October 1992

NASA-CR-190972

UILU-ENG-92-2239
CRHC-92-21

Center for Reliable and High-Performance Computing

SIMULATION AND ANALYSIS OF SUPPORT HARDWARE FOR MULTIPLE INSTRUCTION ROLLBACK

Neil J. Alewine

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

Simulation and Analysis of Support Hardware for Multiple Instruction Rollback

Neal J. Alewine

Center for Reliable and High-Performance Computing
University of Illinois
1308 West Main Street
Urbana, IL 61801
e-mail to alewine@crhc.uiuc.edu

September 10, 1992

Abstract

Recently, a compiler-assisted approach to multiple instruction retry was developed [1]. In this scheme a *read buffer* of size $2N$, where N represents the maximum instruction rollback distance, is used to resolve one type of data hazard. This hardware support helps to reduce code growth, complilation time, and some of the performance impacts associated with hazard resolution. The $2N$ read buffer size requirement of the compiler-assisted approach is worst case, assuring data redundancy for all data required but also providing some unnecessary redundancy. By adding extra bits in the operand field for source 1 and source 2 it becomes possible to design the read buffer to save only those values required, thus reducing the read buffer size requirement. This study measures the effect on performance of a DECstation 3100 running 10 application programs using 6 read buffer configurations at varying read buffer sizes. Two configurations emerged as the most efficient and differed depending on whether split-cycle-saves were assumed. It was determined that while the full $2N$ read buffer size is not required, nearly $2N$ is required to adequately handle most applications. It is also shown that if a buffer size less than $2N$ is chosen, it is possible that some applications will suffer significant performance impacts. This study concludes that no reduction in read buffer size below $2N$ is practical given a wide variety of general applications.

1 Introduction

1.1 Instruction Rollback Schemes

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 [1, 6–8]. Multiple instruction retry within a sliding window of a few instructions [1, 6, 7], or re-execution of a few cycles [9], 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 [10], micro-rollback [7], and compiler-assisted rollback [1] differ in where the updated and old values reside, circuit complexity, CPU cycle times, and rollback efficiency.

In contrast to totally hardware schemes, a compiler-assisted approach to implementing multiple instruction retry was developed where the compiler uses a series of transformations to eliminate anti-dependencies of length $\leq N$ [6]. This approach produces a performance

impact consistent with hardware-based techniques [7] and has the added benefit of making N a compile-time parameter.

More recently the compiler-assisted multiple retry scheme was extended to include a broad class of code execution failures [1]. The error model was expanded to allow any legal path in the control flow graph, thus allowing branch recovery. Possible hazards were shown to be one of two types. Similar compiler techniques to those in [6] were shown to be effective in resolving both types of hazards. Finally, a hardware scheme was introduced to resolve one type of hazard, thus reducing code growth, complilation time, and performance impact.

1.2 Compiler-assisted Multiple Instruction Retry

Within a general error model, data hazards resulting from instruction retry are of two types [1]. On-path hazards are those encountered when the instruction path after rollback is the same as the initial instruction path. As shown in Figure 1(a), r_x represents an on-path hazard. The initial instruction sequence causes r_x to be written. However, after rollback, r_x is read prior to being re-written. Branch hazards are those encountered when the instruction path after rollback is different than the initial instruction path. As shown in Figure 1(b), r_y represents a branch hazard. The initial instruction sequence causes r_y to be written. After rollback, r_y can be read prior to being re-written as with the on-path hazard, however in this case initial path repetition is not guaranteed.

Compiler transformations have been shown to be effective in resolving branch hazards [1]. Hardware support consisting of a read buffer of size $2N$, as shown in Figure 2, was

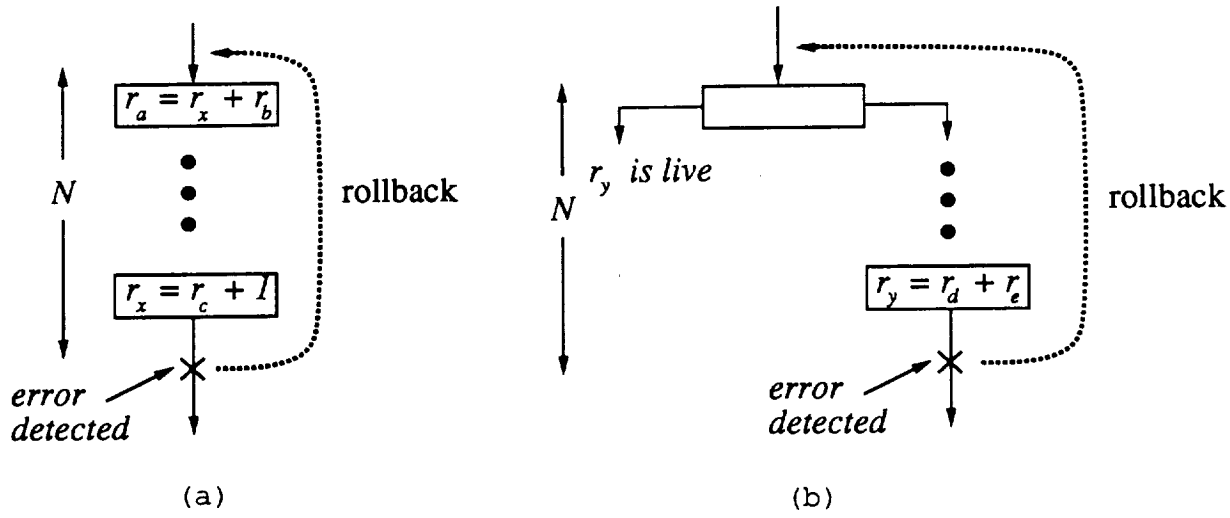


Figure 1: On-path Hazard.

similarly shown to be effective in resolving on-path hazards. The read buffer maintains a window of register read history. At rollback, the read buffer is flushed back to the general purpose register file, restoring the register file to a restartable state.

The read buffer size requirement of $2N$ is worst case. The buffer simple saves the last N register reads from the register file across the source 1 bus (S1) and the source 2 bus (S2). This assures data redundancy for all values required but also saves register reads which are not required during rollback. Register reads which require saving are known at compile time. If this information were added to the instruction at compile time (eg., as a extra bit field for source 1 and for source 2), then the read buffer could be designed to save only those values required. As long as the required values were maintained for N cycles, the read buffer size could conceivably be less than $2N$.

The purpose of this study is to determine the effect on a system's performance given various read buffer configurations for a range of application programs, assessing the viability

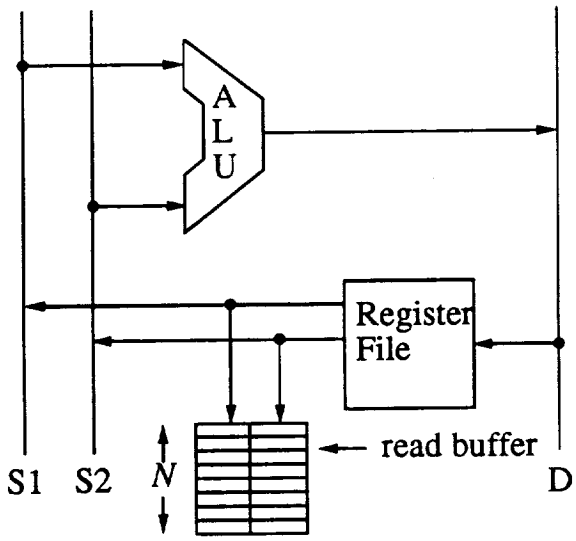


Figure 2: Read Buffer.

of read buffer size reduction while determining the optimal buffer configuration.

Section 2 describes the rollback strategy and various read buffer configurations to be studied, Section 3 discusses the methodology used for the simulations, Section 4 contains results and analysis from the simulations, and Section 5 summarizes the findings.

2 Read Buffer Configurations

2.1 Overall Recovery Strategy

Given a read buffer configuration as shown in Figure 3, rollback is accomplished by first flushing the read buffer back to the general purpose register (GPR) file in the reverse order of which the values were saved. Figure 3 shows the two FIFO read buffers above S1 and S2 to better illustrate the buffer's content given the instruction sequence shown. As long as the depth of the dual FIFO read buffers are N, redundant copies of the appropriate register

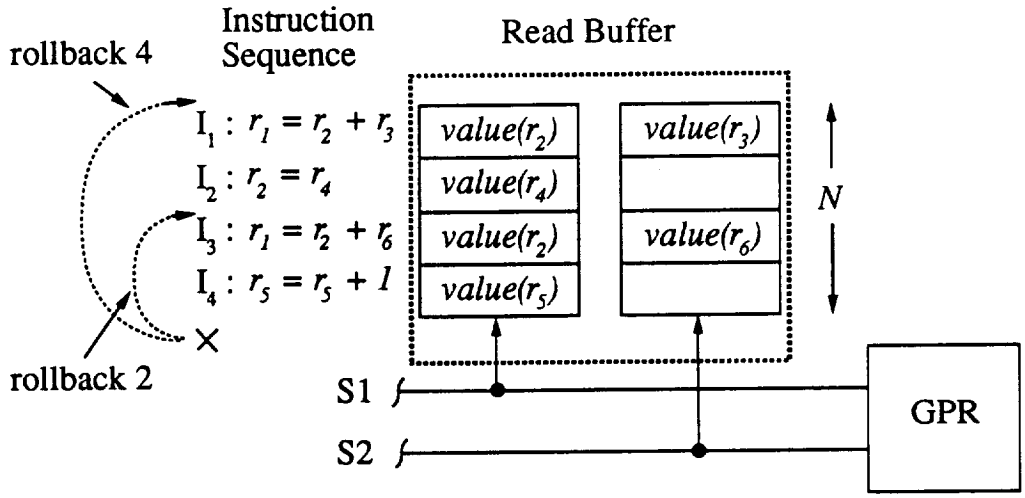


Figure 3: Read Buffer.

values (denoted $value(r_x)$) are available to restore the register file given a rollback of $\leq N$.

Suppose now that only some of the register values need to be saved. This can be determined at compile time when data hazards are detected. Figure 4 shows such a case with the registers to be saved marked with an "*". Since only those values which need to be saved are saved, the read buffer total size can now be less than N. In this case however the instruction count must also be saved so that the value can be maintained for at least N cycles. In the event that the read buffer overflows, the oldest value in the buffer must be pushed to memory and a record kept so that during rollback the value can be retrieved from memory. Given a dual FIFO depth of M, memory would serve the function of the remaining $N - M$ of the two FIFOs. This read buffer design reduces the buffer size while introducing potential performance impacts due to buffer overflows. What will be studied is how the performance impact increases as the buffer size decreases.

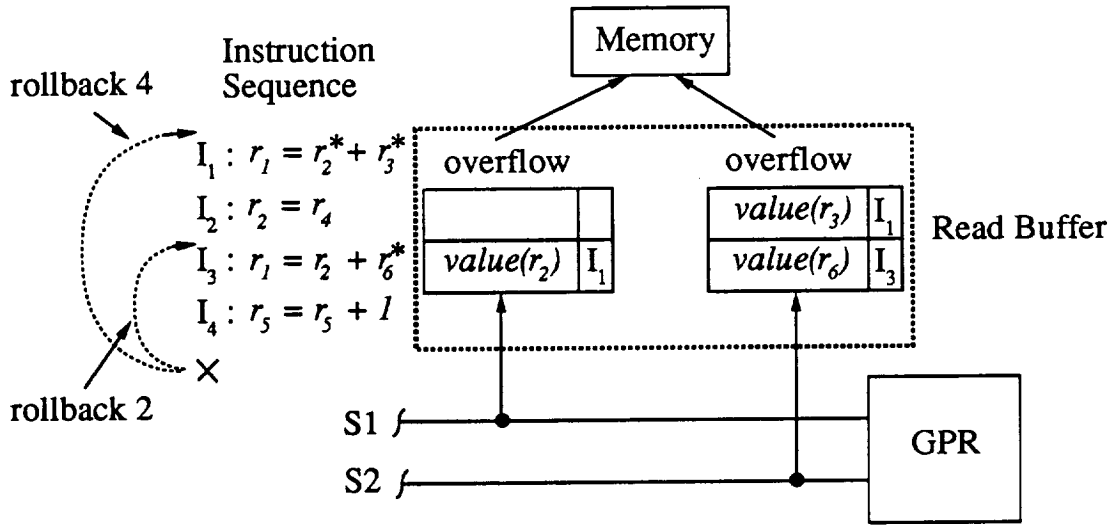


Figure 4: Read Buffer Size $< 2N$.

2.2 Other Considerations

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

Instead, it will be assumed that a read buffer overflow will always cause a single stall. If both FIFOs overflow, two stalls will be incurred. This simplifying assumption is pessimistic relative a store buffer which may have empty locations, while optimistic relative to a full

store buffer requiring a write to cache. These assumptions guarantee that all measured performance impact is directly due to changes in the read buffer size or configuration.

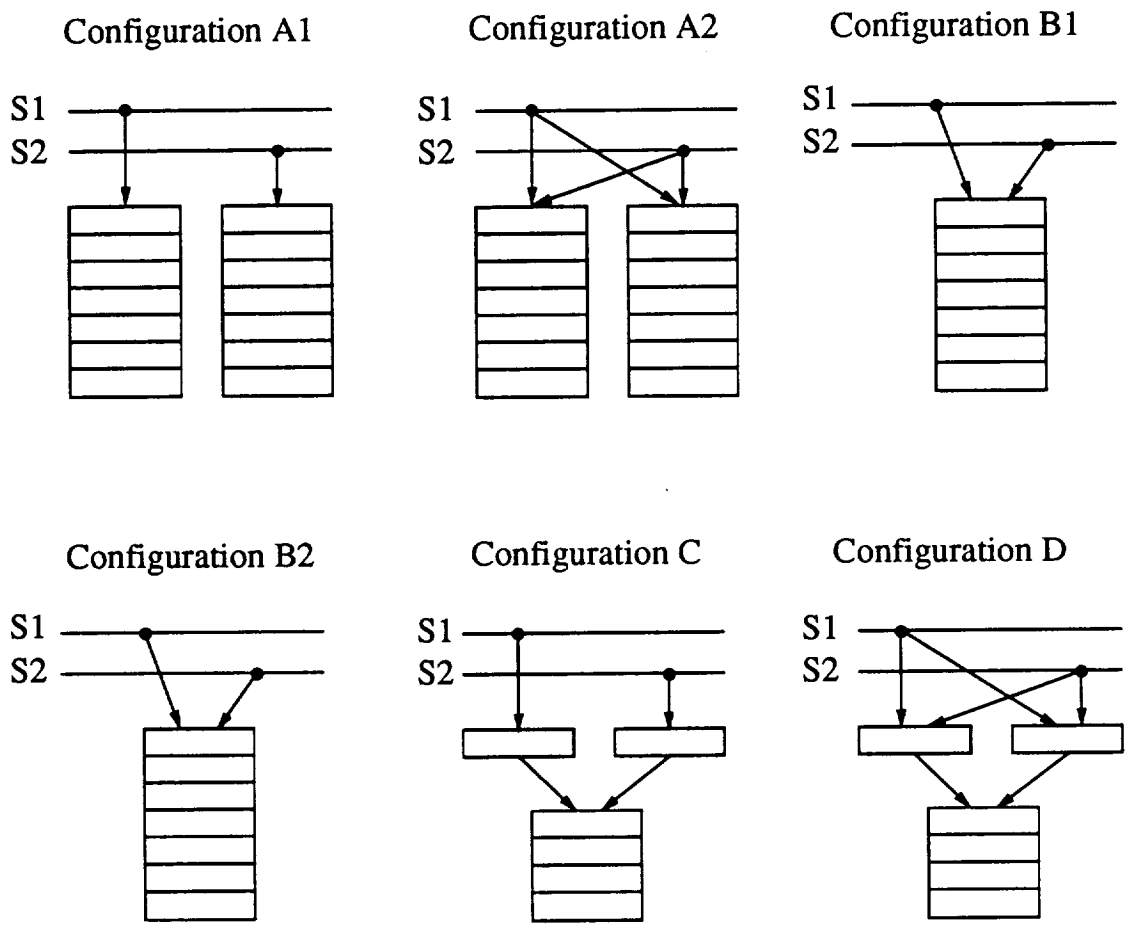
2.3 Read Buffer Models

The most straight forward model for the read buffer is that of configuration A1, shown with configurations A2, B1, B2, C and D in Figure 5. The obvious problem with configuration A1 is that if the FIFO connected to S1 is full and the current S1 value must be saved, a stall occurs due to overflow even though the FIFO connected to S2 may be available. Configuration A2 resolves this inefficiency by allowing either S1 or S2 access to either FIFO.

Configuration B1 also resolves the inefficiency of configuration A1 by having a single FIFO with both S1 and S2 connected to it. Configuration B1 assumes that the S1 value and the S2 value can be saved within the same cycle. This would be possible if the S1 value is saved during the first half of the cycle and the S2 value is saved during the second half of the cycle. This split-cycle-save assumption is consistent with the design of register files which write back during the first half of the cycle and read during the second half of the cycle [11].

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

Configuration C attempts to lessen the impact due to the bottleneck in configuration B2 by adding two single level queues between S1&S2 and the single FIFO. Configuration C can absorb a simultaneous save, processing the first in the current cycle and the second in the



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

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

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

Figure 5: Read Buffer Configurations.

next cycle assuming the next instruction does not also require a simultaneous save.

Configuration D extends configuration C to allow both S1 and S2 access to either queue.

3 Simulation Methodology

3.1 Trace vs Simulation

There are two methods to obtain performance measurements given the various configurations of the previous section. The first is to obtain a trace of the application program and then analyze the read buffer's effect given the actual instruction sequence. Although the register reads required to be saved can be computed at compile time through hazard calculation, this information is difficult to maintain in a trace since the the actual instruction set cannot be altered. Also, traces typically require a great deal of disk space.

A second approach is to simulate the read buffer at the instruction level. Prior to each instruction execution, a procedure is called to update the read buffer model. Parameters such as which register reads to save and instruction type can be passed to the simulation procedure. The difficulty with this approach is the code growth in the original application program and a dramatic reduction in application run time.

Due to the availability of workstations to run the simulations and the lack of available disk space at this time, the second approach was chosen for the simulation.

3.2 Simulation Considerations

Even a modest size hand coded simulation program would be too large to insert prior to each instruction in the application program. It is therefore necessary to branch to a procedure which performs the read buffer simulation. Given the complexity of a software model containing 6 different configurations, use of a high level language like C to program the model is desirable.

The instructions inserted to branch to the simulation procedure prior to each original application instruction can not be added in the high level language. If this were done, the register assignments would be corrupted as the new instructions were compiled and the one-to-one correspondence between original instructions and simulation procedure calls would be lost. Therefore, the calculation of hazards and subsequent determination of which register reads should be saved must be performed at the s-code level (after register assignment) and the appropriate s-code level instructions inserted prior to each original s-code instruction of the application program. The problem is that calling and executing the simulation procedure corrupts current register values of the application, since the compiler was not aware of the inserted instructions. This is further complicated by the fact that since the simulation procedure is coded in C, it's register usages are not known.

3.3 Methodology

To minimize the application code growth, a simple hand written s-code sequence shown in Figure 6 is inserted prior to each instruction. This code sequence pushes register 31 on the

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

Figure 6: s-code Instrumentation

stack (register 31 is used as a return address during procedure calls and therefore will be corrupted), pushes register 4 on the stack, loads register 4 with information relative to the saving of S1 or S2 for this particular instruction, calls *rbuf2_save*, and then pops from the stack and restores registers 31 and 4.

The code sequence of Figure 6 only saves the two registers necessary to branch to a procedure and pass one register's worth of parameters. Prior to actually branching to the read buffer simulation, the remaining registers which are used need to be saved. This was not done in the code sequence of Figure 6 to limit application code growth. The hand written code sequence, *rbuf2_save*, shown in Figure 7 conservatively saves all remaining registers on the stack. Note that both *callee* and *caller* saved registers must be saved since the compiler was unaware of the inserted procedure call.

Finally, the C level read buffer simulation, *rbuf2_sim*, is called from the code sequence shown in Figure 7. The simulation program can be modified and re-compiled without a

```
#Begin rbuf2_save procedure
        .verstamp 2 10
        .extern _iob 60
        .extern _pctype 4
        .extern _ctype_ 0
        .text
        .align 2
        .file 2 "rbuf2_save.c"
        .globl rbuf2_save
        .loc 2 10
        .ent rbuf2_save 2
rbuf2_save:
        .option O1
        subu    $sp,    $sp,    160
        sw      $31,    16($sp)
        sw      $30,    20($sp)
                •
                •
                •
        sw      $2,     132($sp)
        .mask 0x8fffffff, -4
        .frame $sp, 160, $31
        .loc 2 11
        lw      $4,     124($sp)
        .livereg 0x8fffffff,0xfff
        jal rbuf2_sim  ← C-level read buffer simulation program
        .loc 2 12
        lw      $31,    16($sp)
        lw      $30,    20($sp)
                •
                •
                •
        lw      $2,     132($sp)
        addu    $sp,    $sp,    160
        j       $31
        .end rbuf2_save
```

Figure 7: *rbuf2_save* s-code sequence.

corresponding modification to the application program or the two previous s-code sequences.

Similar s-code sequences to handle initialization and summary calculations were also developed. The *initialization* procedure call is placed in the "*main*" module prior to the first instruction. The *summary* procedure calls are placed prior to all "*jal exit*" instructions in all modules and prior to the "*j $31*" instructions in the "*main*" module. Performance impact (% increase) is computed as: $100 * stall\_cycles/base\_cycles$. Stall cycles result from read buffer overflows. All instructions are assumed to require one cycle to complete in a pipelined architecture. This is a pessimistic assumption for performance impact measurement since load and branch delays would give the read buffer an extra cycle to handle an overflow. The assumption is made to again help isolate read buffer effects on performance from those of various delay slot filling strategies.

4 Simulation Results and Analysis

4.1 Implementation

The hazard analysis transformation operates on the s-code emitted by the MIPS code generator of the IMPACT C compiler [12]. The transformation determines which register reads need to be saved by the read buffer and inserts calls to the *initialization*, *simulation*, and *summary* procedures as described earlier. The resulting s-code modules are then compiled and run on a on a DECstation 3100. For the study, a rollback distance of 10 was selected. Given a rollback distance of 10, a read buffer size of 20 (for configurations A1, A2, and B1)

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

Table 1: Application Programs.

will produce zero performance impact.

4.2 Application Programs

Table 1 lists the 10 application programs studied. "Size" is the number of s-level instructions of the application prior to instrumentation.

4.3 Simulation Results: QUEEN

Figures 8 through 13 show changes in performance overhead (**Cycles OH**) for various read buffer sizes and configurations running the QUEEN application. Looking at Figure 8 (configuration A1), it can be seen that significant performance impact is incurred even with modest reduction in read buffer size. As can be seen from the other application runs, shown in Appendix A, configuration A1 is consistently the least efficient of the six configurations studied. This is due to the fact that the dual FIFO's are dedicated to a single source bus.

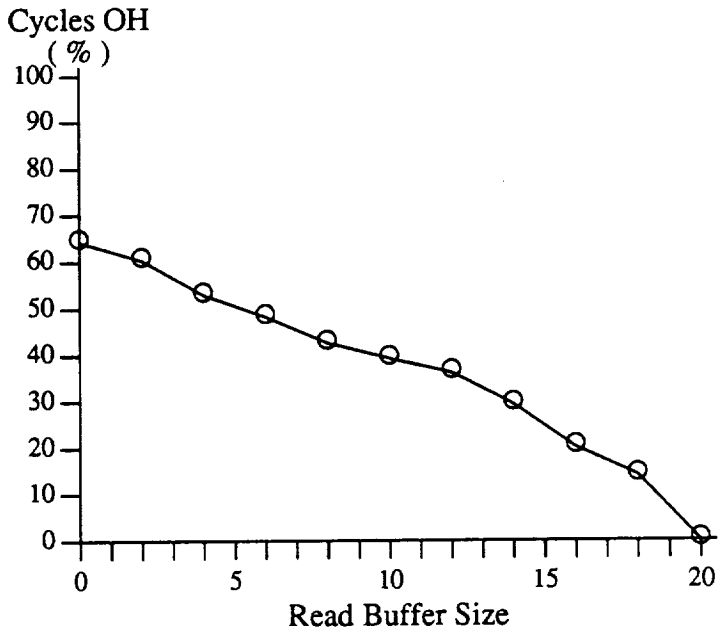


Figure 8: QUEEN: Configuration A1.

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. Figure 9 (configuration A2) shows the improvement gained by allowing either source bus access to either FIFO.

Figure 10 (configuration B1) shows the most efficient of the six configurations. In this configuration a total read buffer size of 13 would produce zero performance impact; a 35% reduction in read buffer size.

Configuration A2 out-performs configuration B1 at the lower buffer sizes but due to the gradual slope of the A2 curve versus the sharp drop-off of the B1 curve, B1 performs better at the low performance overhead values. This characteristic of configuration A2

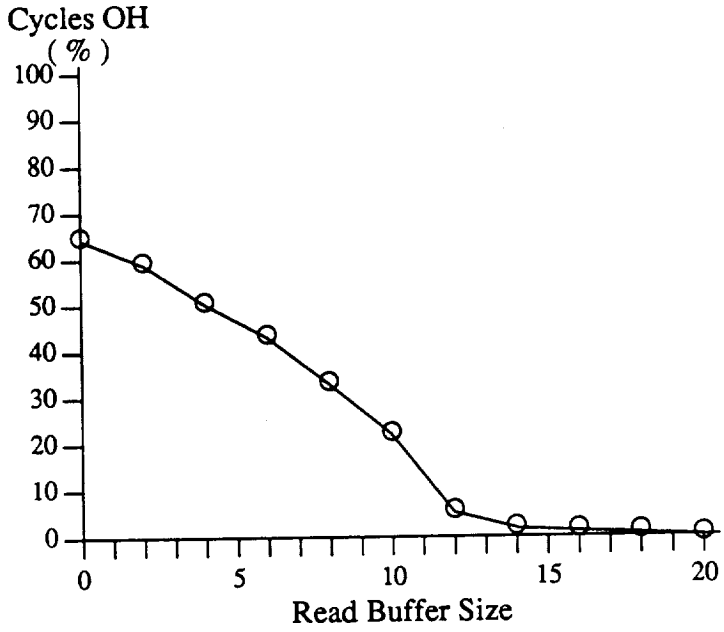


Figure 9: QUEEN: Configuration A2.

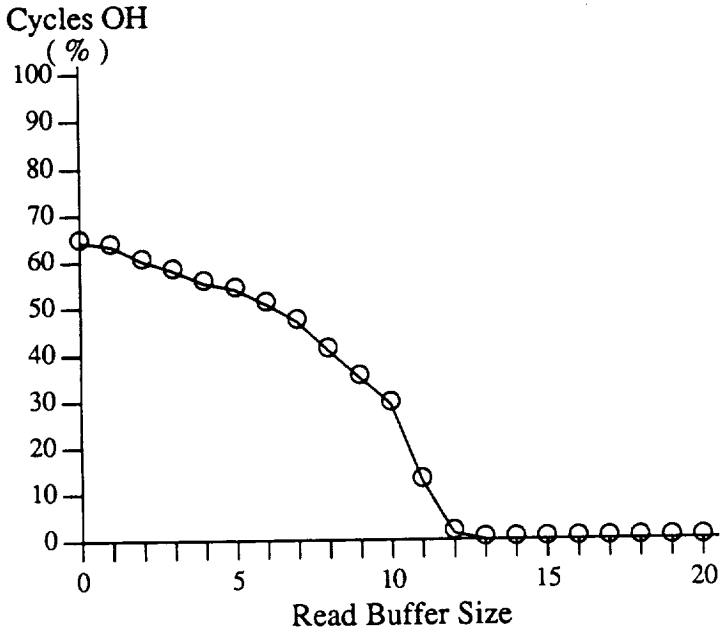


Figure 10: QUEEN: Configuration B1.

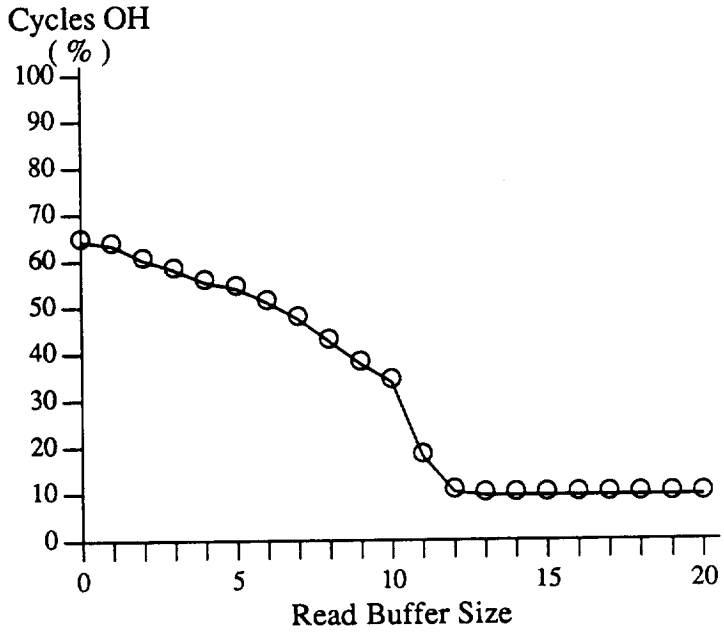


Figure 11: QUEEN: Configuration B2.

versus configuration B1 is present in all of the application results. It should be noted that configuration B1 does assume simultaneous saves of S1 and S2 can be handled within the same cycle. If this latter assumption is invalid, Figure 11 (configuration B2) shows that no less than 9.41% performance impact is achieved regardless of the read buffer size. The "leveling off" of Figure 11 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.

Figure 12 (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 two cycles. A non-zero minimum performance overhead is still present due to cases where the dual queue

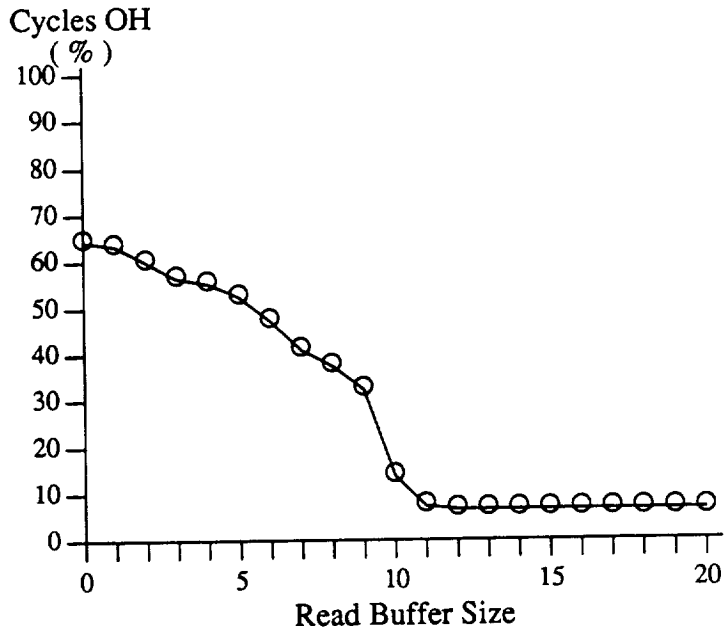


Figure 12: QUEEN: Configuration C.

has not emptied before the next simultaneous save occurs.

Figure 13 (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 (eg., S2 needs to be saved while the queue dedicated to S2 in configuration C is full and the other queue is empty). Configuration D also has a non-zero minimum performance overhead but gives better performance than configuration C.

The simulation results for QUEEN show that configuration A1 is the least efficient and that given the ability to do split-cycle-saves, configuration B1 is the most efficient. Without the split-cycle-save capability, configuration D is the best of the single FIFO designs resulting in a minimum performance overhead of 4.45% and configuration A2 is the best of the dual FIFO designs resulting in a 1.66% performance overhead with a read buffer size of 14. For

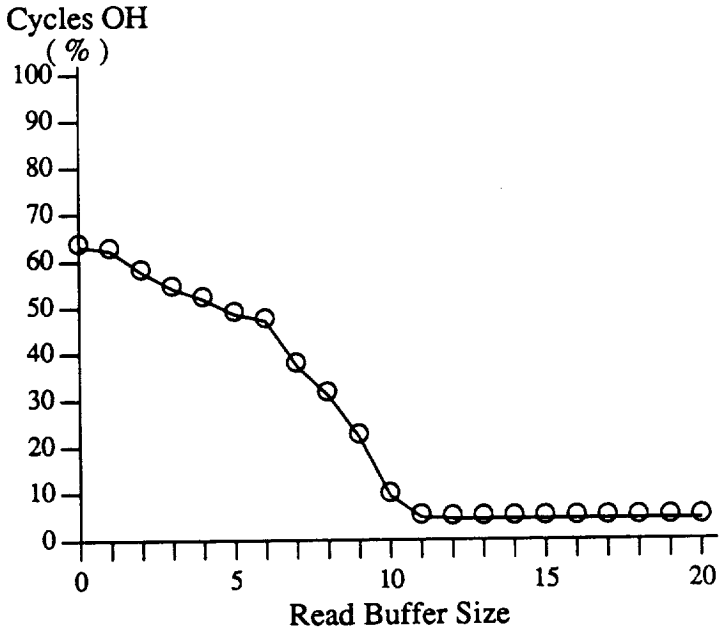


Figure 13: QUEEN: Configuration D.

For configurations B1, B2, C, and D a total read buffer size of 13 is sufficient to maximize performance (note that 2 must be added to each read buffer size value in C and D to account for the dual queues).

4.4 Simulation Results: Other Application Programs

Resulting plot shapes of the other application programs, shown in Appendix A, are similar to those for QUEEN. 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. The former measurement will be called *RB_size* and the latter *OH_level*. Table 2 shows measurements obtained for the 10 applications given the two most efficient configurations, A2 and B1. Configuration A2 does

	RB_size		*OH_level (%)*	
Program	A2	B1	A2	B1
QUEEN	14	13	1.67	0.00
WC	10	9	0.00	0.00
QSORT	16	15	2.28	0.00
CMP	12	11	0.00	0.00
GREP	10	10	0.18	0.00
PUZZLE	10	9	2.87	0.00
COMPRESS	12	13	2.87	0.00
LEX	12	13	2.73	0.00
YACC	16	15	1.07	0.00
CCCP	12	13	2.34	0.00

Table 2: Result Summary.

not level off like configuration D and does not rapidly approach zero like configuration B1. Instead, Configuration A2 gradually approaches zero. The *OH_level* measurement listed in Table 2 for configuration A2 therefore gives the first performance overhead value that is less than 3%, and it's corresponding *RB_size* value.

It can be seen from Table 2 that the read buffer size requirement is roughly the same, per application, regardless of the split-cycle-save assumption. The size requirement does vary quite a bit from application to application; PUZZLE and WC as small as 9 with QSORT and YACC as large as 15. One problem with designing the read buffer capacity to handle the majority of applications is the steepness of the curve (given configuration B1) as read buffer size decreases. For example, if the read buffer size is chosen at 13 (sufficient for all but QSORT and YACC), QSORT would run 15.55% slower and YACC would run 16.35% slower; an unacceptable impact given the minor hardware savings with a read buffer of size 13 versus 20. It is concluded that while the full $2N$ read buffer size is not required

(given the split-cycle-save assumption), nearly $2N$ is required to adequately handle most applications. It seems unlikely that such a small hardware cost reduction would outweigh the cost increase of additional logic to handle buffer overflows. Also, regardless of which buffer size (less than $2N$) is chosen, it is possible that an important application will suffer a significant performance impact similar to QSORT and YACC with a buffer size of 13. It is therefore advisable to use configuration A1 with a total read buffer size of $2N$. Note this is not a contradiction with the previous conclusion that configuration A1 is the least efficient, which is true only when the read buffer size is less than $2N$. With size $2N$, performance impact is always zero for configuration A1.

Given the previous conclusion, the split-cycle-save discussion becomes unnecessary. It should be noted however that significant performance impacts were measured in QUEEN, QSORT, COMPRESS, LEX, YACC, and CCCP regardless of read buffer size using configuration D. Also, using configuration A2, some performance impacts were measured given a similar read buffer size to that used for configuration B1. This result would indicate that if the split-cycle-save assumption were not valid, again only configuration A1 with a total read buffer size of $2N$ would be adequate.

5 Summary

When transient processor errors occur, multiple instruction retry can be an effective alternative to full checkpointing and rollback recovery. Multiple instruction retry within a sliding window of a few instructions can be implemented in parallel with concurrent error detection

for rapid recovery from transient processor errors. Hardware schemes such as reorder buffers, history buffers, future files, and micro-rollback differ in where the updated and old values reside, circuit complexity, CPU cycle times, and rollback efficiency.

In contrast to hardware schemes, a compiler-assisted approach to implementing multiple instruction retry has been recently developed in which a *read buffer* is used to resolve one type of hazard, reducing code growth, complilation time, and performance impact.

The $2N$ read buffer size requirement of the compiler-assisted approach is worst case, assuring data redundancy for all values required but also saving register reads unnecessarily. By adding extra bits in the operand field for source 1 and source 2 it becomes possible to design the read buffer to save only those values required, thus reducing the read buffer size requirement. The cost of the buffer size reduction is occasional overflows resulting in stall cycles. This study determined the effect on performance of a DECstation 3100 running 10 application programs using 6 read buffer configurations.

Simulation was perform by inserting a hand-coded s-level instruction sequence prior to each original instruction. The hand-coded sequence then calls another hand-coded sequence which saves all registers and subsequently calls the simulation procedure. The simulation procedure was written in the C programming language and is compiled separately, allowing modification without disturbing the instrumented application programs. Each instrumented application program was run for various read buffer sizes and 6 read buffer configurations. Performance impact was measured by the number of stall cycles versus the number of base cycles.

Results show that configurations A2 and B1 were the most efficient and differed depending on whether split-cycle-saves were assumed. Performance versus read buffer size plots, by configuration, for the 10 application programs were seen to be of the same shape. There was however significant variances between the buffer sizes required for minimum performance impacts between applications, and the performance stabilization value assuming no split-cycle-save capability. It was determined that while the full $2N$ read buffer size is not required, nearly $2N$ is required to adequately handle most applications. Also, regardless of which buffer size is chosen (i.e., less than $2N$), it is possible that an important application will suffer a significant performance impact. This study therefore concludes that no reduction in read buffer size below $2N$ is practical given a wide variety of general applications.

Appendix A

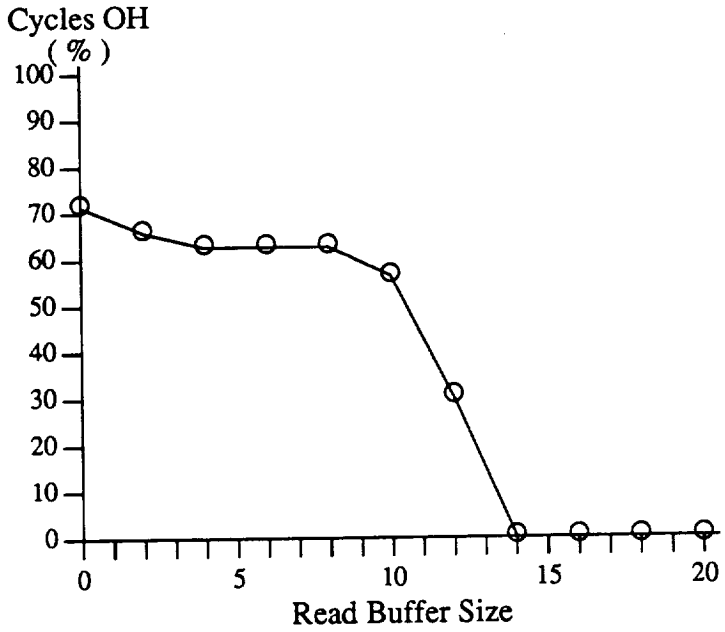


Figure 14: WC: Configuration A1.

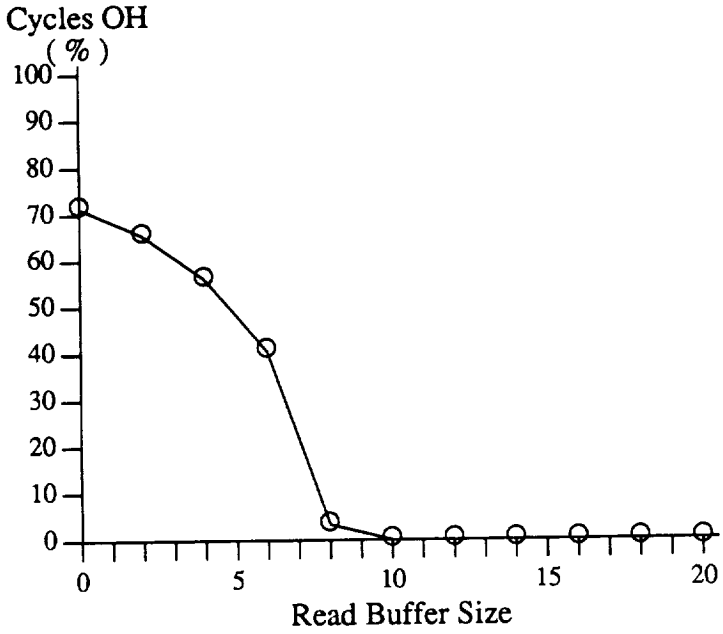


Figure 15: WC: Configuration A2.

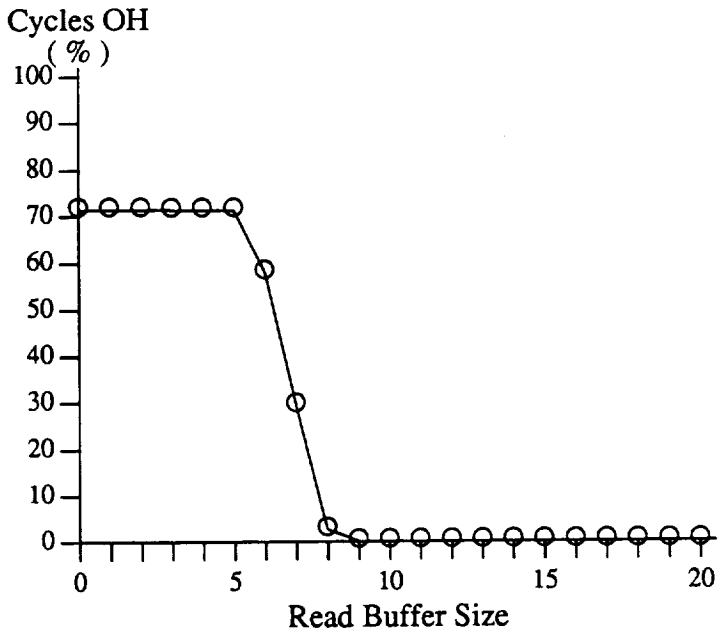


Figure 16: WC: Configuration B1.

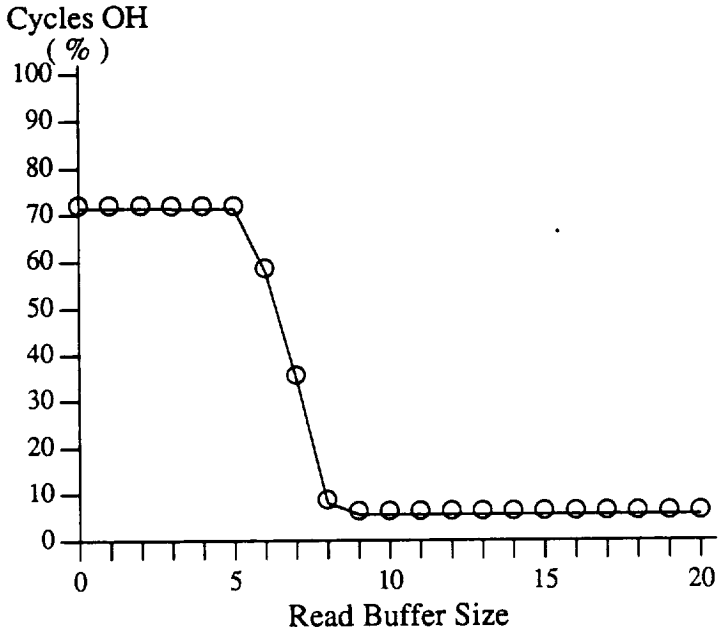


Figure 17: WC: Configuration B2.

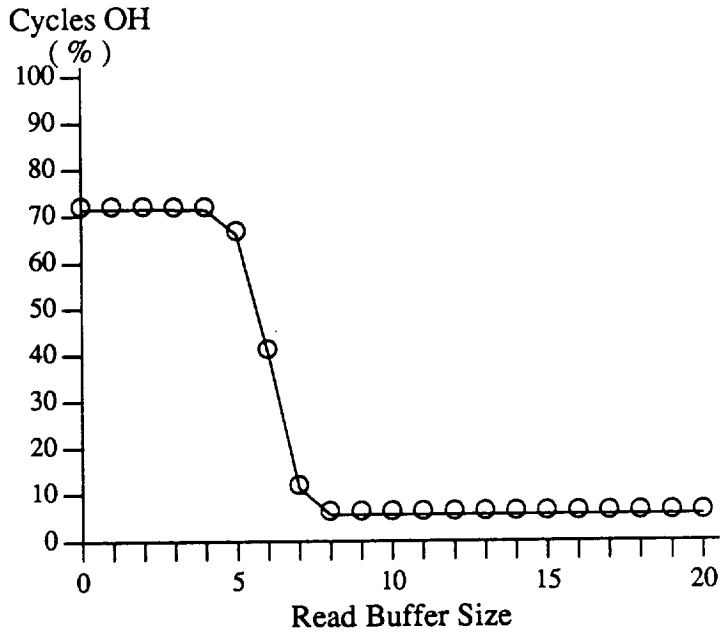


Figure 18: WC: Configuration C.

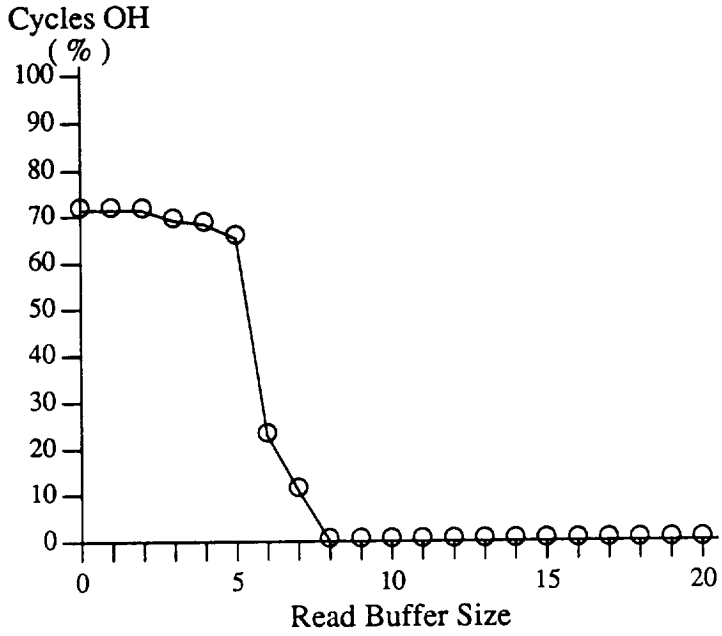


Figure 19: WC: Configuration D.

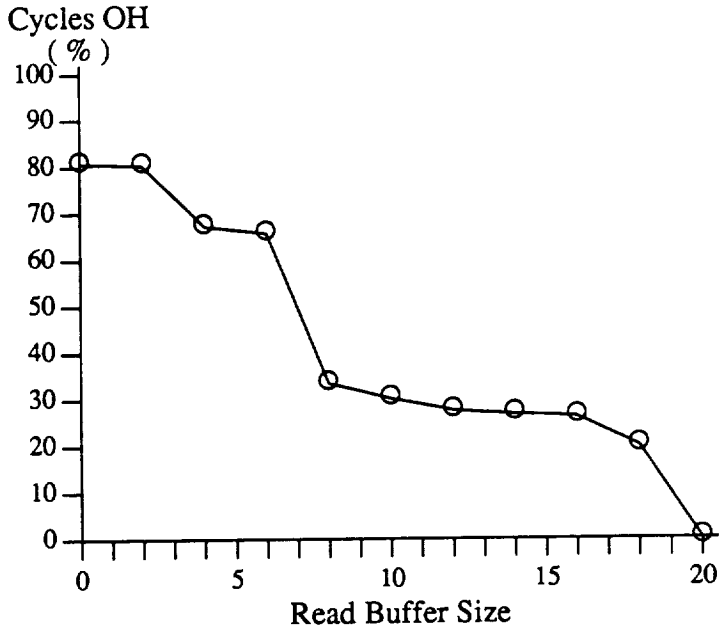


Figure 20: QSORT: Configuration A1.

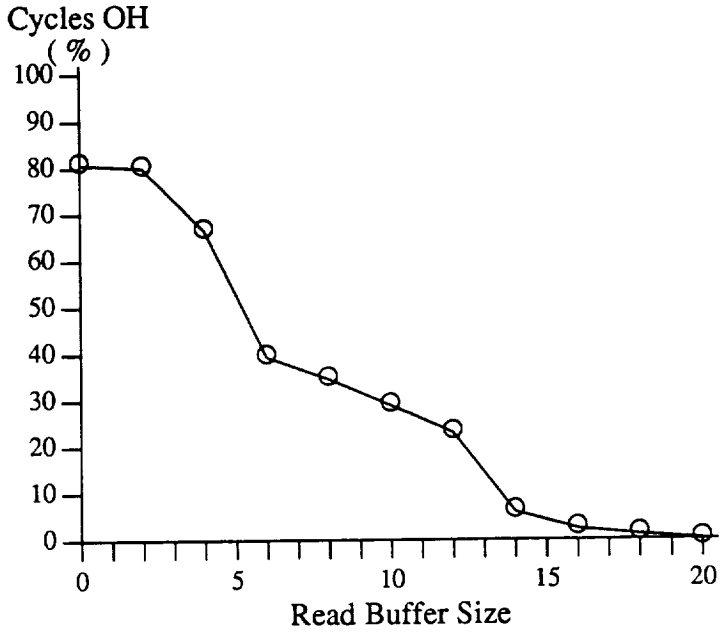


Figure 21: QSORT: Configuration A2.

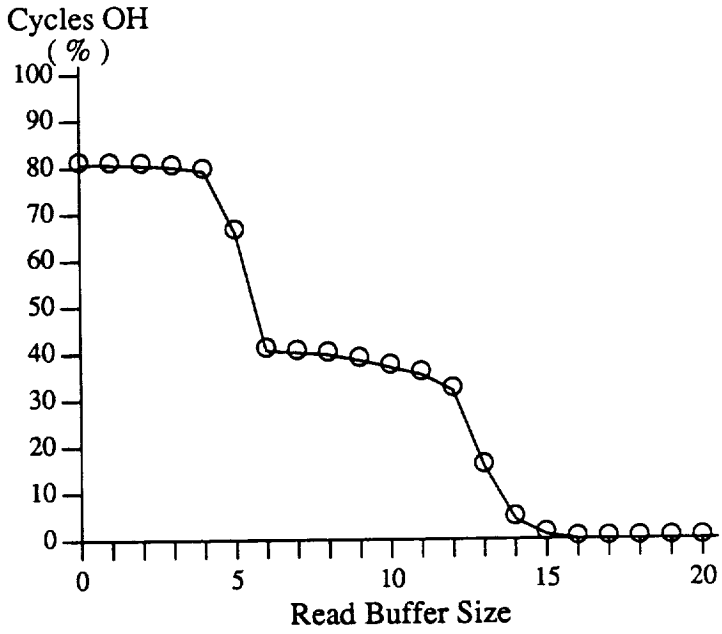


Figure 22: QSORT: Configuration B1.

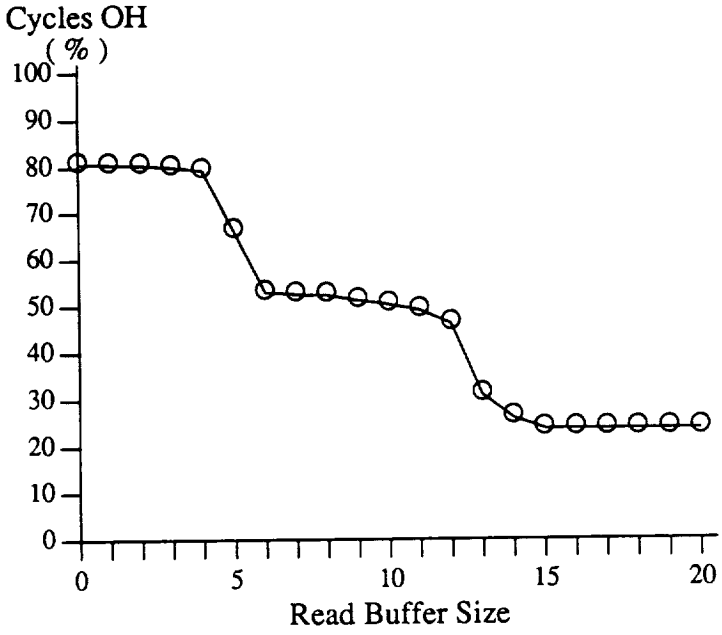


Figure 23: QSORT: Configuration B2.

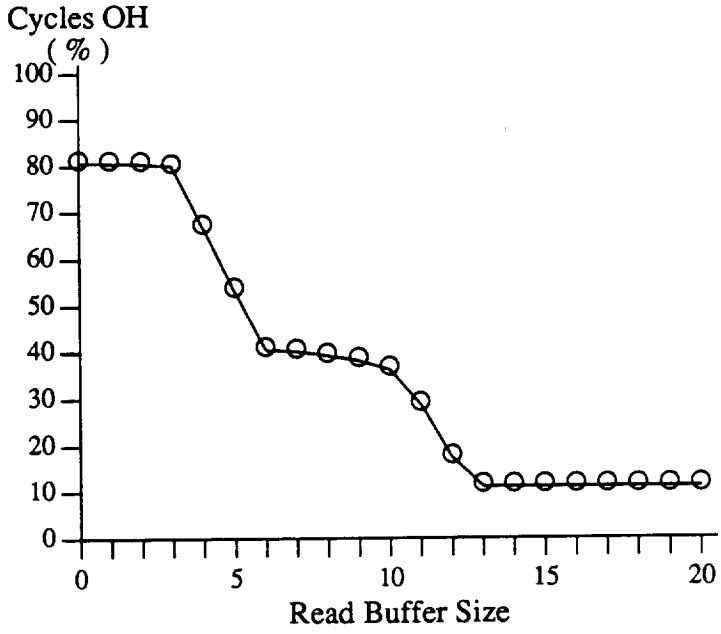


Figure 24: QSORT: Configuration C.

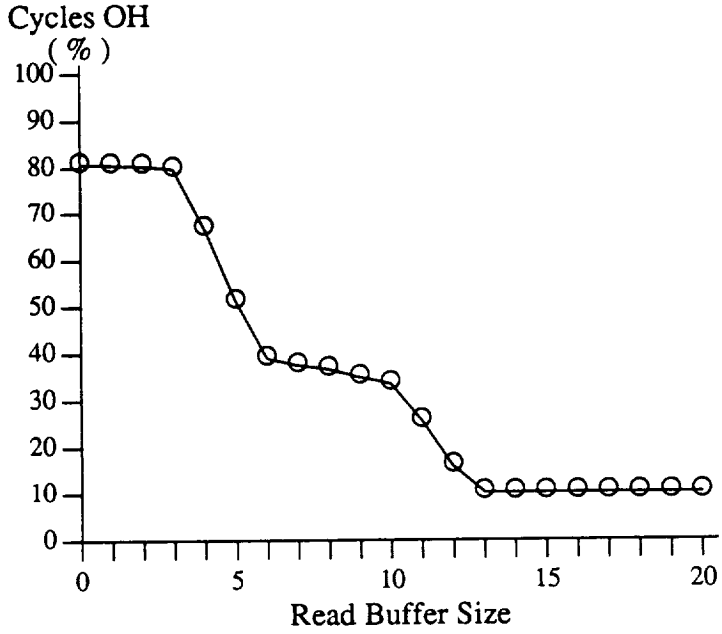


Figure 25: QSORT: Configuration D.

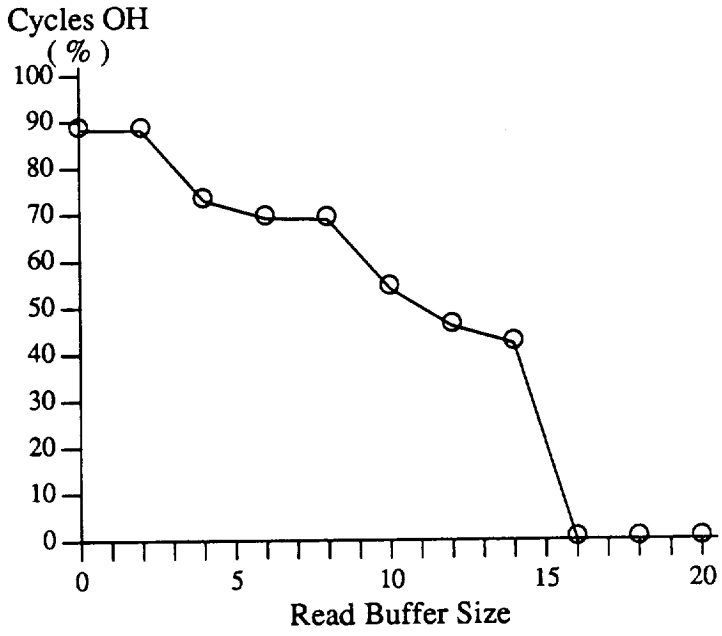


Figure 26: CMP: Configuration A1.

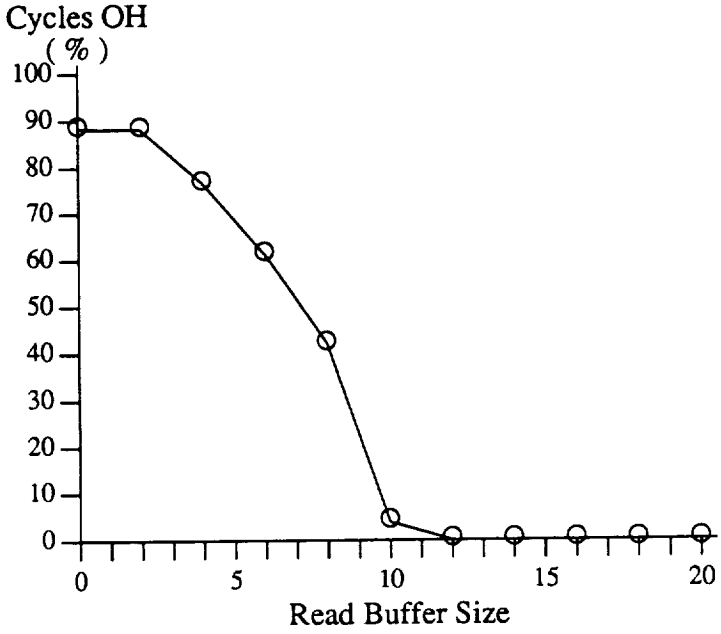


Figure 27: CMP: Configuration A2.

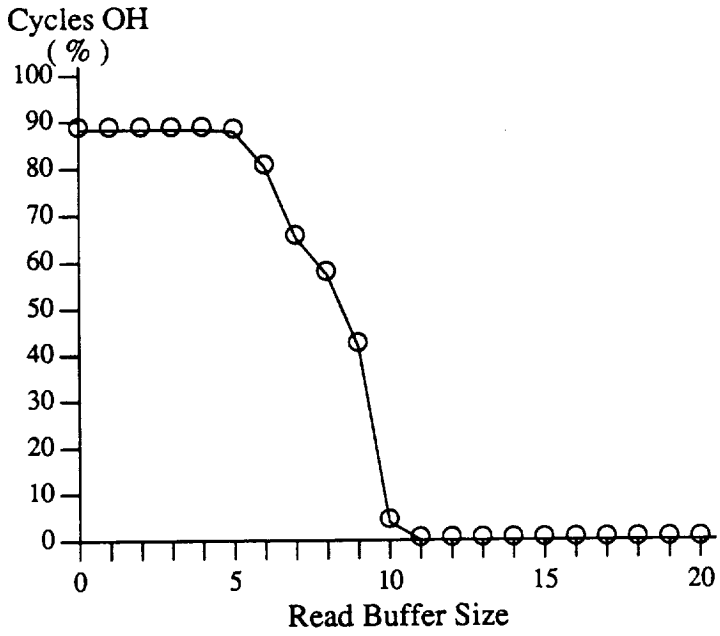


Figure 28: CMP: Configuration B1.

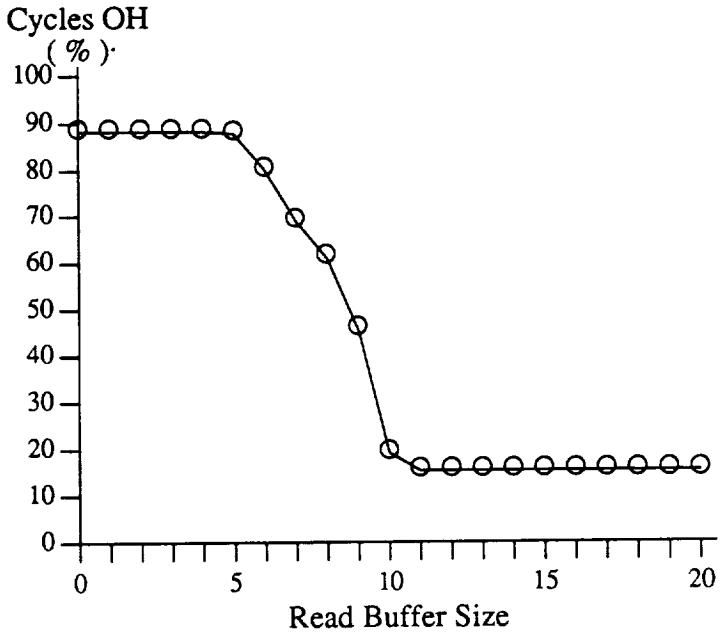


Figure 29: CMP: Configuration B2.

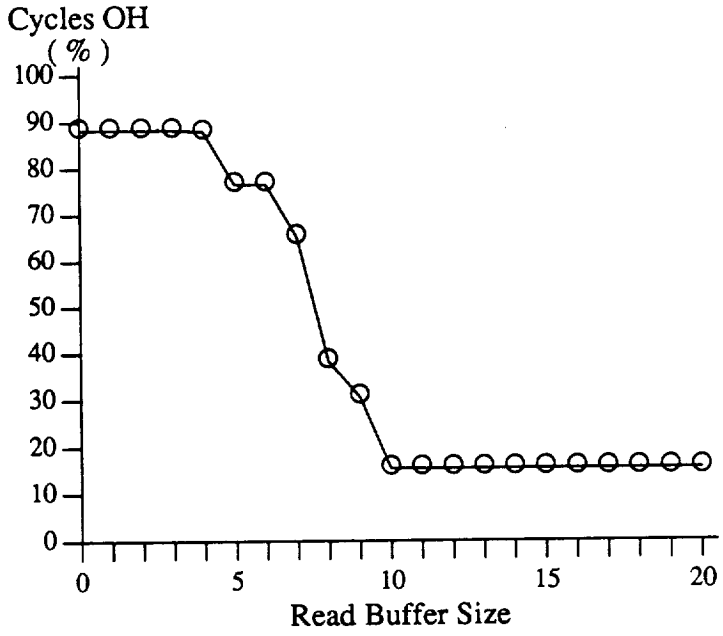


Figure 30: CMP: Configuration C.

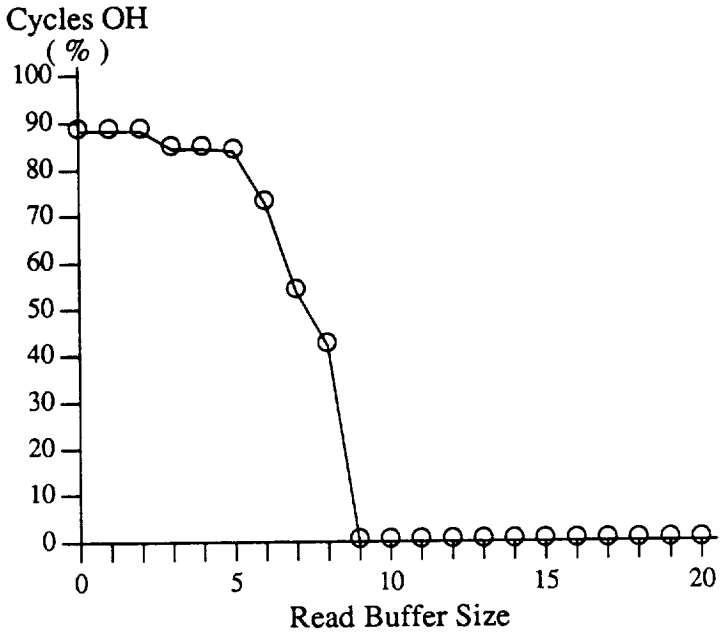


Figure 31: CMP: Configuration D.

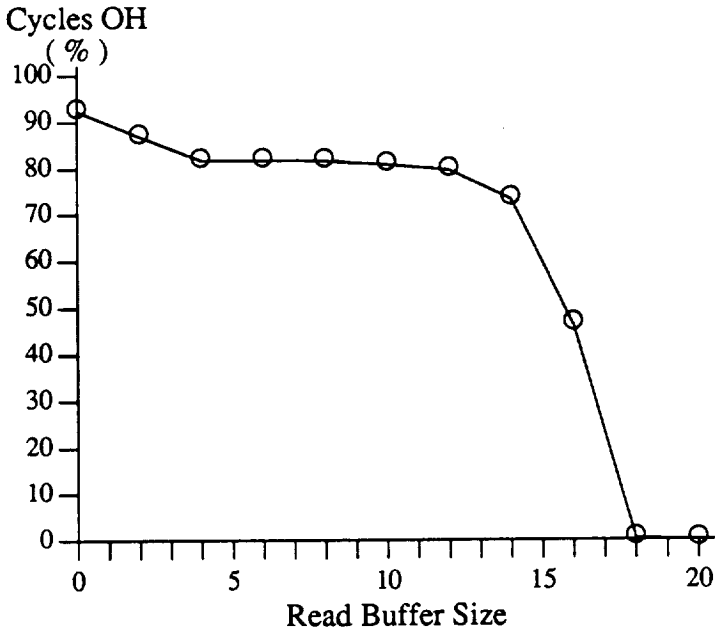


Figure 32: GREP: Configuration A1.

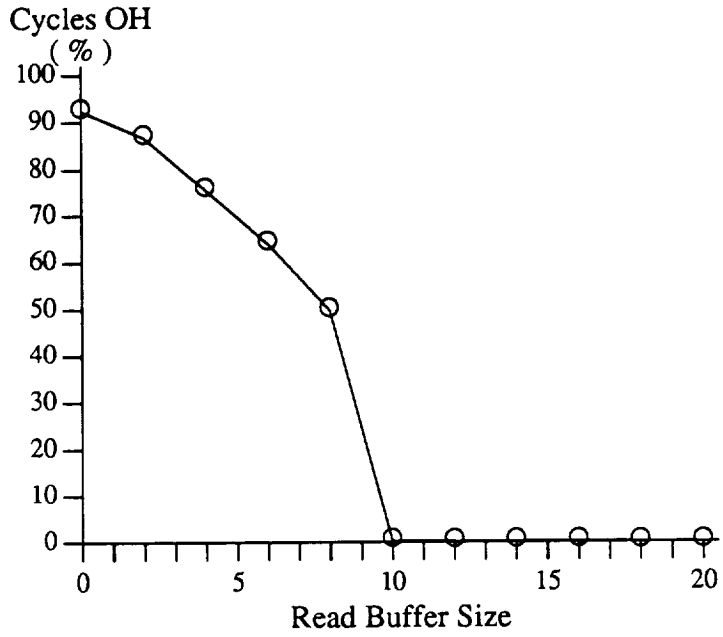


Figure 33: GREP: Configuration A2.

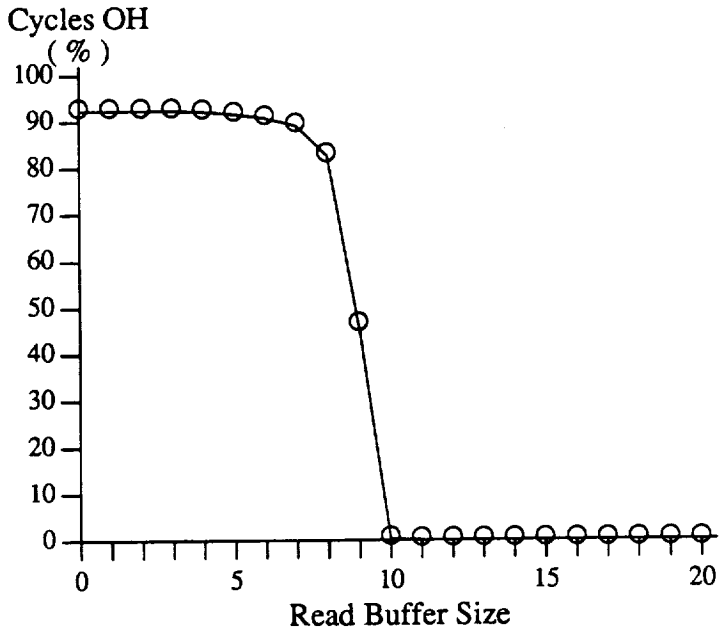


Figure 34: GREP: Configuration B1.

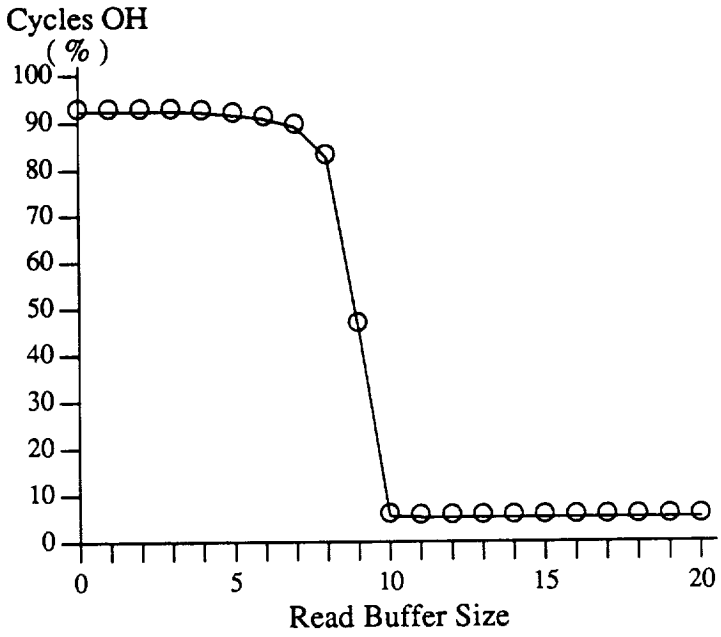


Figure 35: GREP: Configuration B2.

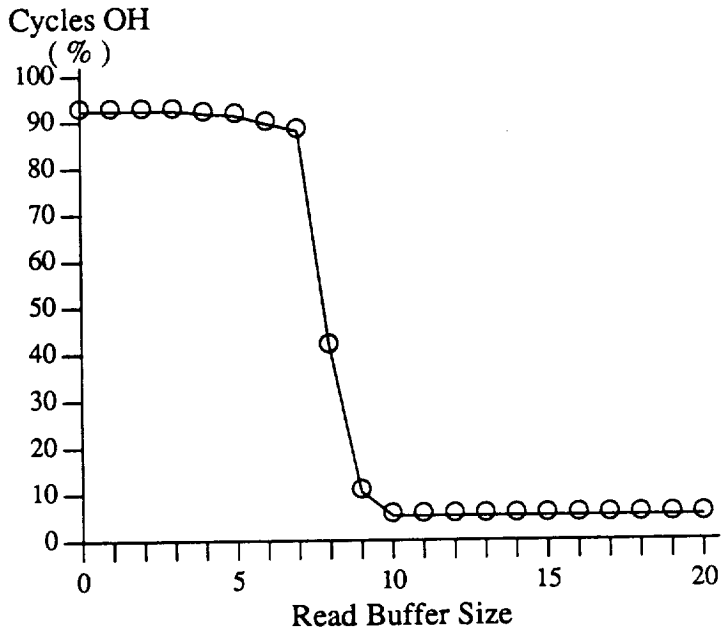


Figure 36: GREP: Configuration C.

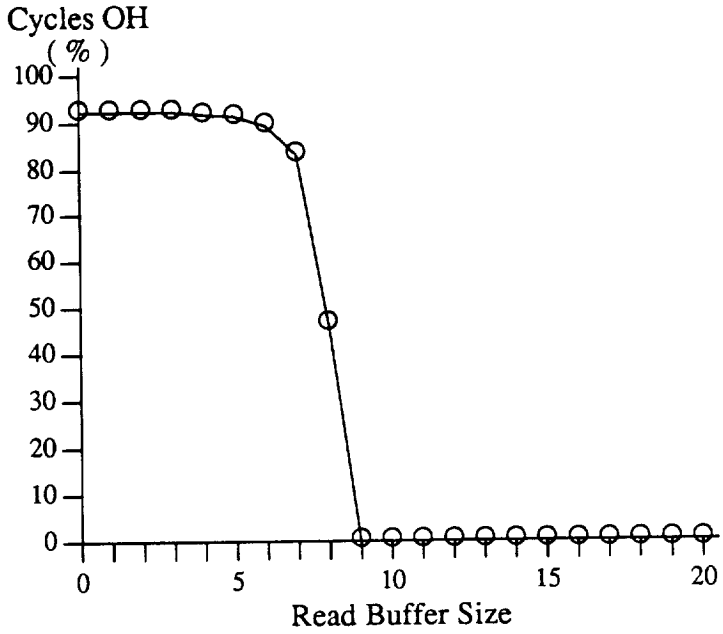


Figure 37: GREP: Configuration D.

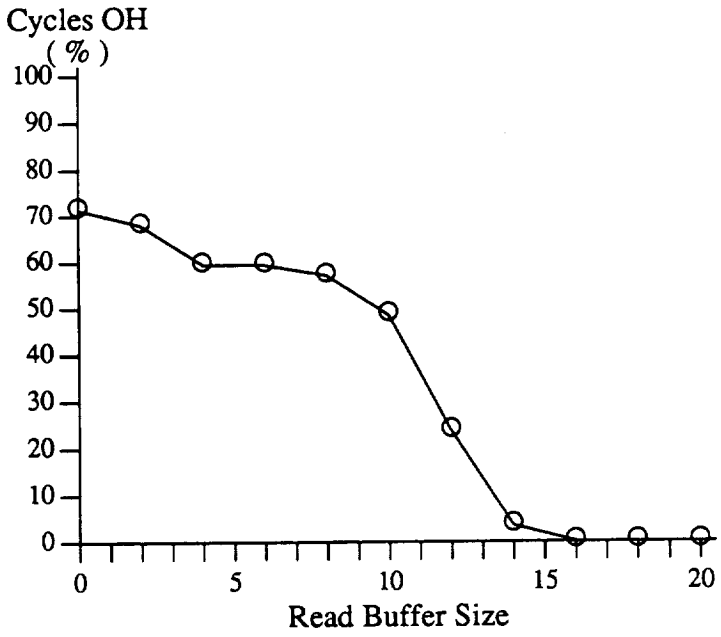


Figure 38: PUZZLE: Configuration A1.

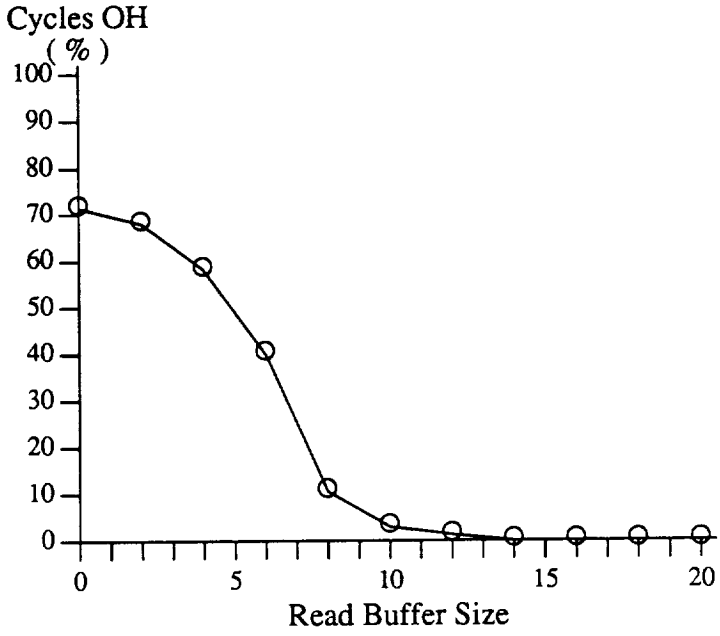


Figure 39: PUZZLE: Configuration A2.

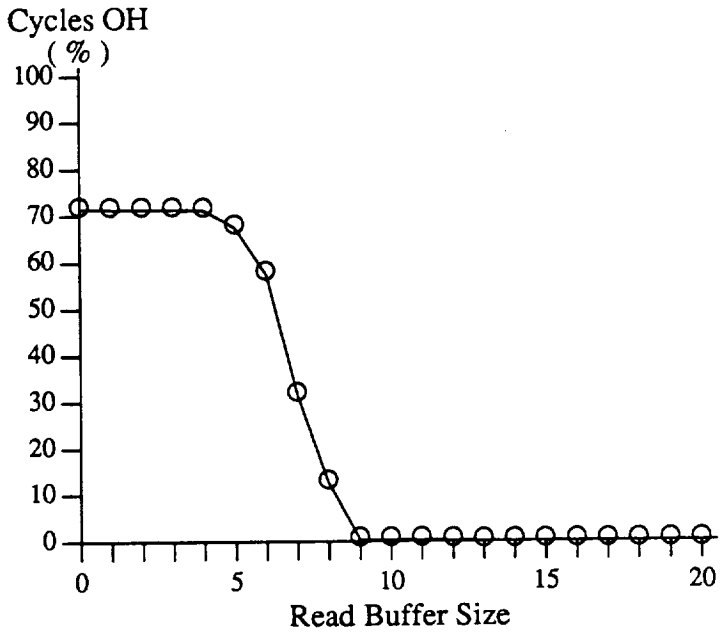


Figure 40: PUZZLE: Configuration B1.

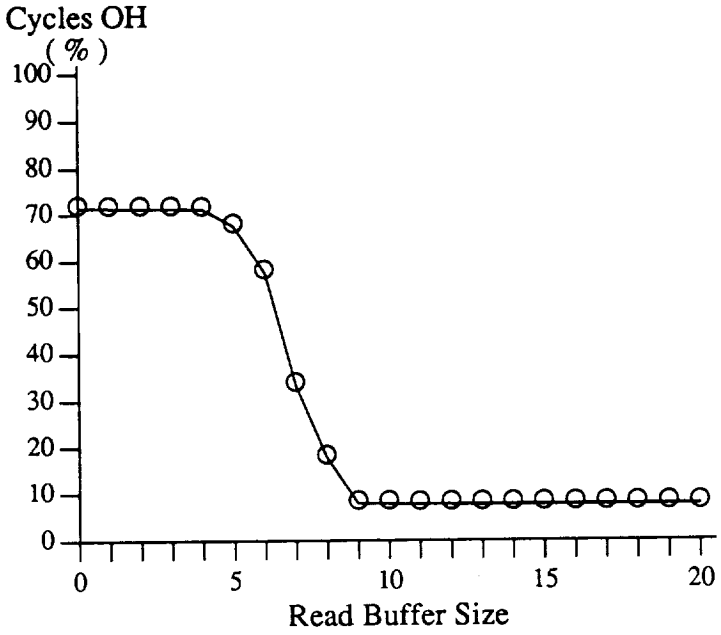


Figure 41: PUZZLE: Configuration B2.

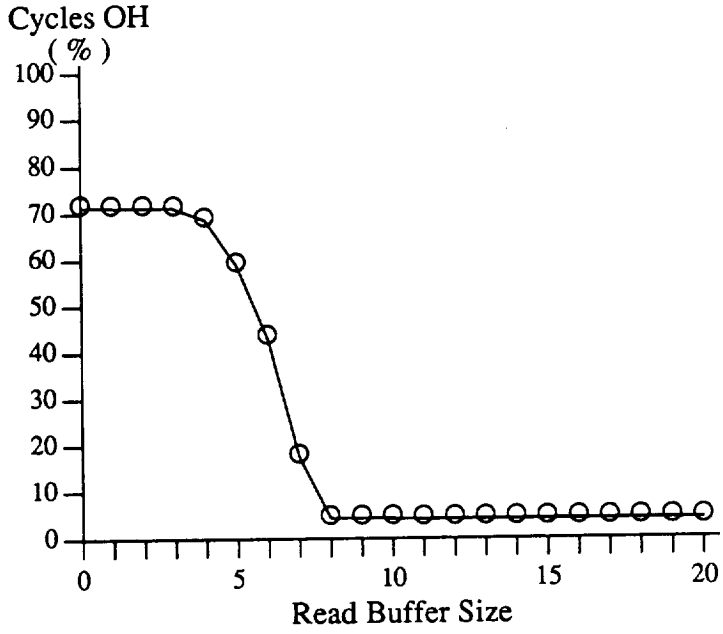


Figure 42: PUZZLE: Configuration C.

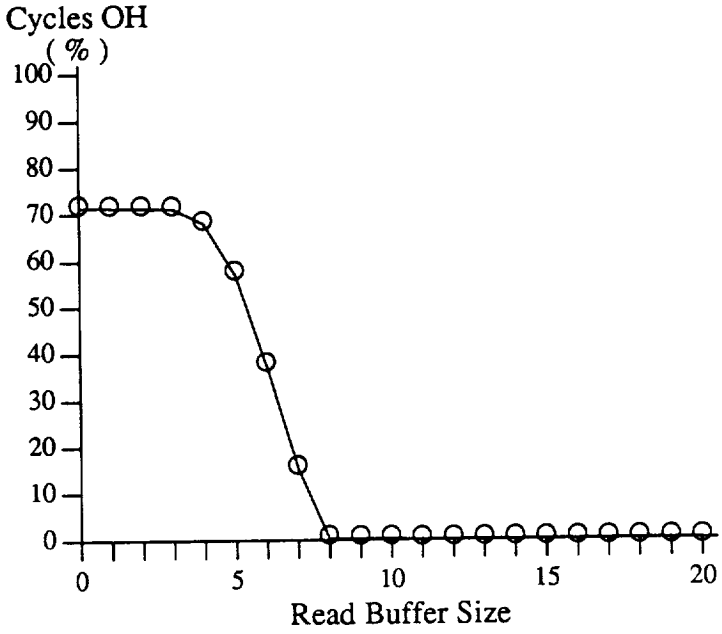


Figure 43: PUZZLE: Configuration D.

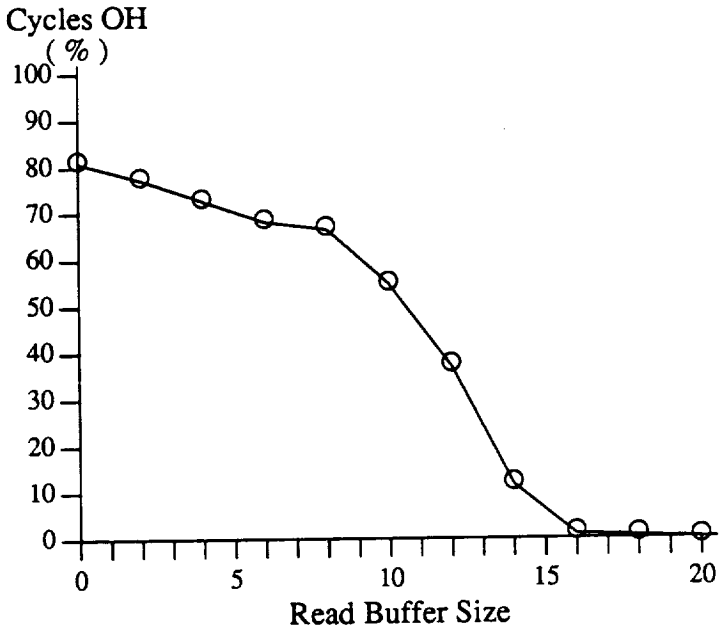


Figure 44: COMPRESS: Configuration A1.

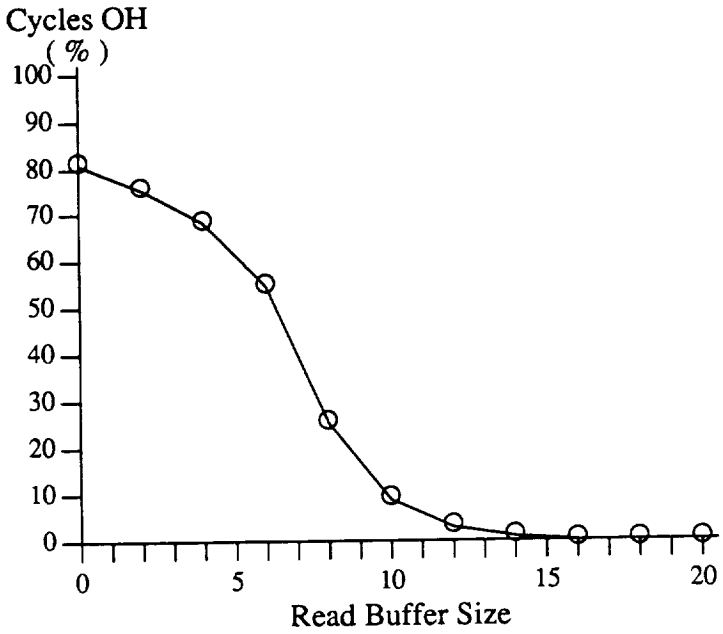


Figure 45: COMPRESS: Configuration A2.

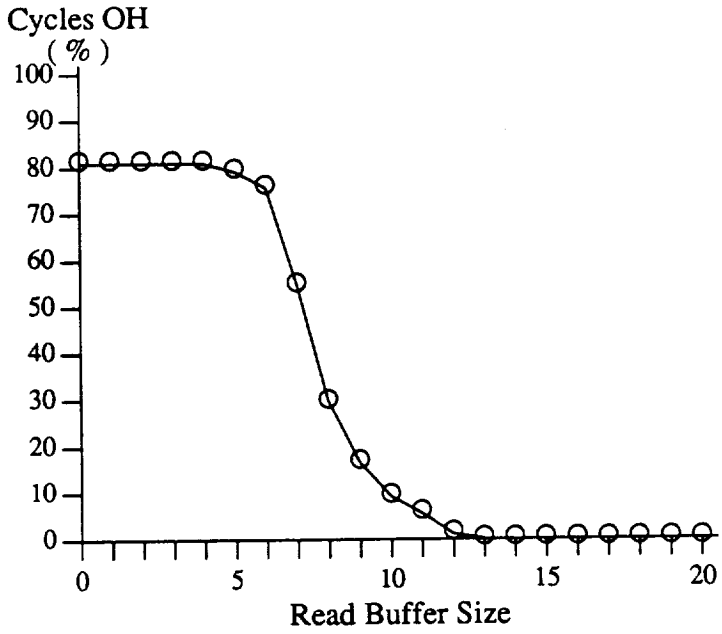


Figure 46: COMPRESS: Configuration B1.

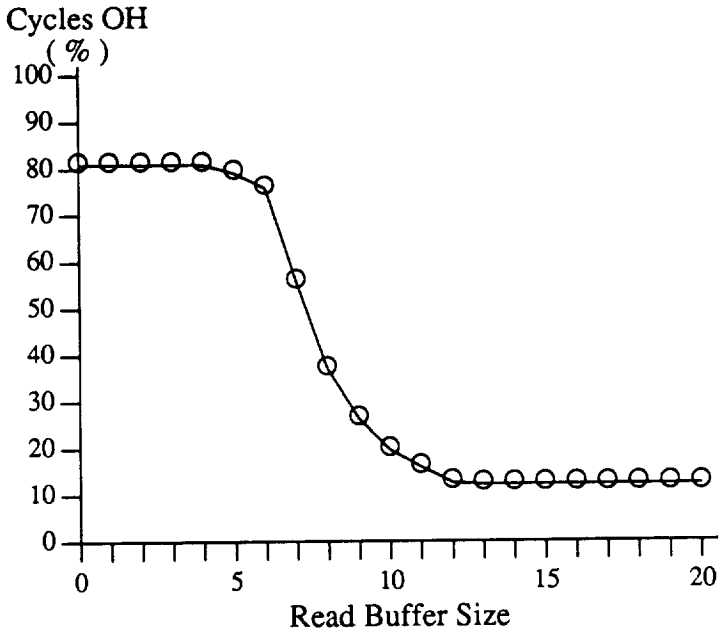


Figure 47: COMPRESS: Configuration B2.

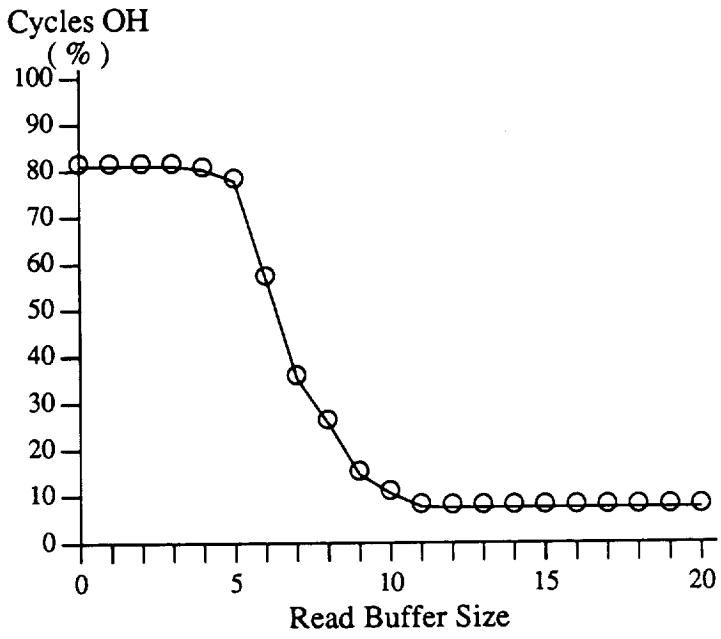


Figure 48: COMPRESS: Configuration C.

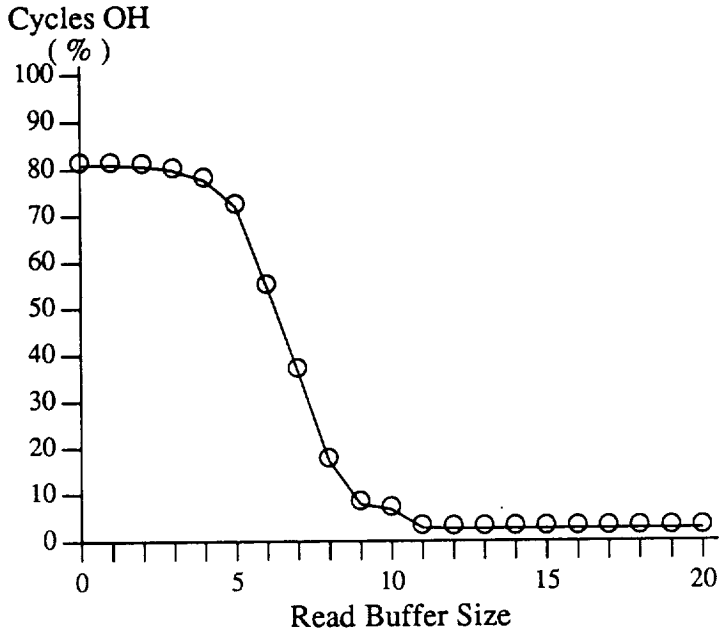


Figure 49: COMPRESS: Configuration D.

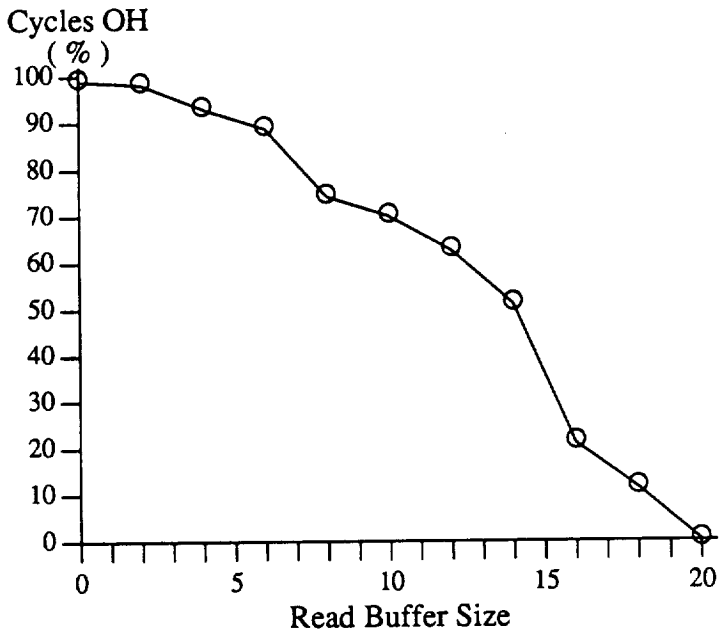


Figure 50: LEX: Configuration A1.

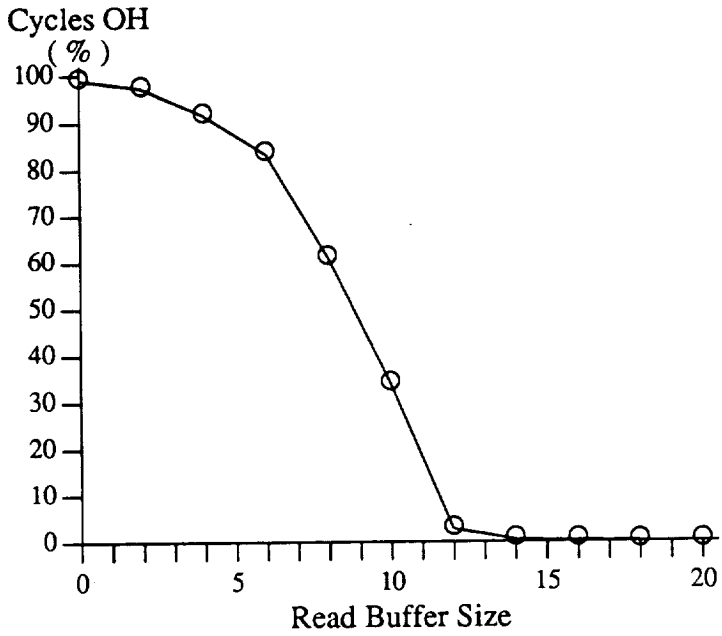


Figure 51: LEX: Configuration A2.

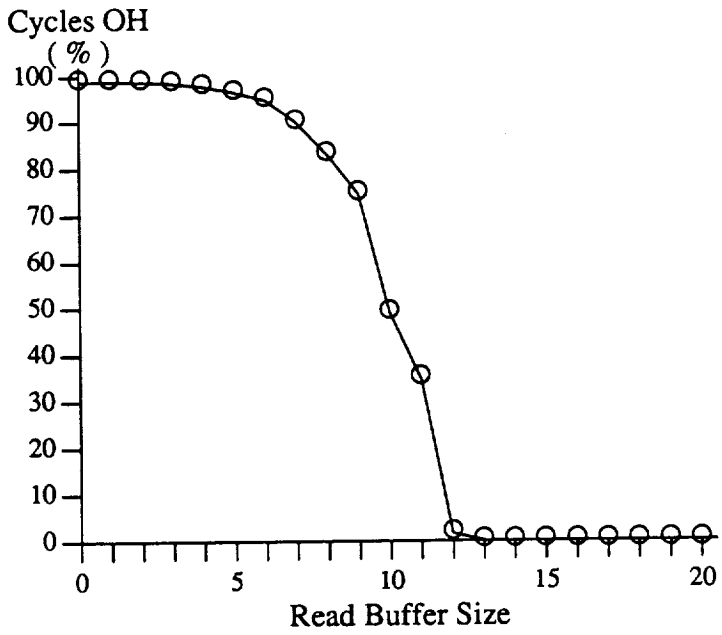


Figure 52: LEX: Configuration B1.

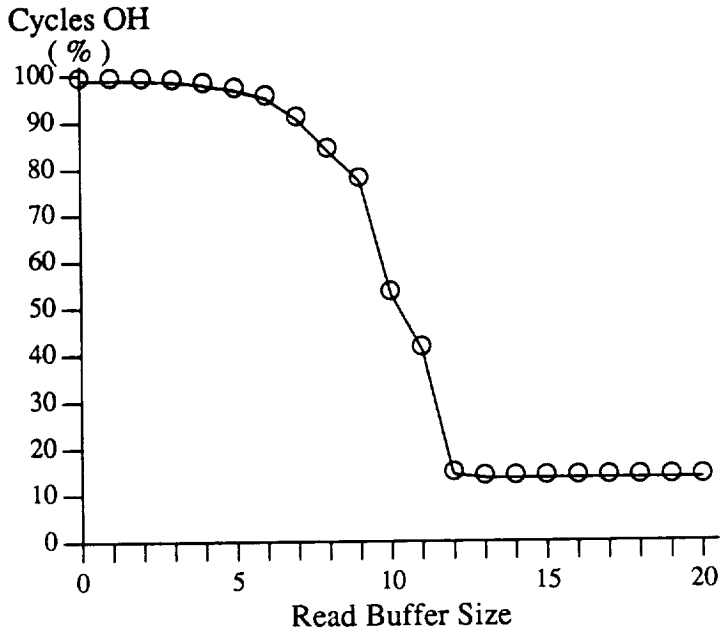


Figure 53: LEX: Configuration B2.

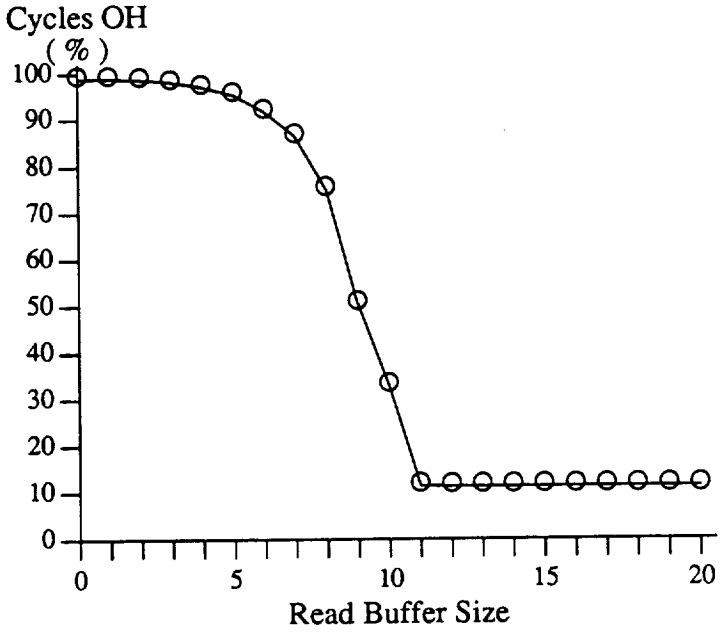


Figure 54: LEX: Configuration C.

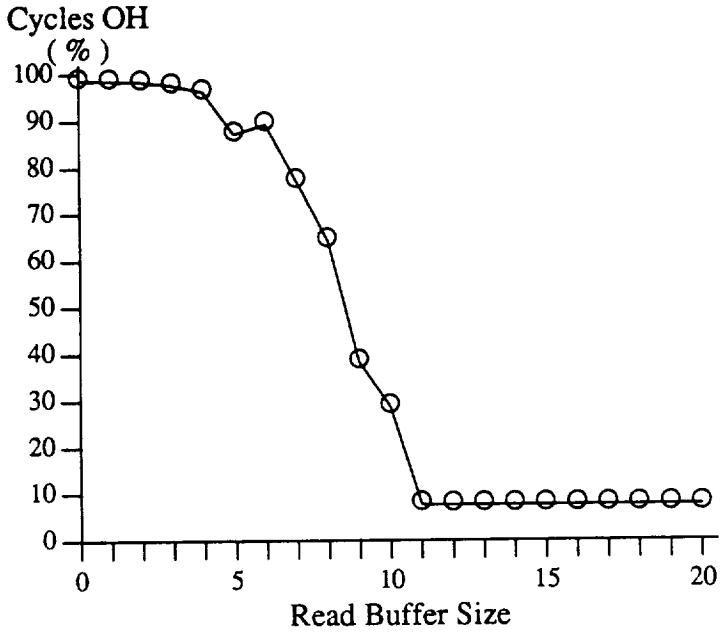


Figure 55: LEX: Configuration D.

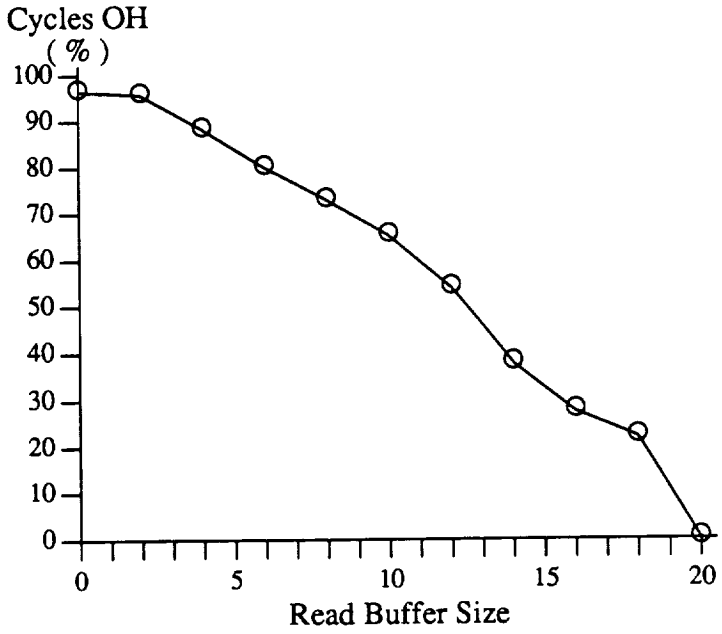


Figure 56: YACC: Configuration A1.

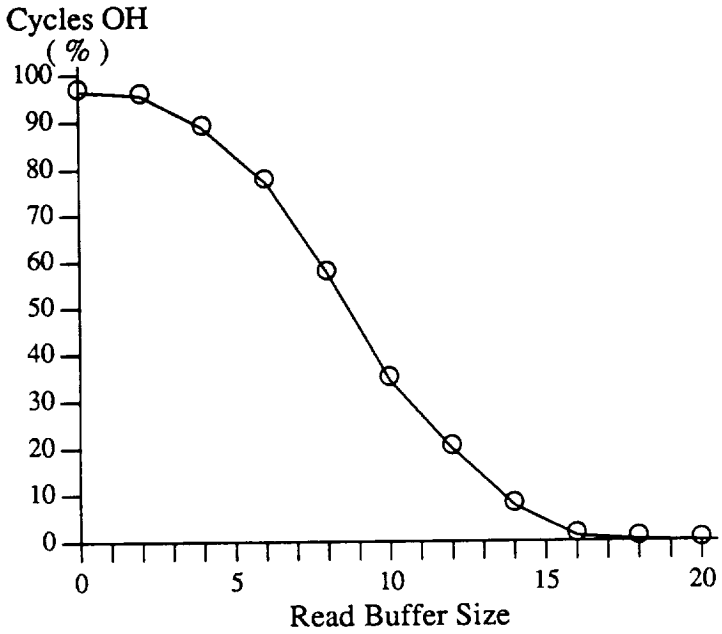


Figure 57: YACC: Configuration A2.

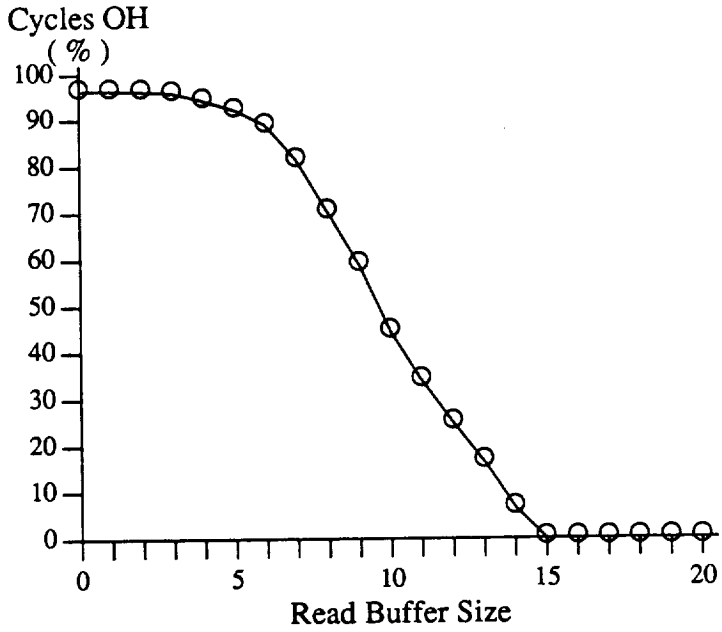


Figure 58: YACC: Configuration B1.

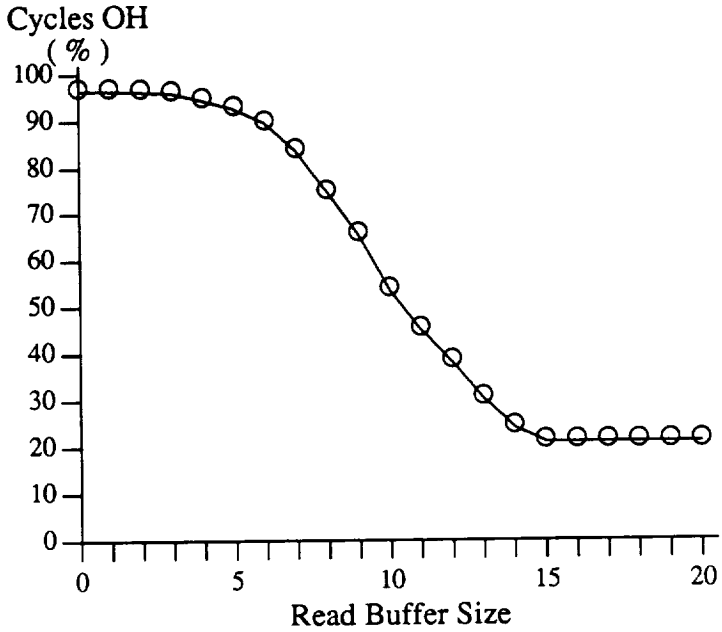


Figure 59: YACC: Configuration B2.

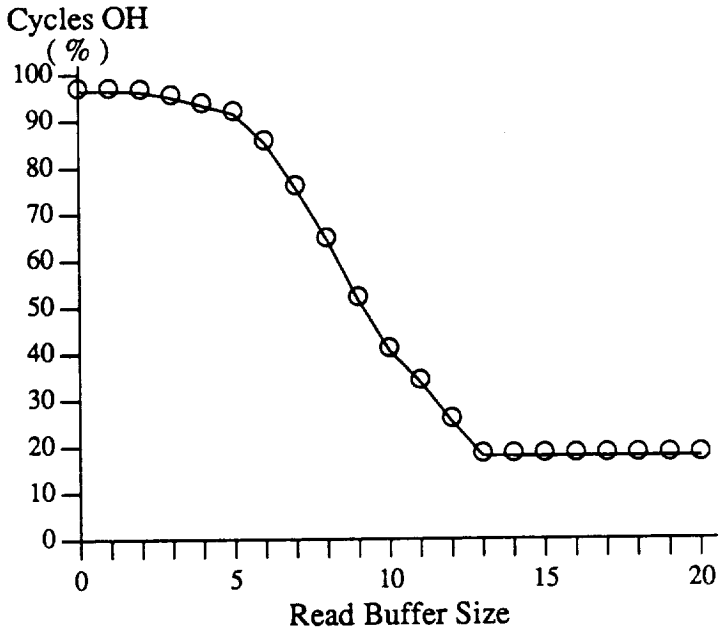


Figure 60: YACC: Configuration C.

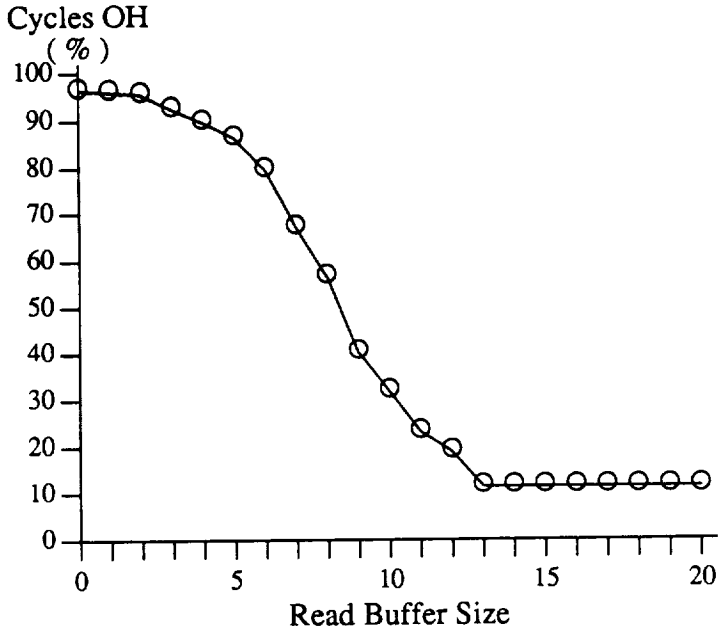


Figure 61: YACC: Configuration D.

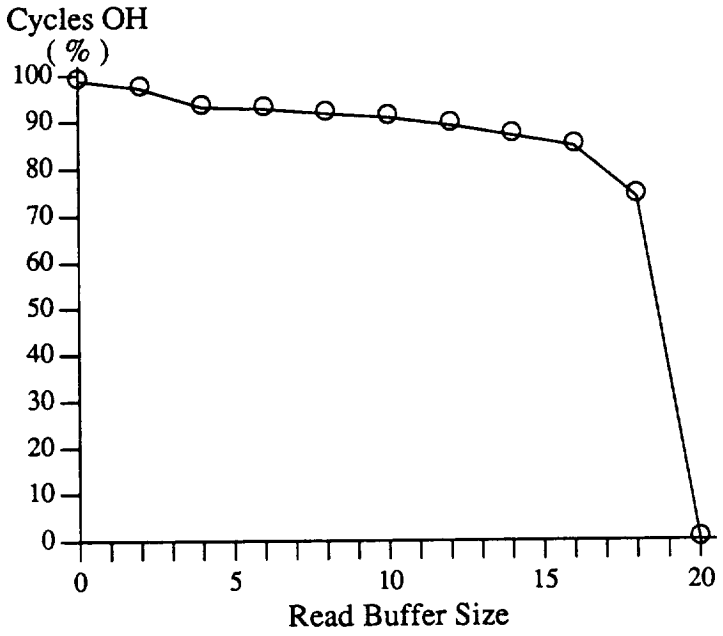


Figure 62: CCCP: Configuration A1.

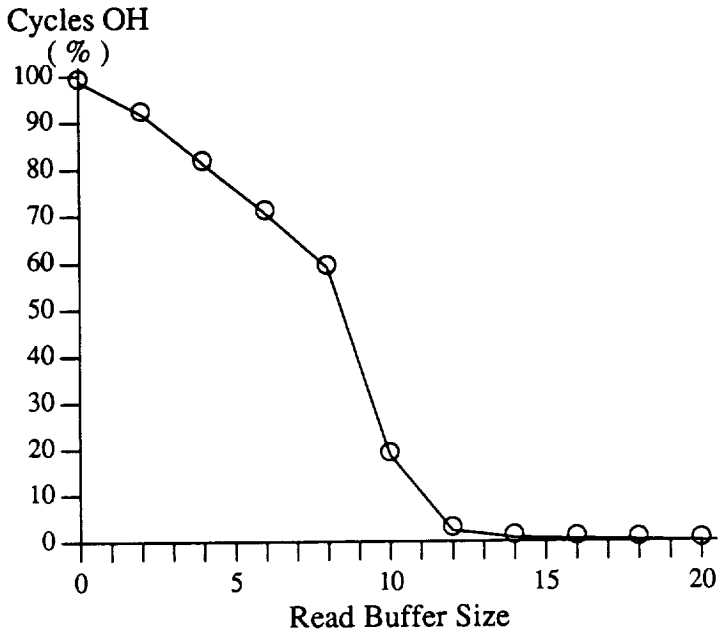


Figure 63: CCCP: Configuration A2.

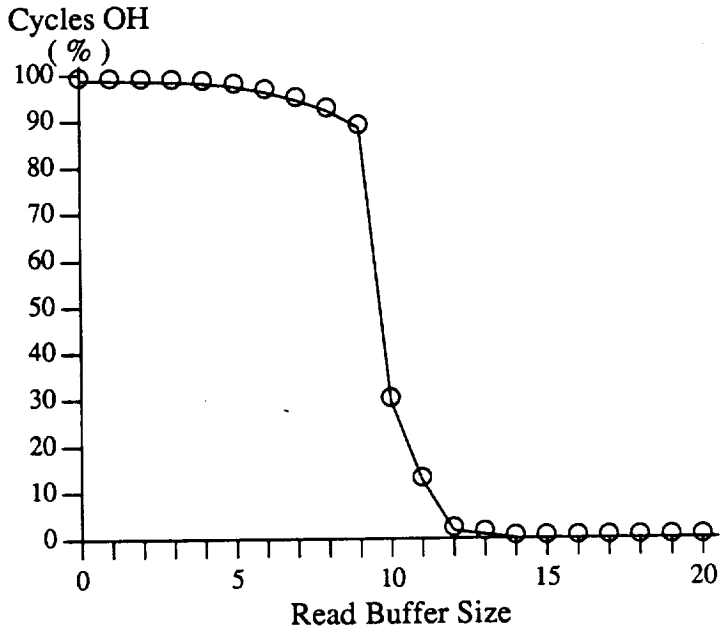


Figure 64: CCCP: Configuration B1.

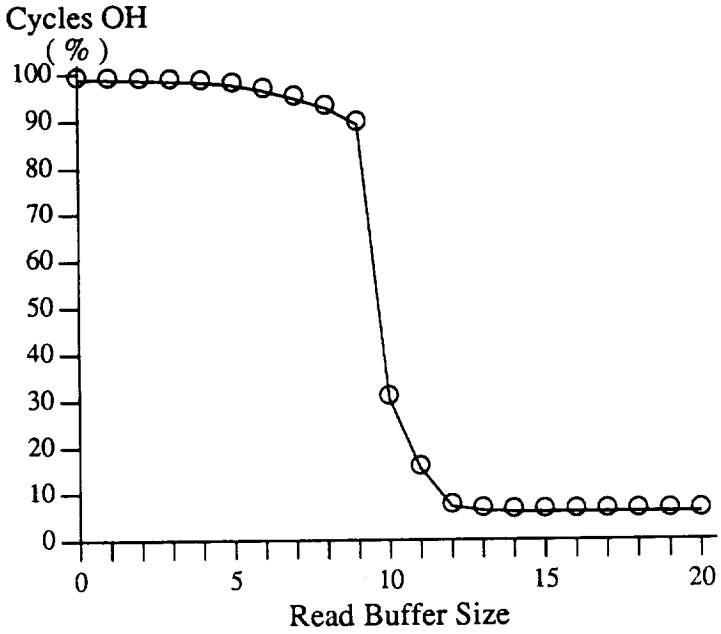


Figure 65: CCCP: Configuration B2.

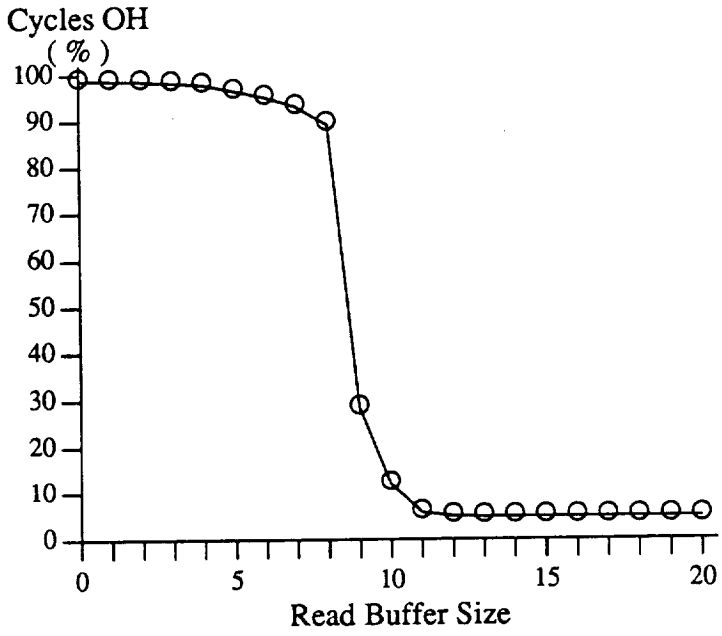


Figure 66: CCCP: Configuration C.

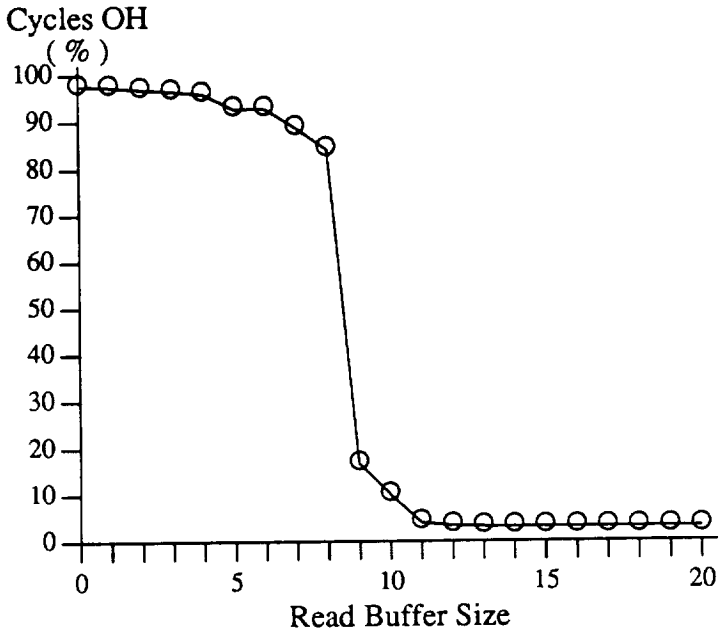


Figure 67: CCCP: Configuration D.

References

[1] 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*, pages 66–73, July 1992.

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

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

[4] 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*, pages 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] C-C. J. Li, S-K. Chen, W. K. Fuchs, and W-M. W. Hwu. Compiler-Assisted Multiple Instruction Retry. Technical Report CRHC-91-31, Coordinated Science Laboratory, University of Illinois, May 1991.

[7] 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.

[8] 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.

[9] Y. Tamir, M. Liang, T. Lai, and M. Tremblay. The UCLA Mirror Processor: A Building Block for Self-Checking Self-Repairing Computing Nodes. In *The Twenty-First International Symposium on Fault-Tolerant Computing*, pages 178–185, June 1991.

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

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

[12] P.P Chang, W.Y. Chen, N.J. Warter, and W-M. W. Hwu. IMPACT: An Architecture Framework for Multiple-Instruction-Issue Processors. In *Conference Proceedings of the 18th Annual International Symposium on Computers*, pages 266–275, May 1991.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-92-2239 CRHC-92-21	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION NASA
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Moffitt Field, CA
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 7a 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 7b	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
Simulation and Analysis of Support Hardware for Multiple Instruction Rollback

12. PERSONAL AUTHOR(S)
ALEWINE, Neil J.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1992 OCT 10	51

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	instruction retry, compilers, hardware assisted retry

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

Recently, a compiler-assisted approach to multiple instruction retry was developed. In this scheme a *read buffer* of size $2N$, where N represents the maximum instruction rollback distance, is used to resolve one type of data hazard. This hardware support helps to reduce code growth, complilation time, and some of the performance impacts associated with hazard resolution.

The $2N$ read buffer size requirement of the compiler-assisted approach is worst case, assuring data redundancy for all data required but also providing some unnecessary redundancy. By adding extra bits in the operand field for source 1 and source 2 it becomes possible to design the read buffer to save only those values required, thus reducing the read buffer size requirement.

This study measures the effect on performance of a DECstation 3100 running 10 application programs using 6 read buffer configurations at varying read buffer sizes.

Continued on back

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL