Turbo-ROB: A Low Cost Checkpoint/Restore Accelerator

Patrick Akl and Andreas Moshovos

University of Toronto, Canada

Abstract. Modern processors use speculative execution to improve performance. However, speculative execution requires a checkpoint/restore mechanism to repair the machine's state whenever speculation fails. Existing checkpoint/restore mechanisms do not scale well for processors with relatively large windows (i.e., 128 or more). This work presents Turbo-ROB, a checkpoint/restore recovery accelerator that can complement or replace existing checkpoint/restore mechanisms. We show that the Turbo-ROB improves performance and reduces resource requirements compared to a conventional Re-order Buffer mechanism. For example, on the average, a 64-entry TROB matches the performance of a 512-entry ROB, while a 128- and a 512-entry TROB outperform the 512-entry ROB by 6.8% and 9.1% respectively. We also demonstrate that the TROB improves performance with register alias table checkpoints effectively reducing the need from more checkpoints and the latency and energy increase these would imply.

1 Introduction

Modern processors use control flow speculation to improve performance. The processor does not wait until the target of a control flow instruction is calculated. Instead, it predicts a possible target and speculatively executes instructions at that target. To preserve correctness, recovery mechanisms restore the machine's state on mispeculation. Recovery involves reversing any changes done by the incorrectly executed instructions and resuming execution at the correct target instruction. Modern processors utilize two such recovery mechanisms. The first is the reorder buffer (ROB) which allows recovery at any instruction in addition to mispeculated branches. Recovering from the ROB amounts to *squashing*, i.e., reversing the effects of each mispeculated instruction, a process that requires time proportional to the number of squashed instructions. The second recovery mechanism uses a number of global checkpoints (GCs) that are allocated prior to executing a branch and in program order. A GC contains a complete snapshot of all relevant processor state. Recovery at an instruction with a GC is "instantaneous", i.e., it requires a fixed, low latency. GCs are typically embedded into the Register Alias Table (RAT) since virtually all other processor structures do not need a checkpoint/restore mechanism for most of their resources (they maintain a complete record of all in-flight instructions and thus recovery is possible by simply discarding all erroneous entries).

Ideally, a GC would be allocated at every instruction such that the recovery latency is always constant. In practice, because the RAT is a performance critical structure only a limited number of GCs can be implemented without impacting the clock cycle significantly and thus reducing overall performance. For example, RAT latency increases respectively by 1.6%, 5%, and 14.4% when four, eight, and 16 GCs are used relative to a RAT with no GCs for a 512-entry window 4-way superscalar processor with 64 architectural registers and for a 130nm CMOS commercial technology [12]. Recent work suggested using selective GC allocation to reduce the number of GCs necessary to maintain high performance [1,2,5,11]. Even with these advances, at least eight and often 16 GCs are needed to maintain performance within 2% of that possible with an infinite number of checkpoints allocated at all branches with a 256-entry or larger window processor. These GCs increase RAT latency and the clock cycle and thus reduce performance. Moreover, RAT GCs increase RAT energy consumption. This is undesirable as the RAT exhibits high energy density. Accordingly, methods for reducing the number of GCs or for eliminating the need for GCs altogether would lead to improved performance and avoid an increase in power density in the RAT.

This work proposes *Turbo-ROB*, or *TROB*, a checkpoint recovery accelerator that can complement or replace the ROB and the GCs. The TROB is off the critical path and as a result its latency and energy can be tuned with greater flexibility. The Turbo-ROB is similar to the ROB but it requires a lot fewer entries since the TROB only stores information necessary to recover at a few selected branches, called *repair points*. Specifically, the TROB stores recovery information for the first update to each register after a repair point. Because programs tend to reuse registers often, many instructions are ignored by the TROB. In contrast, the ROB stores information for all instructions because it allows recovery at *any* instruction. While the ROB is a general mechanism that treats all recoveries as equal, the TROB is optimized for the common case of branch-related recoveries. The TROB can be used to accelerate recovery in conjunction with a ROB or GCs, or it can be used as a complete replacement for the ROB and the GCs. Unlike previous proposals for reducing the recovery latency, the Turbo-ROB does not require modifications to the RAT.

This paper makes the following contributions: (1) It proposes "Turbo-ROB", a ROB-like recovery mechanism that requires less resources than the ROB and allows faster recovery on the frequent case of control flow mis-speculations. Given that the TROB is off the critical path it alleviates some of the pressure to scale the register alias table and the re-order buffer. (2) It shows that the TROB can be used to improve performance over a ROB-only recovery mechanism. (3) It shows that the TROB can replace a ROB offering performance that is close to that possible with few GCs. Eliminating the ROB is desirable since it is an energy and latency inefficient structure [2]. (4) It shows that the TROB improves performance even when used with a GC-based recovery mechanism. More importantly, the TROB reduces GC pressure allowing implementations that use very few GCs. For example, the results of this paper demonstrate that

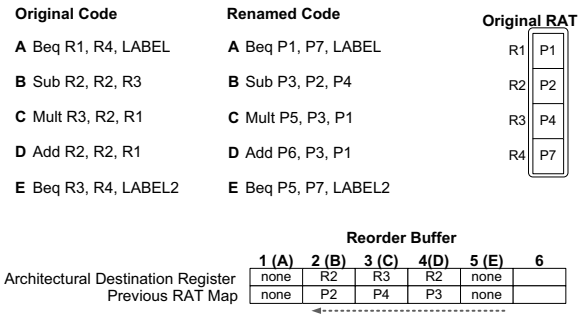

Fig. 1. *Given an instruction to recover at, it is not always necessary to process all subsequent instructions recorded in the ROB.*

a TROB with one GC performs as well as an implementation that uses four GCs. A single GC RAT implementation is simpler than one that uses more GCs.

The rest of this paper is organized as follows: Section 2 presents the TROB design. Section 3 reviews related work. Section 4 presents the experimental analysis of TROB. Finally, Section 5 concludes this work. In this paper we restrict our attention to recovery from control flow mispeculation. However, the TROB can also be used to recover from other exceptions such as page faults. In the workloads we study these exceptions are very infrequent. We also focus on relatively large processors with up to 512-entry instruction windows. However, we do demonstrate that the TROB is beneficial even for smaller window processors that are more representative of today's processor designs. We note, however, that as architects are revisiting the design of large window processor simplifying the underlying structures and as multithreading becomes commonplace it is likely that future processors will use larger windows.

2 Turbo-ROB Recovery

For clarity, we initially restrict our attention to using the TROB to complement a ROB recovery mechanism. In Sections 2.4 and 2.5 we discuss how the TROB can be used without a ROB or with GCs respectively. The motivation for Turbo-ROB is that not all instructions inserted in the ROB are needed for every recovery. We motivate the Turbo-ROB design by first reviewing how ROB recovery works.

The ROB maintains a log of all changes in program order. Existing ROB designs allocate one entry per instruction in the window. Each ROB entry contains sufficient information to reverse the effects of the corresponding instruction. For the RAT it is sufficient to keep the architectural register name and the previous physical register it mapped to. On a mis-speculation, *all* ROB entries for the wrong path instructions are traversed in reverse program order. While the ROB design allows recovery at *any* instruction, given a specific instruction to recover at, not all ROB entries need to be traversed. Specifically, for every RAT entry, only the first corresponding ROB entry after the mispredicted branch is needed.

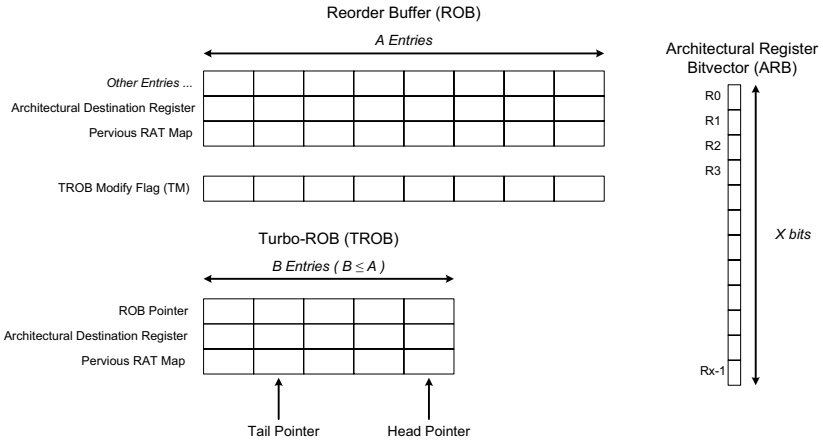

Fig. 2. *Turbo-ROB organization. This figure assumes that the Turbo-ROB complements a ROB.*

Moreover, branch instructions occupy ROB entries but do not modify the RAT. Figure 1 illustrates via an example these two points. The branch instruction A is mispredicted and the wrong path instructions B to E are fetched and decoded before the misprediction is discovered. The ROB-only recovery mechanism traverses the ROB entries *5* to *2* in reverse order updating the RAT. However, traversing just the entries *3* and *2* is sufficient: Entry *5* contains no state information since it corresponds to a branch; entry *4* corresponding to *R2* at instruction D can be ignored because the correct previous mapping (*P2*) is preserved by entry *2*. A mechanism that exploits these observations can reduce recovery latency and hence improve performance. The TROB mechanism presented next, exploits this observation. To do so, it allows recovery only on branches and relies on the ROB or in re-execution as in [2] to handle other exceptions.

2.1 Mechanism: Structure and Operation

We propose TROB, a ROB-like structure that requires fewer resources. TROB is optimized for the common case and thus allows recovery at *some* instructions, which we call *repair points*. The TROB records a subset of the information recorded in the ROB. Specifically, given a repair point B, the TROB contains at most one entry per architectural register corresponding to the first update to that register after B. Recoveries using the TROB are thus potentially faster than ROB-only recoveries. To ensure that recovery is still possible at all instructions (for handling exceptions), a normal ROB is used as a backup.

Figure 2 shows that the TROB is an array of entries that are allocated and released in program order. Each entry contains an architectural register identifier and a previous RAT map. Thus, for an architecture with X architectural registers and Y physical registers, each TROB entry contains $\log_2 X + \log_2 Y$ bits. A mechanism for associating TROB entries with the corresponding instructions

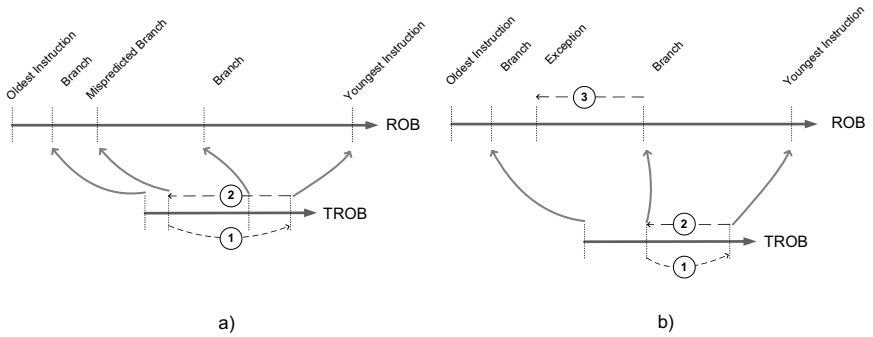

Fig. 3. *Turbo-ROB recovery scenarios when repair points are initiated at all branches. a) Common scenario: recovery at any mispredicted branch uses only the fast Turbo-ROB. b) Infrequent scenario: recovery at any other instruction is supported using the Turbo-ROB and the ROB.*

is needed. In one possible implementation, each TROB entry contains a ROB pointer. Thus an extra $\log_2 A$ bits per TROB entry are needed for an architecture with an A-entry reorder buffer. Detection of the first update to each register after a repair point is performed via the help of a single "Architectural Register Bitvector" (ARB) of size equal to the number of architectural registers (X bits). A "TROB modify" (TM) bit array with as many bits as the number of ROB entries is used to track which instructions in the ROB allocated a TROB entry. This is needed to keep the TROB in a consistent state.

Updating the Turbo-ROB: In our baseline design, repair points are initiated at all branches. Following a repair point, we keep track of the nearest subsequent *previous RAT map* for every RAT entry in the TROB. The ARB records which architectural registers have had their mapping in the RAT changed since the last repair point. The ARB is reset to all-zeros at every repair point and every time a TROB entry is created the corresponding ARB bit is set. TROB entries are created only when the corresponding ARB is zero. Finally, the corresponding TM is set, indicating that the corresponding instruction in the ROB modified the TROB. The TM facilitates in-order TROB deallocation whenever an instruction commits or is squashed. If the TROB has less entries than the ROB, it is possible to run out of free TROB entries. The base implementation stalls decode in this case.

Recovery: There are two recovery scenarios with the TROB: Figure 3(a) shows the first scenario where we recover at an instruction with a repair point. RAT recovery proceeds by traversing the TROB in reverse order while updating the RAT with the previous mappings. Figure 3(b) shows the second recovery scenario where we recover at an instruction without a repair point. In the base design that allocates repair points for all branches, this scenario is not possible for branches and applies only to other exceptions. Accordingly, this will happen infrequently.

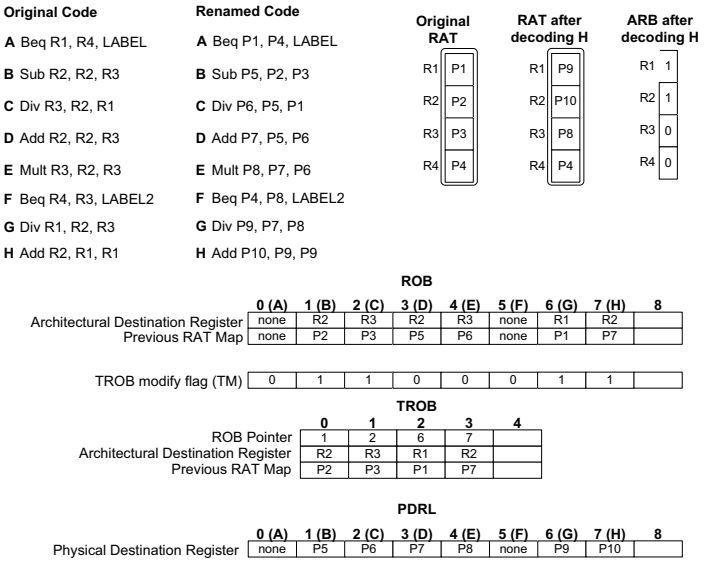

Fig. 4. *Example of recovery using the Turbo-ROB: repair points are initiated at instructions* **A** *and* **F**. *Recovering at* **A** *requires traversing entries* **3** *to* **0** *in the TROB. The physical register free list is recovered by traversing entries* **7** *to* **0** *in the PDRL.*

We first quickly recover the RAT partially at the closest subsequent TROB repair point. We then complete the recovery using the ROB, starting at the instruction corresponding to the repair point at which partial recovery took place. If no such repair point is found, we rely solely the ROB for recovery.

Reclaiming Physical Registers: Since the TROB contains only first updates, it can't be used to free all the physical registers allocated by wrong path instructions. We assume that the free register list (FRL) contains embedded checkpoints which are allocated at repair points. The free register list is typically implemented as a bit vector with one bit per register. Accordingly, checkpoints similar to those used for the RAT can be used here also. Since the FRL is a unidimensional structure embedding checkpoints does not impact its latency greatly as it does in the RAT. Upon commit, an instruction freeing a register, must mark it as free in the current FRL and in all active checkpoints by clearing all corresponding bits. Assuming that these bits are organized in a single SRAM column, clearing them requires extending the per column reset signal over the whole column plus a pull-down transistor per bit.

2.2 Recovery Example

Figure 4 illustrates an example of recovery using the TROB. Following the decode of branch A, instructions B and C perform first updates of RAT entries $R2$

and *R3* and allocate TROB entries *0* and *1*. The ARB is reset at instruction *F* and new TROB entries are allocated for instructions *G* and *H*, which perform new first updates to the RAT entries *R1* and *R2*. If branch *A* is mispredicted, recovery proceeds in reverse order starting from the TROB head (entry *3*) and until entry *0* is reached. Recovering via the ROB would require traversing seven entries as opposed to the four required by the TROB. If branch *F* was mispredicted, recovery would only use TROB entries *3* and *2*. If instruction *C* raised an exception, recovery proceeds using the ROB by traversing entries *7* to *3* in reverse order. Alternatively, we could recover by first recovering at the closest subsequent repair point using the TROB (recovering at instruction *F* by using the TROB entries *3* and *2*) and then use the ROB to complete the recovery (by using ROB entries *4* to *3*).

2.3 Selective Repair Point Initiation

Our baseline mechanism initiates repair points at all branches. Creating repair points less frequently reduces the space requirements for the TROB and makes TROB recovery faster. In the example shown in Figure 4, not creating a TROB repair point at branch *F* reduces TROB pressure since instruction *H* would not be saved. Since TROB entry *3* would not be utilized, repairing the RAT state using the TROB if instruction *A* was mispredicted would be faster. However, if *F* was mis-speculated then we would have to use the slower ROB to recover. To balance between decreasing the utilization and the recovery latency of the TROB on one side, and increasing the rate of slow recoveries that do not utilize solely the TROB on the other side, we can selectively create repair points at branches that are highly likely to be mispredicted. Confidence estimators dynamically identify such branches. Zero-cost confidence estimation has been show to work well with selective RAT checkpoint allocation [11]. In the rest of this study, we refer to the method that uses selective repair point initiation as *sTROB*.

2.4 Eliminating the ROB

The TROB can be used as a complete replacement for the ROB. In one design, the TROB replaces the ROB and repair points are initiated at every branch. In this case, recovery is possible via the TROB at every branch. Other exceptions can be handled using re-execution and a copy of the RAT that is updated at commit time. This policy was suggested by Akkary et al. [2] for a design that used only GCs and no ROB. To guarantee that every branch gets a repair point we stall decode whenever the TROB is full and a new entry is needed. This design artificially restricts the instruction window. However, as we show experimentally, this rarely affects performance. Alternatively, we can allow some branches to proceed without a repair point, or to abandon the current repair point and rely instead on re-execution as done for other exceptions.

In another design, the TROB replaces the ROB but repair points are initiated only on weak branches as predicted via a confidence estimation mechanism. In this design, it is not always possible to recover via the TROB. Whenever no

repair point exists, we rely instead on re-execution from an earlier repair point or from the beginning of the window.

2.5 TROB and In-RAT Global Checkpoints

Finally, the TROB can be used in conjunction with GCs with or without a ROB. If a ROB is available, recovery proceeds first at the nearest subsequent GC or repair point via the GCs or the TROB respectively. Then, recovery completes via the ROB. If no ROB is available, recovery first proceeds to the nearest earlier GC or repair point. Recovery completes by re-executing the intervening instructions as in [2]. In this paper we study the design that combines a TROB with few GCs and a ROB.

3 Related Work

Mispeculation recovery has been extensively studied in the literature. Related work can be classified into the following categories: 1) Reducing the mispeculation recovery latency, 2) Confidence estimation for speculation control, and 3) Multipath execution and instruction reuse. Due to space limitations, we restrict our attention to the first two categories noting that in principle the TROB is complementary to techniques in the third category.

Reducing the Mispeculation Recovery Latency: Aragon et al. analyzed the causes of performance loss due to branch mispredictions [3] and found that the *pipeline-fill* penalty is a significant source of performance loss due to mispredictions. The TROB reduces a significant component of this latency.

The Reorder Buffer, originally proposed by Smith and Pleszkun, is the traditional checkpointing mechanism used to recover the machine state on mispredictions or exceptions [13]. As previous studies have shown, recovering solely using the reorder buffer incurs significant penalties as the processor window increases [1,2,11,16]. To alleviate this concern, non-selective in-RAT checkpointing has been implemented in the MIPS R10000 processor [15]. Moshovos proposed an architecture where the reorder buffer is complemented with GCs taken selectively at hard-to-predict branches to reduce the GC requirements for large instruction window processors [11]. Akkary et al. proposed an architecture that does not utilize a reorder buffer and instead creates GCs at low confidence branches [1,2]. Recovery at branches without a GC proceeds by recovering at an earlier GC and re-executing instruction up until the mis-speculated branch. As we show in this paper, the TROB can be used in conjunction with in-RAT checkpointing. Modern checkpoint/recovery mechanisms have evolved out of earlier proposals for supporting speculative execution [7, 13, 14].

Zhou et al. proposed Eager Misprediction Recovery (EMR), which allows some instructions whose input registers' map were not corrupted to be renamed in parallel with RAT recovery [16]. While the idea is attractive, the implementation complexity has not been shown to be low. While Turbo-ROB can in principle be used with this approach also, this investigation is left for future work.

Confidence Estimation for Speculation Control: Turbo-ROB relies on a confidence estimator for identifying weak branches. Manne et al. propose throttling the pipeline's front end whenever too many hard-to-predict branches are simultaneously in-flight for energy reduction [10]. Moshovos proposed used a similar estimator based on the bias information from existing branch predictors, for selective GC allocation [11]. Jacobsen et al. proposed a more accurate confidence estimator whose implementation requires explicit resources [8]. This confidence estimator was used by Akkary et al. for GC prediction [1, 2] and by Manne et al. for speculation control [10]. Jimenez and Lin studied composite confidence estimators [9].

4 Experimental Results

Section 4.1 discusses our experimental methodology and Section 4.2 discusses the performance metric used throughout the evaluation. Section 4.3 demonstrates that TROB can completely replace the ROB offering superior performance with less resources. Section 4.4 studies the performance of a TROB with a backing ROB. Finally, Section 4.5 demonstrates that TROB improves performance with a GC-based configuration. For the most part we focus on a 512-entry instruction window configuration. We do so since this configuration places much higher pressure on the checkpoint/recovery mechanism. However, since most commercial vendors today are focusing on smaller windows we also show that TROB is useful even for smaller window processors.

4.1 Methodology

We used Simplescalar v3.0 [4] to simulate the out-of-order superscalar processor detailed in Table 1. We used most of the benchmarks from the SPEC CPU 2000 which we compiled for the Alpha 21264 architecture using HP's compilers and for the Digital Unix V4.0F using the SPEC suggested default flags for peak optimization. All benchmarks were run using a reference input data set. It was not possible to simulate some of the benchmarks due to insufficient memory resources. To obtain reasonable simulation times, samples were taken for one billion committed instructions per benchmark. To skip the initialization section in order to obtain representative results, we collected statistics after skipping two billion committed instructions for all benchmarks.

4.2 Performance Metric

We report performance results relative to an oracle checkpoint/restore mechanism where it is always possible to recover in one cycle. We refer to this unrealizable design as *PERF*. PERF represents the upper bound on performance for checkpoint/restore mechanisms if we ignore performance side-effects from mispeculated instructions (e.g., prefetching). Accordingly, in most cases we report performance *deterioration* compared to PERF. The lower the deterioration the

Table 1. *Base processor configuration.*

Branch Predictor	**Fetch Unit**
8K-entry GShare and 8K-entry bi-modal 16K Selector 2 branches per cycle	Up to 4 or 8 instr. per cycle 64-entry Fetch Buffer Non-blocking I-Cache
Issue/Decode/Commit	**Scheduler**
Up to 4 instr. per cycle	128-, 256- or 512-entry/half size LSQ
Functional Unit Latencies	**Main Memory**
Default simplescalar values	Infinite, 200 cycles
L1D/L1I Cache Geometry	**UL2 Cache Geometry**
64KBytes, 4-way set-associative with 64-byte blocks	1MByte, 8-way set-associative with 64-byte blocks
L1D/L1I/L2 Cache Latencies	**Cache Replacement**
3/3/16 Cycles	LRU
Fetch/Decode/Commit Latencies	
4 cycles + cache latency for fetch	

better the overall performance. Practical designs can perform at best as well as PERF and in most cases they will perform worse.

4.3 TROB as a ROB Replacement

In this section we study the performance of a design that replaces the ROB with a TROB. In this design repair points are initiated at every branch so that it is always possible to recover from a control flow mis-speculation using the TROB. Other exceptions are handled by re-executing from a preceding repair point similar to what was proposed in [1]. Whenever a new entry must be written into the TROB and the TROB is full, decode stalls until a TROB entry becomes available.

Figure 5 shows the per-benchmark and average performance deterioration relative to PERF with TROB as a function of the number of TROB entries. The first bar per benchmark represents the deterioration with ROB-only recovery. Lower deterioration implies higher performance. On the average, the 512-entry TROB outperforms the similarly sized ROB by 9.1%. As we decrease the number of TROB entries, performance deteriorates since decode occasionally stalls. However, on the average, even a 64-entry TROB performs slightly better than the 512-entry ROB. With a 128-entry TROB performance is just 2.5% short of that of a 512-entry TROB and 6.8% better than the ROB.

Per-benchmark behavior varies. For many programs, such as *gzip* and *vpr*, a 32-entry TROB performs better than a 512-entry ROB. In this case, the TROB reduces the number of cycles that are needed for recovery. By comparison, a 32-entry ROB reduces performance by more than 50% on the average. However, *swim*, *mgrid*, *applu* and *lucas* suffer when using a TROB with less than 256 entries. In these programs the instruction window is at full capacity most of the time because they exhibit a low rate of branches and nearly perfect predic-

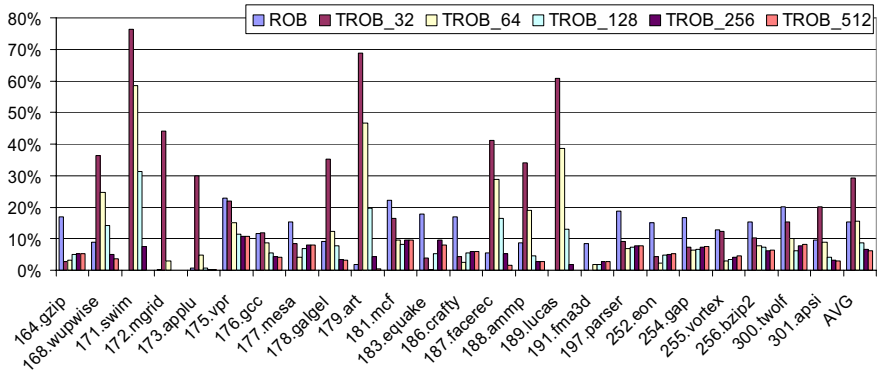

Fig. 5. *Per-benchmark and average performance deterioration relative to PERF with ROB-only recovery and TROB-only recovery as a function of the number of the TROB entries.*

tion accuracy. Moreover, these benchmarks tend to utilize most of the registers most of the time. As a result, a smaller TROB artificially restricts the instruction window. While not shown on the figure, a 384-entry TROB eliminates this problem for all programs. Thus, TROB always outperforms the ROB even while requiring fewer resources. Since mis-speculations are virtually non-existent for these benchmarks we expect that selective repair point allocation coupled with re-execution recovery as in [1] will be sufficient to avoid performance degradation for these benchmarks even with a smaller TROB. However, due to time limitations this study is left for future work.

The results of this section demonstrate that TROB-only recovery can improve performance significantly over ROB-only recovery. When TROB is given the same entries as ROB it always performs better. A 384-entry TROB that requires 25% less resources performs better or as well as a 512-entry ROB. With half the resources, TROB performs significantly better than ROB for most benchmarks and virtually identically for those that exhibit very low misprediction rates. With just 25% the resources of ROB, TROB achieves an average performance deterioration of 7.5% compared to PERF which is significantly lower than the 17% deterioration observed with the 512-entry ROB.

Smaller Window Processors Figures 6 and 7 report performance for TROB for processors with 128- and 256-entry instruction windows. The trends are similar to those observed for the 512-entry window processor, however, in absolute terms the differences are smaller. In either case, a TROB that has half the entries than the ROB it replaces is sufficient to improve performance. These results demonstrate that TROB is a viable ROB replacement for today's processors also.

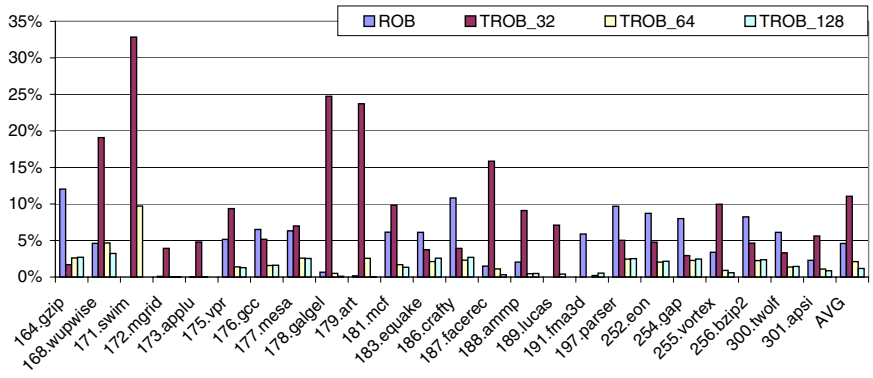

Fig. 6. *Per-benchmark and average performance deterioration relative to PERF with ROB-only recovery and TROB-only recovery as a function of the number of the TROB entries for a* **128-entry window processor**.

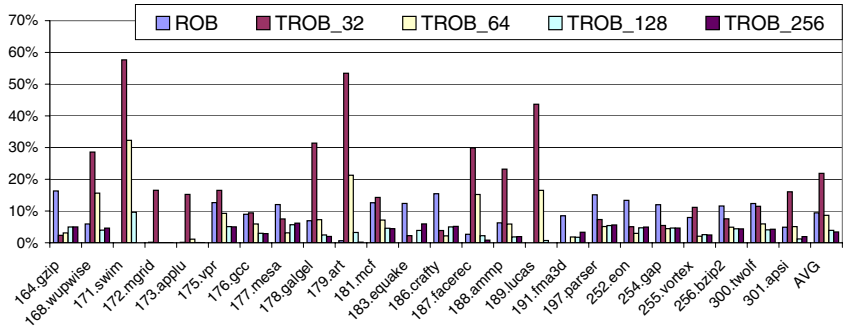

Fig. 7. *Per-benchmark and average performance deterioration relative to PERF with ROB-only recovery and TROB-only recovery as a function of the number of the TROB entries for a* **256-entry window processor**.

4.4 Selective Repair Point Initiation

In this section we complement a ROB with an sTROB which is a TROB that uses selective repair point initiation. In this case, the sTROB acts as a recovery accelerator. This design requires more resources than the ROB-only recovery mechanism, however, these resources are off the critical path. Recovery at a branch with a repair point proceeds via the TROB only, otherwise, it is necessary to use both the ROB and the sTROB. In the latter case, recovery proceeds first at the nearest subsequent repair point via the sTROB. It takes a single cycle to locate this repair point if it exists provided that we keep a small ordered list of all repair points at decode. Recovery completes via the ROB for the remaining instructions if any. Whenever the TROB is full, decode is not stalled but the current repair point if any is marked as invalid. This repair point and any preceding ones can no longer be used for recovery. Figure 8 shows the per-benchmark

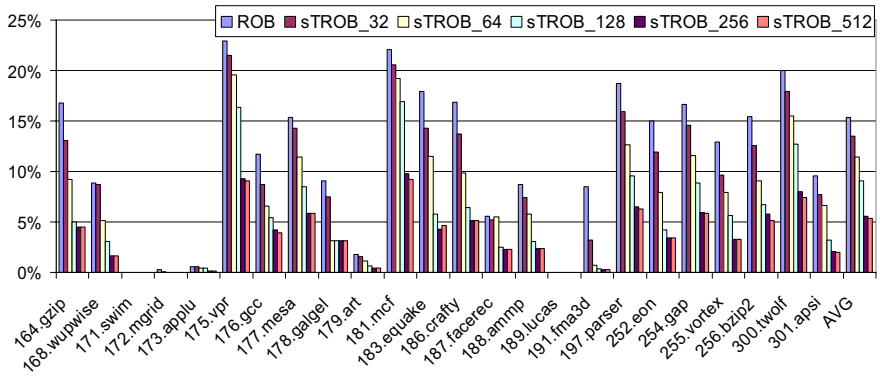

Fig. 8. *Per-benchmark and average performance deterioration relative to PERF with ROB-only recovery and sTROB recovery as a function of the number of sTROB entries.*

and average performance deterioration for sTROB recovery as a function of the number of sTROB entries. We use a 1K-entry table of 4-bit resetting counters to identify low confidence branches [8]. Very similar results were obtained with the zero-cost, anyweak estimator [11]. Average performance improves even with a 32-entry sTROB. These results demonstrate that the TROB can be used as a recovery accelerator on top of a conventional ROB. Comparing with the results of the next section, it can be seen that sTROB offers performance that is comparable to that possible with RAT checkpointing.

4.5 Turbo-ROB with GCs

This section demonstrates that the TROB successfully improves performance even when used with a state-of-the-art GC-based recovery mechanism [1,2,11]. In this design, the TROB complements both a ROB and in-RAT GCs. The TROB accelerates recoveries for those branches that could not get a GC. Recent work has demonstrated that including additional GCs in the RAT (as it is required to maintain high performance for processors with 128 or more instructions in their window) greatly increases RAT latency and hence reduces the clock cycle. Accordingly, in this case the TROB reduces the need for additional RAT GCs offering a high performance solution without the impact on clock cycle. The TROB is off the critical path and it does not impact RAT latency as GCs do.

Figure 9 shows the per-benchmark and average performance deterioration relative to PERF with (1) GC-based recovery (with one or four GCs), (2) sTROB, and (3) sTROB with GC-based recovery (sTROB with one or four GCs). We use a 1K-entry table of 4-bit resetting counters to identify low confidence branches [8]. A low confidence branch is given a GC if one is available, otherwise, a repair point is initiated for it in the TROB. If the TROB is full, decode stalls until space becomes available. In this experiments we use a 256-entry sTROB. On average, the sTROB with only one GC outperforms the GC-based only mechanism that utilizes four GCs. When four GCs are used with the

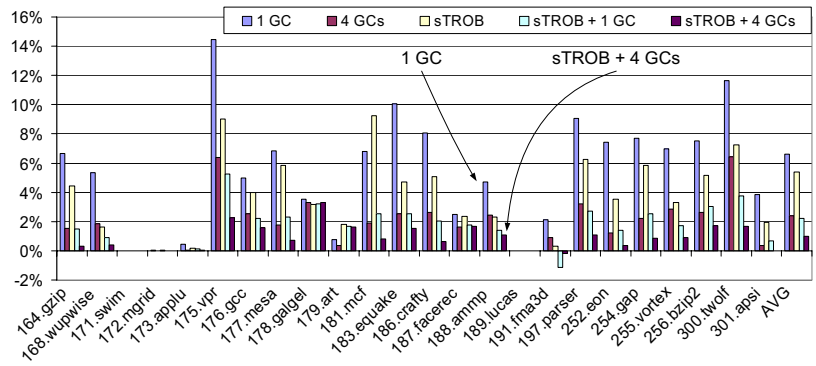

Fig. 9. *Per-benchmark and average performance deterioration relative to PERF with GC-based recovery (one or four GCs), sTROB, and sTROB with GC-based recovery (sTROB with one or four GCs).*

sTROB, performance deterioration is 0.99% as opposed to 2.39% when only four GCs are used. This represents a 59% reduction in recovery cost. These results demonstrate that the TROB is useful even with in RAT GC checkpointing.

5 Conclusions

In this work, we presented "Turbo-ROB", a recovery accelerator that can complement or replace existing checkpoint/restore mechanisms. The TROB is a novel low cost and complexity structure that can be used to reduce the negative performance effects of checkpointing and restore in modern architectures. We have shown that TROB can completely replace a ROB and studied its performance assuming that repair points are initiated at every branch. We have also shown that the TROB can complement a conventional ROB acting as an accelerator. Additional benefits were possible with larger TROBs. Finally, we have shown that TROB can reduce the pressure for global checkpoints in the RAT.

6 Acknowledgements

We would like to thank the anonymous reviewers for their time and comments. This work was supported by an NSERC Discovery Grant, an equipment grant from the Canada Foundation for Innovation and from an equipment donation from Intel Corporation.

References

[1] H. Akkary, R. Rajwar, and S. Srinivasan, "An Analysis of Resource Efficient Checkpoint Architecture", *ACM Transactions on Architecture and Code Optimization (TACO)*, Volume 1, Issue 4, Dec. 2004.

[2] H. Akkary, R. Rajwar, and S. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Instruction Window Processors", *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2003.

[3] J. L. Aragon, J. Gonzalez, A. Gonzalez, and J. E. Smith, "Dual Path Instruction Processing", *Proceedings of the 16th International Conference on Supercomputing, pp. 220-229*, Jun., 2002.

[4] D. Burger and T. Austin, "The Simplescalar Tool Set v2.0, Technical Report UW-CS-97-1342", *Computer Sciences Department, University of Wisconsin-Madison*, June 1997.

[5] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Kilo-Instruction Processors", *Proceedings The 5th International Symposium on High Performance Computing (ISHPC-V)*, Oct., 2003.

[6] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun, "Confidence Estimation for Speculation Control", *in Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[7] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines", *Proceedings of the 14th Annual Symposium on Computer Architecture*, June 1987.

[8] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions", *Proceedings of the 29th Annual International Symposium on Microarchitecture*, Dec. 1996.

[9] D. A. Jimenez and C. Lin, "Composite Confidence Estimators for Enhanced Speculation Control", *Technical Report TR-02-14, Department of Computer Sciences, The University of Texas at Austin*, Jan. 2002.

[10] S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction", *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[11] A. Moshovos, "Checkpointing Alternatives for High Performance, Power-Aware Processors", *Proceedings of the IEEE International Symposium Low Power Electronic Devices and Design (ISLPED)*, Aug. 2003.

[12] E. Safi, P. Akl, A. Moshovos, A. Veneris and A. Arapoyianni, "On the Latency, Energy and Area of Superscalar Renaming Tables", *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Devices*, Aug. 2007.

[13] J. Smith and A. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors", *IEEE Transactions on Computers, 37(5)*, May 1988.

[14] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers", *IEEE Transactions on Computers*, March 1990.

[15] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor", *IEEE MICRO*, 1996.

[16] P. Zhou, S. Onder, and S. Carr, "Fast Branch Misprediction Recovery in Out-of-Order Superscalar Processors", *Proceedings of the International Conference on Supercomputing*, June 2005.