since nop insertion can be costly to performance, previous compiler transformations removed all hazards possible, leaving only unresolvable hazards to be removed by the post-pass transformation.

In Section 3.1.2, a new post-pass transformation was introduced in which nop insertion was replaced by read insertions as the primary hazard removal technique. As illustrated in Figure 6, up to two branch hazards can be removed by a single read instruction. The new post-pass transformation is very efficient and in some cases can resolve branch hazards with less performance impact than pseudo-level transformations. Figures 11 and 13 of Section 4.2 show performance overhead comparisons between compiler-driven data-flow manipulations and the post-pass transformation for the PUZZLE and TBL applications described in Table 3 of Section 4.1. *Comp/PP* indicates that hazards are resolved by the compiler where possible, with the remaining hazards being resolved at the post-pass level. *PP* (post-pass) indicates that compiler transformations have been disabled and that all hazards are removed at the post-pass phase.

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

Figure 7 illustrates the potential effect on performance given the following two types of hazard removal: 1) hazard removal using register renaming that results in loop protection, and 2) hazard removal using read insertion. If the protected loop of Figure 7 is executed 20 times and the hazard instruction is executed two times, loop protection would require the execution of 40 additional instructions, where read insertion would require the execution of only two additional instructions. If the loop and hazard instruction execution frequencies were reversed, then read insertion would produce more performance impact than loop protection. As shown in Figure 7, profiling data can be used to aid in loop protection decisions.

3.3.2 Profiling effectiveness

Profiled data was included in the pseudo-level transformations of Section 3.2. The profile data is comprised of both dynamic profile sampling and static prediction. The static prediction is used as a supplement for areas of the application code that are unexecuted during profile sampling. For static profiling, a loop is assumed to iterate ten times. Inner loops, therefore, iterate multiples of 10 times depending on the depth of loop nesting. All loop header nodes and hazard nodes are assigned weights based on the profile data.

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

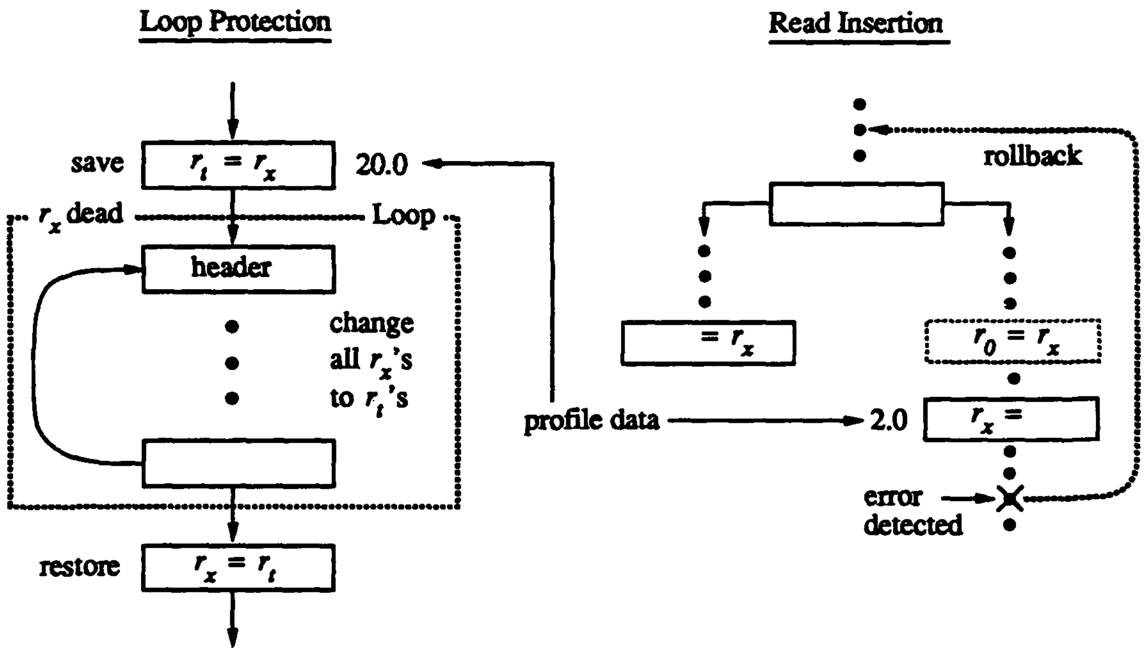


Figure 7: Loop protection versus read insertion.

from the save/restore instruction pair shown in Figure 7. Indirect loop protection costs result from: 1) an increased number of hazards which in turn required more node splitting and more loop protection, and 2) increased register usage due to the save/restore instructions which can result in additional register spills. Figure 8 shows the run-time overhead for the TBL application with rollback distances from 1 to 10. *Prof/PP* indicates that profiling data was used in loop protection decisions.

The results show that the use of profile data can improve application performance by postponing some hazard resolutions until the post-pass phase. Using profile data to aid in loop protection decisions did not produce performance equal to that for the post-pass transformation, for the TBL application. As an extension to this work, profile data can be used to aid in register allocation. As discussed in Section 3.2, hazards that are present after pseudo register renaming are resolved by adding hazard constraints to live range constraints prior to register allocation. These additional constraints can cause increased register spillage and impact performance. Similar techniques to those developed for loop protection can be used to enhance register allocation decisions.

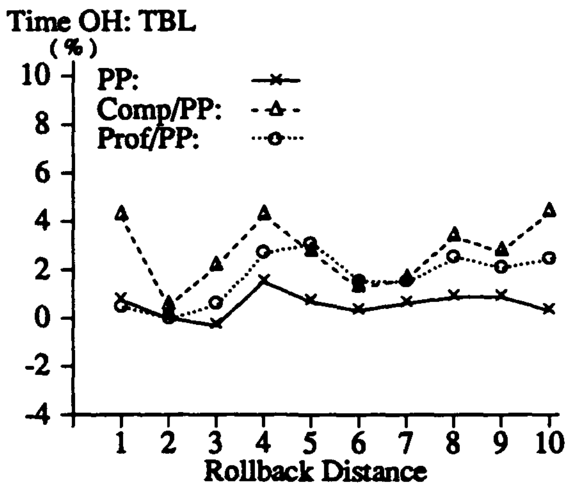


Figure 8: TBL: profile data used for loop protection decisions.

4 Performance Evaluation

4.1 Implementation and Application Programs

The hazard removal transformation algorithms have been implemented in the MIPS code generator of the IMPACT C compiler [18]. Transformations resolving pseudo register hazards (loop protection, node splitting, and loop expansion) are called just before register allocation. Transformations resolving machine register hazards are called after the live range constraints have been generated and before physical register allocation. The nop insertion algorithm, or post-pass algorithm, is called before the assembly code output routine.

Table 3 lists the eleven application programs used in the evaluations. The applications were cross-compiled on a SPARCserver 490 and then the compiled program was run on a DECstation 3100. *Static Size* is the number of assembly instructions emitted by the code generator, not including the library routines and other fixed overhead.

The results are summarized in Figures 9 through 13. Each figure contains two plots, the first plot shows the percent of run-time overhead (*Time OH*) of the referenced hazard resolution scheme, and the second plot shows the percent of code growth overhead (*Size OH*) relative to the base values in Table 3.

Four hazard resolution techniques were evaluated. *Compiler 1* resolves on-path hazards only, using the compiler-driven data-flow manipulations. *Compiler 2* extends the compiler transformations

Table 3: Application programs.

Program	Static Size	Description
QUEEN	148	eight-queen program
WC	181	UNIX utility
QSORT	252	quick sort algorithm
CMP	262	UNIX utility
GREP	907	UNIX utility
PUZZLE	932	simple game
COMPRESS	1826	UNIX utility
LEX	6856	lexical analyzer
YACC	8099	parser-generator
TBL	8197	table formatting preprocessor
CCCP	8775	preprocessor for gnu C compiler

to resolve both on-path and branch hazards. *PP* (post-pass) disables the compiler transformations and relies solely on the post-pass transformation presented in Section 3.1.2. *Comp/PP* uses compiler transformations to resolve branch hazards with the techniques described in Section 3.2, assumes a read buffer to resolve on-path hazards, and uses the post-pass transformation to remove remaining branch hazards. *Comp/PP* represents the compiler-assisted multiple instruction rollback scheme.

Due to the excessive compile times of the previous *Compiler 1* and *Compiler 2* algorithms for large applications, the evaluations of these schemes were restricted to applications QUEEN, WC, COMPRESS, CMP, PUZZLE, and QSORT. Both *Comp/PP* and *PP* were evaluated for all eleven applications.

4.2 Performance analysis

Compiler transformations used for the removal of data hazards can impact performance in several ways. Loop protection inserts save/restore operations at the head and tail of the loop. This increases the path length and, therefore, the run time. Additional arcs in the dependency graph can cause more spill code to be generated, increasing memory references and cache misses. Nop insertion can be costly since up to N nops could be inserted for each unresolved hazard. The insertion of MOV r_k, r_k instructions to create covering on-path hazards in the post-pass transformation also increases path lengths, although typically less than with nop insertions. Finally, the increase in code size, mainly due to loop expansion, may cause more run-time cache misses. The performance numbers shown in Figures 9 through 13 are for execution of the eleven application programs on a DECstation 3100 after they have been compiled with the transforms described.

4.3 Results: *Compiler 2*

As can be seen in Figures 9 through 11, extending the compiler hazard resolution scheme to include branch hazards introduces little incremental performance impact or code growth overhead. Given a rollback distance of 10, resolving both on-path and branch hazards using compiler transformations resulted in a maximum performance impact of 32.6% and an average performance impact of 12.6%. This compares with maximum and average impacts of 35.4% and 15.4%, respectively, for compiler-driven on-path hazard resolution only. The maximum code size overhead measured for the extended compiler-based technique was 328% with an average overhead of 207%, for a rollback distance of 10. This compares with a maximum and average overhead of 372% and 225%, respectively, for the unextended compiler-based scheme.

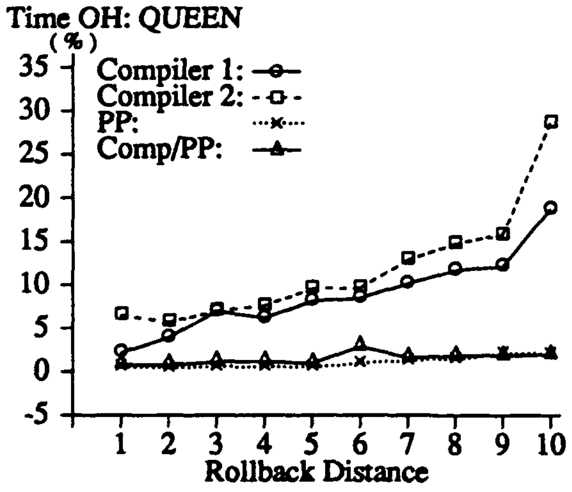
These results indicate a small incremental run-time performance overhead and a small code size overhead given compiler-based branch hazard removal compared to compiler-based on-path hazard removal alone. Three factors account for these small incremental impacts. First, on-path hazards dominate in frequency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

4.4 Results: *PP*

Figures 9 through 13 show the run-time and code size overheads for each application studied using the read buffer to resolve on-path hazards and the post-pass transformation described in Section 3 to cover all branch hazards. The results are worst case in that many of the branch hazards could have been resolved with no performance impact using the compiler techniques; instead, they are resolved by the insertion of MOV instructions which cause a guaranteed, although small, performance impact. Given a rollback distance of 10, the post-pass transformation produced a maximum performance impact of 7.69% with an average performance impact of 2.43%, significantly below the levels produced by the compiler-based scheme. Code growth overhead measurements were correspondingly lower with a maximum overhead of 13.0% and an average overhead of 8.59%.

4.5 Results: *Comp/PP*

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



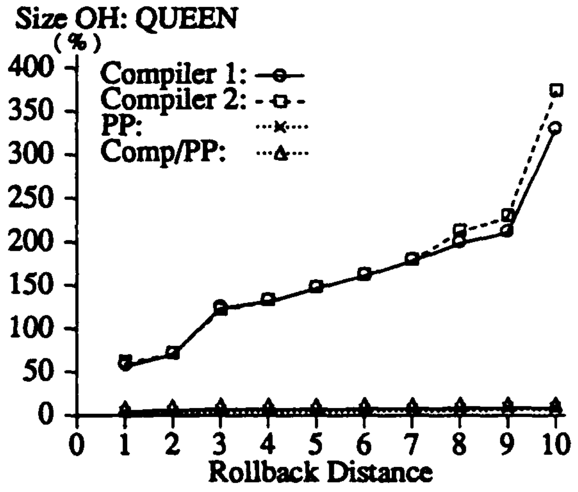


Figure 9: Run-time overhead and code size overhead: QUEEN.

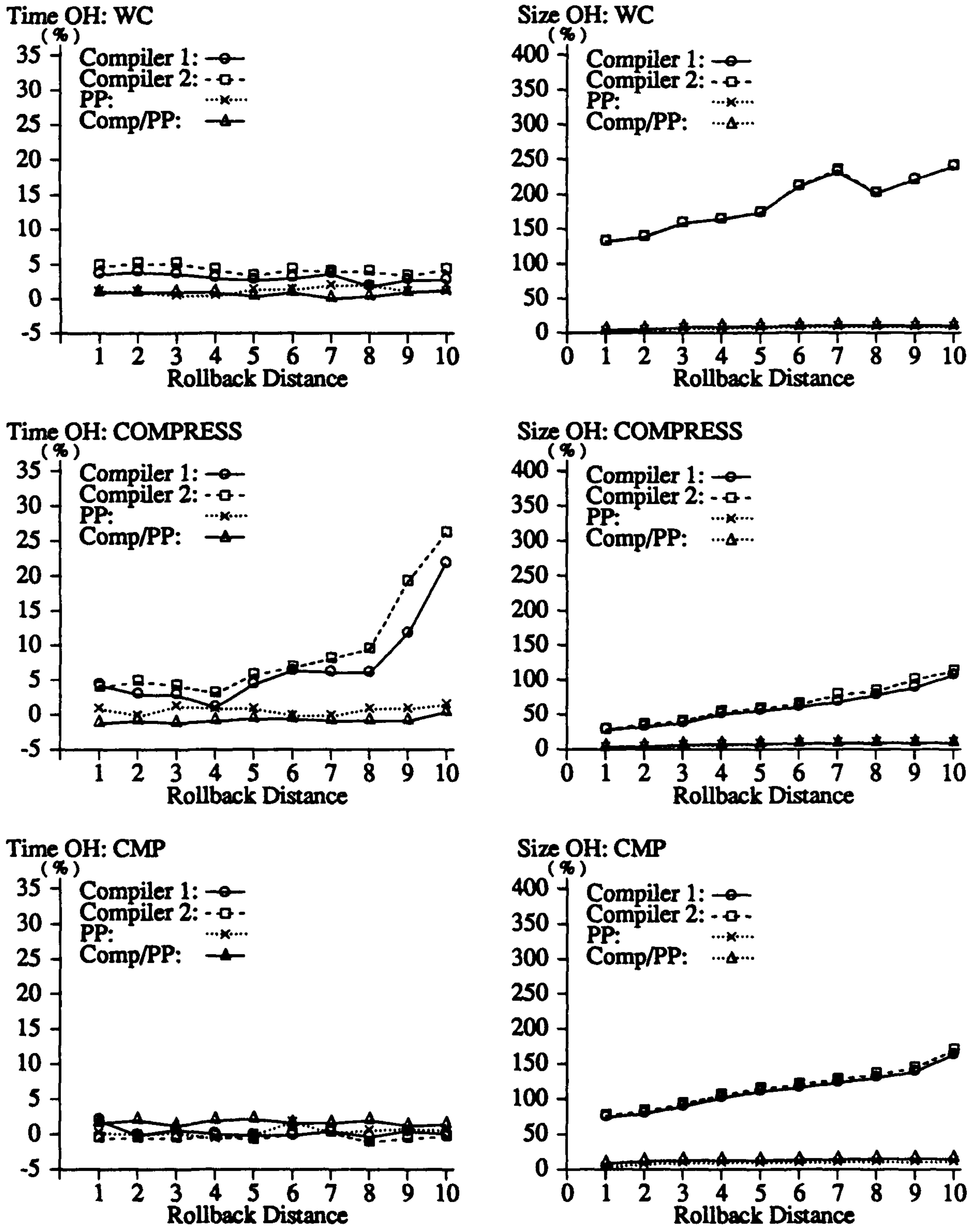


Figure 10: Run-time overhead and code size overhead: WC, COMPRESS, and CMP.

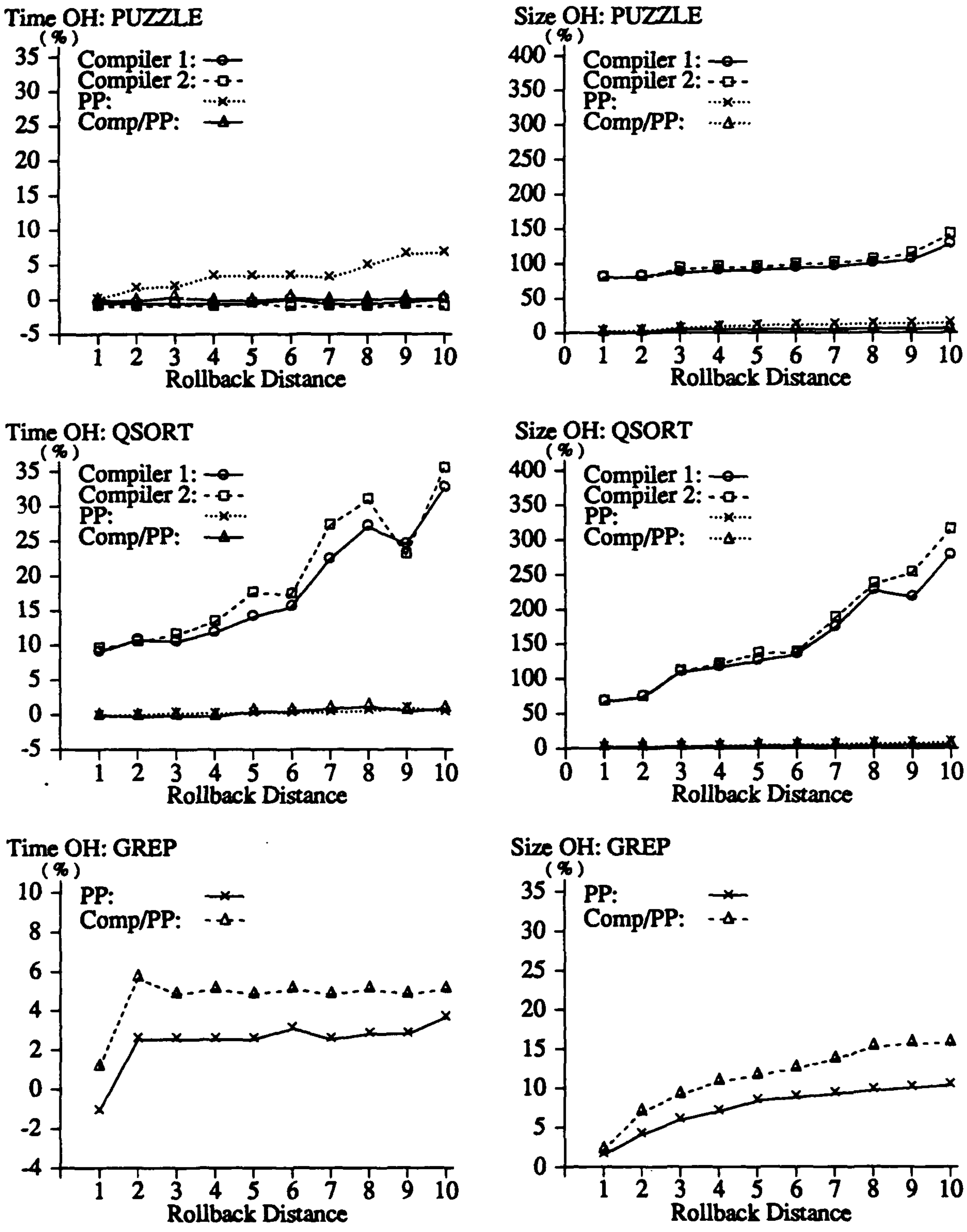


Figure 11: Run-time overhead and code size overhead: PUZZLE, QSORT, and GREP.

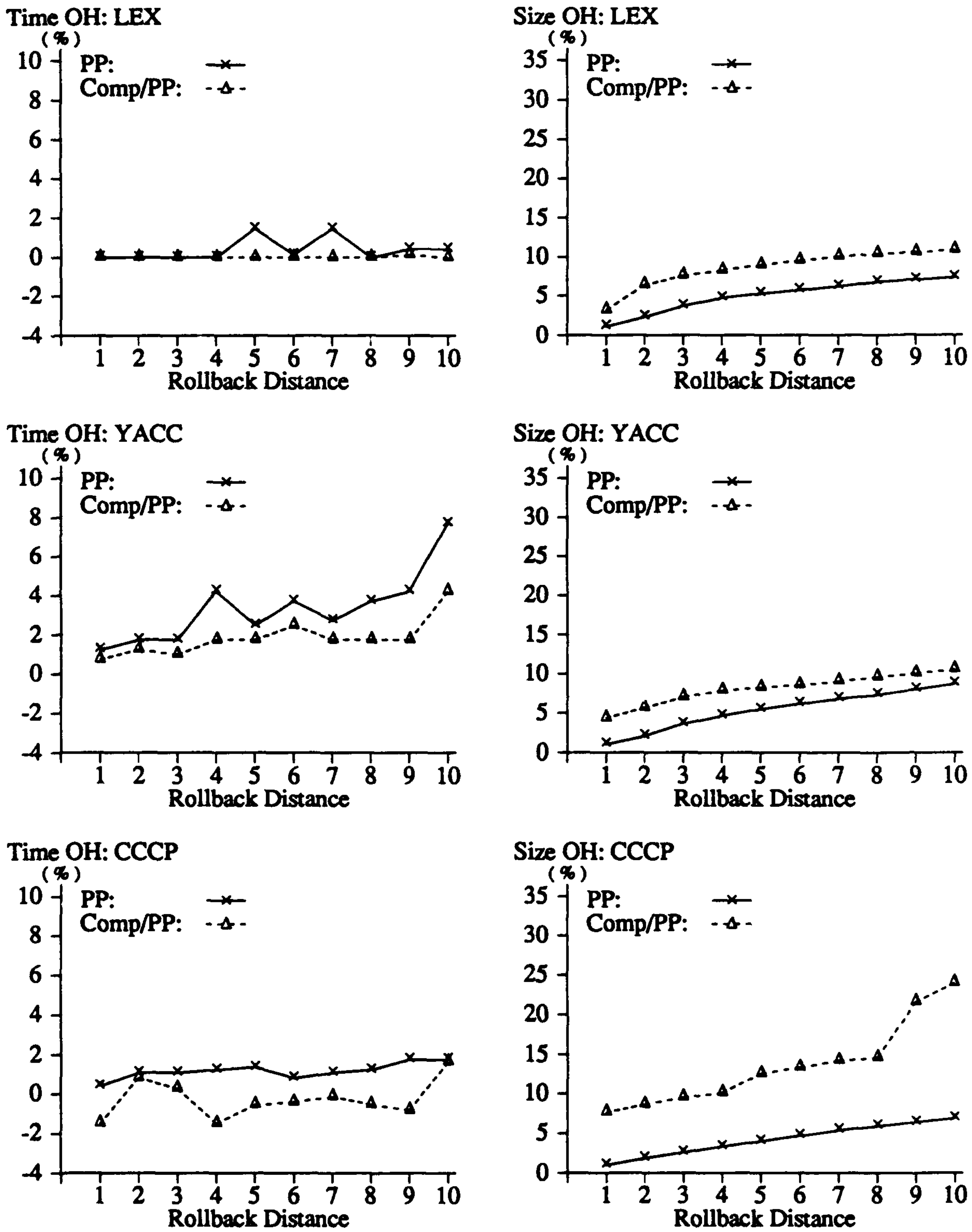


Figure 12: Run-time overhead and code size overhead: LEX, YACC, and CCCP.

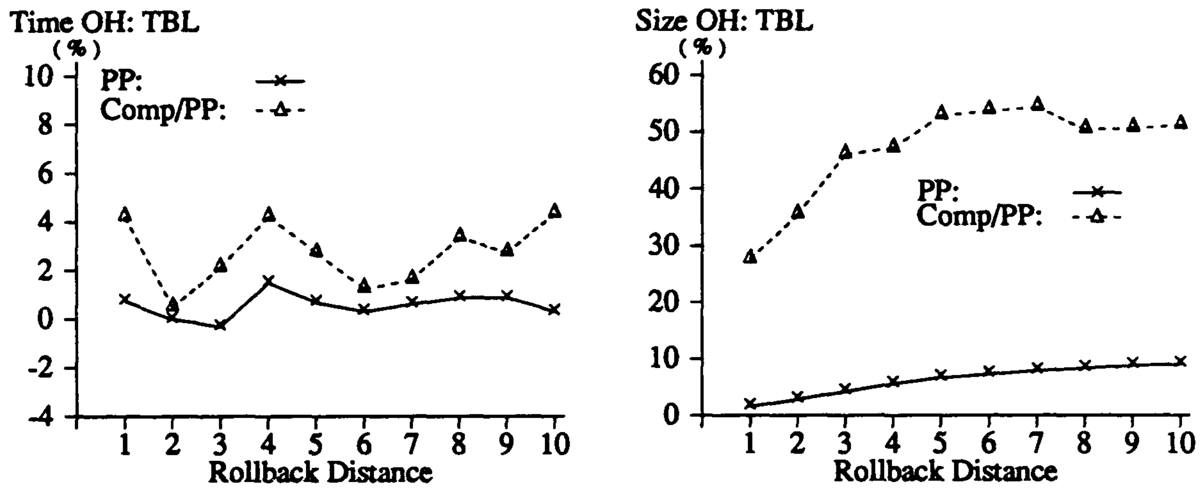


Figure 13: Run-time overhead and code size overhead: TBL.

5 Read Buffer Size Requirement

A practical lower bound and average size requirement for the read buffer are established in this section by modifying the design to save only the data required for rollback. The study measures the effect on the performance of ten application programs using six read buffer configurations with varying read buffer sizes. Two alternative configurations are shown to be the most efficient.

Given a read buffer, rollback is accomplished by first flushing the read buffer back to the general purpose register GPRF in the reverse order of which the values were saved. Provided that the depth of the dual first-in-first-out (FIFO) read buffers are N, redundant copies of the appropriate register values are available to restore the register file given a rollback of $\leq N$.

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

Figure 14 illustrates a case in which all register reads do not have to be placed in the read buffer. The register values (denoted $value(r_x)$) which require saving are marked with an "*." Since

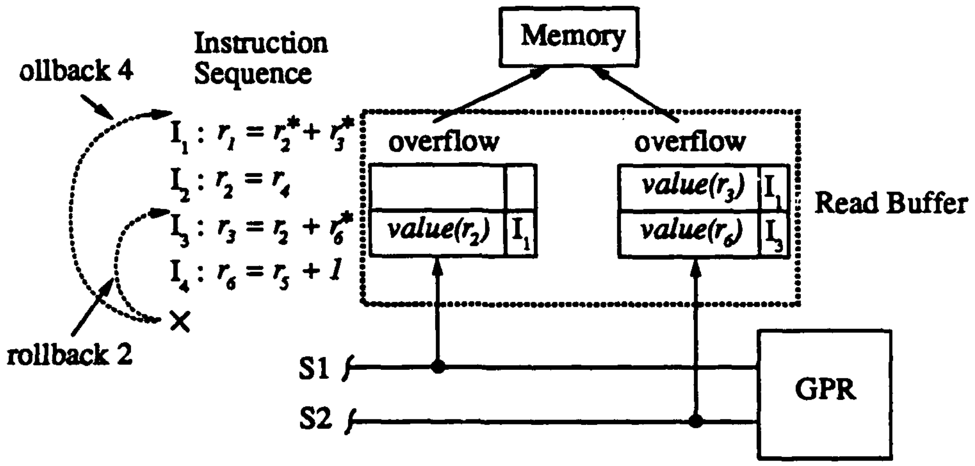


Figure 14: Read buffer of size $< 2N$.

only the required values are saved, the read buffer total size can now potentially be less than N. In this case, however, the instruction count must also be saved so that the value can be maintained for at least N cycles. In the event that the read buffer overflows, the oldest value in the buffer must be pushed to memory and a record kept so that during rollback the value can be retrieved from memory. Given a dual FIFO depth of M, memory would serve the function of the remaining $N - M$ of the two FIFOs.

5.1 Read Buffer Designs and Evaluation Methodology

Six read buffer configurations were studied. Configuration A1, shown in Figure 15, has a separate FIFO for each source bus. Configuration A2 allows access to either FIFO from either source bus. Configuration B1 contains a single FIFO and assumes that both source operands can be written into the single FIFO within the same cycle. This latter *split-cycle-save* assumption is consistent with a register file design that writes during the first half of the cycle and reads during the second half of the cycle [19]. Configuration B2 assumes no split-cycle-save capability. Configuration C contains a single level dual queue to absorb a simultaneous operand save and configuration D extends this design to allow access to either queue from either source bus.

The read buffer was simulated at the instruction level. The s-code emitted by the IMPACT C compiler [18] was instrumented with procedure calls to a simulation program containing models for the six read buffer configurations. Branch hazards were removed by the compiler for a rollback

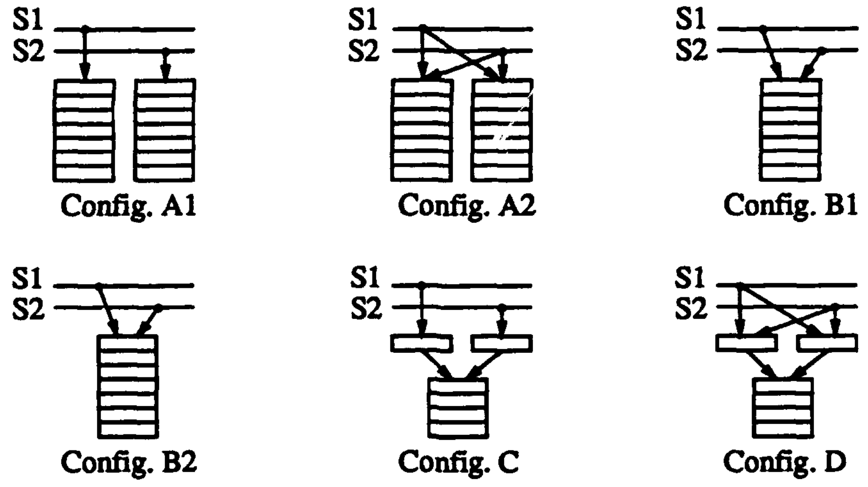


Figure 15: Read buffer configurations.

distance of 10. Parameters such as which operands require saving in the read buffer were determined at the post-pass level and instrumentation code segments were adjusted to pass this information to the simulation program. Table 3 lists the ten[6] application programs used in the evaluations. The applications were cross-compiled on a SPARCserver 490 and run on a DECstation 3100 with read buffer sizes ranging from 0 to 20 (note that 20 represents the maximum read buffer size of $2N$).

5.2 Evaluation Results

5.2.1 Detailed analysis: QUEEN

Figure 16 shows changes in performance overhead (Cycles OH) for various read buffer sizes and configurations running the QUEEN application. Looking at Figure 16, configuration A1, it can be seen that significant performance impact is incurred even with a modest reduction in read buffer size. Configuration A1 was consistently the least efficient of the six configurations across the ten applications studied.[7] This is due to the fact that the dual FIFO's are dedicated to a single source bus. In many cases saving S1 will cause an overflow because the S1 FIFO is full, even though there is room in the S2 FIFO. Configuration A1 does allow for simultaneous saves of S1 and S2, given sufficient room in each, but this feature does not compensate for the latter inefficiency.

[6]The TBL application was not included in the read buffer size evaluation.

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

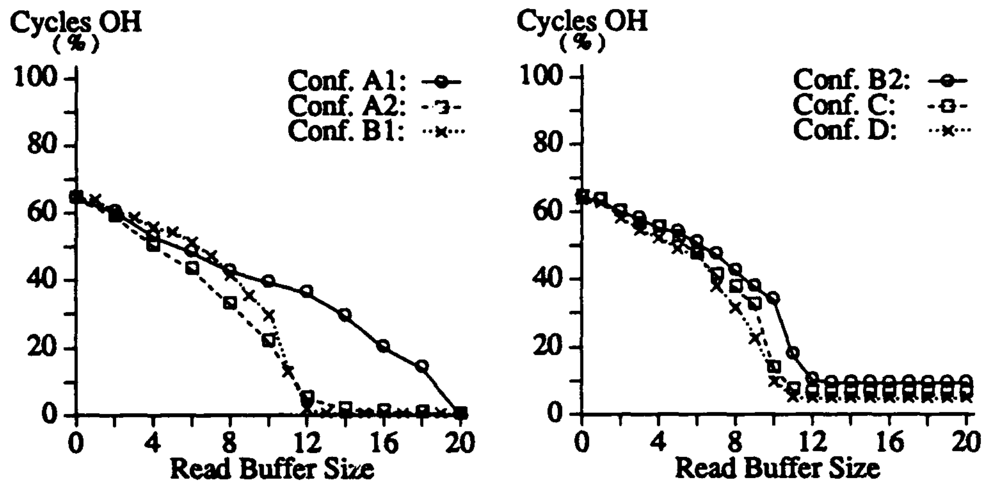


Figure 16: Cycle overhead: QUEEN.

Configuration A2 demonstrates the improvement gained by allowing either source bus access to either FIFO. Configuration B1 was the most efficient of the six configurations for the QUEEN application. In this configuration a total read buffer size of 13 would produce zero performance impact with a 35% reduction in read buffer size.

It should be noted that configuration B1 assumes that simultaneous saves of S1 and S2 can be handled within the same cycle. If this latter assumption is invalid, Figure 16, configuration B2, shows that no less than 9.4% performance impact is achieved regardless of the read buffer size. The "leveling off" of B2 is due to the bottleneck at the single FIFO entry point and not the depth of the FIFO. The flat part of the curve shows the percent of instructions requiring simultaneous saves of S1 and S2 in the QUEEN application.

Figure 16, configuration C, shows how a single level dual queue placed between the source bus and the single FIFO can alleviate some of the bottleneck effects. The dual queue can absorb a single simultaneous save of S1 and S2, distributing the saves over multiple cycles. A nonzero minimum performance overhead is still present due to cases in which the dual queue has not emptied before the next simultaneous save occurs.

Figure 16, configuration D, shows the results of an improved queue structure which permits saves from either bus into either queue. This configuration avoids stalls in some cases (e.g., S2 must be saved while the queue dedicated to S2 in configuration C is full and the other queue is empty). Configuration D also has a nonzero minimum performance overhead but gives better

Table 4: Read buffer size evaluation summary.

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

performance than configuration C.

The simulation results for QUEEN show that configuration A1 is the least efficient and that given the ability to do split-cycle-saves, configuration B1 is the most efficient. Without the split-cycle-save capability, configuration D is the best of the single FIFO designs resulting in a minimum performance overhead of 4.5%, and configuration A2 is the best of the dual FIFO designs resulting in a 1.7% performance overhead with a read buffer size of 14. For configurations B1, B2, C, and D, a total read buffer size of 13 is sufficient to maximize performance.[8]

5.2.2 Evaluation of all application programs

Results for the other nine application programs are similar to those for QUEEN [17]. The differences between the application results are the points at which the curve "levels off" (i.e., the buffer size) and, in the case of configurations B2 through D, at what level the performance overhead stabilizes. Table 4 summarizes measurements obtained for the ten applications given the two most efficient configurations, A2 and B1. It is assumed for this study that minimal performance overhead can be tolerated as a result of read buffer size reduction. For this reason, configuration comparisons are made at read buffer size values which produce low values of performance overhead. Configuration A2 does not level off like configuration D and does not rapidly approach zero like configuration

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

B1. For a better comparison of configurations A2 and B1, Table 4 gives the read buffer size value where the performance overhead value drops below 3%. The read buffer size value is referred to as *RB_size* and the performance overhead value is referred to as *OH_level.*

It can be seen from Table 4 that the read buffer size requirement is roughly the same, per application, regardless of the split-cycle-save assumption (i.e., comparing configurations A2 and B1). The size requirement is application dependent - from 8 for WC, to 15 for QSORT and YACC. The measurements show that a considerable reduction in read buffer size is achievable. Given the split-cycle-save assumption and configuration B1, a minimum of 25%, a maximum of 60%, and an average of 42% reduction was achieved. For configuration A2 and no split-cycle-save assumption, a minimum of 20%, a maximum of 50%, and an average of 38.0% reduction was achieved. The measurements indicate that care should be taken relative to the ultimate selection of read buffer size. Given the steepness of the B1 curve around the *RB_size* value, small decreases in size can produce large performance overheads.

5.2.3 Read buffer size requirement summary

Results show that two read buffer configurations were the most efficient. A dual FIFO with source bus access to each (configuration A2) and the single FIFO with the split-cycle-save capability (configuration B1) consistently out-performed the other four configurations. There were moderate variances between the buffer sizes required for minimum performance impact between the ten applications studied and the performance stabilization value assuming no split-cycle-save capability. Up to a 55% read buffer size reduction was achieved with an average reduction of 39.5% given the most efficient read buffer configuration for the applications. It was also found that given the split-cycle-save assumption and single FIFO configuration, significant changes in the performance overhead result from small changes in the read buffer size. Our results indicate that care should be taken in the final selection of read buffer size in any given design.

6 Concluding Remarks

This paper has presented a compiler-assisted multiple instruction rollback scheme which combines compiler-driven data-flow manipulations with dedicated data redundancy hardware to remove data hazards that result from multiple instruction rollback. Experimental evaluation of the proposed compiler-assisted scheme with a maximum rollback distance of ten showed performance impacts of no more than 6.57% and an average impact of 1.80%, over the eleven application programs studied. The performance evaluation indicates lower performance penalties than for previous compiler-only approaches or comparable hardware-only approaches. Six read buffer configurations were studied to determine the minimum size requirement for general applications. It was found that a 55% read buffer size reduction is achievable with an average reduction of 39.5%, but that additional control logic to handle read buffer overflows may limit the overall hardware savings.

Future research includes application of compiler-assisted multiple instruction rollback recovery to super-scalar, VLIW, and parallel processing architectures. Evaluations of compiler-assisted rollback recovery applied to speculative execution repair would include modifying compiler transformations to operate in a super-scalar and VLIW environment.

7 Acknowledgements

The authors wish to thank C.-C. Jim Li for his help with the compiler aspects of this paper, and Scott Mahlke and William Chen for their invaluable assistance with the IMPACT compiler. We also express our thanks to Janak Patel for his contributions to this research.

References

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

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

[3] 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.

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

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

[6] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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