Almost Scalable Pipeline Flush

Arthur Perais, Trevor E Carlson

Compas'2024 : Parallélisme/ Architecture / Système
IETR/LS2N/Nantes-U - Nantes - France, 2-5 juillet 2024

Almost Scalable Pipeline Flush

Arthur Perais†, Trevor E. Carlson‡

†Univ. Grenoble Alpes, CNRS, Grenoble INP / ‡ National University of Singapore
Laboratoire TIMA - 46 avenue Félix Viallet / 4 Engineering Drive 3
3800 Grenoble - France / Singapore 117583
arthur.perais@univ-grenoble-alpes.fr / tcarlson@comp.nus.edu.sg

Résumé
Modern high-performance general purpose processors rely on an instruction window from which instructions are executed in a dataflow manner. Instruction window scaling is notably tied to register renaming. Indeed, on a pipeline flush some designs repair the Rename Map Table (RMT, maps architectural to physical registers) by iteratively undoing flushed mappings. This incurs a latency penalty that grows linearly with the instruction window size. To address this, modern designs use RMT checkpoints, which makes the latency of restoring the RMT state constant. However, RMT checkpoints increase the delay and power consumption of the renaming process, and can cause correct path instructions to be thrown away. In this paper, we revisit the algorithms used to restore the RMT state and propose an alternative to full-fledged checkpointing that provides comparable performance using a single checkpoint.

Mots-clés : out-of-order, pipeline flush, general-purpose, microarchitecture

1. Introduction

Technology advances have allowed major structures to keep growing across processor generations. For instance, over the past 25 years, the Reorder Buffer (ROB) of Intel processors has grown from 40 entries in P6 to 512 entries in Golden Cove. A larger ROB can impact the time required to recover the microarchitectural state when a mispeculation takes place. Indeed, after a branch misprediction, new instructions cannot be renamed until the Rename Map Table (RMT), which contains the most recent (i.e., speculative) mappings of architectural registers to physical registers, [1] has been restored to a consistent state. Simpler designs use the ROB to repair the RMT, by walking it from the tail to the flush point (Figure 1 (a)). Variations of this algorithm exist, e.g., walking from the commit RMT to the flush point, or even dynamically walking from the FIFO end that is closest to the flush point [1]. However, those variations still perform recovery iteratively and suffer from the same inherent scaling limit that a small, fixed number of mappings can be restored each cycle. To accelerate recovery, modern microarchitectures leverage checkpointing [1]. Checkpointing can be complementary to the ROB (Figure 1 (b)) [24, 19], in which case instruction commit is still driven by the ROB, or a complete replacement of the ROB [1], in which case all instructions within the oldest checkpoint have to be ready to commit for the checkpoint to commit. Moreover, checkpointing is more hardware-intensive and a larger number of RMT checkpoints can significantly impact its energy consumption and latency,

1. A Commit RMT may also be implemented. It contains the mappings corresponding to the current architectural state.

reducing performance gained from faster RMT repair [21]. As a result, this is not necessarily the best tradeoff.

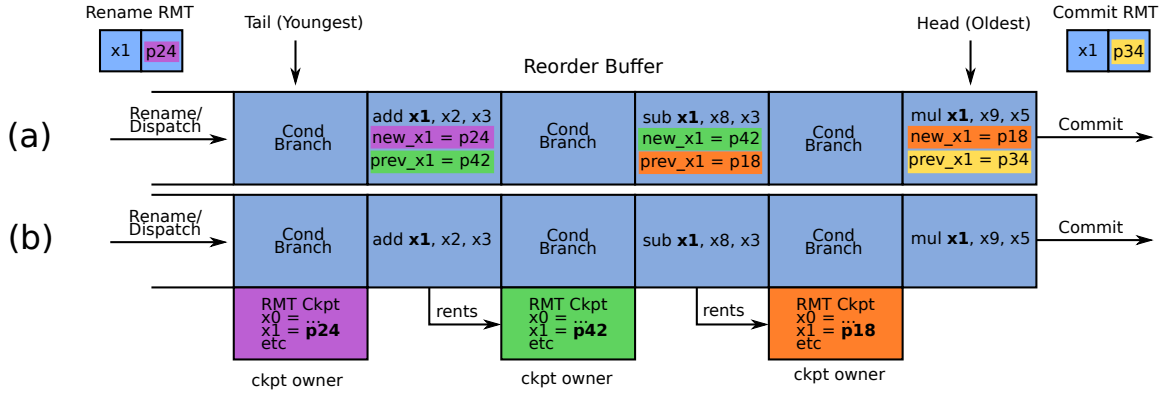

FIGURE 1 – (a) Recovering the RMT through logging mappings in the ROB (iterative). (b) Recovering the RMT through whole RMT checkpoints attached to conditionnal branches.

2. Motivation

We first introduce the following definitions. A dynamic instruction is i) **Unsafe** if it can still cause a pipeline flush (e.g., unresolved branch), ii) **Safe** if it is not unsafe iii) A checkpoint **owner** if an RMT checkpoint corresponds to the state of the RMT immediately prior the instruction is renamed and iv) A checkpoint **renter** if it is not a checkpoint *owner*.

2.1. RMT Repair Scalability

The RMT recovery mechanism should satisfy two requirements : i) Constant recovery latency while the window scales and ii) Require as little hardware as possible. Iterative schemes that walk the ROB do not provide i) although they provide ii) [21]. Conversely, checkpoint-based schemes generally provide i) since recovering the RMT requires a single bulk copy. However, they do not provide ii) as in-RMT checkpoints do incur a latency penalty and can be detrimental to performance [21]. In this work, we consider the following algorithms :

- **Shortest Walk (Iterative) :** The ROB is walked either from the head or the tail at the pace of 8 instructions per cycle, whichever is shorter.
- **Checkpoint Processing & Recovery (CPR) [1] (Immediate) :** All indirect and conditionnal branches take a checkpoint of the RMT at Rename. If a non-branch instruction flushes the pipeline, the youngest older checkpoint is restored. 48 checkpoints are implemented (Intel Sunny Cove). An alternative scheme, *CPR LoConfBr*, allocates checkpoints only to non-confident branches (non saturated branch prediction counter [23]). In *CPR*, checkpoints completely replace the ROB.
- **CPR + ROB (Immediate + Iterative) :** Same as CPR, except if a non-branch instruction flushes the pipeline, the youngest older checkpoint is restored and the ROB is walked from the youngest older checkpoint to the flush point (8 insts. per cycle.)
- **Idealistic :** All instructions are associated with an RMT checkpoint.

2.2. Limitations of Checkpoint-based Schemes

2.2.1. CPR

Depending on branch density, if all branches obtain a checkpoint, a fixed number of checkpoints might provide a more or less large instruction window. That is, if branch density is too

high, performance may degrade compared to a simple ROB. As a result, CPR schemes often propose to allocate checkpoints selectively, leveraging branch confidence estimators [15, 10, 23].
Moreover, if a *renter* triggers a flush (e.g., load that suffers a hard page fault), useful work has to be thrown away because the pipeline restarts from the youngest older checkpoint. In addition, in highly predictable workloads or workloads with few branches, checkpoints can grow very large. The need for all instructions in a checkpoint to be *safe* before retiring the checkpoint can stall resource reclamation. For instance if large quantities of Store Queue (SQ) entries are associated with the oldest checkpoint, a long time may pass until a single new instruction is allowed to Dispatch, because no SQ entry is available.
For both these reasons, CPR can benefit from enforcing a finite span for each checkpoint [9], so as to bound the number of instructions that are thrown away if a *renter* triggers a flush, and to limit the pipeline resources held by each checpoint. Unfortunately, this limits the size of the instruction window to the number of checkpoints times the maximum size of a checkpoint.

2.2.2. CPR + ROB

As in CPR, letting checkpoints grow arbitrarily can be detrimental for *CPR + ROB* as a flush by a *renter* instruction can still require several cycles to repair the RMT iteratively. Limiting the span of each checkpoint is a solution to mitigate this issue.
Another limitation compared to CPR is that since a ROB is still present, the window size is limited by the ROB wisze, whereas CPR can implement a very large window with only a few checkpoints, unless checkpoint instruction span is capped to a small number.

3. Speculatively Retiring from the ROB to Improve Scaling

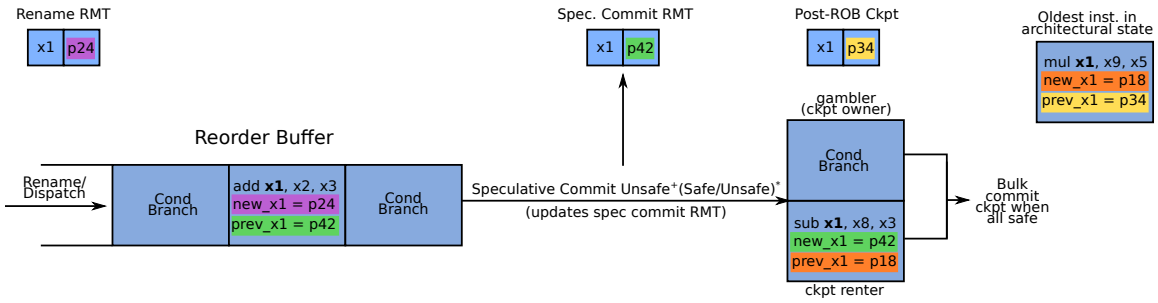

FIGURE 2 – Allowing older instructions to speculatively retire from the ROB

"Post-ROB" RMT checkpoints have previously been proposed in several other contexts [5, 18]. The overall idea is to track instruction state coarsely, which allows costly "per-instruction" resources to be reallocated to newer instructions. The high-level idea is depicted in Figure 2, in which older *unsafe* instructions are allowed to make space in the ROB for newer instructions and "pile-up" in a single RMT checkpoint, increasing the instruction window size. Previous work is conservative in what instructions are allowed to leave the realm of precise state tracking and enter the "post-commit" checkpoint [18]. Indeed, although many instructions at the head of the ROB are not yet *safe*, they are very likely to become *safe* in the future (e.g., confident branches). By allowing those instructions to take a checkpoint, we can retire them from the ROB, as well as any subsequent instructions that are already *safe* or likely to become *safe*. Practically, we implement a single commit RMT checkpoint, for a total of three versions of the RMT : The rename RMT, the commit RMT (CRMT), and the checkpoint. When an *unsafe* ins-

truction is at the head of the ROB and the number of free ROB entries is below a threshold, it takes a checkpoint of the CRMT and is removed from the ROB, although it retains any other resource it may need to become *safe* (e.g., SQ entry). We refer to this instruction as the *gambler*.

3.1. Who can Gamble? Who can Rent?

Intuitively, the *gambler* should be likely to become safe, hence be an unresolved but *confident* branch, or a memory instructions since hard page faults are rare. However, and perhaps counter-intuitively, an unresolved *non-confident* branch is also a good candidate to be the *gambler*. Indeed, if the *gambler* turns out to cause a flush, then there is no additional cost to restoring the checkpoint rather than the CRMT, which would have been restored if the *gambler* had remained stuck at the head of the ROB. Conversely, if the *gambler* completes without issue, which is the case most of the time even for *non-confident* branches, then we have succesfully increased the window size. In other words, all branches can gamble. After the checkpoint has been created, instructions can continue to retire from the ROB into the checkpoint. Both *safe* and *unsafe* but likely to become *safe* instructions retire from the ROB and become *renters* of the *gambler's* checkpoint. In our proposal, this includes unresolved *confident* branches and unresolved memory operations that are still subject to potential memory ordering hazards. However, this is implementation dependent and the designer could for instance choose to not retire loads that are still subject to *read-after-write* hazards, as in Cherry [18].

3.2. Checkpoint Size

Like in CPR, the checkpoint cannot retire until there are still *unsafe* instructions covered by it. By not capping its size, it would therefore be possible that the checkpoint never retires as more *unsafe* instructions keep becoming *renters*. As a result, in this work, we explicitly limit the size of the checkpoint to 512 instructions, to allow a big enough instruction window while limiting the amount of work thrown away if a *renter* flushes. An alternative would be to implement multiple "post-ROB" checkpoints, which, although it would defeat the purpose of having a single checkpoint to limit hardware cost, would not be as costly as multiple RMT checkpoints as we take CRMT checkpoints and the CRMT is less timing critical than the RMT as it does not participate in source renaming.

4. Experimental Framework

We evaluate our proposal in a cycle-level, full-system simulator, gem5 [4]. We implement different RMT repair algorithms, including *CPR* and *CPR + ROB*. Table 1 summarizes the processor modelled in this study, which is on par with Intel Sunny Cove and Apple M1. [2] We scale the pipeline structures (Instuction Queue, Load/Store Queue, ROB and physical registers) of the baseline configuration by 1x, 2x and 4x to study the scalability of our proposal. In our proposed microarchitecture, referred to as *SPCOM* in this Section, the retire logic starts speculatively retiring from the ROB when the ROB is close to being full (24 entries left in our case), regardless of ROB size. To distinguish between confident and non-confident branches, we use the counter value from the TAGE branch predictor [23] and have implemented 2-bit confidence counters in the indirect branch predictor. We set the saturating probability of counters to $\frac{1}{256}$. We run SPEC2k17 [6] speed workloads (*reference* inputs) using the Simpoints sampling methodology [11]. A total of 215 100M instructions simpoints are obtained from running the first 1000B instructions in 28 workloads. Workloads were compiled for Aarch64 using gcc 8.3 -O3. Prior to

2. The L1I latency is set to one cycle to allow 0-cycle taken branch penalty. The TLBs are sized optimistically as we do not model a second level TLB and the impact of TLB hitrate on performance is not the concern of this work.

TABLE 1 – Gem5 processor configuration (13-stage pipeline, 3GHz)

Branch Prediction	64KB, 1+15-table TAGE predictor [22] Min/Max Hist. length : 5/640, 8192-entry BTB, 1k-entry Indirect Branch Target Cache, 32-entry Return Address Stack
Fetch	16-wide fetch from 64B Line Buffer 1 taken branch/cycle 32-instruction fetch queue, 1-cycle taken branch penalty, 3-cycle Fetch to Decode
Decode	8-wide, Mistarget detection (BTB miss), 1-cycle Decode to Rename
Rename	8-wide, 3-cycle Rename to Dispatch, 48 checkpoints (CPR and CPR + ROB)
Dispatch/ Commit	8-wide, 352-entry Reorder Buffer, 160-entry Instruction Queue, 148-entry Load Queue, 106-entry Store Queue, 280 INT Regs, 280 FP/SIMD Regs
Issue	Up to 15 instructions per cycle into : 4 simple ALU, 2 (simple ALU or IntMul(3c)), 1 IntDiv(20c, not pipelined), 3 (simple FP/SIMD(3c) or FP/SIMD Mul(4c mul/5c mac)) 1 (simple FP/SIMD(3c) or FP/SIMD Mul(4c mul/5c mac) or FP/SIMD Div (12c, not pipelined)), 2 Loads, 2 Stores Store Sets [7] mem. dep. pred. (2k-entry LFST, 2k-entry SSIT)
Caches	128KB 8-way L1D, 64B line size, 4c load-to-use, 56 MSHRs, LRU 128KB 8-way L1I, 64B line size, 1c load-to-use, 8 MSHRs, LRU 2MB 8-way L2, 64B line size, 12c load-to-use, 64 MSHRs, LRU 8MB 16-way L3, 64B line size, 37c load-to-use, 64 MSHRs, LRU
Memory	4-channel 16GB DDR3_1600 (Micron MT41J512M8 datasheet)
TLBs	256-entry 1-way L1I (0c) + 1k-entry 1-way L1D (0c) TLBs
Prefetchers	L1D : Stride Prefetcher, degree 4 [8], L2 : AMPM Prefetcher [14]

running simpoint regions, we warm up the processor structures for 50M instructions.

5. Experimental Results

We generally consider three different *SPCOM* configuration for our proposal : *SPCOM 0.5x ROB*, *SPCOM 0.75x ROB* and *SPCOM 1x ROB*. Those configurations share the same parameters regarding what and how to speculatively retire from the ROB, and differ only in the actual number of ROB entries they implement, relative to the baseline. For instance, when using the unscaled baseline (1x), the ROB has 352 entries, and so *SPCOM 0.5x ROB* only implements 176. Figure 3 reports performance normalized to *Idealistic* for the unscaled pipeline (1x). For both *SPCOM* and *CPR (+ ROB)*, we can observe that some workloads show performance gain, while some show performance loss. The former typically stem from the larger instruction window provided by both *SPCOM* and *CPR*, e.g., *perlbench_3*, *mcf*, *omnetpp*, *wrf*, *deepsjeng* and *imagick*. Interestingly, in *deepsjeng*, only *CPR* shines as although a larger window is beneficial, unconfident branches prevent the *SPCOM* checkpoint from growing large in *SPCOM 1x ROB* (35 instruction on average). In *imagick*, only *SPCOM 1x ROB* outperforms *Idealistic* as it icreases the window size without holding onto SQ entries for too long because checkpoints are very large. Conversely, performance loss arises for multiple reasons. For *CPR*, losses come from i) Wasted work caused by *renter*-caused pipeline flushes (*cactuBSSN*, *pop2*) and ii) Staggered resource recycling (SQ in *exchange_2*, *lbm*). For *SPCOM* configurations, the loss arises from i) A smaller window (smaller ROB) and the inability to speculatively retire due to the presence of an unresolved non-confident branches near the head of the ROB and ii) Wasted work being thrown away when a *renter* flushes the checkpoint. Yet, focusing on *SPCOM 0.75x ROB*, we observe that although performance is occasionally lost, there are no "dip" such as those of *CPR* in *cactuBSSN* and *pop2* and performance is thus more "even". Overall, *SPCOM 0.75x ROB* achieves performance on par with *Idealistic*, with 9 workloads out of 28 losing between 1 and 2% performance, and 6 workloads out of 28 gaining more than 1% performance (max. around 6% in *imagick*). Further tuning of the *SPCOM* parameters (e.g., checkpoint size, ROB occupancy threshold, branch confidence, etc.) is left for future work.

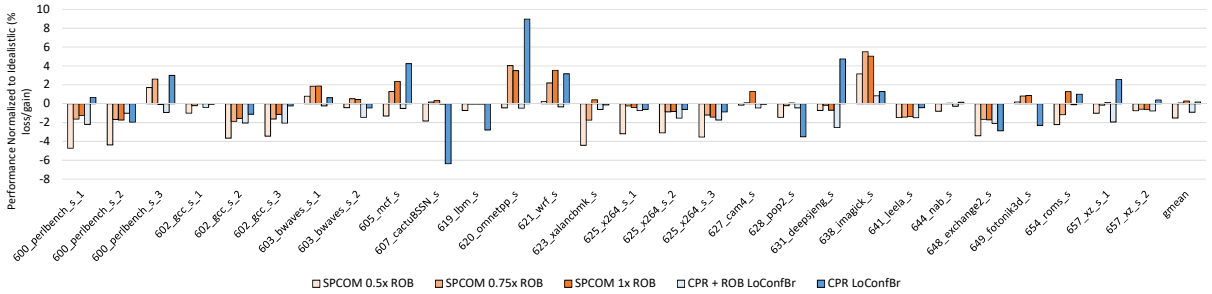

FIGURE 3 – Performance of various speculative commit configurations along checkpoint-based configurations, normalized to *Idealistic* using the **1x** pipeline configuration.

By lack of space, we do not show performance results for the 2x and 4x configurations as the the trend is generally similar, with a handful notable differences. In 4x, the ROB has become much larger (1452 entries). As a result, the iterative repair penalty starts limiting and even taking over performance in some workloads. For instance, in *x264_2*, the performance degrades as the ROB size of *SPCOM* is increased from 0.5x to 1x. Nonetheless, average performance of *SPCOM 0.75x ROB* remains on par with *CPR LoConfBr*, although both provide a slowdown of around 1% compared to *Idealistic* in 4x, when they matched its performance in 1x and 2x configurations.

6. Related Work

The MIPS R10K featured a 32-entry *Active List* to undo mappings iteratively in case of exceptions [24], from the youngest instruction to the oldest one. An alternative is to implement a commit RMT and to immediately copy it to the rename RMT on a pipeline flush. The mappings are then reapplied to the rename RMT from the head of the ROB to the flush point [1]. Several other schemes rely on the observation that repairing all the RMT entries does not require performing an entire ROB walk. The idea is that only the most recent mapping older than the flush point have to be restored [17, 3, 25]. Our proposal is orthogonal of the iterative algorithm, although poorly scaling algorithms require *SPCOM* to be more aggressive.
Hwu and Patt first formalize the notion of using checkpoints to recover from branch mispredictions and exceptions and provide circuit level implementations [13]. Safi et al. [21, 20] point out that implementing RMT checkpoints directly in the RMT array does not scale and can lead to performance degradation simply because the RMT is on the critical path. Moshovos proposes two improvements that improve the efficiency of checkpointing [19]. Akkary et al. first introduced CPR as a way to both increase the window size and hasten RMT repair [1]. CPROB improves on CPR by allowing precise recovery even if a *renter* triggers a misprediction [12] following the same observation as [17, 25, 3]. Both Golander and Weiss [9] and Akl and Moshovos [2] provide improvements on checkpoint allocation heuristics.

7. Conclusion

In this work, we introduced an alternative way to limit the RMT repair latency, which relies on scaling the ROB *down* and backing it with a single "post ROB" checkpoint. This design achieves average performance on par with both an idealistic RMT repair scheme as well as a machine without a ROB but only checkpoints, and scales reasonably well to larger machines despite fixed per-cycle ROB walk rate.

Bibliographie

1. Akkary (H.), Rajwar (R.) et Srinivasan (S.). – Checkpoint processing and recovery : towards scalable large instruction window processors. – In *Proc. of the Intl. Symp. on Microarchitecture*, pp. 423–434, 2003.
2. Akl (P.) et Moshovos (A.). – Branchtap : Improving performance with very few checkpoints through adaptive speculation control. – In *Proc. of the Intl. Conf. on Supercomputing*, pp. 36–45, 2006.
3. Akl (P.) et Moshovos (A.). – Turbo-rob : A low cost checkpoint/restore accelerator. – In *Proc. of the Intl. Conf. on High-Performance Embedded Architectures and Compilers*, pp. 258–272. Springer, 2008.
4. Binkert (N.), Beckmann (B.), Black (G.), Reinhardt (S. K.), Saidi (A.), Basu (A.), Hestness (J.), Hower (D. R.), Krishna (T.), Sardashti (S.) et al. – The gem5 simulator. *ACM SIGARCH computer architecture news*, vol. 39, n2, 2011, pp. 1–7.
5. Blundell (C.), Martin (M. M.) et Wenisch (T. F.). – Invisifence : performance-transparent memory ordering in conventional multiprocessors. – In *Proc. of the Intl. Symp. on Computer architecture*, pp. 233–244, 2009.
6. Bucek (J.), Lange (K.-D.) et v. Kistowski (J.). – Spec cpu2017 : Next-generation compute benchmark. – In *Companion of the 2018 ACM/SPEC Intl. Conf. on Performance Engineering*, pp. 41–42, 2018.
7. Chrysos (G.) et Emer (J.). – Memory dependence prediction using store sets. – In *Proc. of the Intl. Symp. on Computer Architecture*, pp. 0142–0142. IEEE, 1998.
8. Fu (J. W.), Patel (J. H.) et Janssens (B. L.). – Stride directed prefetching in scalar processors. – In *Proc. of the Intl. Symp. on Microarchitecture*, pp. 102–110, 1992.
9. Golander (A.) et Weiss (S.). – Checkpoint allocation and release. *ACM Transactions on Architecture and Code Optimization*, vol. 6, n3, 2009, pp. 1–27.
10. Grunwald (D.), Klauser (A.), Manne (S.) et Pleszkun (A.). – Confidence estimation for speculation control. – In *Proc. of the Intl. Symp. on Computer architecture*, pp. 122–131, 1998.
11. Hamerly (G.), Perelman (E.), Lau (J.) et Calder (B.). – Simpoint 3.0 : Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, vol. 7, n4, 2005, pp. 1–28.
12. Hilton (A.), Eswaran (N.) et Roth (A.). – Cprob : Checkpoint processing with opportunistic minimal recovery. – In *Proc. of the ACM Intl Conf. on Parallel Architectures and Compilation Techniques*, pp. 159–168. IEEE, 2009.
13. Hwu (W.-m. W.) et Patt (Y. N.). – Checkpoint repair for out-of-order execution machines. – In *Proc. of the Intl. Symp. on Computer architecture*, pp. 18–26, 1987.
14. Ishii (Y.), Inaba (M.) et Hiraki (K.). – Access map pattern matching for data cache prefetch. – In *Proc. of the Intl. Conf. on Supercomputing*, pp. 499–500, 2009.
15. Jacobsen (E.), Rotenberg (E.) et Smith (J. E.). – Assigning confidence to conditional branch predictions. – In *Proc. of the Intl. Symp. on Microarchitecture*, pp. 142–152. IEEE, 1996.
16. Jiménez (D. A.). – Composite confidence estimators for enhanced speculation control. – In *Proc. of the Intl. Symp. on Computer Architecture and High Performance Computing*, pp. 161–168. IEEE, 2009.
17. Jin (Z.), Aşilioğlu (G.) et Önder (S.). – Mower : A new design for non-blocking misprediction recovery. – In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 285–294, 2015.
18. Martínez (J. F.), Renau (J.), Huang (M. C.) et Prvulovic (M.). – Cherry : Checkpointed early resource recycling in out-of-order microprocessors. – In *Proc. of the Intl. Symp. on Microarchitecture (MICRO)*, pp. 3–14. IEEE, 2002.

19. Moshovos (A.). – Checkpointing alternatives for high performance, power-aware processors. – In *Proc. of the Intl. Symp. on Low Power Electronics and Design*, pp. 318–321, 2003.
20. Safi (E.), Akl (P.), Moshovos (A.), Veneris (A.) et Arapoyianni (A.). – On the latency, energy and area of checkpointed, superscalar register alias tables. – In *Proc. of the Intl. Symp. on Low power electronics and design*, pp. 379–382, 2007.
21. Safi (E.), Moshovos (A.) et Veneris (A.). – On the latency and energy of checkpointed superscalar register alias tables. *IEEE transactions on very large scale integration systems*, vol. 18, n3, 2009, pp. 365–377.
22. Seznec (A.). – A new case for the tage branch predictor. – In *Proc. of the Intl. Symp. on Microarchitecture*, pp. 117–127, 2011.
23. Seznec (A.). – Storage free confidence estimation for the tage branch predictor. – In *Proc. of the Intl. Symp. on High Performance Computer Architecture*, pp. 443–454. IEEE, 2011.
24. Yeager (K.) et [a]The (M.). – R10000 superscalar microprocessor. *IEEE Micro*, vol. 16, n4, 1996.
25. Zhou (P.), Önder (S.) et Carr (S.). – Fast branch misprediction recovery in out-of-order superscalar processors. – In *Proc. of the Intl. Conf. on Supercomputing*, pp. 41–50, 2005.