The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

PERFORMANCE-RELIABILITY TRADEOFFS IN DESIGNING RE-ORDER BUFFERS

A Thesis in

Computer Science and Engineering

by

Srinath Sridharan

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2008

The thesis of Srinath Sridharan was reviewed and approved* by the following:

Vijaykrishnan Narayanan
Professor of Computer Science and Engineering
The Pennsylvania State University
Thesis Advisor

Chita R. Das
Professor of Computer Science and Engineering,
The Pennsylvania State University

Raj Acharya
Professor of Computer Science and engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School

ABSTRACT

Recent research efforts to enable quantitative analysis of the architectural transient fault-tolerance solutions has led to a key metric named Architectural Vulnerability Factor (AVF). AVF quantifies the relative contributions of the key processor structures to the overall processor error rate. In this paper, we propose a new design of Re-Order Buffer (ROB), a key structure in superscalar architectures, to reduce its AVF without disrupting the AVF of other structures. This new design, which is employed using a technique named Late Instruction Binding (LIB), provides better reliability without compromising on performance. LIB postpones the binding of instruction information to the ROB from the time of allocation to the time when the results are written back from the execution units.

In order to enable LIB, we propose a Target Address Buffer (TAB) which stores only the target addresses of the taken branch instructions. With TAB and the Program Counter (PC) of the last committed instruction, the PC of the exception raising instruction can be easily obtained, thus eliminating the need to buffer the PCs all in-flight instructions in ROB. This also reduces the overall ROB size. Detailed simulation with SPEC CPU 2000 benchmarks show that LIB achieves an average of 33.4% and up to 56.5% reduction in AVF over the conventional instruction binding in ROB. Furthermore, with a slight increase in the width of the issue queue and execution units, LIB achieves an average of 38.7% and up to 64.5% reduction in AVF of ROB. This increase in width also lend towards an average improvement of 9.5% and up to 57.2% in IPC.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Vijay Narayanan, for being such a great mentor. But for his moral support, I would not have successfully completed my Masters program at Penn State. I learned invaluable skills working under him. He taught me the nuances of executing and completing a task within the specified time bound. Though I was exceedingly irregular, he was able to put up with me and took pains to teach me the value of hard work. He taught me the ways to improve my writing skills and made me realize how better presentation can make a difference. Another person whom I would like to personally thank is Prof. Chita R Das. He along with Vijay baled me out of a career slump and helped me set my career path right. I am always indebted to both of them.

I would also like to thank Dr. Angshuman Parashar for mentoring during the time when I was without an advisor. He was there when I really needed help in making progress in research. He is the most brilliant person I have ever come across so far and I am really proud to have known him and worked alongside with him. Period.

Chapter 1

INTRODUCTION

Mitigating the effects of transient faults has become one of the key design constraints in building modern day processors. With transistor density trends adhering to Moore's law, errors on chip are starting to grow as devices become more susceptible to neutron and alpha particle strikes. A tangible approach in solving this problem of increasing error rate is to provide ECC mechanisms [**1**] or employ redundancy execution mechanisms such as [**2**] [**3**] [**4**] [**20**] [**21**]. While these approaches try to provide 100% reliability, they come with a significant increase in cost in the form area and performance. This loss in area and performance for 100% reliability can be overkill for most of the microprocessor market segments as they require only good reliability and not bulletproof operation. Hence, in this work, we try to explore alternate mechanisms to curb the effect due to transient faults and keep it under control at an acceptable level without having any negative impact on area and performance.

The probability that a transient fault affect the valid state in a processor structure can be attributed to the life-time of ACE-bits staying in it [**11**]. ACE bits are those bits which are required for architecturally correct execution [**13**]. That is, a bit is termed ACE, if error in that bit manifests itself into an observable output. A structure can exhibit high vulnerability when longer delays are incurred to discard instructions out of the structure

from the time they become available for usage. In other words, a processor structure is more susceptible to transient faults, when longer delays are incurred between the time when the ACE-information is pushed into the structure and the time when it is discarded. Definitely, eliminating these long delays will aid to keep the processor error rate at an acceptable level. There are two major approaches to solving this problem.

The first approach, as mentioned in [**11**], involves squashing instructions from the structure when the processor encounters events that trigger these long delays. These triggers can be events such as cache misses, pipeline bubbles due to the non-availability of hardware resources, memory clogging, page faults, etc [**11**]. While this approach is quite efficient in drastically reducing the exposure of the long staying instructions to the radiation, it takes a toll on performance. The second approach can be to reduce the difference between the time when the instruction binds to the structure and the time when the instruction gets utilized or discarded from the structure. In this work, I choose the second approach. We focus on reducing the probability of the transient fault occurring in the processor with specific applications to Re-Order Buffer (ROB) and show how the reliability of the structure is affected when the binding time of the instructions is altered. In order to do so, I propose Late Instruction Binding (LIB).

LIB reduces the life-time of the ACE-bits in ROB by postponing the binding or writing of this information into ROB from the time of allocation to the time when the instructions complete execution. In a typical ROB, the lifetime of the ACE-bits is inherently more when compared against other storage-based structures such as issue

queue and load store queue. Figure **1-1** shows the average lifetime of instructions in ROB across all 26 benchmarks of the SPEC CPU 2000 suite. This is primarily due to the fact that ROB holds the information corresponding to instructions from the time they are allocated an entry in ROB to the time until they are successfully committed to the architected state, while other structures discard information from their storage space once the instruction or the data corresponding to the instruction is manipulated upon.

The contributions of this work are as follows:

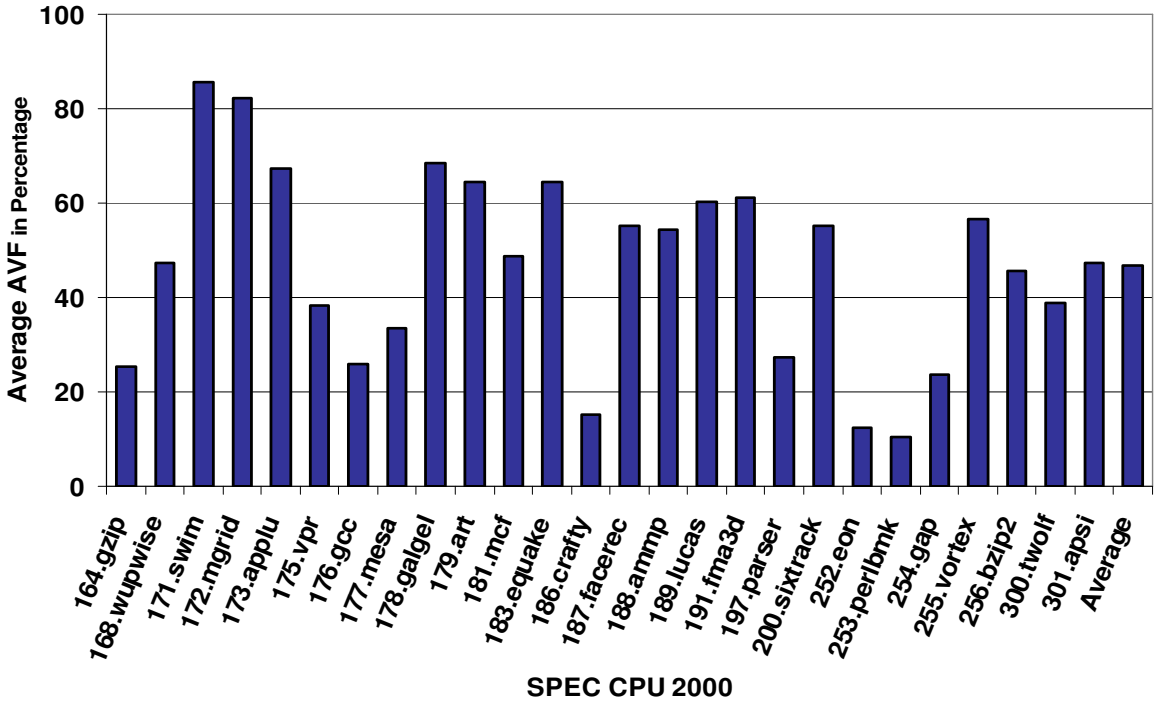


Figure **1-1**: Average AVF of a 128-entry ROB in percentage

1. We propose Late Instruction Binding (LIB) which reduces the life-time of the ACE-bits in ROB by postponing the binding or writing of information into ROB from the time of allocation to the time when the instructions complete execution.

2. We then elaborate how LIB can be enabled and exploited to reduce the probability of transient fault affecting the valid state in ROB.

3. We exhibit how to extract performance benefits using our technique with minimal impact on the vulnerabilities of the issue queue and execution units.

4. We demonstrate the efficiency of a well known vulnerability control mechanism, VCT (Vulnerability Control via dispatch Throttling) when implemented on top of our design.

The rest of the thesis is organized as follows. Chapter 2 gives a brief description about measuring vulnerabilities and meeting reliability budgets. Chapter 3 discusses about techniques to enable LIB in ROB followed by experimental setup in chapter 4. Chapter 5 provides thorough analysis of the trade-offs between performance and reliability along with results. Related work is given in chapter 6 and concluding remarks in chapter 7.

Chapter 2

Information binding and Reliability

2.1 Computing System Vulnerability

Typically, the processor error rate is expressed in terms of Failure in Time (FIT) rate [**13**]. The effective FIT rate of a processor is the summation of FIT rates of individual processor components. Initially, a raw FIT_{raw} rate estimate for each component is obtained by performing a circuit level analysis. With architectural and micro-architectural effects further reducing the probability of occurrence of a transient fault, this raw estimate is further de-rated. Hence, the effective FIT_{eff} of a component is given by

$$FIT_{eff} = AVF \times FIT_{raw}$$

Where AVF is the Architectural Vulnerability factor and is defined as the average-over-time of the ratio of ACE bits in structure to the total number of bits in the structure [**13**].

AVF is a key metric to enable quantitative analysis of the transient fault tolerance solutions. It isolates the architectural and microarchitectural effects on soft error rate from that of the device technologies, thus making it a powerful metric to architects to build solutions independent of process technology. Since circuit-level estimates are not known to the architects at the design time, AVF has been accepted as a powerful metric

during the high-level architectural design stages to determine the relative contributions of various micro-architectural components to the overall processor error rate.

2.2 Information binding and AVF – the connection

The AVF contributions of the microarchitectural structures that occupy significant portion of real estate, such as ROB, issue queue and load-store queue, can be defined as a measure proportional to the difference between the time when the information binds to them and the time when the information gets utilized from them. A structure will exhibit high AVF if the difference in time between binding and utilization is more. While it is not possible to alter the time of utilization, it is always possible to alter the time of binding. In this work, we show techniques to reduce the AVF of ROB by altering the binding time of instructions to it.

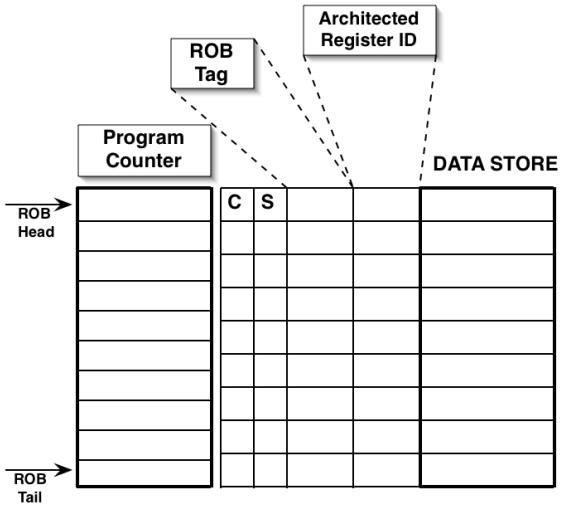


Figure **2-1**: Structure of ROB

In ROB, the information binding can happen in two different time frames of the processor pipeline: instruction *allocation time* and instruction *completion time*. Various information bits corresponding to an instruction bind to ROB either at *allocation time* or *completion time* and are discarded from ROB at the *commit time* after the instruction is resolved to be non-speculative. Figure **2-1** depicts the typical structure of ROB and Table **2-1** shows the binding time of various information bits in ROB along with their functionality.

Table **2-1**: Functionality and binding time of various information bits in ROB

Information	Functionality	Binding Time
A 64-bit Program Counter (PC)	To handle instructions that raise exceptions as well as mis-predictions (Control and Data)	Allocation time
A completed bit	Essential information for the instructions to commit. Signifies that an instruction has finished execution and the results are written back.	Instruction Completion time
A Speculative	To ensure speculative instructions never get to the head of the reorder buffer because previous instructions (including branches) must complete first. Thus a branch will already be resolved and recovery initiated if needed before any (control dependent) instructions after the branch are allowed to complete.	Instruction Completion time
ROB Tag	Essential for the allocator to know which ROB entry writes into the architected register so as to de-allocate that entry and re-allocate it for some other instruction.	Instruction Completion time
Architected Destination Register (ADR)	Required by the commit logic to correctly update the architected register file.	Allocation time
Data Store	Contains the results of the instructions that have completed execution.	Instruction Completion time

When the entries in the ROB are allocated, certain fields in them are written with the data from the instruction. To be exact, Program Counter (PC) and the Architected Destination Register (ADR) are written. These information bits can be either written at

the allocation time or with the results written back by the execution units. But in order to reduce the width of the *issue queue* entries as well as to reduce the amount of information which must be circulated to the execution units or memory subsystem, these information bits which are required to correctly retire or commit an instruction is written into the ROB strictly at the allocation time. While this approach definitely reduces the design complexity of the issue queue, it increases the vulnerability of the structure by increasing the life time of the ACE-bits in it.

As mentioned in the previous chapter, the average lifetime of instructions sitting in ROB is unusually more unlike other structures due to the fact that they hold the ACE-bits from the time of allocation to the time when instructions commit. This increase in vulnerability is unwarranted because there is no necessity for the ACE-bits to be available so early in ROB before they are utilized. It is enough for those bits to be written into ROB after the corresponding instruction completes execution. LIB tries to achieve exactly this by lessening the gap between the availability and utilization of the ACE-bits in ROB.

When the entries in the ROB are allocated, certain fields in them are written with the data from the instruction. To be exact, Program Counter (PC) and the Architected Destination Register (ADR) are written. These information bits can be either written at the allocation time or with the results written back by the execution units. But in order to reduce the width of the *issue queue* entries as well as to reduce the amount of information

which must be circulated to the execution units or memory subsystem, these information bits which are required to correctly retire or commit an instruction is written into the ROB strictly at the allocation time. While this approach definitely reduces the design complexity of the issue queue, it increases the vulnerability of the structure by increasing the life time of the ACE-bits in it.

As mentioned in the previous chapter, the average lifetime of instructions sitting in ROB is unusually more unlike other structures due to the fact that they hold the ACE-bits from the time of allocation to the time when instructions commit. This increase in vulnerability is unwarranted because there is no necessity for the ACE-bits to be available so early in ROB before they are utilized. It is enough for those bits to be written into ROB after the corresponding instruction completes execution. LIB tries to achieve exactly this by lessening the gap between the availability and utilization of the ACE-bits in ROB.

Chapter 3

Late Instruction Binding

LIB reduces the life-time of the ACE-bits in ROB by postponing the binding or writing of this information into ROB from the time of allocation to the time when the instructions complete execution. From Table **2-1**, it can be seen that except for the PC and the ADR all other information bits are written only after the instruction completes execution. These bits are not carried through the rest of pipeline as it would drastically increase the width of the already complex issue queue by a bizarre number of bits. For example, for Alpha ISA, the width of the issue queue as well as the execution units should increase by 69 bits (64-bits for PC and 5-bits for ADR). This would not only make the design of these structures complex but also circulate unwanted information through the structures present in back end (or the out-of-order-end) of the pipeline increasing their vulnerabilities. While carrying ADR through the back end of the pipeline can be tolerated to some extent, carrying PC is impossible to tolerate because of its size. It is also not possible to eliminate storing the PCs entirely as they are required to handle exceptions and mis-predictions. Hence, in order to enable LIB, an intelligent solution is required to carry PC through the structures present in the back end of the pipeline without increasing their complexity and at the same time having zero or negligible impact on their vulnerabilities. In order to do so, we propose Target Address Buffer (TAB).

3.1 Target Address Buffer (TAB)

In a typical program flow, the instruction addresses are sequential until a branch instruction is encountered. Branching instructions are the potential deviants to the sequential flow of instruction addresses. If assumed that there are no branching statements in the application program (like streaming media applications), it is possible to recover the PC of any instruction at the head of the ROB if the PC of the first instruction and the count of instructions committed after the first instruction are known. This can be accomplished with just having a 64-bit *commit register* that holds the PC of the first committed instruction and a 64-bit increment by one *commit counter*, thus eliminating the need to have a PC array of size *ROB_SIZE * 64-bits*. When an instruction raises exception or mis-prediction, it is allowed to reach the head of the ROB first and its PC is computed with just adding the values in the *commit register* and the *commit counter*. The clear advantage of this technique is the reduction in the vulnerability of the structure as well as chip area. Also, this technique is not a costly one to implement considering the accuracy of predictors in modern day superscalar processors and the rarity of exceptions getting raised.

The problem slightly complicates when branching instructions are encountered in the program. Since every branch, which is asserted to be taken, is a potential deviant to the sequential flow of the instruction addresses, the address that every branch points to should be buffered in order to successfully track the PC of any instruction. In order to do so, we propose Target Address Buffer (TAB).

TAB stores the target addresses of the branch instructions which are asserted to be taken. It is a direct mapped structure with each entry being 64-bit wide and hashed using a *TAB index*. The structure of the ROB with TAB is shown in Figure **3-1**. Every branch instruction, whose prediction is asserted to be true, is assigned a *TAB-index* by the allocator and the index value is stored in ROB. The TAB is then accessed using the *TAB-index* and the target address value, which is effectively the PC of the next instruction to branch, is written. With TAB there are two inherent advantages. (1) It eliminates the need to carry PC of any instruction through structure present in the back end of the pipeline; (2) it enables LIB by allowing the address values to bind to it only after the corresponding branch instruction completes execution and asserted to be taken.

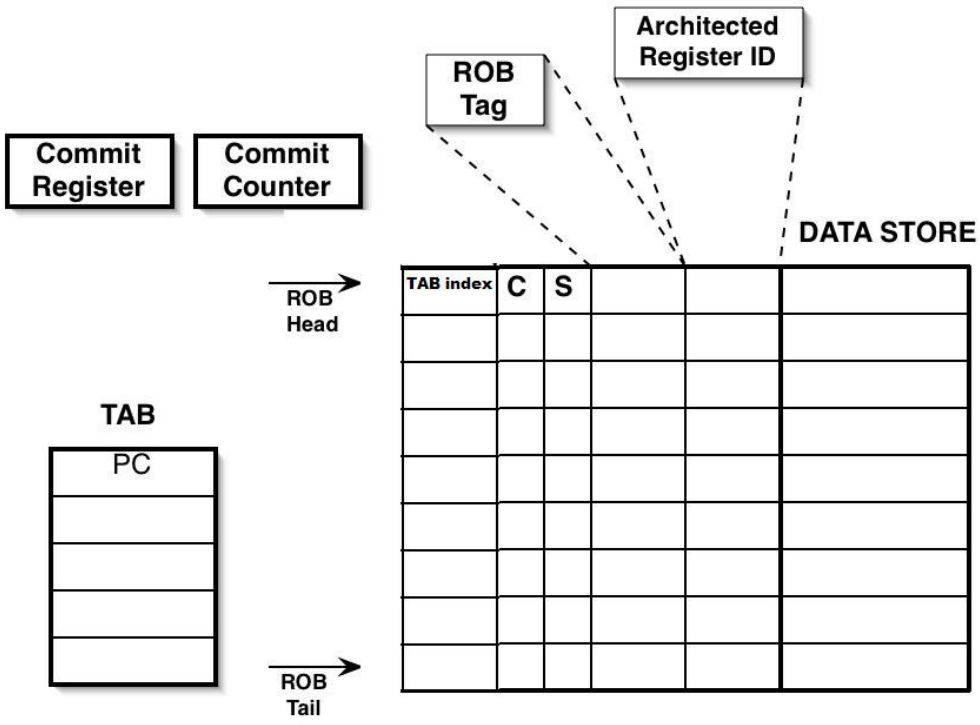


Figure **3-1**: Structure of ROB with TAB

The only disadvantage would be to provide a separate data path between the execution units and TAB. Though this added data path will have negligible impact on the overall AVF of ROB, we have not included the impact of this data path in any of our AVF computations.

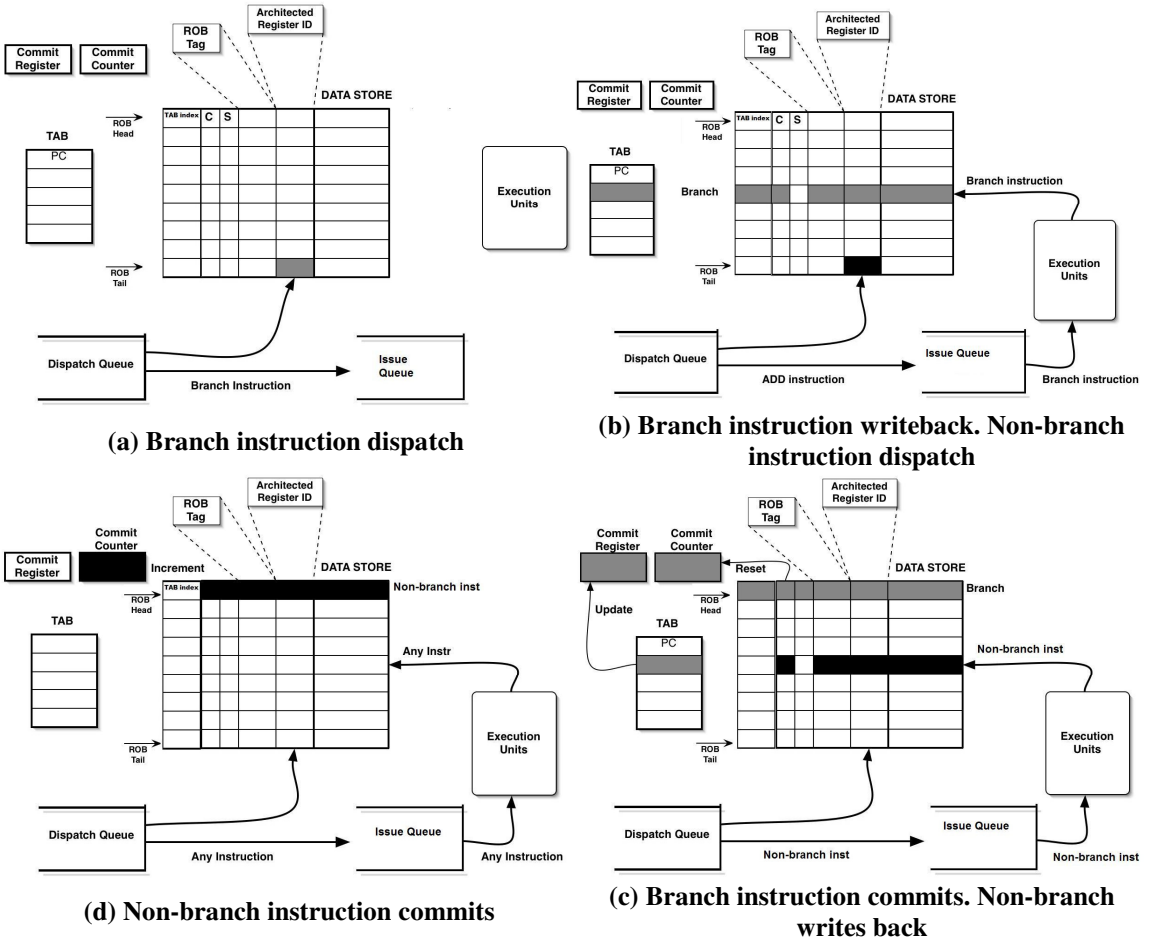
The process of tracking and recovering the PC incase of an exception or branch mis-prediction with TAB, *commit counter* and *commit register* is explained in section 3.3. The following section discusses the functioning of LIB enabled ROB (ROB_{LIB}) for both branching and non-branching instructions.

3.2 Operation – ROB_{LIB}

The ROB is active in three different stages of the pipeline: allocation and dispatch stage, instruction completion stage and instruction retirement stage [**14**]. Figure **3-2** depicts the functioning of ROB_{LIB} for both branching and non-branching instructions during these pipeline stages.

Allocation and dispatch Stage: Soon after an instruction is decoded, the allocator allocates each instruction with a ROB slot, called ROB tag. From this point, this information becomes part of the in-flight instruction and is carried through the rest of the pipeline stages. This information is required by the execution units to index into the corresponding ROB entry of an instruction and write the computed results back. The protocols for dispatching an instruction slightly differ based on whether an instruction is a branch or not. A non-branch instruction is dispatched into the issue queue soon after it

is allocated a ROB slot. Whereas, a branching instruction is dispatched only after determining that a free slot is available in TAB. Incase of TAB being full, *dispatch throttling* for branches is employed. This is to ensure that a TAB slot is deterministically available when the branch targets are written back from the execution units. Further, we prefer "dispatch throttling" because (1) it is impossible to throttle write backs deterministically without a replay-based scheduling mechanism [**15**] and (2) it is extremely costly to throttle issue as the structure is already complex handling multiple functionalities at the same time [**22**] [**23**].



(a) Branch instruction dispatch

(b) Branch instruction writeback. Non-branch instruction dispatch

(d) Non-branch instruction commits

(c) Branch instruction commits. Non-branch writes back

Figure **3-2**: Functioning of ROB_{LIB} during various pipeline stages

Once the instructions are dispatched, the information that is necessary to successfully commit those instructions, is entered in the ROB and the rest of the information that are required to execute the instruction is pushed into issue queue as shown in the Figure **3-2** (a).To be exact, the ARD number is entered into the ROB as this information is required to update correct architected register when the instruction commits. Unlike the conventional instruction dispatch, no PC information is entered into the ROB.

Writeback Stage: After the instruction finishes execution, it exits the functional unit and enters the ROB. The *data-store* field in ROB is written with the computed result. For branch instructions, soon after the branch target is computed, the value is compared with the predicted address. If both of them match, the branch is marked to be taken. It is then assigned a *TAB-index* and the target address is written into that entry. The *TAB-index* value is then saved in ROB. This scenario is depicted in Figure **3-2** (b). But for non-branch instructions, no PC information is required to be stored and hence no TAB slot is allocated. When a non-branch instruction is written back as shown in Figure **3-2** (c), it is marked complete and the data-store field is updated with the computed result. TAB is untouched in this case.

Commit Stage: An instruction is ready to commit or retire if it has completed execution and when its state has become non-speculative. Committing an instruction involves updating the architected state or the memory sub-system. Every instruction other than the

taken branches increments the *commit counter* by one while retiring. When a taken branch instruction commits, the TAB is indexed using the *TAB-index* and the *commit register* is updated with the value from TAB and the *commit counter* is set to zero. At the outset of the program execution, the *commit register* will hold the PC of the first dispatched instruction. Figure **3-2** (c) shows the steps involved in committing a branch instruction. The target address of the corresponding branch instruction is retrieved from the TAB using *TAB-index* and the *commit register* is updated with that value. The *commit counter* is then reset to zero. When a non-branch instruction commits as shown in Figure **3-2** (d), the *commit counter* is incremented by one and the *commit register* is untouched.

3.3 Handling Exceptions and branch mis-predictions

When an exception occurs, the exception raising instruction is tagged in ROB [**14**]. The commit logic checks each instruction before that instruction is committed. At this instant, the *commit register* has the target address of last committed taken branch and the *commit counter* has the number of instructions surpassed after that branch. When a tagged instruction is encountered; it is not allowed to commit. All the instructions prior to the tagged one are allowed to commit. During this process, the *commit register* and *commit counter* are updated as explained in the section 3.2. Once the tagged instruction reaches the head of ROB, the PC of the exception raising instruction is obtained as follows:

$$PC_{exception} = \text{commit register} + \text{commit counter}$$

The machine state is then check pointed. The machine state includes all the architected registers and the $PC_{exception}$. The remaining instructions in pipeline, some of which may have already completed are flushed. After the exception is handled, the checkpointed machine state is restored and execution resumes with the fetching of the instruction that triggered the original exception. Control and data mis-predictions are handled in a similar fashion. It is essential to recover the PC of the mis-predicted instruction as all the prediction tables are indexed using the PC. Once the PC is obtained the corresponding prediction tables are index and the values are updated.

Chapter 4

Experimental Set-up

4.1 Methodology

Table **4-1**: Simulation parameters: Latencies of ALUs and caches are given in parentheses. All ALU operations are pipelined except division and square root.

Parameter	Value
Fetch/Decode/Issue/Commit Width	4
Pipeline Stages	15
Fetch Queue size	128
ROB Size	128
Reservation Station size	64
BTB size	2K-entry 4-way
LSQ Size	128 entries
TAB size	16
L1 D-Cache	64KB, 4-way with 32B block (2)
Branch Predictor	Combined predictor with 16K entry meta-table. 2-lev predictor with 16K-entry L1, 16K-entry L2, 14-bit history XORed with address
L1 I-Cache	64KB, 4-way with 32B block (2)
L2 Unified Cache	512 KB, 4-way with 64B line-size (12)
I-TLB	512-entries 4-way set-associative
D-TLB	1K-entries 4-way set-associative
TLB Miss-Latency	30 cycles
Memory Latency	200 cycles
Integer ALU	6 (1-cycle)
Integer Multipliers/Dividers	2 (3,20)
FP ALUs	4 (2)
FP Mult./Div./Sqrt.	2 (4,12,24)
L1 D-Cache Ports	4
TAB size	16-entry
Commit register	64-bits
Commit counter	12-bits

The baseline microarchitectural parameters for all our experiments are shown in Table **4-1**. Our experiments were conducted using via execution driven simulation using an in-house simulator based on simplescalar 3.0 toolset [**16**].

Table **4-2**: Spec CPU 2000 sim-points. M = 1 million

Benchmark	Instructions Skipped	Benchmark	Instruction Skipped
164.gzip	29000 M	187.facerec	64100 M
168.wupwise	23800 M	188.ammp	50900 M
171.swim	78100 M	189.lucas	123500 M
172.mgrid	200 M	191.fma3d	23600 M
173.applu	500 M	197.parser	71400 M
175.vpr	49200 M	200.sixtrack	4100 M
176.gcc	16,600 M	252.eon	73000 M
177.mesa	73300 M	253.perlbmk	0 M
178.galgel	5000 M	254.gap	18800 M
179.art	36,400 M	255.vortex	59,300 M
181.mcf	26,200 M	256.bzip2	48,900 M
183.equake	1,500 M	300.twolf	185400 M
186.crafty	120600 M	301.apsi	100 M

We evaluated the performance and AVF trade-offs for all 26 benchmarks of SPEC CPU 2000 suite. All the benchmarks were compiled for Alpha-ISA with *–fast* optimization. We studied the results of the first 100 million instructions after fast forwarding each benchmark to its first sim-point [**17**]. Table **4-2** lists the skip interval and input set selected for each of the SPEC CPU 2000 programs used for our analysis.

4.2 Optimal TAB size

Figure **4-1** shows the CDF of the number of taken branches existing in ROB at any instant in time for all 26 benchmarks of SPEC CPU 2000 suite. On an average, well over 99% of the fraction of processor cycles, the number of taken branches existing in a 128-entry ROB is not more than 16. Henceforth, in all our experiments we have assumed a size of 16 for TAB unless mentioned otherwise.

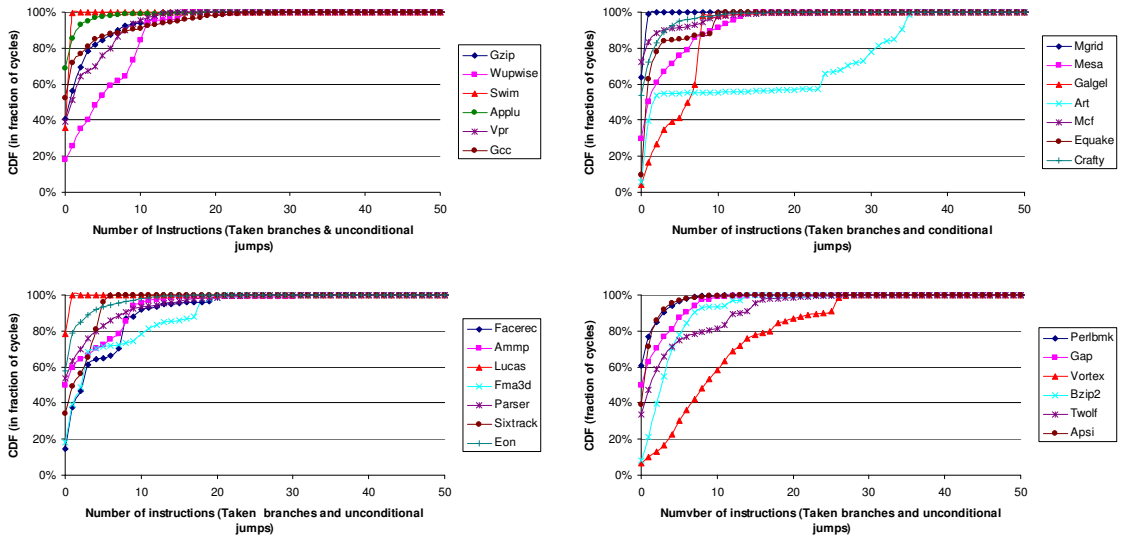


Figure **4-1**: CDF (in fraction of cycles) of the number of taken branch instructions existing in a 128 entry-ROB at any instant in time

4.3 Optimal commit counter size

Table **4-3** shows the maximum number of instructions surpassed between two subsequent taken branches in a 128-entry ROB. The *commit counter* should have the provision to hold these values in-order to ensure correct recovery of PC during

exceptions and mis-predictions. From Table **4-3** it can be seen that the maximum value the counter need to hold is 1157 which means that the width of the *commit counter* should be at least 11-bits. Henceforth, in all our experiments we have assumed 11-bits to the size of the *commit counter* unless mentioned otherwise.

Table **4-3**: Maximum instructions surpassed between two taken branch instructions

Benchmark	**Instructions Skipped**	**Benchmark**	**Instruction Skipped**
164.gzip	48	187.facerec	250
168.wupwise	63	188.ammp	381
171.swim	318	189.lucas	213
172.mgrid	783	191.fma3d	1157
173.applu	1570	197.parser	68
175.vpr	24	200.sixtrack	389
176.gcc	48	252.eon	148
177.mesa	216	253.perlbmk	62
178.galgel	78	254.gap	59
179.art	390	255.vortex	62
181.mcf	49	256.bzip2	38
183.equake	492	300.twolf	162
186.crafty	95	301.apsi	675

Chapter 5

Evaluation and Results

This chapter discusses the impact on reliability of ROB_{LIB} (section 5.1) and improving the efficiency of VCT (section 5.2). Section 5.3 articulates on the trade-offs between performance and reliability in using ROB_{LIB}.

5.1 Reliability of ROB_{LIB}

In this section, the impact of using LIB on a 128 entry ROB is shown. Figure **5-1** shows the reduction in average AVF of ROB_{LIB} over the conventional ROB. The bar marked *average* shows the arithmetic mean of the percentage reduction in AVF across all 26 benchmarks of the SPEC CPU 2000 suite. With ROB_{LIB} there is a reduction in AVF on an average 33.8 % and up to 55.7%. This sizable reduction can easily be justified using Figure **4-1**. From Figure **4-1**, it can be seen that for benchmarks such as 197.*parser and 253.perlbmk*, the fraction of total processor cycles the TAB is empty is well over 60%. Hence, these benchmarks exhibit more than 50% reduction in AVF. Similar kind of justification can be given to *200.sixtrack, 176.gcc, 164.gzip,* and *252.eon*. These benchmarks show well over 40% reduction. With *256.bzip2*, the TAB was never full at any instant in time. This contributed to a sizeable reduction of 55.7% in average AVF. *189.lucas* show more than 40% reduction in AVF due to the following reasons: (1) the fraction of branch instruction found in ROB at instant in time is not more than 12.5%.

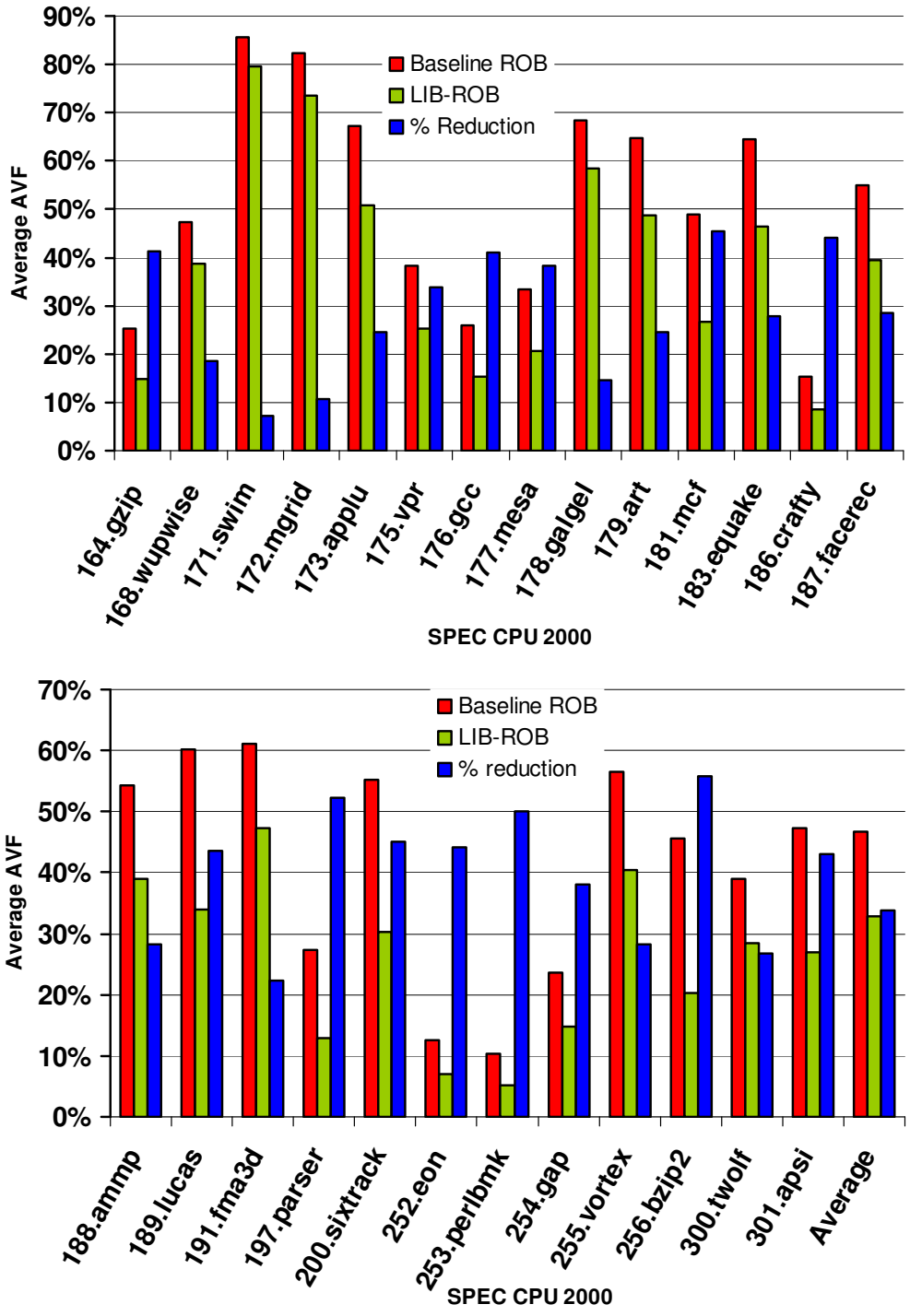


Figure **5-1**: Average Reduction in AVF of a 128-entry ROB

(2) For around 80% of the total processor cycles the TAB is empty. (3) The average distance between any two branches is around 170 instructions.

Benchmarks that show very less reduction in AVF are primarily due to the fact that TAB is full for most fractions of the processor clock cycles excepting *171.swim*. It exhibits a peculiar behavior. Even though the number of entries in TAB is not more than 3 for most fractions of the total processor cycles (and the average distance between two branch instructions is around 205 instructions), it is not able extract sizable AVF gains like *189.lucas* due to the fact that the retirement of the instructions are blocked for a substantial fraction of the total execution time due to L2 cache misses [**18**]. This effect over-shadows the contributions of TAB in reducing AVF. Blocking of the instruction retirement is the prime reason for the high average AVF exhibited by this benchmark as shown in Figure **1-1**.

5.1.1 Limitation

As mentioned in the previous section, ROB_{LIB} is able to provide sizeable reduction in AVF with almost no degradation in performance. But this is on an assumption that the task of committing instructions does take more than a cycle. For commit widths greater than one, there is a possibility that more than one branch instruction waiting to commit at the same time. In those cases, the task of committing instructions cannot be accomplished within a cycle as there is a fair bit of work involved in identifying branch instructions and then updating the *commit register* and *commit counter* with correct values. In all our experiments, we have considered this scenario as a special case as the average branch

distance of all the applications in the SPEC CPU 2000 suite is typically greater than five. We have assumed that the commit logic would be able to handle this scenario and commit instructions within a cycle.

5.2 Efficiency of VCT with ROB_{LIB}

Recent studies have proposed online monitoring mechanisms at cycle-level granularity to control vulnerabilities of processor structures and bound them under specified AVF budget. One such mechanism is Vulnerability Control via dispatch Throttling (VCT), a proactive vulnerability control mechanism for ROB which stalls the instruction dispatch to the ROB when the vulnerability of the structure exceeds the specified AVF bound [**19**]. This is a relatively simple technique to implement with no additional hardware overhead. We have implemented VCT on top of our processor with ROB_{LIB} and compared the behavior of this mechanism ($VCT\text{-}ROB_{LIB}$) with respect to the one implemented on top of the processor with conventional ROB ($VCT\text{-}ROB_{conv}$). We found that the pressure exerted on the dispatch logic (in terms of number of times the dispatch logic is stalled) by VCT reduces on an average by 54% and up to 100% across varying AVF bounds.

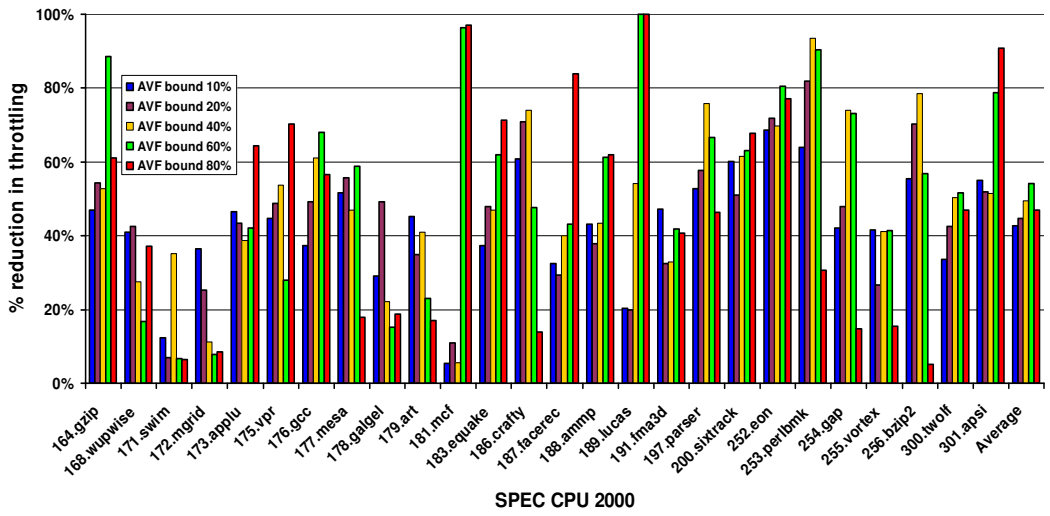


Figure **5-2**: Percentage Reduction in the number of time throttling is called for different AVF bounds

Figure **5-2** shows the percentage reduction in the number of dispatch stalls exhibited by *VCT- ROB_{LIB}* over *VCT- ROB_{conv}* for different AVF bounds. This reduction in the number of dispatch stall cycles *VCT-LIB* can be easily explained with the cycle-level amplitude variations of AVF. Figure **5-3** depicts the cycle level AVF variations of *177.mesa*. It can be seen that the amplitude of AVF variations in ROB_{LIB} is reduced by almost 50% for most fractions of the processor cycles. This means that the number of times the bound getting violated is drastically reduced. This reduction in amplitude is partly due to the reduction in the lifetime of the information bits resident in ROB_{LIB} and partly due to the

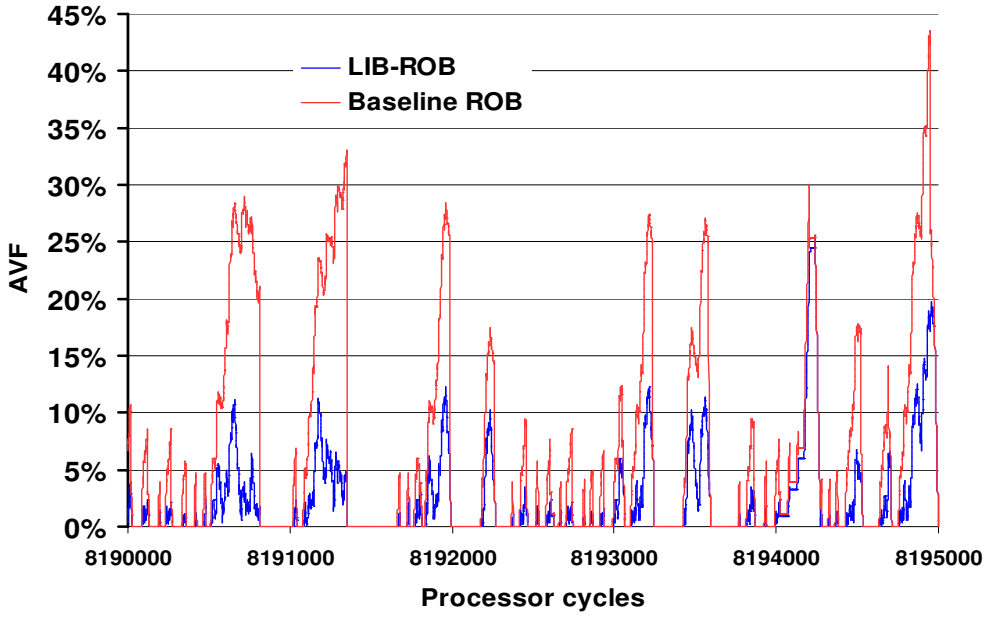


Figure **5-3**: Variations in amplitude of AVF at cycle level granularity for 177. Mesa

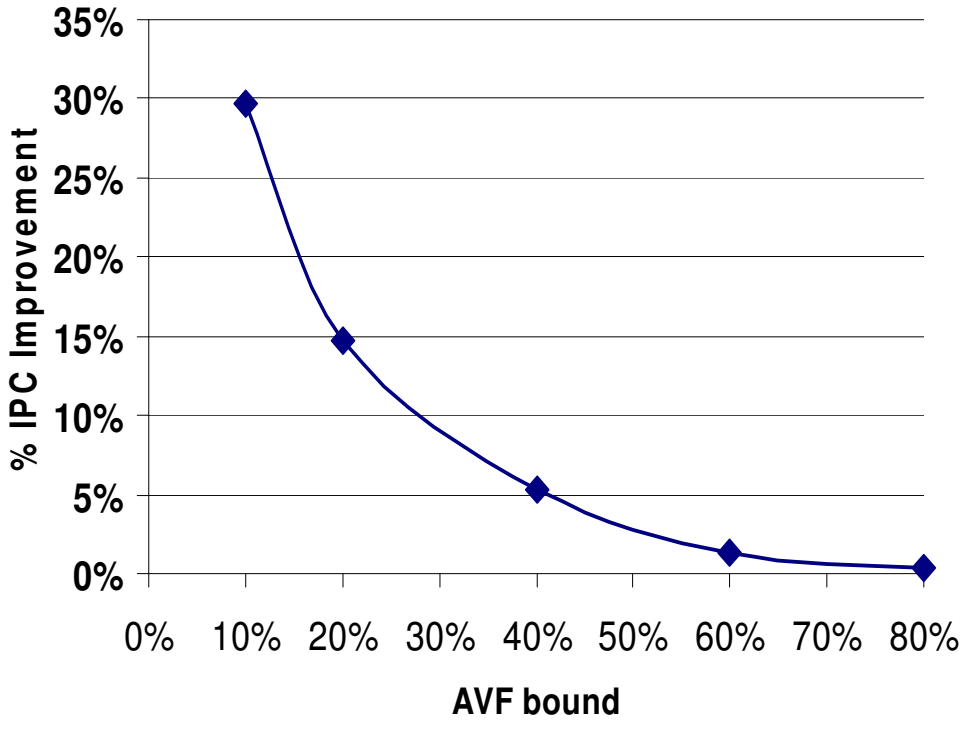


Figure **5-4**: Improvement in IPC when VCT is implemented on top of LIB-ROB for varying AVF bounds

reduction in the number of PC values that is to be stored in ROB_{LIB}. This has a direct impact on the number of dispatch stalls imparted by VCT. With the reduction in the processor dispatch stalls, the total execution time of the processor also decreases proportionately. Hence, for stricter AVF budgets (such as, 10% or 20%), *VCT- ROB_{LIB}* buys back a substantial fraction of the performance which is otherwise lost due to *VCT- ROB_{conv}*. Figure **5-4** depicts the improvement in IPC of *VCT- ROB_{LIB}* over *VCT- ROB_{conv}*. For an AVF bound of 10%, *VCT- ROB_{LIB}* buys back 30% the lost performance. But the performance gain slowly diminishes when the vulnerability bound is relaxed.

5.3 Extracting performance using ROB_{LIB}

As mentioned in Section 3.2, when an instruction is dispatched, the information necessary to successfully commit that instruction is entered in the ROB (which is nothing but ADR) and the rest of the information that are required to execute the instruction is pushed into the issue queue. Imagine the situation in which ADR is not written into ROB_{LIB} at the time of instruction dispatch but is pushed into the issue queue along with the other information bits that are required to execute the instruction. This can be easily accomplished with slight increase in the width of the issue queue and execution units. For example, for processors that execute Alpha ISA, the width of their issue queue as well as the execution units should increase by 5-bits as Alpha ISA has 32 architected registers.

By not pushing ADR into ROB at the time of dispatch has two undue advantages: (1) Any information corresponding to an instruction stays in ROB only between the time

of instruction completion and retirement, thereby reducing the overall lifetime of the instructions present in ROB. (2) There is a big gap between the time at which a ROB entry is allocated for an instruction and the time at which information bits become available for usage. This gap in time can be exploited to let the pipeline proceed without any stalls incases of ROB being full.

Instead of introducing pipeline bubbles when ROB is full, the allocator speculatively assigns the next possible but currently occupied ROB entry to another instruction which is waiting to be dispatched and push that instruction into the issue queue. It is enough for that ROB entry to be available at the time when latter is being written back along with computed results. The dispatch logic is throttled only in case of either issue queue being or TAB being full. Given that the dispatch logic need not be throttled in case of ROB being full, there is a possibility that more than one instruction sitting in the issue queue having the same ROB tag. To eliminate the ambiguity and maintain age ordering, sequence numbers have to be appended along with ROB tag. In all our experiments, we have used a single bit sequence number as we never found a case in which we need to assign identical ROB tags to more than two instructions. The first instruction to a ROB slot is assigned a sequence number of ‘0’ and the second instruction to the same slot is assigned a sequence number of ‘1’. When the instruction with sequence number ‘1’ is about to be issued into the execution unit, the ROB slot corresponding to that instruction is checked for emptiness. If the slot is found to be empty, then the instruction is executed. Otherwise, the instruction issuing that instruction is throttled until the slot frees up (that is, the older instruction with sequence number ‘0’

commits). Figure **5-5** compares the IPC values of three different ROB configurations, baseline 128-entry ROB, 128-entry ROB_{LIB} and baseline 256-entry ROB.

To prove the effectives of the above mentioned technique, we have assumed an ideal Load Store Queue (LSQ) to nullify the stalls introduced in case of LSQ being full. Across 26 SPEC CPU benchmarks a 128-entry ROB_{LIB} shows an improvement in IPC on an average 9.5% and up to 57.2% over a 128-entry conventional ROB. The performance of ROB_{LIB} for most of the benchmarks is as good as a 256-entry ROB. *179.art* shows 19% degradation in performance with respect to a 128-entry ROB because of the stalls induced by TAB. It can be inferred from Figure **4-1** that *179.art* requires more than twice the number of TAB entries to buy back the lost performance. For benchmarks like *181.mcf, 186.craft, 164.gzip, 252.eon, 253.perlbmk, 256.bzip and 300.twolf*, there is very little or no performance gains. This is because of the fact that the occupancy rate for these benchmarks is very low. This technique will be really handy to applications which are hungry for more physical resources. *189.lucas* gained enormous improvement in performance because of this reason.

Apart from providing performance gains, this technique further reduces the AVF of the ROB_{LIB} by 8.3% as shown in Figure **5-6**. This is because the binding of ADR in ROB is postponed until the time of instruction completion. But all the above mentioned benefits come with a trade off. The width of the issue queue should increase slightly in-order to hold the bits of the ADR (five bits incase of Alpha ISA as it has 32 architected registers or 3 bits in case of X86 ISA as it has 8 architected registers) as well as the sequence number (1-bit).

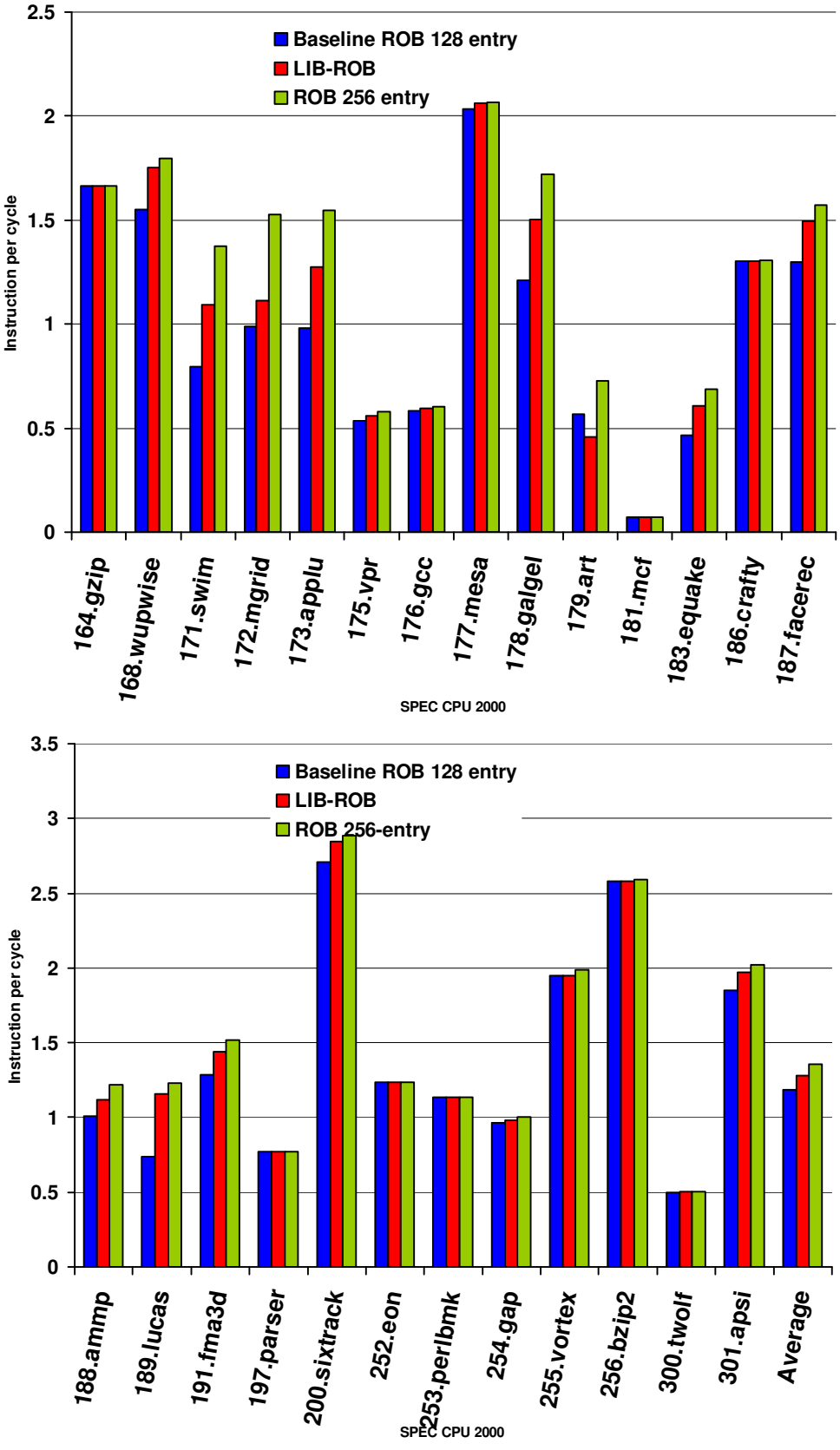


Figure **5-5**: IPC improvement of LIB-ROB over the conventional ROB

This will increase the AVF of the issue queue and execution units and it is depicted in Figure **5-6**. On an average across all 26 benchmarks, there is an 8.1% aggregate increase in the AVF of issue queue and execution units.

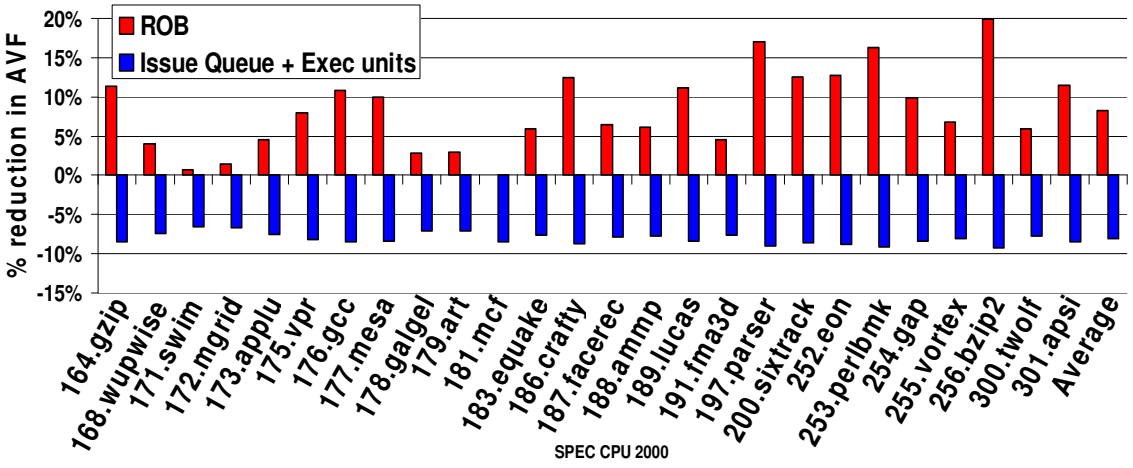


Figure **5-6**: Impact on AVF of ROB as well as Issue queue and Exec units

5.4 Storage Cost

The additional hardware for LIB-ROB consists of the following (1) a 64 bit Commit register, (2) a 11 bit commit counter, (3) a 16 entry TAB.

The total storage cost comparison between ROB_{LIB} and baseline ROB is shown in Table **5-1**. From Table **5-1**, the storage overhead of the ROB_{LIB} reduces by 38.3%. As the number of ROB entries increases, the storage overhead further reduces (which means that the percentage increases). This is because the increase in TAB size is not linear with the increase in the number of ROB entries. For example, a 256 entry ROB could do with a 20-entry TAB without any degradation in performance.

Table **5-1**: Comparison of Storage Cost

Parameter	**Baseline ROB**	**LIB-ROB**
Total number of entries	128	128
Size of PC array	128 * 64-bit = 8192 bits	16 * 64-bits = 1024 bits
Status bits (Completed + Speculative + Sequence no bit)	128 * (1+1+0) =256 bits	128 * (1+1 +1) = 384 bits
Architected Destination register number	128 * 5 bits = 640 bits	128 * 5 bits = 640 bits
ROB Tag	128 * 7 bits = 896 bits	128 * 7 bits = 896 bits
Data store	128 * 64 bits =8192 bits	128 * 64 bits = 8192 bits
Commit register	0 bits	64 bits
Commit counter	0 bits	11 bits
Total	**18176 bits**	**11211 bits**

Chapter 6

Related Work

A tangible approach in solving the problem of transient errors is to provide mechanisms such as parity or ECC [**1**]. But unfortunately these mechanisms, which are typically applied in memory sub-systems, have a significant impact on the area, power and performance cost. Another class of solution to mitigate this problem is by redundant execution schemes [**2**] [**3**] [**4**] [**20**] [**21**]. But these schemes require additional hardware resources, such as threads or cores. Recently, researchers proposed numerous cost-effective techniques to both detect and recover from the transient faults at the thread or processor level [**5**] [**6**] [**7**] [**4**] [**8**] [**9**] [**10**] [**3**] . However, my work does not propose any technique to detect or correct soft errors. Instead, it tries to explore techniques to reduce the soft error rate of the processor structures by reducing the lifetime of the instructions staying in it.

Earlier work in reducing soft error rate involves squashing instructions from the structures when the processor encounters events that can trigger long delays [**11**]. These triggers can be events such as cache misses, pipeline bubbles due to the non-availability of hardware resources, memory clogging, page faults, etc. While this approach is quite efficient in drastically reducing the exposure of the long staying instructions to the radiation, it takes a toll on performance. My approach tries to reduce the soft error rate with zero or negligible impact on performance by binding the instructions to the structure as late as possible close to the time of utilization. Recent studies have proposed online

monitoring mechanisms to monitor and control vulnerabilities at cycle level granularity [**19**] [**24**]. These papers provide mechanisms by treating reliability as first order design constraint and then optimize performance as long as reliability requirements are satisfied. I do not provide any mechanism to control the vulnerabilities of structures in my work. Instead, I try to provide a new design to the processor structures which can inherently have a low error rate.

The idea of late binding is not entirely new. Sethumadhavan et al [**12**] proposed unordered-LSQs to improve area and power efficiency of LSQs by allocating entries only when the instruction issues ("late binding") and not when the instruction dispatches. While this paper employs late binding to solve the scalability problems of LSQ for large instruction windows, we exploit this technique to improve the reliability of ROB.

Chapter 7

Conclusion

This work introduces Late Instruction Binding (LIB) in ROB and demonstrates how LIB can be enabled and exploited to achieve better reliability as well as performance. LIB reduces the life-time of the ACE-bits in ROB by postponing the binding or writing of this information into ROB from the time of allocation to the time when the instructions complete execution. Though LIB reduces the AVF of ROB, it may increase the complexity and vulnerability of issue queue and execution units. Hence, in order to enable LIB and at the same time not to impact the complexity and vulnerability of other structures in the back end of the pipeline, we proposed Target Address Buffer (TAB). TAB eliminates the need to store PCs of all instructions and stores only the target address of taken branch instructions. TAB works on the intuition that it is only the branch instruction that deviate the sequential flow of instruction addresses. Furthermore, addresses bind to TAB only when the corresponding branch instruction completes execution. By integrating TAB with ROB, we are able to achieve an average of 33.4 % and up to 56.5% reduction in average AVF of ROB. Also, with slight increase in the width of the issue queue and execution units, we are able to achieve an average 38.7% and up to 64.5% reduction in average AVF of ROB. This increase in width also lend towards an average improvement of 9.5% and up to 57.2% in IPC.

Bibliography

1. D. R. Siewiorek and R. S. Swarz, “Reliable Computer Systems Design and Evaluation”, Published by A.K.Peters, Ltd., 1998.

2. T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," ACM/IEEE 32nd Annual Symposium on Microarchitecture (MICRO-32), November 1999.

3. M. Gomaa, C. Scarbrough, T.N. Vijaykumar, and I. Pomeranz, "Transient Fault Recovery for Chip Multiprocessors," International Symposium on Computer Architecture (ISCA), 2003.

4. T.N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient Fault Recovery via Simultaneous Multithreading," International Symposium on Computer Architecture (ISCA), 2002.

5. A. Baniasadi and A. Moshovos, “Instruction Flow-Based Frontend Throttling for Power-Aware High-Performance Processors,” International Symposium on Low Power Electronics and Design, 2001.

6. E. Rotenberg, “AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors,” Proceedings of the Fault-Tolerant Computing Systems (FTCS), 1999

7. S. K. Reinhardt and S. S. Mukherjee, “Transient Fault Detection via Simultaneous Multithreading,” 27th Annual International Symposium on Computer Architecture (ISCA), June 2000.

8. S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, “Detailed Design and Evaluation of Redundant Multithreading Alternatives,” 29th Annual International Symposium on Computer Architecture (ISCA), 2002.

9. J. Ray, J. Hoe, and B. Falsafi, “Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery,” International Symposium on Microarchitecture (MICRO), December 2001.

10. S.-C. Lai, S.-L. Lu, K. Lai and J.-K. Peir, “Ditto Processor,” International Conference on Dependable Systems and Networks (DSN), 2002.

11. Weaver, C.; Emer, J.; Mukherjee, S.S.; Reinhardt, S.K., "Techniques to reduce the soft error rate of a high-performance microprocessor," In the proceedings of

31st Annual International Symposium on Computer Architecture (ISCA), 2004, vol., no., pp. 264-275, 19-23 June 2004.

12. Sethumadhavan, S., Roesner, F., Emer, J. S., Burger, D., and Keckler, S. W., "Late-binding: enabling unordered load-store queues", SIGARCH Computer Architecture News 35, 2 (Jun. 2007), 347-357.

13. Mukherjee, S.S.; Weaver, C.; Emer, J.; Reinhardt, S.K.; Austin, T., "A systematic methodology to compute the architectural vulnerability factors for a high performance microprocessor," In the proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003 (MICRO-36), pp. 29-40, 3-5 Dec. 2003.

14. J. P. Shen and M. H. Lipasti. Modern Processor Design: Fundamentals of Superscalar Processors. McGraw-Hill, 2005.

15. I. Kim and M. H. Lipasti. "Understanding scheduling replay schemes". In HPCA'04: Proceedings of the 10th International Symposium on High Performance Computer Architecture, page 198, Washington, DC, USA, 2004. IEEE Computer Society.

16. D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.com

17. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2002.

18. Basu, Arkaprava; Kirman, Nevin; Kirman, Meyrem; Chaudhuri, Mainak; Martinez, Jose, "Scavenger: A New Last Level Cache Architecture with Global Block Priority," In the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007 (MICRO 2007), pp.421-432, 1-5 Dec. 2007.

19. Soundararajan, N. K., Parashar, A., and Sivasubramaniam, A. 2007. "Mechanisms for bounding vulnerabilities of processor structures." In Proceedings of the 34th Annual international Symposium on Computer Architecture (San Diego, California, USA, June 09 - 13, 2007). ISCA '07.

20. Parashar, A., Sivasubramaniam, A., and Gurumurthi, S. 2006. "SlicK: slice-based locality exploitation for efficient redundant multithreading." In Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems (San Jose, California, USA, October 21 - 25, 2006). ASPLOS-XII.

21. M. K. Qureshi, O. Mutlu, and Y. N. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05), pages 434–443, 2005.

22. Abella, J.; Canal, R.; Gonzalez, A., "Power- and complexity-aware issue queue designs," Micro, IEEE, vol.23, no.5, pp. 50-58, Sept.-Oct. 2003.

23. Moreshet, T. and R. I. Bahar. Complexity-Effective Issue Queue Design Under Load-Hit Speculation. In Workshop on Complexity-Effective Design, May 2002, Anchorage / AK.

24. Walcott, K. R., Humphreys, G., and Gurumurthi, S. 2007. Dynamic prediction of architectural vulnerability from microarchitectural state. In *Proceedings of the 34th Annual international Symposium on Computer Architecture* (San Diego, California, USA, June 09 - 13, 2007). ISCA '07.