AD-A281 648

The Application of Compiler-Assisted Multiple Instruction Retry To VLIW Architectures*

Shyh-Kwei Chen, W. Kent Fuchs, and Wen-Mei W. Hwu

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

Correspondent: Shyh-Kwei Chen
Tel: (217) 244-7180
FAX: (217) 244-5686
Email: skchen@crhc.uiuc.edu

94-22046

Abstract

Very Long Instruction Word (VLIW) architectures enhance performance by exploiting fine-grained instruction level parallelism. In this paper, we describe the development of two compiler assisted multiple instruction word retry schemes for VLIW architectures. In the first scheme, compiler generated hazard-free code with different degrees of rollback capability for uni-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 uni-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 results are shown for six benchmark programs.

Index terms: fault-tolerant parallel computing, instruction retry, compilers, VLIW architectures, instruction level parallelism.

---

94 7 14 016

DTIC QUALITY INSPECTED 5

1   Introduction

VLIW machines can simultaneously execute multiple instructions which are grouped as an instruction word [3, 4, 5]. 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 lockstep 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 [6, 7], superblock scheduling [8], and software pipelining [9] can effectively increase performance, especially for scientific applications where conditional branches are highly predictable.

Periodic checkpointing can recover from transient faults by rolling back the system to a previous checkpointed consistent state [10, 11, 12]. Checkpointing schemes allow long error detection latencies, but suffer from the high cost of long recovery time. In environments where error detection latency is only a few instructions [13, 14], instruction retry can be an effective scheme for providing fast recovery.

To preserve the status of the register files, and to allow the system to rollback a few instructions in the event of transient processor failures, hardware schemes [14], compiler-based schemes [1], and hardware-software combined schemes [2] have been developed. Write buffers can hold the *writes* to register files and delay the potentially contaminated data for a few cycles until the error detecting unit validates its correctness [14]. Read buffers save the
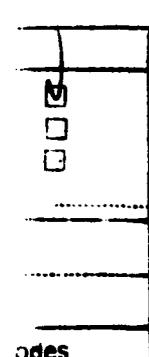
correct data for every read for a few instructions [2]. By compile time manipulation of data dependencies, such as the anti-dependencies [15], data required for rollback can be preserved in a register for a specific number of instructions [1].

The application of concurrent error detection, signature monitoring, and TMR to VLIW has recently been described by several researchers [16, 17, 18]. In this paper, we describe two compiler assisted multiple instruction word retry schemes for VLIW architectures. The first scheme is a compiler-based approach which applies the compiler-generated hazard-free code for various rollback distances for uni-processors to a modified trace scheduling algorithm [6]. 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 uni-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].

2 The Machine Model and Data Hazards

In this section, we describe the machine model, the assumptions, and the two hazard classes. 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.

2

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. We assume 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 [14], 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 [5].

Data hazards are those that cause inconsistencies during several retries of the same instruction word sequence. Two types of hazards are classified for the machine model. *On-path hazards* [1], appear in the form of anti-dependencies [15] 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* [20] at the other branch path.

3   Our Approach

3.1   Previous Results for Uni-processors

Register renaming is the main technique we previously utilized to resolve hazards for uni-processors. To eliminate on-path hazards, we remove all anti-dependencies of length $\leq N$.

Loop protection, *node splitting*, and *loop expansion* achieve this goal. We illustrate these techniques in Figure 1. Figure 1(a) is a partial program segment of QSORT, a recursive quick sort program. U(a) denotes reading register a, and a node represents an instruction. Suppose $N = 6$. Node 3 is a hazard node and register a is a hazard register since after executing node 3 any retry that rolls back above node 3 has a wrong value for register a due to the definition at node 3. We can not simply rename the definition at node 3 to resolve the hazards, since doing so will result in all references of pseudo register a to be renamed. Additionally, register a is used outside of the loop. We need to protect the loop in order to control the code growth due to later splitting, as shown in Figure 1(b), where the shaded nodes denote the save and restore nodes. Figure 1(c) shows the equivalent program segment after node splitting and renaming, while Figure 1(d) after expanding the loop twice and renaming. All the hazards within the loop are eliminated, and all anti-dependencies have distances greater than 6. Although there still are hazards for register a of distance 5, they belong to the outer loop and will be resolved since we always process inner loops first. Note that the new pseudo registers have very short live ranges, which is a benefit to our VLIW scheduling algorithm.

Figure 2 outlines the entire hazard removal procedure for uni-processors. The input code is generated by the IMPACT C compiler [19]. Hazards are resolved at three different stages, i.e., pseudo register, machine register, and nop insertions [1]. The machine register stage performs the register allocation. The pseudo register stage provides a large number of pseudo registers that can be assigned, as shown in Figure 1. This stage is the *pre-pass stage*
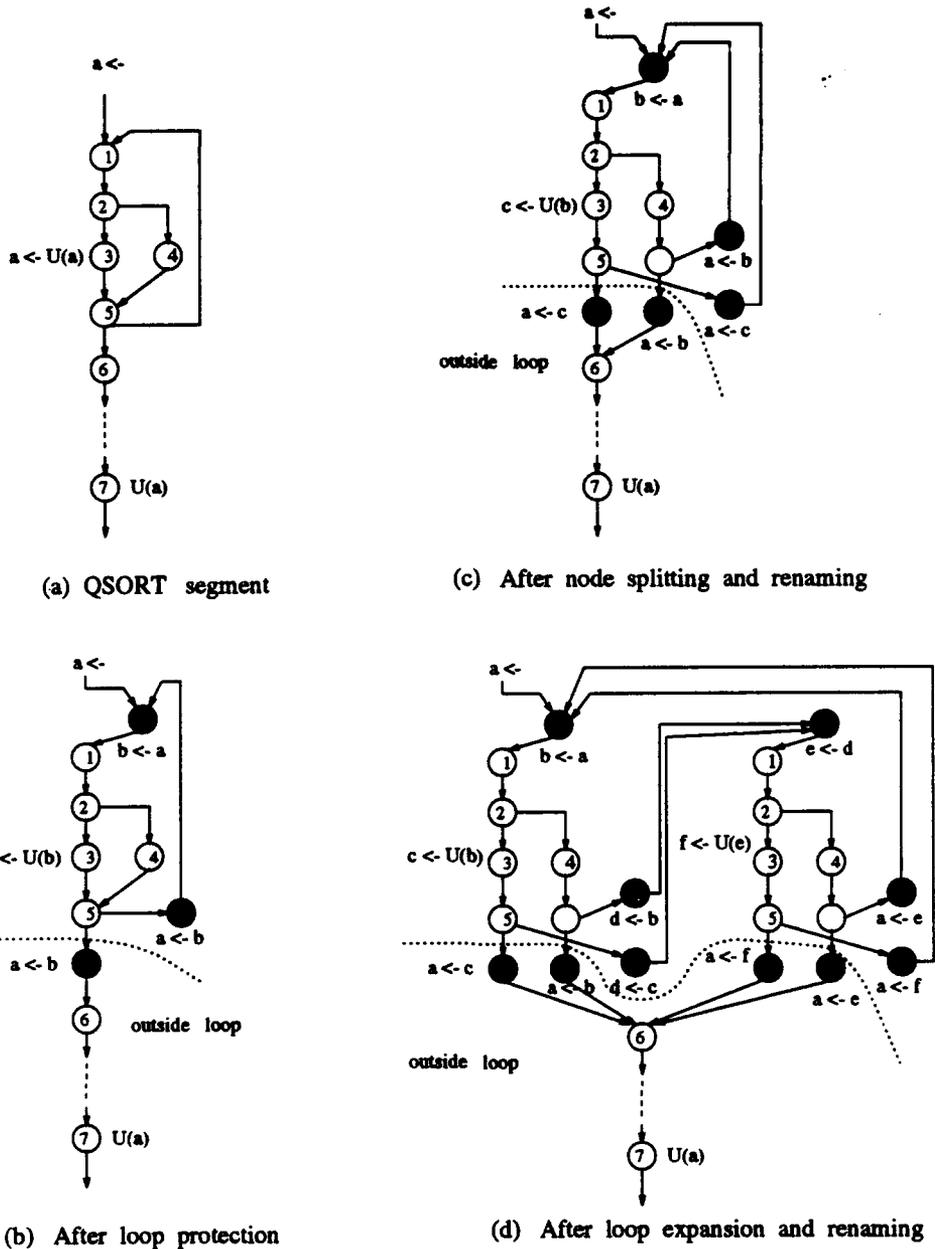
(a) QSORT segment

(c) After node splitting and renaming

(b) After loop protection

(d) After loop expansion and renaming

Figure 1: Renaming – loop protection, node splitting and loop expansion
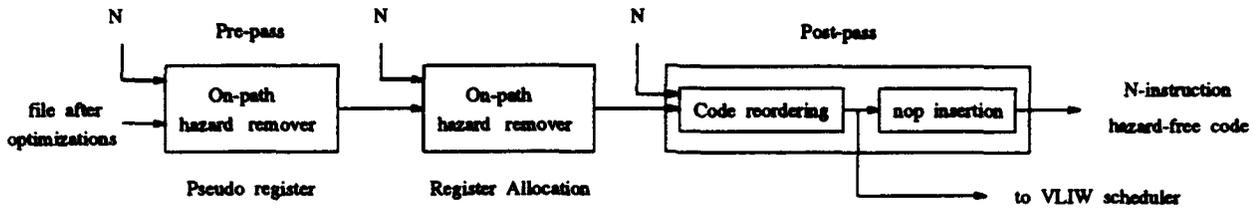
5

Figure 2: Hazard removal for uni-processors

while the nop insertion stage is the *post-pass stage*. The post-pass stage handles hazards

across interprocedural boundaries, and the new hazards induced by the spill registers and

parameter passing registers. Sufficient nops are then inserted to make the code free of any

remaining hazards.

3.2   Compiler-Based Scheme for VLIW Architectures

In this section, we describe how the instruction retry scheme for uni-processors can be ex-

tended to VLIW architectures. We have implemented a modified trace scheduling algorithm

for trace-based simulation. Profiling is implemented to guide the trace selection [6, 7]. The

most frequently 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 cor-

rect 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 uni-processors, we can apply our previous approach to

generate N-instruction hazard-free code. One direct extension would schedule such hazard-
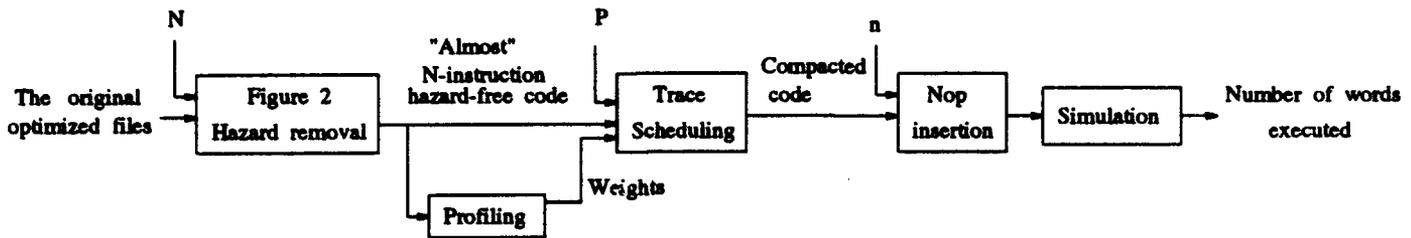
Figure 3: The hazard removal, scheduling and simulation process for VLIW

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 as shown in Figure 2 is scheduled. Nop instruction words are

then inserted to resolve both types of hazards in the compacted code. Figure 3 outlines the

entire process for hazard removals, scheduling and simulation.

Further enhancement can be made by introducing priorities for selecting feasible instruc-

tions during list scheduling. For example, consider the instruction word segment of QSORT

generated by trace scheduling, as shown in Figure 4(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 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 4(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 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 trick 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

8

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 4: Enhanced list scheduling algorithm

9

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.

4   Simulation Results

Six benchmarks are utilized. QUEEN is the 8-queen problem, and QSORT is the quick sort algorithm. Both programs include recursive subroutines. CMP, 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.

For simplicity, we assume each instruction word takes 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 word instruction. As the compacted code is executed on a uni-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 uni-processors (N) and the desired rollback distance for VLIW architectures (n) with a varying number of functional units P.

The original optimized code generated by 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 5 – 10 illustrate the performance
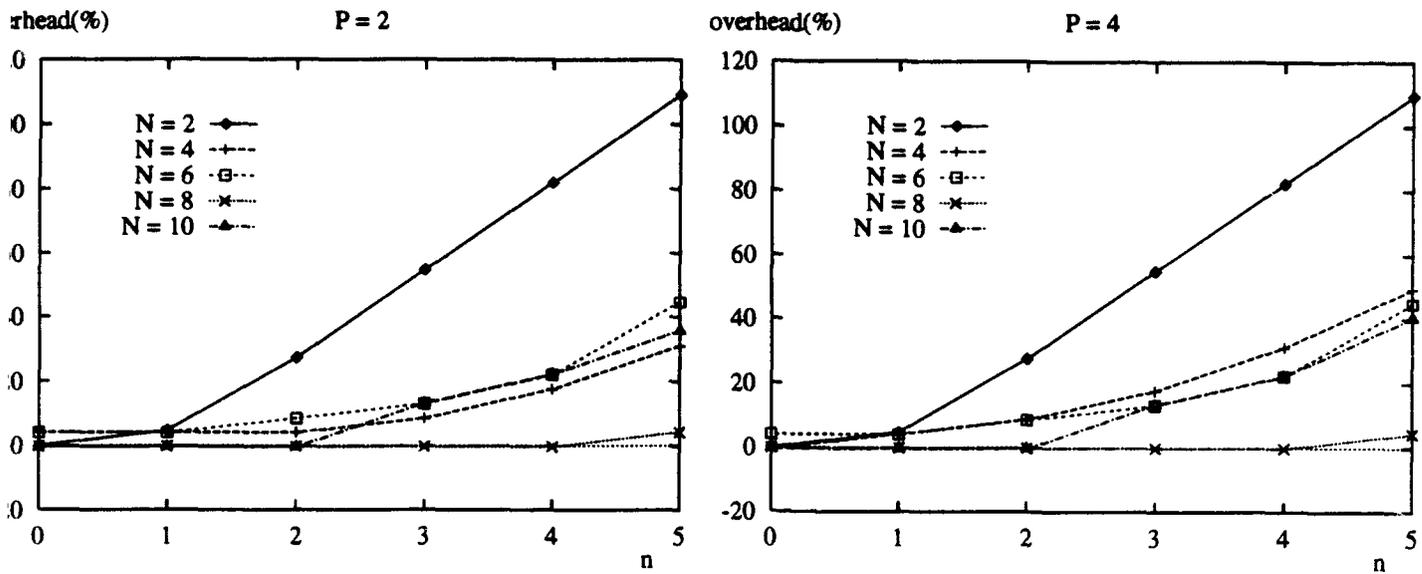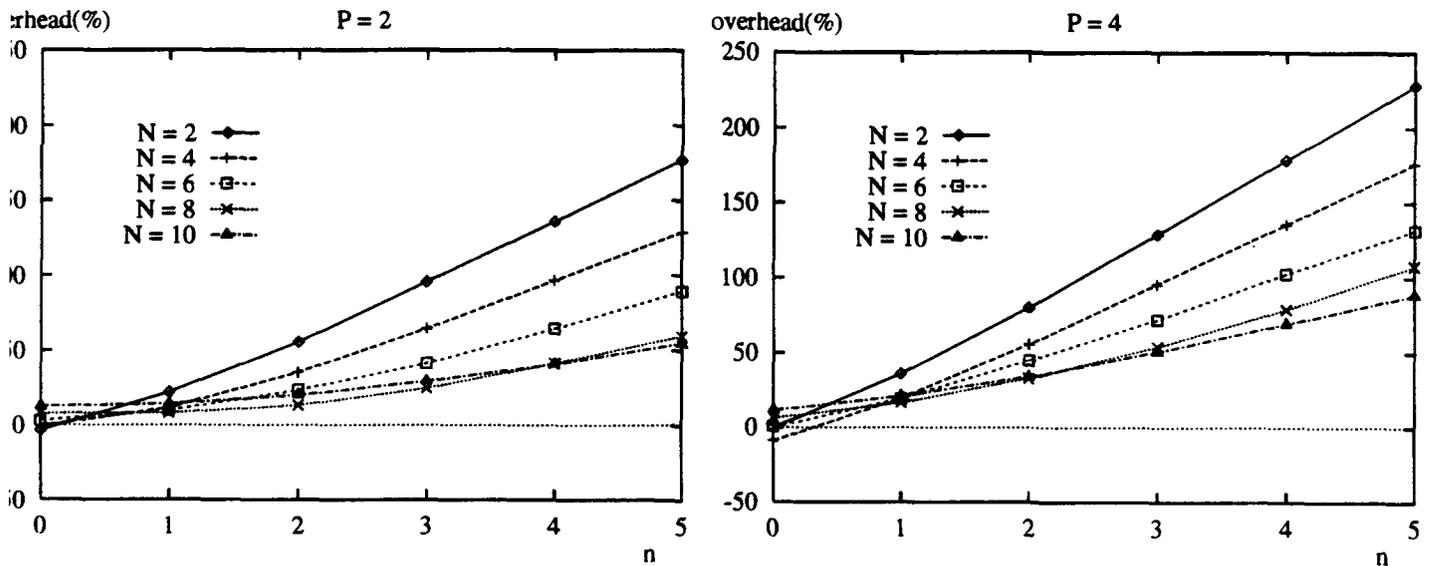
Figure 5: Performance overhead for CMP

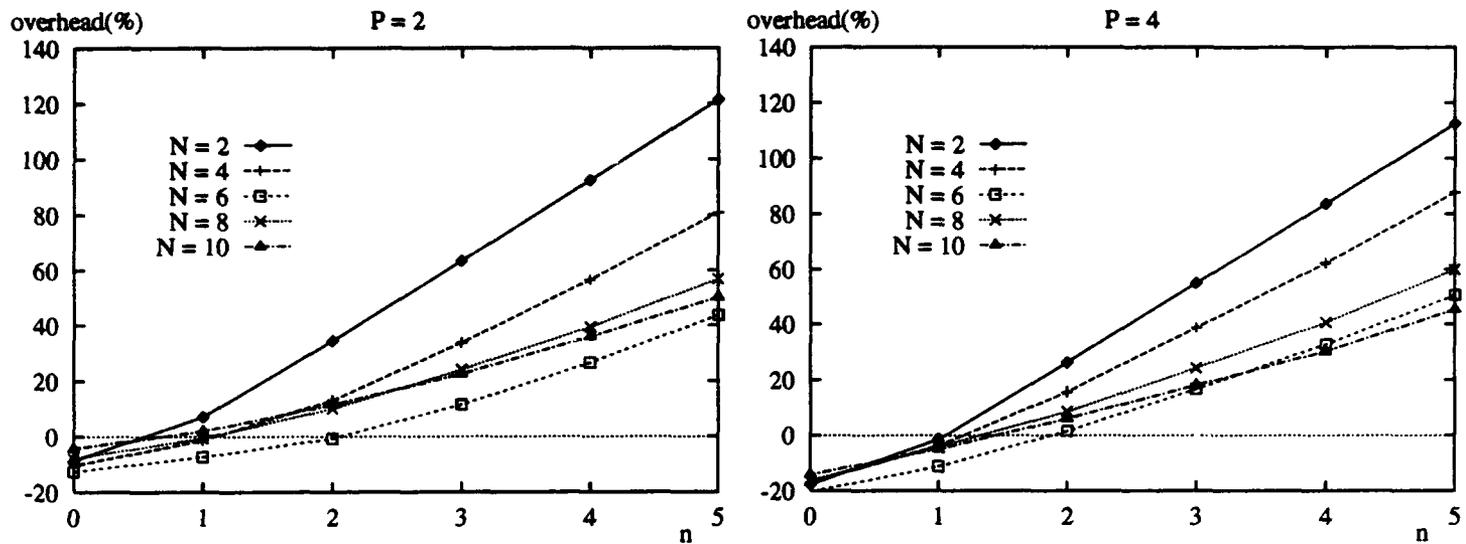Figure 6: Performance overhead for COMPRESS
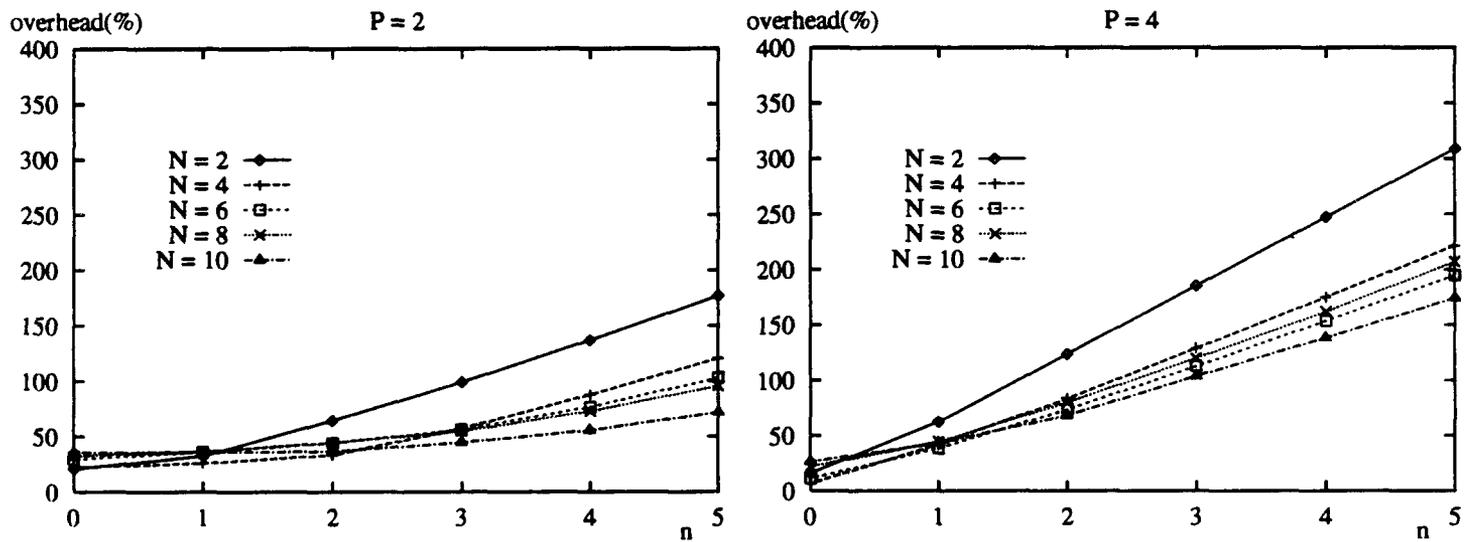
11

Figure 7: Performance overhead for PUZZLE

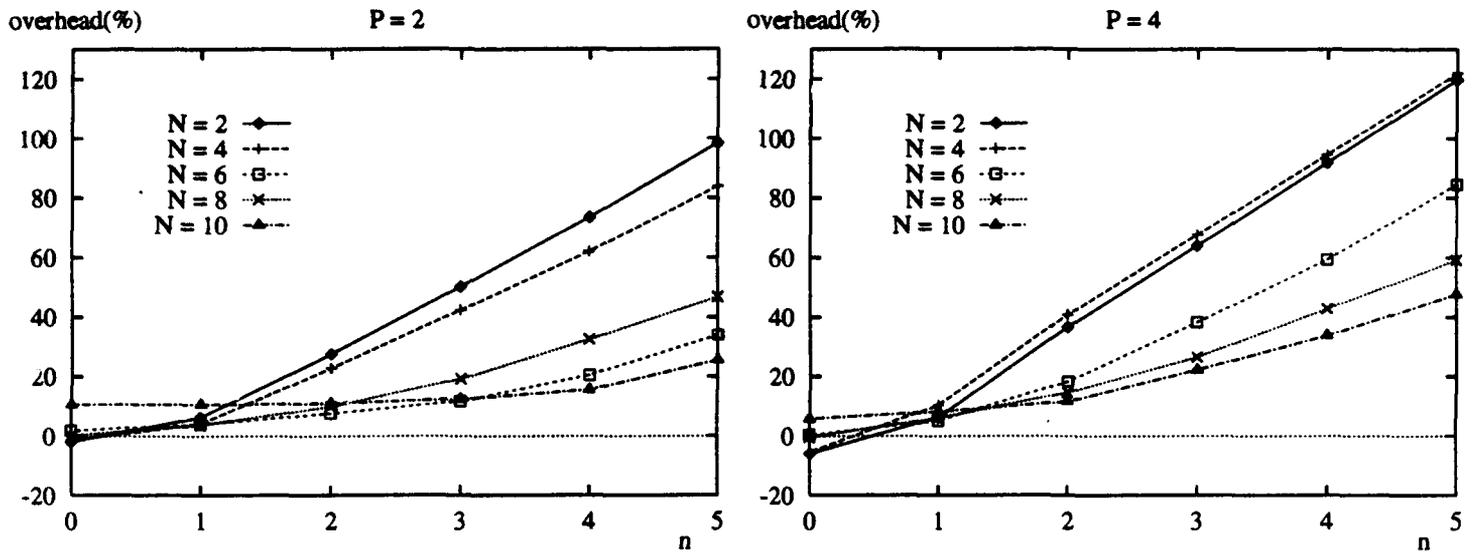Figure 8: Performance overhead for QSORT
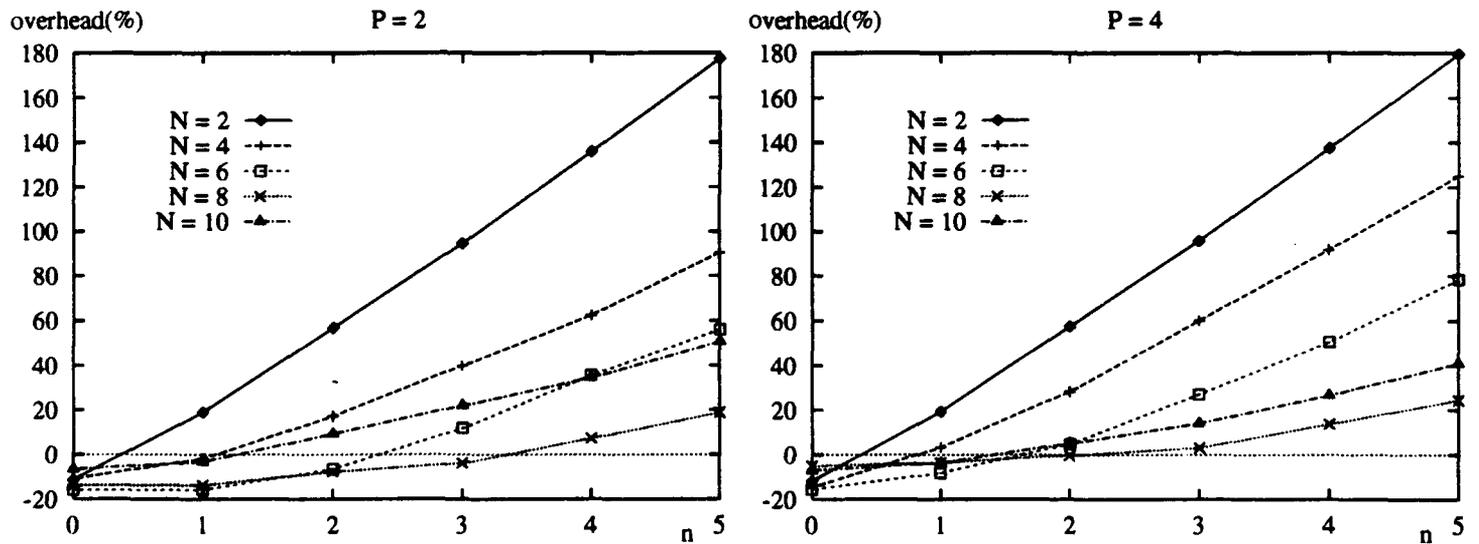
Figure 9: Performance overhead for QUEEN

Figure 10: Performance overhead for WC

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 it has a frequently called recursive subroutine, *merge_sort*. The prioritized list scheduling algorithm helps to improve its performance, as shown in Figure 11.

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
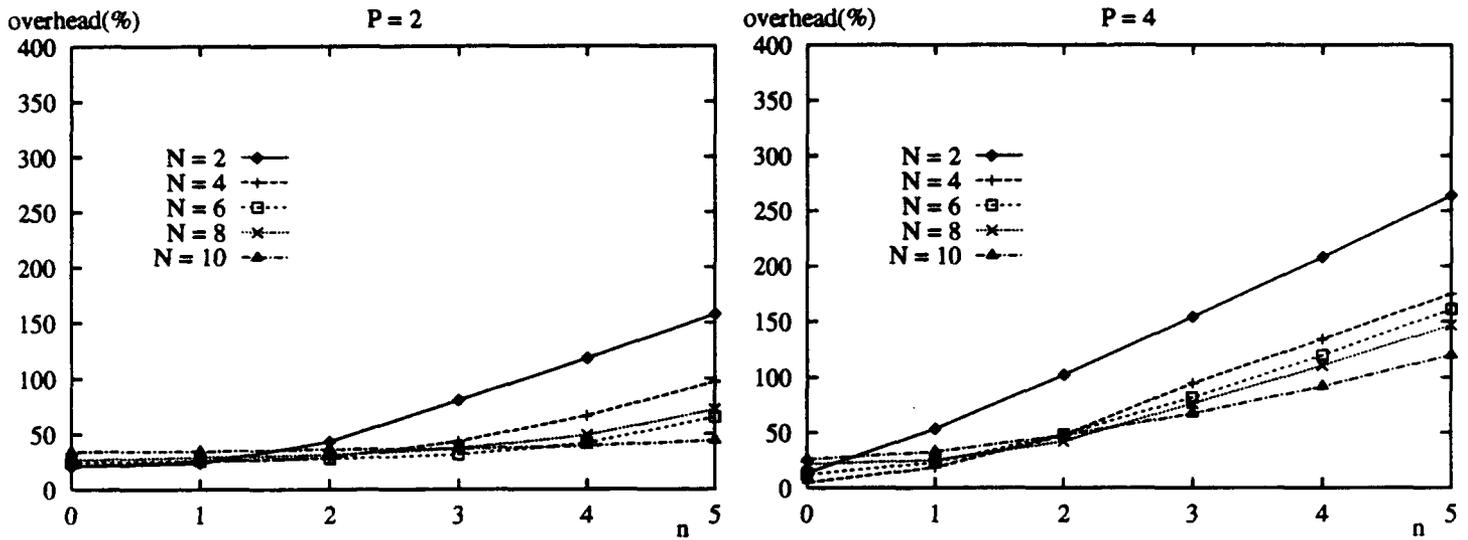
14

Figure 11: Performance overhead for QSORT under the prioritized list scheduling algorithm 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 12 – 17 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.

Figures 18(a) and (b) illustrate the performance overhead when employing a read buffer. For $P = 2$ and $P = 4$, both figures almost have the same shape. PUZZLE has a near 0
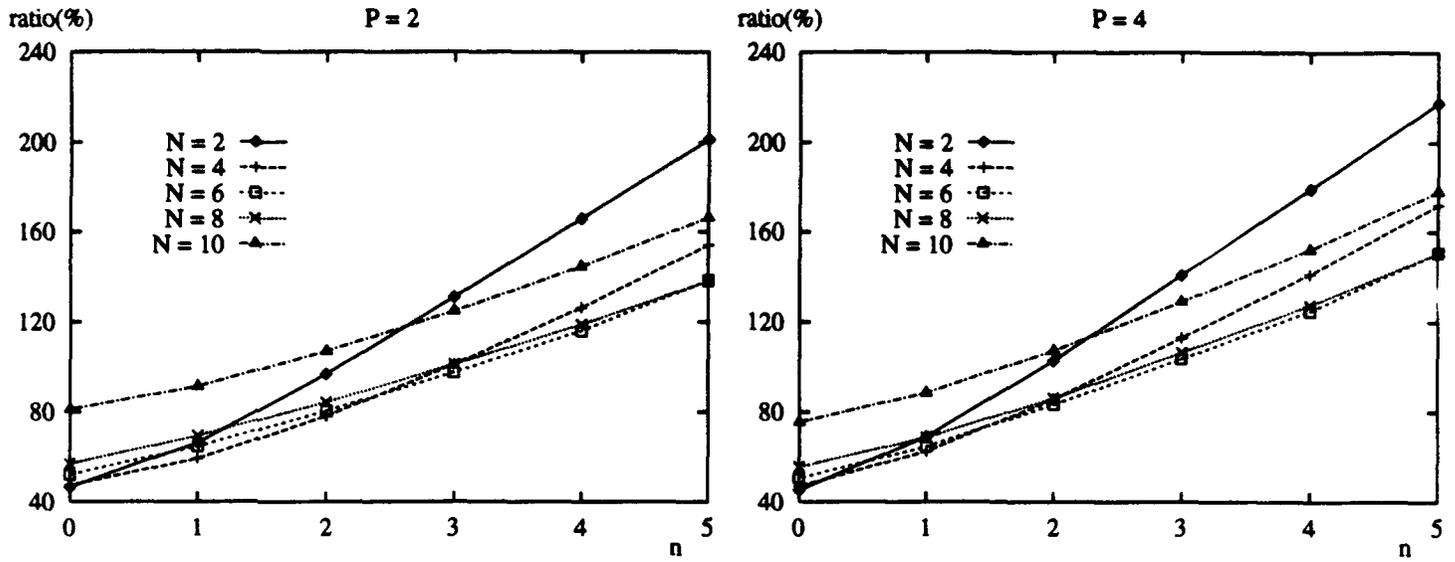
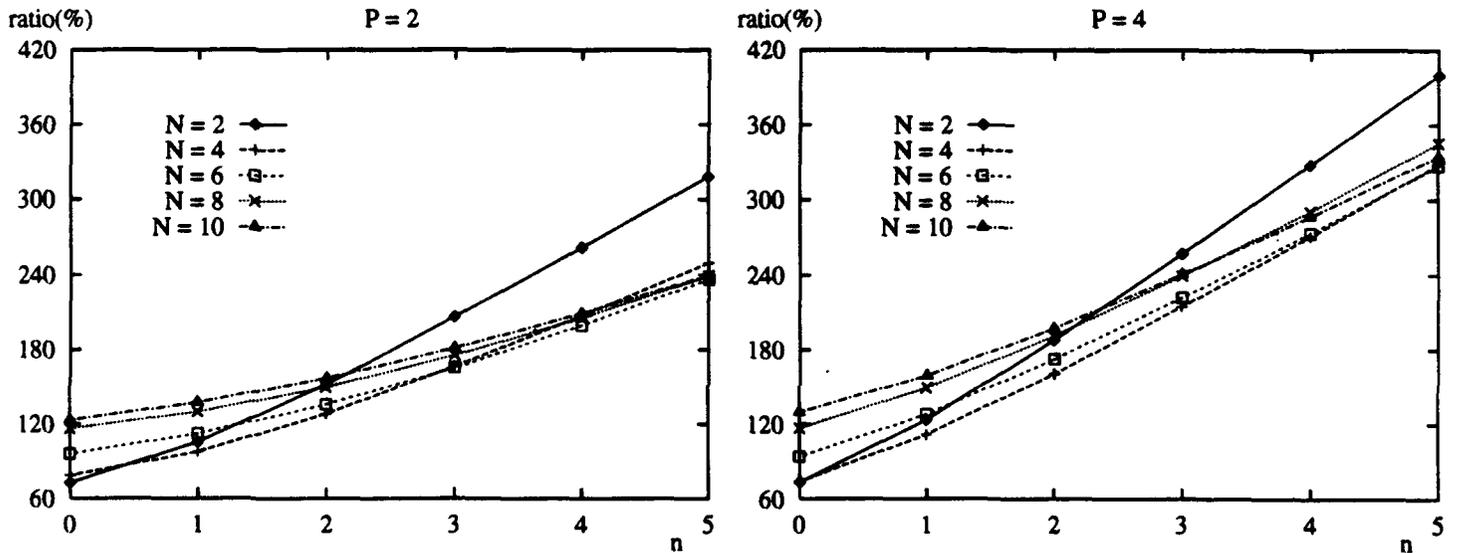Figure 12: Code growth ratio for CMP
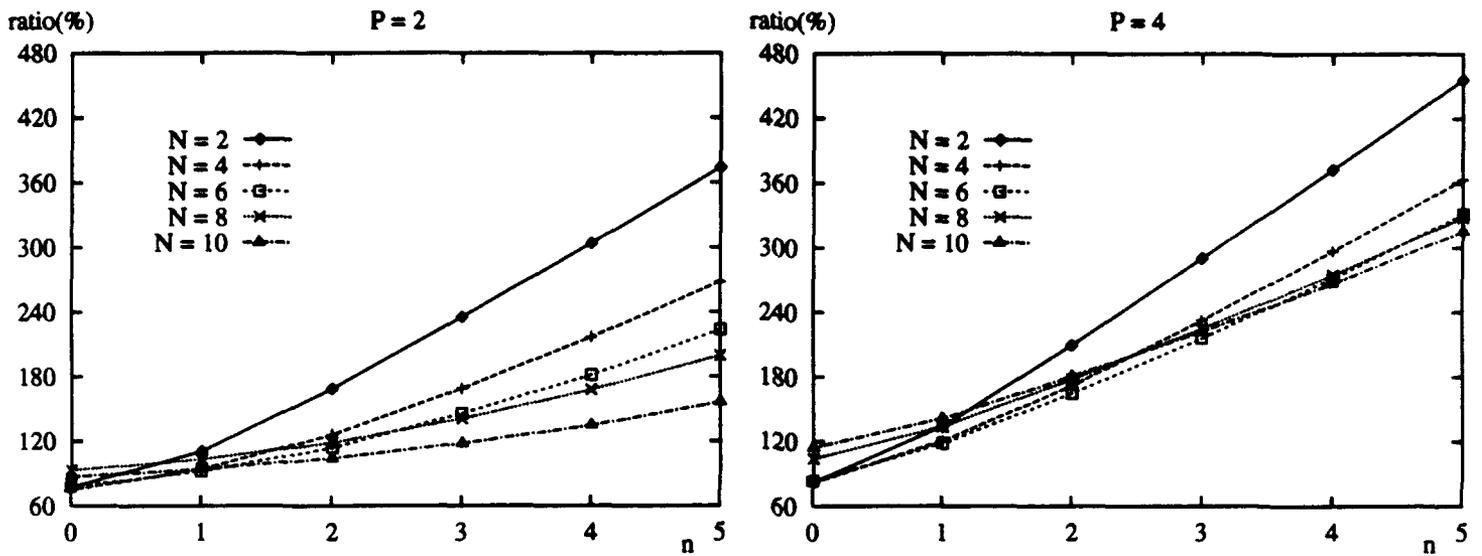

Figure 13: Code growth ratio for COMPRESS

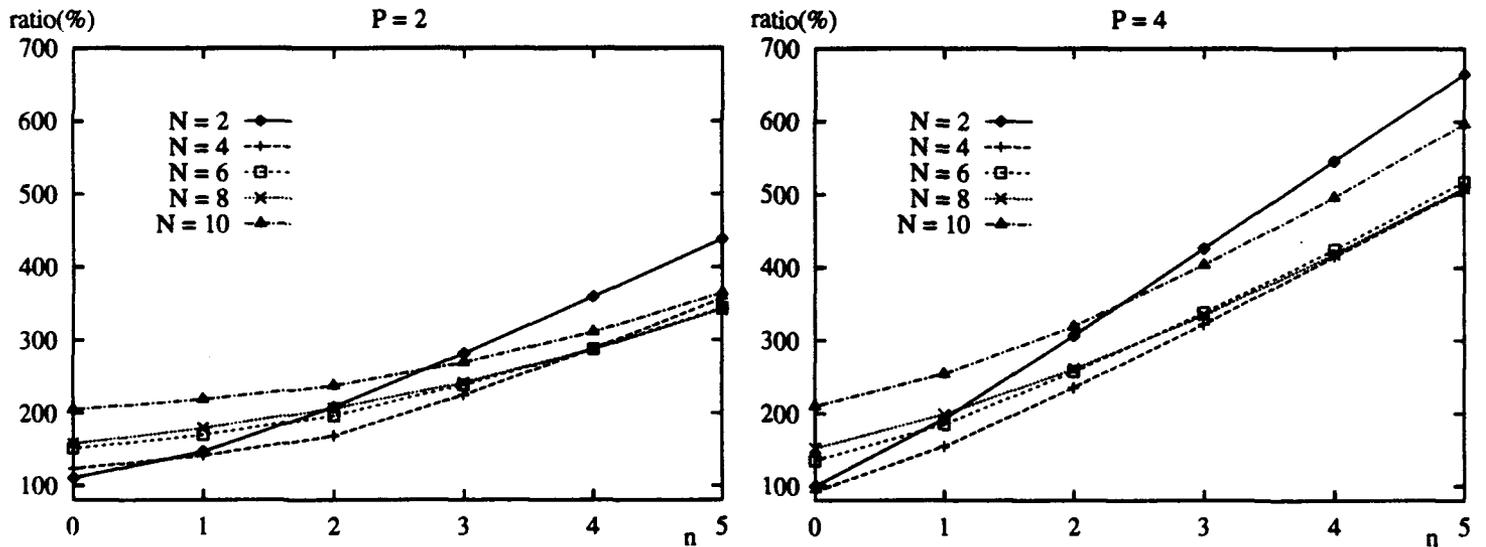Figure 14: Code growth ratio for PUZZLE

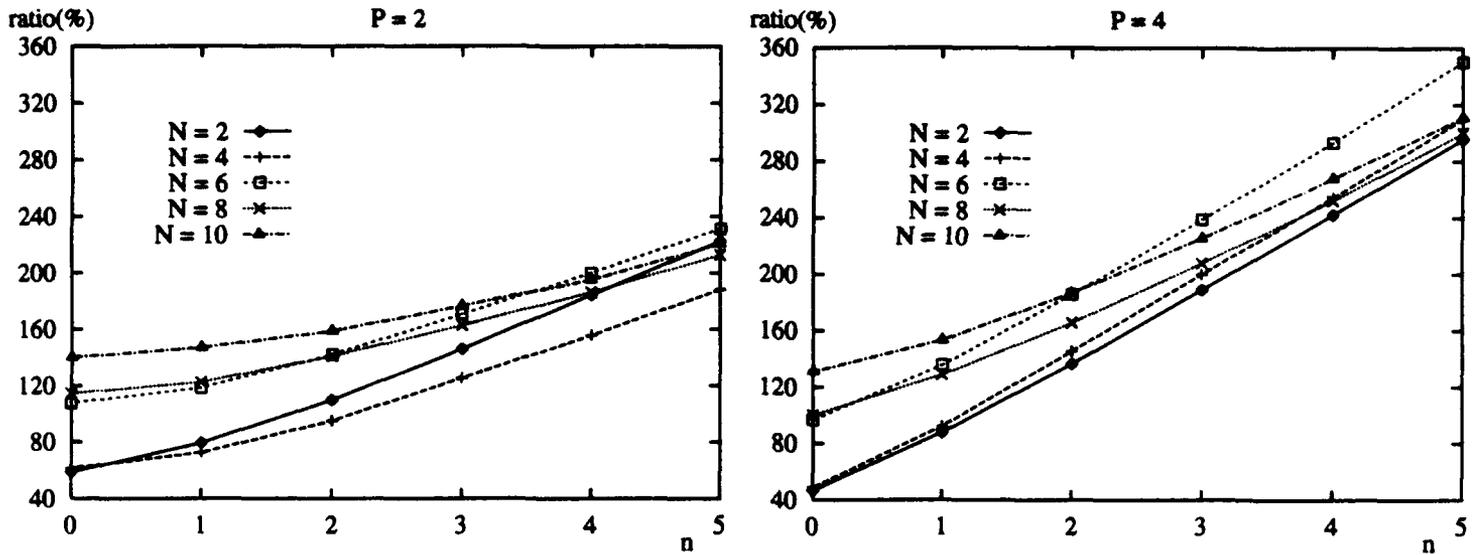Figure 15: Code growth ratio for QSORT

17

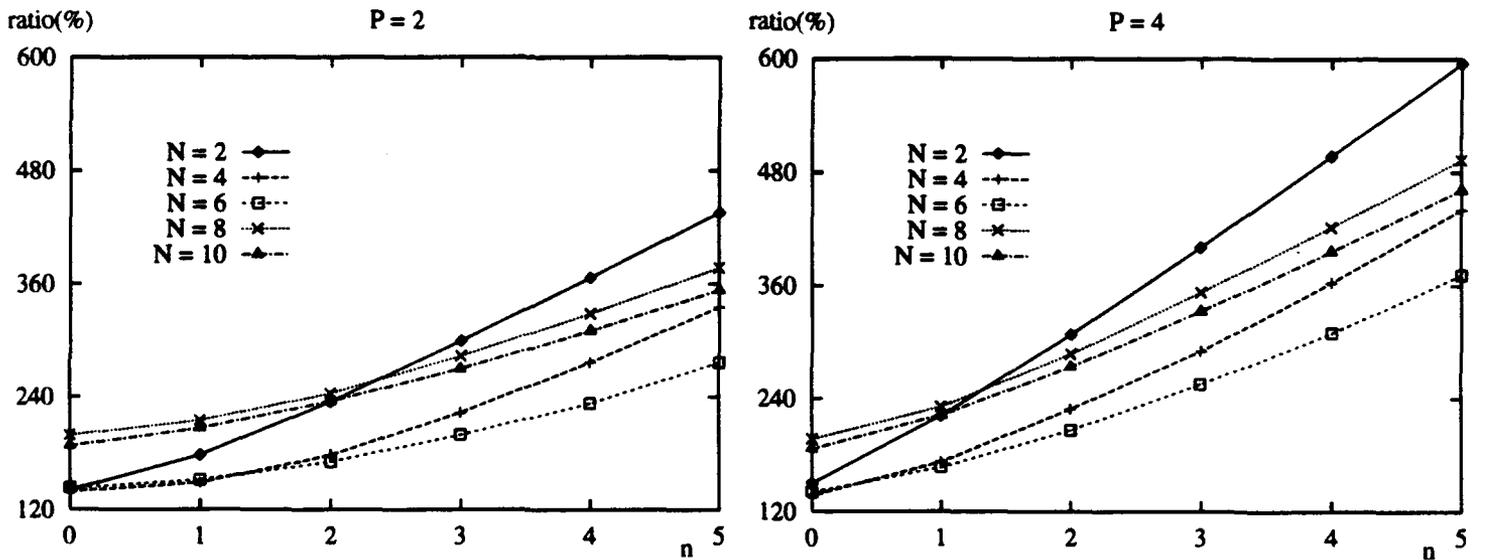Figure 16: Code growth ratio for QUEEN

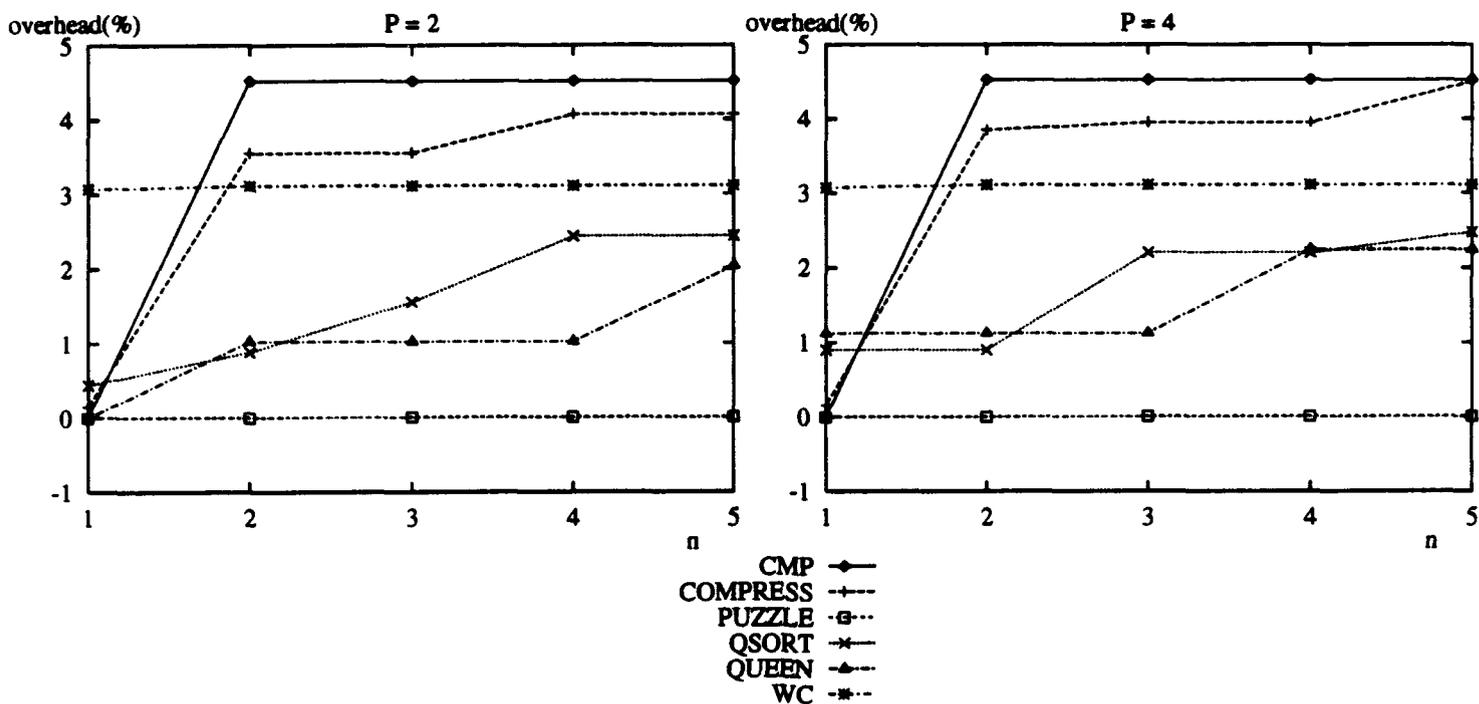

Figure 17: Code growth ratio for WC

Figure 18: Performance overhead for using a read buffer

performance impact, and CMP has the highest overhead, 4.52% for $P = 2$, $n = 5$, and 4.53% for $P = 4$, $n = 5$. In average, when $n = 5$, the performance overhead for $P = 2$ is 2.7% and for $P = 4$ is 2.81%. Figures 19(a) and (b) illustrate 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 $P = 4$.

5 Summary

We described the development of compiler assisted multiple instruction word retry for VLIW architectures. Both a software-based scheme and a hardware read buffer with compiler-assisted scheme were proposed. The first scheme employs compiler technology to resolve all
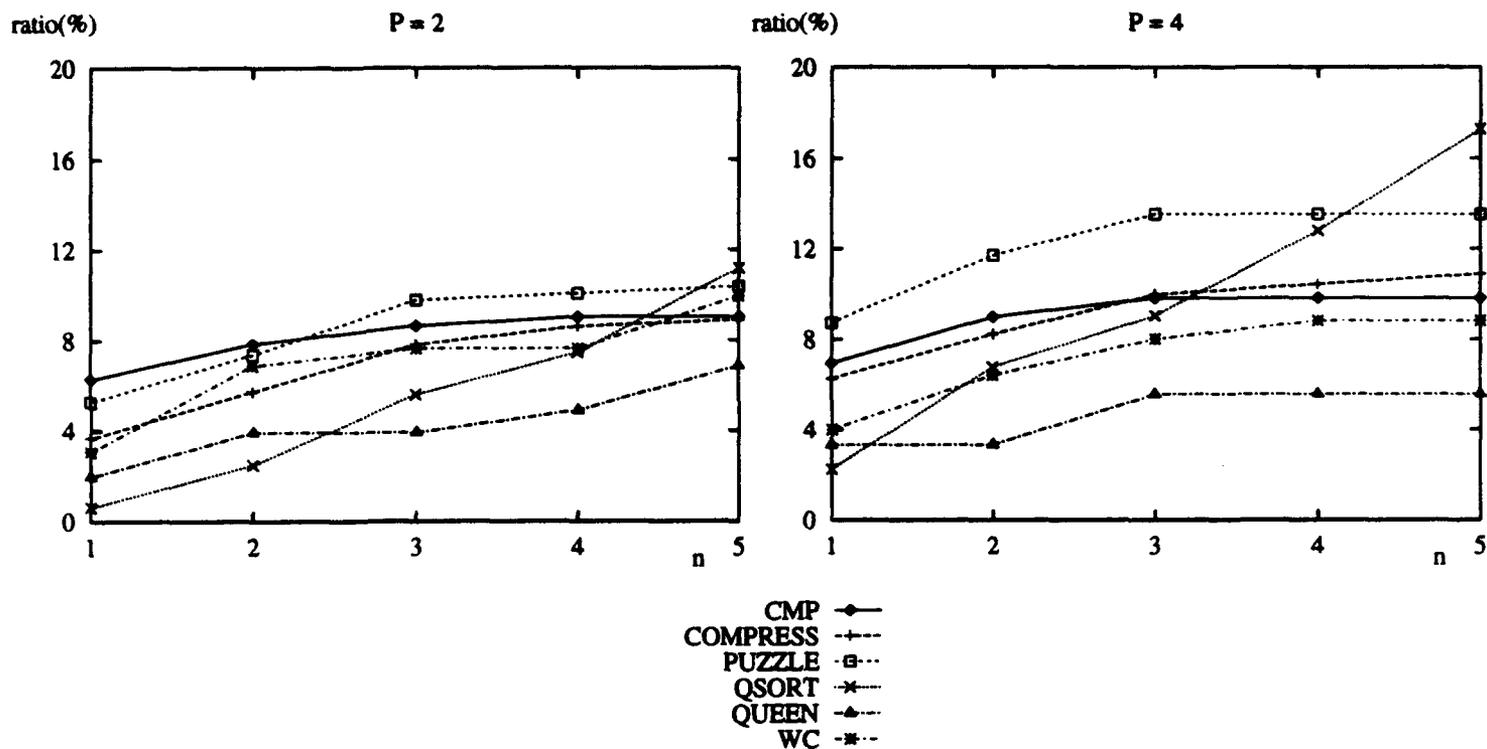
Figure 19: Code growth ratio for using a read buffer

hazards. The performance costs are higher than in the second scheme, ranging from 20% to 70% for 2 functional units, and from 5% to 170% for 4 functional units.. The second scheme employs a hardware read buffer to retain reads within the last n instruction words. The mechanism resolves the frequently occurring on-path hazards. Branch hazards are resolved by the compiler. 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.

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 Transactions 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] 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.

[4] 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.

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

[6] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, July 1981.

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

[8] W.-M. W. Hwu *et al.*, "The Superblock: An effective technique for VLIW and superscalar compilation," *the Journal of Supercomputing*, Kluwer Academic Publishers, pp. 229–248, July 1993.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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*, vol. I, pp. 192-195, 1992.

[17] 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.

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

[19] 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.

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