BRANCH RECOVERY WITH COMPILER-ASSISTED MULTIPLE INSTRUCTION RETRY

N. J. Alewine*, S.-K. Chen, C.-C. Li†, W. K. Fuchs, W.-M. Hwu

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Abstract

In processing systems where rapid recovery from transient faults is important, schemes for multiple instruction rollback recovery may be appropriate. Multiple instruction retry has been implemented in hardware by researchers and also in mainframe computers.

This paper extends compiler-assisted instruction retry to a broad class of code execution failures [1]. Five benchmarks were used to measure the performance penalty of hazard resolution. Results indicate that the enhanced pure software approach can produce performance penalties consistent with existing hardware techniques. A combined compiler/hardware resolution strategy is also described and evaluated. Experimental results indicate a lower performance penalty than with either a totally hardware or totally software approach.

1 Introduction

Checkpointing is a well understood method for implementing rollback recovery when system errors occur [2–4]. In case of a detected fault, the system is rolled back to a previous checkpoint containing a consistent state of the system [5]. Full checkpointing may permit long error detection latency at the expense of long recovery times.

When transient processor errors occur, multiple instruction retry can be an effective alternative to full checkpointing and rollback recovery [6–8]. Multiple instruction retry within a sliding window of a few instructions [6], or re-execution of a few cycles [8], can be implemented in parallel with concurrent error detection for rapid recovery from transient processor errors.

The issues associated with instruction retry are similar to those with exception handling in out-of-order instruction execution. If an instruction is to write to a register and N is the maximum error (or exception) detection latency, two copies of the data must be maintained for N cycles. Hardware schemes such as reorder buffers, history buffers, future files [9], and micro-rollback [6] differ in where the updated and old values reside, circuit complexity, and rollback efficiency.

A compiler-assisted approach to implementing multiple instruction retry has recently been developed by the authors [1]. In this technique, a series of compiler transformations are used to eliminate anti-dependencies of length $\leq N$. Our work was inspired by the hardware-based micro-rollback design of Tamir and Tremblay [6]. Our software approach produces a performance impact consistent with hardware-based techniques [6] and has the added benefit of making N a compile-time parameter.

This paper extends compiler-assisted multiple instruction retry to include a broad class of code execution failures. The error model is expanded to allow any legal path in the control flow graph (CFG) thus allowing branch recovery. Similar compiler techniques to those we previously developed [1] are shown to be effective in resolving the hazards. Finally, a combined compiler/hardware scheme is introduced which reduces code growth, compilation time, and performance impact.

The remainder of the paper is organized as follows: Section 2 describes the error model and classifies hazards. Section 3 describes the compiler techniques for resolving the hazards. Section 4 introduces a simple hardware scheme to resolve some of the hazards. Section 5 presents performance results using the compiler-only scheme and also the hardware-assisted compiler scheme.

*Currently on Resident Study leave from IBM, Boca Raton Fl.

†Symbol Technologies Inc., Bohemia, NY.

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

0731-3071/92 $03.00 © 1992 IEEE

2 Error Model and Hazard Classification

The model of targeted errors is summarized as follows. First, the maximum error detection latency is N instructions. Second, memory and I/O have delayed write buffers and can rollback N cycles. Third, the states of the program counter are preserved by an external recording device or by shadow registers [6]. Finally, the CPU state can be restored by loading the correct contents of the register file and the program counter.

In addition to the above, any error which does not manifest itself as an illegal path in the CFG, is also allowed 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 as follows:

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

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 instruction i, denoted I_i, to I_j, where $I_i \in V$ and $I_j \in V$, then there is an edge $(I_i, I_j) \in E$.

The hazard set H of the error model is defined as the set of pseudo or machine registers whose values are inconsistent during different executions of the same instruction sequence due to retry.

Hazards can be of two types; those that appear as anti-dependencies [10] of length $\leq N$ in $G(V, E)$, and those that appear at branch boundaries. Figure 1 illustrates both types hazards. If an error occurring prior to instruction I_j is detected after instruction I_i and a rollback is attempted, an incorrect value may be contained in register x. The first type of hazard occurs if, after rollback, x is used in an instruction along the original path (e.g., I_j). The second type of hazard occurs if x is used in an instruction along a new path (e.g., I_k). This can happen if the error causing the rollback results in an incorrect branch decision

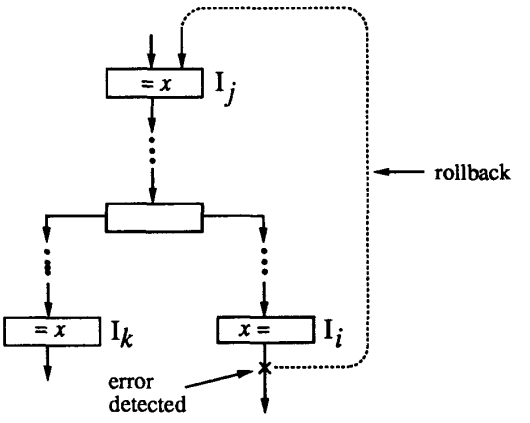

Figure 1: On-path and Branch Hazards.

during execution of the original instruction sequence. Hazards of the first type will be referred to as on-path hazards. Hazards of the second type will be referred to as branch hazards.

3 Compiler Based Hazard Resolution

Our previous techniques resolved on-path hazards in four phases [1]. Phase 1 resolved pseudo register hazards, phase 2 resolved machine register hazards, phase 3 resolved inter-procedural register hazards, and phase 4 used nop insertion to resolve the remaining hazards. This section describes compiler techniques for resolving branch hazards.

3.1 Pseudo Registers

The on-path hazard of Figure 1 can be resolved by renaming the definition register in I_i from x to y. Node splitting and loop expansion are used to break all data dependencies which require the *use* register in I_j to be renamed as a result of renaming x [1, 10]. Loop protection is used to maintain loop integrity during node splitting and loop expansion. Renaming is also effective in resolving branch hazards. The next step is to see how node splitting, loop expansion and loop protection apply to branch hazards.

Figures 2(a) and 2(b) show a typical data dependence (requiring node splitting) and the node splitting technique respectively. In Figure 2(a), renaming x in I_i to y will ultimately require the renaming of the *use* register x in I_j to y since multiple definitions of x reach I_k. To break this dependence, the following

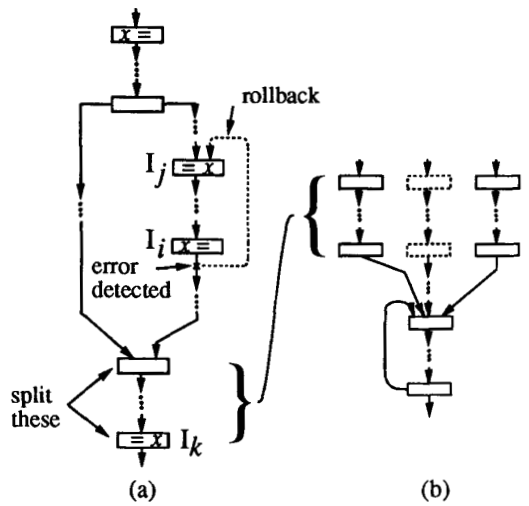

Figure 2: Node Splitting.

node splitting criterion is used: If multiple definitions of x reach I_k and x is in the live_in set of I_k, I_k will be split into two identical nodes. This "unzipping" is shown in Figure 2(b). Loop protection assures that no loop header is split [1].

Node splitting and loop protection operate on the data flow parameters of the CFG. Since these parameters are unaffected by the type of hazard considered, both techniques work equally well for branch hazards. This is not the case for the loop expansion transformation.

Loop expansion is used for resolving a hazard which traverses a loop back edge. We have experimentally observed a low rate of occurrence for branch hazards traversing loop back edges. We therefore resolve all such branch hazards in the nop insertion phase.

3.2 Machine Registers

Once hazards have been eliminated through renaming they can reappear as the physical registers are assigned. Figure 3 shows the elimination of on-path and branch hazards by adding arcs to the dependency graph used for register allocation.

3.3 Inter-Procedural Hazards

Inter-procedural register saving conventions can create immediate on-path hazards [1]. Branch hazards are not immediately created at procedural boundaries and therefore no additional action is taken.

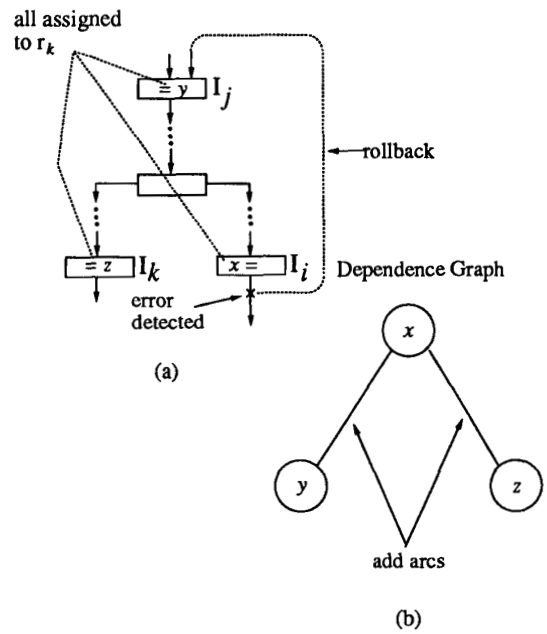

Figure 3: Machine Register Hazards.

3.4 Nop Insertion

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

4 Hardware Assisted Hazard Resolution

It was shown in Section 3 that both on-path and branch hazards can be resolved using compiler techniques. However, based on unique characteristics of the two hazard types, we can design an efficient combined compiler/hardware resolution technique.

The on-path hazard shown in Figure 1 is such that after rollback of N instructions, the original N instructions are re-executed leading back to I_i. Recovery in the presence of on-path hazards can be aided by maintaining a register *read* history of depth N in hardware.

Examining the branch hazard in Figure 1, it is seen that at the time x is defined in I_i it is not possible to determine dynamically whether it is a hazard. After retry, control flow may proceed down a path that uses x prior to redefinition (in this case it is a hazard) or it may proceed down a path where x is redefined before being used (in this case there is no hazard). In contrast to on-path hazards, branch hazards are a function of possible future paths (i.e., after rollback). A delayed write or history buffer could be used to resolve such hazards, however these schemes conservatively resolve on-path hazards as well. An alternative is to use compiler transformations to resolve branch hazards and hardware to assist in on-path hazard resolution.

4.1 Hardware Assist

Figure 4 shows a hardware scheme to resolve on-path hazards. In contrast to a *write* buffer [6] which

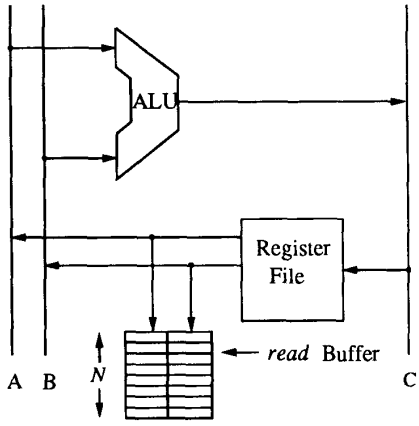

Figure 4: *read* Buffer.

is attached to the input port of the register file, 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 pushed into 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 popped and loaded back into the register file. For an on-path hazard, the path taken after the rollback will be the same as the path taken prior to rollback and each read of r_k will produce the same value as before. Branch hazards will be removed by the compiler. It is assumed that the *read* buffer is an integral part of the register file and any error in the system does not corrupt the transfer to the *read* buffer or its contents.

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

4.2 Combined Approach

On-path hazards were 3 to 4 times more frequent than branch hazards across the five benchmark programs evaluated. This would imply an improvement in performance when resolving only branch hazards using compiler techniques. The difficulty arises in determining which branch hazards to resolve. In addition to resolving all on-path hazards, the *read* buffer will resolve some branch hazards.

Figure 5 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 respectfully. 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. In this case, resolution of the branch hazard using compiler techniques is not necessary.

The current software calculates on-path hazards and total hazards, however it is not yet capable of accounting for *read* buffer resolution of branch hazards. Compiler resolution of (total hazards minus on-path hazards) would be overly optimistic and result in incorrect performance impact. In lieu of direct measurements, a conservative measure of the range of improvement that could be expected with the *read* buffer was obtained using a transformation on the unmodified (i.e., original) assembler level code.

The transformation creates on-path hazards when necessary to assure that all branch hazards are resolved by the *read* buffer. Given one such branch haz-

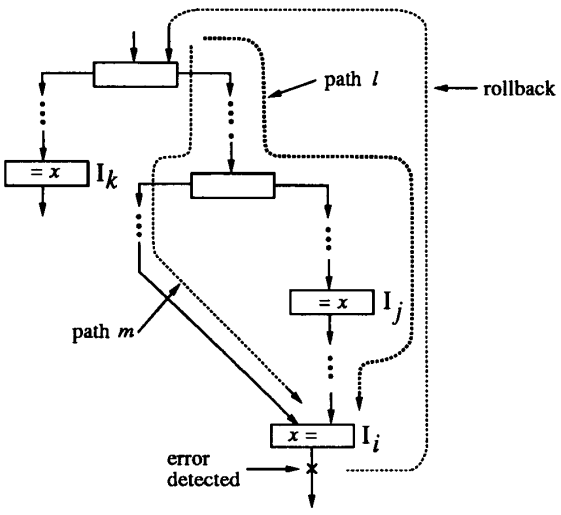

Figure 5: *read* Buffer Resolution of Branch Hazards

ard which defines physical register r_k at instruction I_i, the transformation inserts a 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 5. Section 5 includes experimental results using this transformation.

5 Performance Evaluation

5.1 Implementation

The transformation algorithms have been implemented in the MIPS code generator of the IMPACT C compiler [11]. 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 is called before the assembly code output routine.

5.2 Benchmarks

Five benchmark programs were cross-compiled on a SPARCserver 490 and run on a DECstation 3100. QUEEN is based on the eight-queen program but with 12 queens as input. QSORT implements the quick sort algorithm to process a randomly generated array. Both QUEEN and QSORT use recursive calls. WC and CMP are well-known UNIX utilities and PUZZLE is a simple game.

The results are summarized in Figures 6 through 15. Two groups of results are shown for each benchmark. The first group shows performance measured by run time overhead (**OH**), in seconds, on the DECstation 3100 and the second group by code size overhead, in number of assembly instructions emitted by the code generator, not including the library routines and other fixed overhead. **s/w: op** represents performance impact using software transformations to remove on-path hazards only and **s/w: op & br** shows performance impacts using similar transformations to remove both on-path and branch hazards.

5.3 Performance Analysis

The compiler transformations introduce performance impact in several ways. Loop protection inserts save/restore operations at the head and tail of the loop. This increases the path length and therefore increases 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. Finally, the increase in code size (mainly due to loop expansion) may cause more runtime cache misses.

The loop expansion transformation, however can improve performance over a compiler that does not have this optimization technique as demonstrated by the run-time results for CMP and PUZZLE [12]. Once the loop is expanded, some condition checks and index operations can be eliminated. Also the save/restore operations from loop protection shorten the live ranges of some registers thus allowing more efficient register allocation. Only the latter optimization is implemented in the software described in this paper.

5.3.1 Compiler-Only

It is interesting to note that there is negligible incremental performance impact introduced by resolving branch hazards in addition to on-path for the benchmarks evaluated. Two key factors account for this result. First, on-path hazards dominate in frequency of occurrence and second, resolving an on-path hazard at instruction I_i through renaming can sometimes resolve a branch hazard at instruction I_i. Additionally, resolving a similar on-path hazard with nop insertion may resolve a corresponding branch hazards by increasing the distance between it and its nearest predecessor branch.

5.3.2 Compiler/Hardware

Figures 6 through 15 also show the run time and code size overheads for each benchmark assuming the *read* buffer to resolve on-path hazards and the assembler level transformation described in Section 4. These measurements are labeled **h/w: op, s/w: br**.

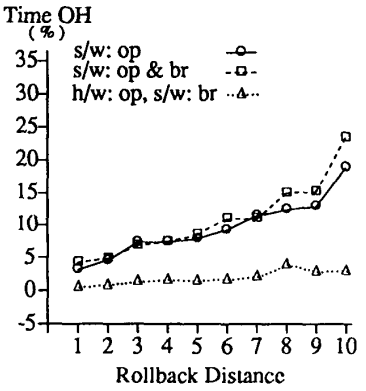

Figure 6: QUEEN, Runtime Overhead.

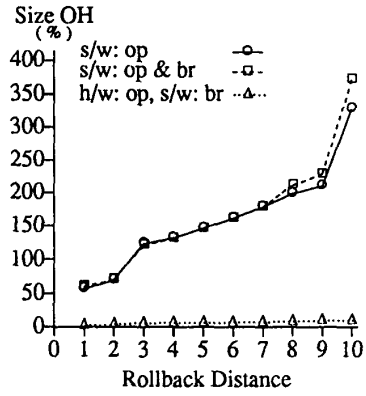

Figure 7: QUEEN, Program Size Overhead.

The results are worst case in the sense that many of the branch hazards could have been resolved with no performance impact using the compiler techniques of Section 3. Instead, they are resolved by insertion of MOV instructions which causes a guaranteed (although small) performance impact.

All benchmarks except one have less than 4% performance impact and all benchmarks have less than 14% code size increase. Given the *read* buffer feature and the option to use compiler techniques as well, all benchmarks are below 4% performance impact with an average impact of 1%. The run time results of

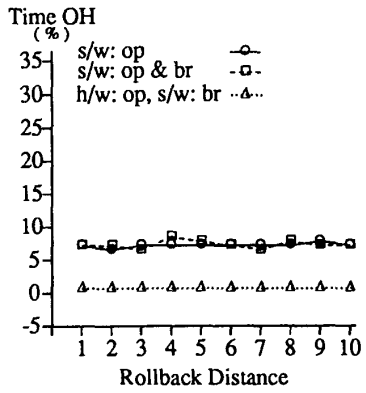

Figure 8: WC, Runtime Overhead.

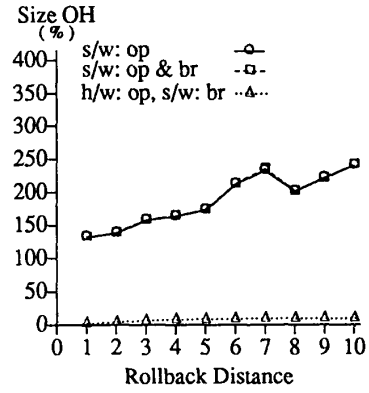

Figure 9: WC, Program Size Overhead.

PUZZLE indicate that compiler techniques are still useful in reducing performance penalties. These compiler techniques however, have the disadvantages of requiring recompilation, long compilation times, and significant code growth.

Current work is underway to modify the compiler transformations to allow branch hazard resolution only. All indications are that the performance impact, code growth and compilation time will be reduced below the current levels. Our experiments indicate that a combined compiler/hardware scheme for hazard resolution can produce lower performance penalties than either a compiler-only scheme or a hardware-only scheme. It is also noted that the code size is reduced relative to a compiler-only scheme and that hardware complexity is reduced relative to a delayed write scheme.

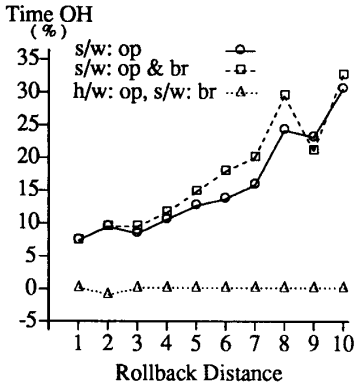

Figure 10: QSORT, Runtime Overhead.

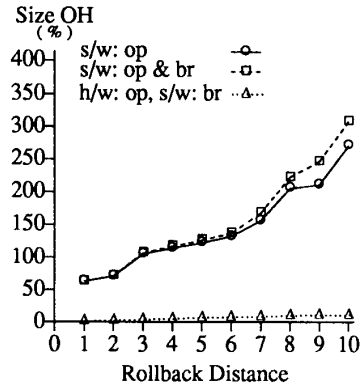

Figure 11: QSORT, Program Size Overhead.

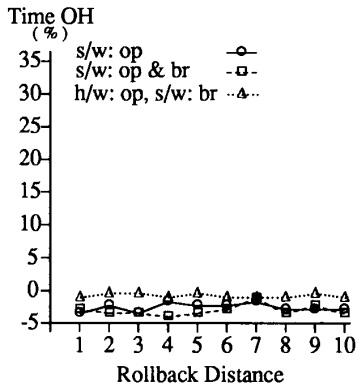

Figure 12: CMP, Runtime Overhead.

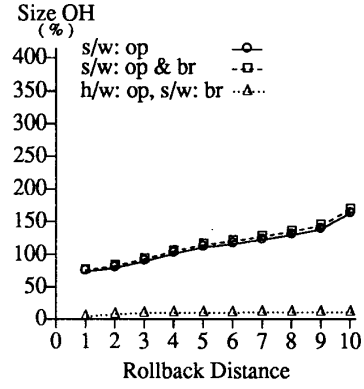

Figure 13: CMP, Program Size Overhead.

6 Concluding Remarks

Two schemes have been described to efficiently support multiple instruction rollback with branch recovery, a compiler-only scheme and a combined compiler/hardware scheme. Hazard classification has proved useful in construction of the combined scheme. Compiler transformations such as pseudo register renaming, node splitting, loop protection, and loop expansion were shown to be effective in resolving on-path and branch hazards with negligible performance impacts over resolving on-path hazards alone. The compiler-only approach yields performance impacts consistent with previous compiler techniques [1] and hardware techniques [6]. A hardware assisted scheme was introduced to resolve on-path hazards by maintaining a window of instruction *read* history.

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

7 Acknowledgements

The authors wish to thank 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.

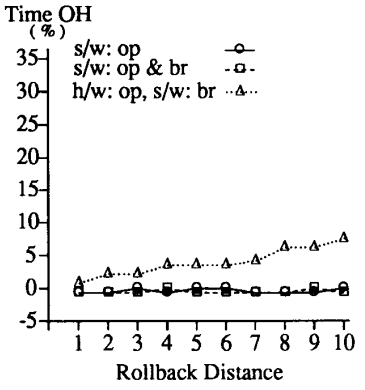

Figure 14: PUZZLE, Runtime Overhead.

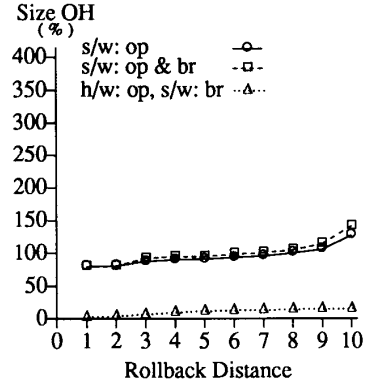

Figure 15: PUZZLE, Program Size Overhead.

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.

[2] L. Svobodova, "Resilient Distributed Computing," *IEEE Transactions on Software Engineering*, vol. SE-10, No. 3, pp. 257-267, May 1984.

[3] L. Lin and M. Ahamad, "Checkpointing and rollback-recovery in distributed object based systems," in *Proc. 20th International Symposium on Fault-Tolerant Computing*, pp. 97–104, 1990.

[4] K. Tsuruoka, A. Kaneko, and Y. Nishihara, "Dynamic Recovery Schemes for Distributed Processes," in *Proc. IEEE 2nd Symp. on Reliability in Distributed Software and Database Systems*, pp. 124–130, 1981.

[5] 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.

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

[7] 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.

[8] 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 International Symposium on Fault-Tolerant Computing*, pp. 178–185, June 1991.

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

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

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

[12] S. Weiss and J. E. Smith, "A Study of Scalar Compilation Techniques for Pipelined Supercomputers," in *Proc. 2nd International Conference on Architectural Support for Programming*, pp. 105–111, Oct. 1987.