Reliability-Aware Runahead

Ajeya Naithani
Ghent University, Belgium

Lieven Eeckhout
Ghent University, Belgium

***Abstract*—Decreasing voltage levels and continued transistor scaling have drastically increased the chance of a processor bit encountering a soft error. We find that the microarchitecture state in an out-of-order core is vulnerable to soft errors especially while waiting for data to return from memory. The severity of the problem is further aggravated by the increasingly large size of microarchitecture state with every new processor generation. Prior solutions are ineffective as they incur too high overhead in terms of chip area, energy consumption and/or performance.**

In this paper, we make the observation that runahead execution, which was originally conceived to improve performance, also improves soft-error reliability as an unintended side effect. While the state-of-the-art runahead technique, Precise Runahead Execution (PRE), leads to substantial performance improvements, reliability is suboptimal still. We propose Reliability-Aware Runahead (RAR) which substantially improves soft-error reliability over the current state-of-the-art by rendering the microarchitecture state non-vulnerable during runahead execution and by initiating runahead execution early. Across a set of memory-intensive applications — the primary target for runahead execution — RAR improves the mean-time-to-failure (MTTF) by on average 4.8× (and up to 35.8×) relative to an out-of-order baseline while at the same time improving performance by 33.5% on average (and up to 2.6×). Across a broader set of compute- and memory-intensive benchmarks, RAR improves MTTF by on average 2.5× while at the same time improving performance by 11.9% on average. We explore the runahead design space and conclude that RAR is the only design point that improves both reliability and performance by such a significant margin. We find that RAR is more effective for increasingly large processor architectures, making RAR an effective microarchitecture technique for future high-reliability high-performance microprocessors.

I. Introduction

Technology scaling and reduced operating voltages have rendered soft errors or transient faults of critical concern for the reliability of modern-day computer systems [12, 21, 24, 27, 37, 51, 63, 70, 83]. Soft errors due to radiation and energy particle strikes may result in spurious bit flips that corrupt the architectural state leading to reduced system reliability, increased vulnerabilities, unexpected data loss, and catastrophic system crashes [8, 20, 31, 82]. Memory structures such as caches and TLBs are commonly protected using error detection and correction [6, 28, 38, 67]. Core microarchitecture on the other hand is much more difficult to protect and has become increasingly vulnerable to soft errors not just because of technology scaling, but also because of microarchitecture enhancements. Several of the core structures, such as the reorder buffer (ROB) and related back-end structures (issue queues, register file, etc.) have increased dramatically over the past decade, e.g., the ROB has increased from 128 entries for Intel's 2008 Nehalem to 352 entries for the current Ice Lake [4, 23, 29]. Larger structures contain more architectural state and therefore increase the vulnerability to soft errors.

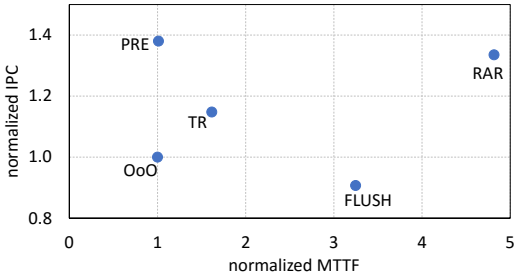

Fig. 1: Performance (IPC) versus reliability (MTTF) for Flushing (FLUSH), Precise Runahead (PRE), traditional runahead (TR), and Reliability-Aware Runahead (RAR), relative to a baseline OoO core for our set of memory-intensive benchmarks. *RAR is the first technique to provide both high reliability and high performance.*

Core microarchitecture vulnerability to soft errors is particularly problematic for memory-intensive workloads. A memory access typically stalls the processor for (at least) a couple hundreds of processor cycles. In particular, a long-latency load miss that waits for memory blocks commit, while the front-end keeps on dispatching instructions into the processor's back-end structures, including the ROB, issue queue (IQ), Load Queue (LQ) and Store Queue (SQ); eventually, these back-end structures fill up with instructions. This exposes a large microarchitecture state for a long period of time to potential soft errors. We find that on average 70.4% (and up to 87.7%) of the vulnerable correct-path state is exposed due to long-latency load misses for a set of memory-intensive SPEC CPU workloads. The current trend towards OoO cores with increasingly large back-end structures exacerbates the problem. Devising techniques that reduce the vulnerability to soft errors for memory-intensive workloads is thus of critical importance.

Prior work to improve core microarchitecture reliability can be categorized in three major groups. The first group of techniques adds extra hardware to detect and possibly correct soft errors. Unfortunately, these techniques incur significant overhead. In particular, ECC protection increases cycle time for latency-sensitive pipeline structures [10, 14, 78]. Parity detection incurs significant chip area, power and energy overheads [14]. The second group engages redundancy (e.g., helper threads) to duplicate the execution and verify correctness by cross-checking results [18, 19, 57, 58, 65, 73, 77]. Unfortunately, redundant execution leads to substantial runtime overheads in terms of performance and/or energy consump-

2378-203X/22/$31.00 ©2022 IEEE
DOI 10.1109/HPCA53966.2022.00062

tion [40, 68, 72]. The third group improves reliability by limiting or completely removing the microarchitecture state in the processor's back-end upon a long-latency memory access through dispatch throttling [68] or pipeline flushing [80], respectively. Although this leads to a substantial improvement in reliability, it comes at the cost of a significant performance degradation because it limits the amount of memory-level parallelism (MLP) that an out-of-order core can exploit.

With increasingly large microarchitectural structures and the ever large processor-memory latency gap, there is an urgent need to revisit microarchitectural vulnerability to soft errors. Nonetheless, high performance is indispensable, and any technique that aims at improving soft-error reliability must also deliver high performance. Therefore, the goal of this paper is to devise a microarchitectural technique that does not incur high hardware cost nor runtime overhead, and yet improves both soft-error reliability and performance. While surveying previously proposed microarchitecture techniques, we find that runahead execution does improve soft-error reliability, as an unintended benefit, next to significantly improving performance. Runahead execution is a microarchitecture technique that was originally conceived to improve performance by speculating into the future instruction stream in search for distant MLP while the processor is stalled on a long-latency memory access. We find that runahead also improves soft-error reliability because the speculative state during runahead execution is, by definition, non-vulnerable. The state-of-the-art Precise Runahead Execution (PRE) technique [48], while it delivers high performance, faces two major shortcomings when it comes to reliability: (i) it initiates runahead execution too late, and (ii) it maintains vulnerable microarchitecture state in the processor back-end during runahead to optimize performance.

In this paper, we propose Reliability-Aware Runahead (RAR) by enhancing PRE with two critical optimizations. First, RAR does not maintain *any* vulnerable microarchitecture state during runahead execution, i.e., it flushes the back-end when exiting runahead execution. This renders the microarchitecture state in the back-end non-vulnerable during runahead execution. Second, RAR initiates runahead as early as a memory access blocks commit at the head of the ROB, i.e., it does not wait until the ROB completely fills up. These optimizations are synergistic and lead to a significant improvement in reliability while achieving high performance.

Our evaluation using a set of memory-intensive SPEC CPU benchmarks demonstrates the effectiveness of Reliability-Aware Runahead, see also Figure 1. RAR improves mean-time-to-failure (MTTF) by on average 4.8× (and up to 35.8×) while improving performance by 33.5% on average (and up to 2.6×) compared to an OoO baseline core. While PRE does not improve reliability, traditional runahead (TR) [43] on the other hand provides only a modest improvement of 1.6× in reliability. Flushing performs better in terms of reliability (3.2×) but significantly degrades performance (by 9.3% on average) relative to the baseline OoO core. Overall, we find that RAR is the only microarchitecture technique that significantly improves both soft-error reliability and performance compared to a baseline out-of-order core. RAR maintains the high performance delivered by PRE while providing an average 4.8× improvement in reliability (MTTF) and up to 35.8×. Across a broader set of compute- and memory-intensive workloads, we find that RAR outperforms the OoO baseline by on average 2.5× for MTTF and by 11.9% for performance.

We make the following contributions in this paper:

- We demonstrate that long-latency memory accesses lead to vulnerable state being exposed in the back-end of an OoO core, even if the memory access does not lead to a full ROB stall. The reliability problem increases with increasing back-end structure sizes as we are witnessing in recent commercial designs.
- We analyze how runahead execution techniques improve soft-error reliability as an unintended side effect, and we identify major shortcomings in the state-of-the-art Precise Runahead Execution (PRE) technique in terms of reliability.
- We propose Reliability-Aware Runahead (RAR) by enhancing PRE with two critical optimizations, namely flush at runahead exit to render microarchitecture state non-vulnerable, and early start to eagerly initiate runahead as soon as a memory access blocks commit.
- We report that RAR significantly improves soft-error reliability and performance compared to a baseline OoO core, making it the most effective design point compared to previously proposed runahead techniques.
- We comprehensively explore the design space of runahead techniques, clearly demonstrating RAR's effectiveness.

II. Motivation

We first analyze under what circumstances out-of-order (OoO) processors expose vulnerable state. We then argue why the problem of soft error vulnerability is worsening as OoO processors evolve over time.

A. Vulnerable Microarchitectural State in OoO Processors

OoO processors expose a significant amount of microarchitectural state that is vulnerable to soft errors, especially in the back-end of the pipeline. Recall that an OoO core fetches, decodes and renames instructions in the front-end of the pipeline, before dispatching them into the back-end. Dispatch allocates an entry in the reorder buffer (ROB) and issue queue (IQ), and, depending on the instruction type, in the load queue (LD) or store queue (SQ). An instruction is issued to a functional unit (FU) when its operands are ready. When the instruction is executed, its result is stored in the physical register file (RF), dependent instructions are woken up, and the reorder buffer is updated to reflect that the instruction is ready to commit. An instruction is committed when all instructions before it in program order have been committed. Upon commit, all back-end resources allocated by an instruction are released.

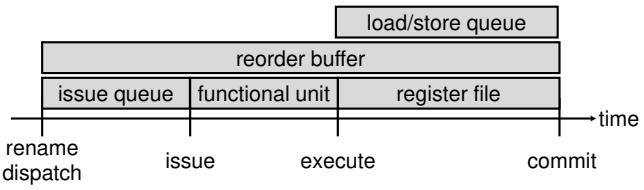

Fig. 2: Timeline representing the allocation and release of back-end resources in a OoO core. *An instruction allocates various back-end resources during its execution.*

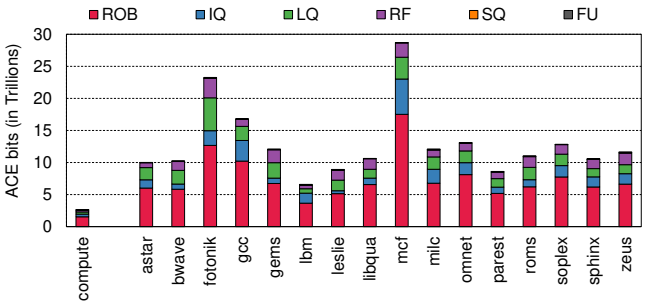

Fig. 3: ABC stacks for our set of memory-intensive benchmarks; the figure also shows the average ABC stack for the compute-intensive benchmarks. *Memory-intensive workloads expose vulnerable state in primarily the reorder buffer, issue queue, load queue and register file.*

Instructions expose microarchitecture state that is potentially vulnerable to soft errors while occupying resources in the back-end of the pipeline. Of course, wrong-path instructions do not expose vulnerable state because those instructions are never committed. In contrast, correct-path instructions do expose vulnerable state, and all bits that they allocate expose vulnerable state (i.e., from resource allocation to release). How much vulnerable state instructions expose depends on their type as well as their execution flow. To understand how much state a correct-path instruction exposes, we have to reason about the different back-end structures, see also Figure 2. Specifically, an ROB entry allocated by a correct-path instruction is vulnerable from dispatch to commit. The address and data values in the load/store queue are vulnerable between execute and commit. An issue queue entry is vulnerable from dispatch to issue. Architecture registers are vulnerable throughout the entire execution; a physical register is vulnerable between execute and commit, assuming a physical register transitions into an architecture register upon commit. The number of vulnerable bits exposed on a functional unit amounts to its bit width times the number of execution cycles per instruction.

We now quantify the amount of vulnerable state exposed by an OoO core when executing a set of memory-intensive SPEC CPU benchmarks, see Section IV for details about our experimental setup. We use ACE Bit Count (ABC) to quantify the amount of vulnerable state exposed by an OoO processor. We provide a formal definition of ABC in Section IV-B, but, intuitively speaking, ABC quantifies the total number of vulnerable bits during the execution of a workload. An ABC stack breaks down the total number of ACE bits exposed by the different microarchitectural structures, namely

TABLE I: Four OoO core configurations.

	Core-1	*Core-2*	*Core-3*	*Core-4*
ROB	128	192	224	352
Issue queue	36	92	97	128
Load queue	48	64	64	128
Store queue	32	64	60	72
Int registers	120	168	180	256
Fp registers	120	168	180	256

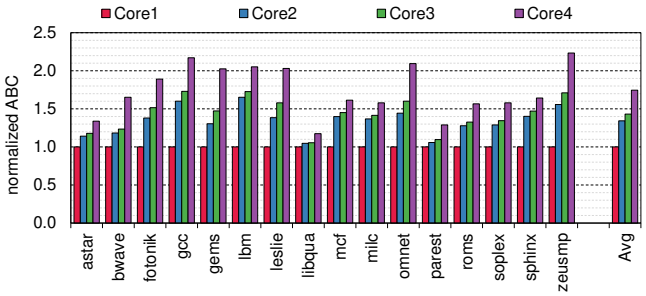

Fig. 4: Normalized ABC for four core configurations with increasingly large back-end structures. *Soft-error vulnerability increases significantly with back-end processor size.*

the ROB, IQ, LQ, SQ, RF and FU, see Figure 3. The memory-intensive benchmarks are sorted alphabetically and are contrasted against a set of compute-intensive benchmarks for which we report the average ABC stack on the far left. The higher the ABC stack, the higher the number of soft errors. There are several important conclusions to be deduced from this result. First, memory-intensive applications tend to expose significantly more vulnerable state than compute-intensive workloads. Second, the reorder buffer is responsible for the bulk of vulnerable state, followed by the issue queue, load queue and physical register file. The reason is that, as we later analyze and quantify more deeply, a long-latency memory access blocks commit at the head of the ROB, which in turn leads to the processor back-end filling up with instructions, thereby exposing vulnerable state.

B. Historic Trend Leads to Increased Vulnerable State

It is interesting to analyze how vulnerability to soft errors has historically evolved as processors have become more powerful. In particular, OoO processors have increased the sizes of their back-end resources by a significant margin over the past decade. For example, the ROB size in recent Intel processors has increased from 128 (Nehalem) in 2008, to 192 (Haswell) in 2013, to 224 (Skylake) in 2015, and 352 (Ice Lake) in 2019. IBM's POWER9 has a 256-entry ROB [59], and Apple's recently released M1 core features a huge 600-entry ROB [5]. To maintain a balanced architecture, the other back-end structures need to be scaled proportionally with the size of the ROB. To analyze how the amount of vulnerable state increases with the size of processor back-end structures, we now quantify total ABC for the four processor configurations from Table I, loosely defined following the four respective Intel processor generations we just discussed.

Figure 4 quantifies vulnerability to soft errors using the ABC metric for these four processor configurations, normalized to the least powerful configuration, for our set of memory-

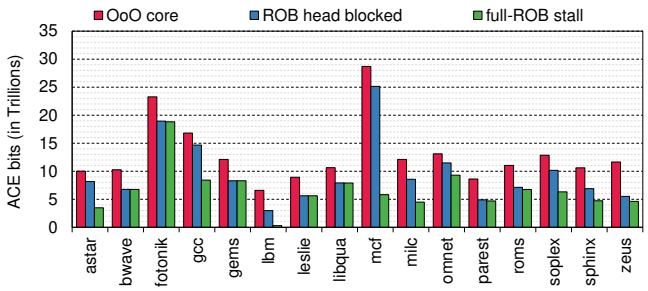

Fig. 5: Impact of memory accesses on ACE bit count (ABC) on an out-of-order core. *A significant fraction of the total ACE bit count results from long-latency memory accesses filling up the ROB and blocking commit at the ROB head.*

intensive benchmarks. It is clear that soft-error vulnerability increases with back-end structure size, i.e., the larger the back-end structures, the higher the vulnerability to soft errors. Compared to Core-1 with a 128-entry ROB, Core-4 with a 352-entry ROB leads to an average 1.83 × higher vulnerability to soft errors. We note an approximate linear relationship between back-end structure size and vulnerability. The reason is that large back-end structures lead to increased accumulated state being allocated in the processor when commit stalls on a long-latency memory access at the head of the ROB. The overall conclusion is that the problem of soft-error vulnerability in OoO cores has become increasingly severe as their back-end structures have increased in size, urging the need for a solution.

C. Understanding Vulnerable State

We now dive deeper to understand how vulnerable state gets exposed in an OoO core. As aforementioned, memory accesses due to LLC load misses frequently lead to a full-ROB stall, i.e., the load miss blocks commit at the head of the ROB, while new instructions are dispatched into the ROB until the ROB fills up, at which point new instructions can no longer be dispatched. To assess the impact of full-ROB stalls on reliability, we perform the following experiment. When a load instruction that misses in the LLC causes a full-ROB stall, we start counting the number of ACE bits exposed in all structures of the pipeline by all in-flight instructions. When the load returns, we add up the ACE bits exposed in all structures, and stop the counters. This is the reliability overhead caused by the LLC miss between the full-ROB stall and the return of the memory access. We repeat the same process for every LLC load miss that results in a full-ROB stall. When we add the number of ACE bits exposed by all such loads, we obtain the reliability overhead caused by all full-ROB stalls. This reliability overhead (expressed in ABC terms) is represented by the 'full-ROB stall' bar in Figure 5. Comparing this bar against the 'OoO core' bar, which is the same as the top height in Figure 3, it is clear that full-ROB stalls are responsible for a significant fraction of the overall soft-error vulnerability. For example, for benchmarks such as fotonik and libquantum, more than 74% of the total ACE bits are exposed during full-ROB stalls due to memory accesses.

It is further worth noting that accounting for full-ROB stalls does not account for the total amount of vulnerable state in an OoO core. For example, for applications such as mcf and lbm, full-ROB stalls lead to only 20.3% and 4.5% of the total ACE bits, respectively. A large percentage of ACE bits in these applications are not exposed during the full-ROB stalls. To understand this remaining gap in soft-error vulnerability, we perform another (but similar) experiment. When an LLC load miss reaches the head of the ROB and blocks commit, we start counting ACE bits exposed in all pipeline structures by all in-flight instructions. Unlike the previous experiment, we start counting ACE bits as soon as the LLC load miss blocks commit — we do not wait for the ROB to fill up as in the previous experiment. When the LLC load miss returns, we sum the ACE bits exposed in all structures, and stop the counters. This is the reliability overhead caused by the LLC load miss between blocking and unblocking commit. We repeat the same process for every LLC load miss that blocks commit at the ROB head. When we add the number of ACE bits exposed by all such loads, we obtain the reliability overhead caused by all memory accesses between ROB blocking and unblocking. This reliability overhead is represented by the 'ROB head blocked' bar in Figure 5. The overall conclusion is that the vast majority of soft-error vulnerability (on average 70.4% and up to 87.7%) is exposed when the ROB head is blocked by a long-latency load. The fine subtlety between the two experiments is important for understanding how to best improve soft-error reliability. A load instruction that blocks the head of the ROB does not necessarily lead to a full-ROB stall. However, a large number of ACE bits are still exposed to soft errors. We encounter such situations when an LLC miss is followed by a branch misprediction, front-end miss or a full issue queue. Benchmarks such as mcf and gcc have a fairly large number of branch mispredictions in the shadow of long-latency load misses [45]; lbm is stalled on a full issue queue for about 20% of the time; soplex and astar also encounter branch mispredictions and some other resource stalls in the pipeline [46]. These experiments suggest that to minimize the vulnerable microarchitecture state under a long-latency memory access, we need a mechanism that withholds instructions beyond a blocking long-latency load from allocating back-end structures.

III. Reliability-Aware Runahead

Before presenting Reliability-Aware Runahead (RAR) in detail, we first revisit prior microarchitecture solutions and their limitations.

A. Revisiting Prior Solutions

A couple microarchitecture techniques have been proposed that improve soft-error reliability: flushing and runahead.

1) Flushing: One solution to limit the accumulation of vulnerable state upon a memory access is to throttle dispatch [68]. A more aggressive and more effective solution is to flush the instructions beyond the blocking memory access in the dynamic instruction stream. Flushing [76, 80] removes the vulnerable state from the processor back-end for the duration

of the memory access. Flushing is an effective technique to reduce the amount of vulnerable state which leads to significant improvements in soft-error reliability, as we will later quantify in Section V.

Flushing to improve soft-error reliability was previously proposed by Weaver et al. [80] in the context of an in-order processor. The performance penalty of flushing the pipeline upon a cache miss is minimal on an in-order processor. In contrast, flushing incurs a severe performance penalty on out-of-order processors as it prevents the core from exploiting memory-level parallelism (MLP) within the ROB by simultaneously servicing multiple independent memory accesses. The performance degradation due to flushing increases with increasing ROB sizes. Our evaluation shows that flushing the pipeline when a long-latency memory access blocks commit at the head of the ROB, degrades performance by 7.6% on average (and up to 15.8%) for the smallest configuration in Table I with a 128-entry ROB, to 12.2% on average (and up to 36.5%) for the largest configuration with a 352-entry ROB. Overall, although flushing is an effective technique to improve soft-error reliability, it is not an appealing design option because of its high performance overhead.

2) Runahead: A well-known microarchitecture technique to speculate beyond the ROB when stalled on long-latency memory accesses is called runahead. Although runahead execution was originally developed to improve performance, we find that runahead also reduces soft-error vulnerability. We believe we are the first to report this unintended side effect.

Runahead execution [15, 42, 43] is a speculative microarchitecture technique that prefetches future memory accesses when the processor is servicing a long-latency load instruction. The blocking load leads to a full ROB stall as the processor keeps dispatching instructions into the processor back-end. Runahead execution mode gets initiated when commit is stalled on a long-latency memory access and the ROB is completely filled up. The processor then takes a checkpoint of the architectural state and starts speculatively executing future instructions beyond the ROB. The speculative execution of load instructions beyond the ROB generates prefetches, which are issued simultaneously with the blocking load, thereby exploiting memory-level parallelism (MLP). When the blocking load returns from memory, the architectural state of the application is restored to the point of entry, and the processor resumes normal execution mode by re-fetching instructions starting from the blocking load. During normal mode, soon after a runahead interval, the load instructions are likely to hit in the cache, thereby improving overall application performance.

Precise Runahead Execution (PRE) [48] is the state-of-the-art runahead technique that significantly improves runahead performance through two key innovations. First, PRE makes the observation that there are typically enough issue queue entries and physical register file entries available when initiating runahead mode. PRE leverages the available issue queue and register file entries to speculate into the future instruction stream while maintaining the instructions in the full ROB. This significantly reduces the overhead of transitioning from runahead mode to normal mode, i.e., the instructions already present in the ROB at the point of entry into runahead mode do not need to be re-fetched and re-filled into the processor back-end. Second, PRE does not execute all instructions beyond the ROB for prefetching future memory accesses. Instead, PRE speculatively executes only those instructions that lead to future memory accesses, i.e., their backward slices or the chains of instructions that a memory access depends upon. In other words, PRE is lean as it only executes those instructions that are needed to generate future memory accesses, in contrast to traditional runahead which executes all instructions in the future instruction stream. These innovations enable PRE to achieve significant performance improvements over prior runahead techniques, which we confirm in our result set.

Although PRE significantly improves performance, it only modestly improves reliability compared to a conventional OoO core, and less so than Flushing. The reason is that PRE maintains vulnerable microarchitecture state in the processor back-end structures during runahead mode.

B. Overcoming Shortcomings in Prior Solutions

The overall conclusion from the previous section is that existing solutions either substantially improve soft-error reliability at the cost of performance degradation (in case of Flushing), or lead to substantial performance improvements with only modest improvements in reliability (in case of PRE). Ideally, we would like to devise a microarchitectural technique that drastically improves soft-error reliability while at the same time substantially improving performance. To achieve this, we need to overcome PRE's two major shortcomings: (i) PRE initiates runahead execution too late — it waits until the ROB completely fills up, and hence misses the important case where a significant amount of vulnerable state is accumulated in the processor back-end even if the ROB does not fill up — and (ii) PRE exposes vulnerable state by maintaining the instructions in the ROB during runahead execution.

C. Reliability-Aware Runahead: Description

Ideally, we want to dramatically reduce the amount of vulnerable state in a way that it does not deteriorate performance (unlike Flushing) and it should initiate speculation as early as possible to minimize the amount of vulnerable state and not miss out on the cases where a lot of vulnerable state is exposed even if the ROB does not fill up (unlike PRE). In other words, we aim at improving soft-error reliability to a level that is comparable to or, if at all possible, surpasses Flushing, while at the same achieving a level of performance that is comparable to PRE. We achieve this goal through Reliability-Aware Runahead (RAR) which significantly improves soft-error reliability over Flushing with the performance of PRE.

RAR takes PRE as a starting point with the goal of maintaining its level of performance. To minimize the vulnerable state during runahead execution, we propose two critical optimizations: (i) flushing and (ii) early-start. We now describe the two optimizations that RAR implements on top of PRE to further improve soft-error reliability.

1) Flush: As aforementioned, PRE maintains the microarchitecture state in the various back-end structures (ROB, issue queues, register file, etc.) during the course of a runahead interval. The reason for doing so is to maximize performance. Indeed, not having to restore the state when exiting runahead mode to return to normal mode reduces the overhead of transitioning between execution modes. On the flip side, the microarchitecture state maintained during runahead is vulnerable to soft errors, and because memory accesses take on the order of hundreds of processor cycles, this leads to a significant reliability degradation.

RAR instead flushes the state in the back-end when exiting runahead mode, i.e., when transitioning from runahead mode to normal mode. By doing so, the accumulated state in the back-end structures during the runahead interval becomes non-vulnerable to soft errors. Any bit flip that may have occurred and could have potentially compromised correctness is hence alleviated, making the looming soft error benign. This leads to a dramatic improvement in reliability. We find that the impact on performance is systematic, albeit small and hence acceptable, as we will quantify in Section V.

Note the subtle but important difference with Flushing as previously proposed by Weaver et al. [80], which flushes the microarchitecture state as soon as a cache miss is detected and the memory access is initiated. This strategy significantly hurts performance as it does not expose any MLP. RAR in contrast flushes the back-end pipeline when the memory access comes back and has been serviced. This enables the core to exploit MLP. In runahead terminology, Flushing flushes the pipeline before the runahead interval while RAR flushes the pipeline after the runahead interval.

2) Early Start: In Section II-C, we made the observation that OoO processors accumulate a lot of vulnerable state even if a long-latency load does not lead to a full-ROB stall. From that perspective, PRE initiates the runahead interval too late as it waits until the ROB is completely filled up. This suggests that it may be beneficial from a reliability perspective to initiate runahead earlier, e.g., as soon as the long-latency load blocks commit at the ROB head. (The next section describes how we determine that the blocking load is a long-latency LLC miss.) Note that the early-start condition does not affect reliability for those cases where the ROB does fill up because RAR flushes the pipeline anyways when exiting runahead mode, thereby making the microarchitecture state non-vulnerable. In contrast, the early-start condition has a major impact on reliability for those cases where the ROB does *not* fill up. By initiating runahead in even those cases — and thus flushing the back-end when exiting runahead mode in even those cases — renders the accumulated state in the back-end non-vulnerable, which leads to a significant improvement in soft-error reliability.

Note that the early-start condition will trigger runahead execution more frequently and, from a performance perspective, too frequently and therefore unnecessarily. Its impact on performance remains to be seen and is subject to evaluation. Performance may improve for two reasons. First, an early start may anticipate a full-ROB stall in the future and therefore initiate runahead execution sooner, which may enable prefetching deeper into the future. Second, an early start will initiate runahead at a point in time at which there are more back-end resources available to use during the runahead interval, which may also contribute to a deeper runahead interval. On the flip side, initiating runahead more frequently may also degrade performance because of increased overhead. This will lead to a performance degradation in those cases where the ROB does not fill up. Overall, we find that the early-start condition is performance-neutral while significantly improving reliability, as we will quantify in Section V.

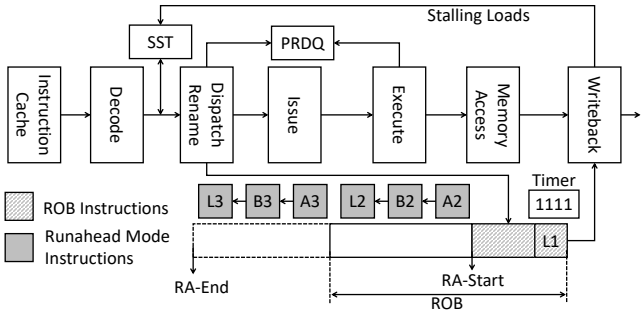

Fig. 6: RAR-enhanced out-of-order pipeline.

D. Reliability-Aware Runahead: Implementation Details

We now dive deeper into RAR's implementation, see also Figure 6. RAR only adds minimal hardware overhead over PRE. More specifically, a 4-bit countdown timer is added at the head of the ROB. This counter is set to 15 (i.e., '1111') whenever a new instruction becomes the oldest instruction in the ROB. The counter is decremented each cycle that the same instruction resides at the ROB head. When the counter reaches zero, runahead mode is triggered. The countdown timer is a low-cost implementation to gauge whether the load at the ROB head is indeed a long-latency load that misses in the LLC. In our setup, the tag lookup times of the L1, L2 and LLC amount to 1, 3 and 10 cycles, respectively. Hence, an instruction that resides at the ROB head for more than 14 cycles is likely to be an LLC miss. An alternative implementation in which the LLC would notify the core upon an LLC miss would involve a dedicated signal from the LLC to the core. The countdown timer is a low-cost core-local implementation.

Figure 6 further illustrates RAR's operation assuming that a long-latency load L1 blocks commit. When the countdown timer expires, we save L1's PC and checkpoint the register allocation table (RAT), and the processor enters into runahead mode (marked as 'RA-Start' in Figure 6). The partly filled ROB is 'freezed', meaning that we do not allocate ROB entries in runahead mode. Instead, we only execute the chains of instructions leading to long-latency loads. For our example, this includes loads L2 and L3 and their producers A2 – B2 and A3 – B3, respectively. The identification of long-latency loads and their backward slices, as well as the allocation of physical registers is done through PRE's stalling slice table (SST) and precise register deallocation queue (PRDQ), respectively [48].

TABLE II: Baseline out-of-order core.

Frequency	2.66 GHz
Type	out-of-order
ROB size	192
Issue queue size	92
Load queue size	64
Store queue size	64
Pipeline width	4
Pipeline depth	8 stages (front-end only)
Branch predictor	8 KB TAGE-SC-L
Functional units	3 int add (1 cyc), 1 int mult (3 cyc), 1 int div (18 cyc), 1 fp add (3 cyc), 1 fp mult (5 cyc), 1 fp div (6 cyc)
Register file	168 int (64 bit) 168 fp (128 bit)
SST size	128 entry, fully assoc, 6r 2w
PRDQ size	192 entry, 4r 4w
L1 I-cache	32 KB, assoc 4, 2 cyc
L1 D-cache	32 KB, assoc 8, 4 cyc, 20 outstanding misses (MSHRs)
Private L2 cache	256 KB, assoc 8, 8 cyc
Shared L3 cache	1 MB, assoc 16, lat 30 cyc
Memory	DDR3-1600, 800 MHz ranks: 4, banks: 32 page size: 4 KB, bus: 64 bits `tRP-tCL-tRCD`: 11-11-11

Runahead mode ends when the blocking load returns (marked as 'RA-End' in Figure 6). RAR flushes the entire back-end, including all the instructions in the ROB. Therefore, soft errors encountered by all the instructions in the back-end during runahead mode are simply discarded. We restore the RAT to the point of entry into runahead mode, and redirect fetch to the restored PC of the blocking load. The processor is now back in normal mode.

IV. Methodology

We now describe the methodology used in this work to evaluate RAR.

A. Experimental Setup

We evaluate RAR using the most accurate, hardware-validated core model in the Sniper 6.0 simulator [13]. We augment Sniper with ACE bit counters to account for bits exposed in the ROB, IQ, LQ, SQ, functional units and register file. NOPs and wrong-path instructions are considered un-ACE. The configuration parameters of the simulated (baseline) out-of-order core are provided in Table II; we model the same baseline core configuration as Precise Runahead Execution (PRE) [48] for fair comparison. We assume a total of 20 miss-status holding registers (MSHRs) at the L1 D-cache level. The branch predictor is an 8 KB TAGE-SC-L from the 2016 Branch Prediction Championship [62]. We do not assume a hardware prefetcher in our baseline setup, however, we evaluate the impact of hardware prefetching in Section V-F.

We consider all the memory-intensive workloads from both the SPEC CPU2006 and CPU2017 suites, by creating their representative 500 M instruction SimPoints [64]. All memory-intensive benchmarks have more than 8 LLC misses per thousand instructions (i.e., MPKI $>$ 8) on the baseline OoO core. All benchmarks with an MPKI of less than 8 from SPEC 2017 suite are considered to be compute-intensive.

To quantify the impact on reliability, we need to know the sizes of the various hardware structures. Since it is impossible to find the size of each entry of different pipeline structures for commercial processor implementations, we model our baseline core microarchitecture following the sizes reported in Table III, which provides a justified balance among the various pipeline structures. We assume that the instruction fetch unit maintains a table that tracks all in-flight instructions between fetch and commit. This table maintains the PCs of all in-flight instructions, which incurs less hardware cost and state than propagating PC information throughout the pipeline. Because PC information is needed to index the branch predictor and to guarantee precise exceptions, each ROB entry holds a 12-bit index to this PC table. Overall, we assume that an entry in the ROB takes 120 bits in total, an issue queue entry takes 80 bits, a load queue entry takes 120 bits and a store queue entry takes 184 bits. Integer registers incur 64 bits while floating-point registers incur 128 bits, see Table II. All integer and floating-point functional units are 64 and 128 bits wide, respectively. We assume that caches, TLBs, the architectural register file, and the register renaming logic and RAT checkpoints are protected using error detection and/or correction codes [6, 17, 28, 38, 67].

B. Soft Error Vulnerability: Metrics

A number of metrics and methodologies have been reported and used in the literature to evaluate a processor's vulnerability to soft errors. We use *Architecturally Correct Execution (ACE) analysis* proposed by Mukherjee et al. [41] to assess soft-error reliability in our simulation infrastructure.[1] An *ACE bit* is a bit that must be correct for the correct execution of a program on a processor. ACE cycles is the total number of cycles a bit must be ACE. For a program running on a processor with N bits, the total *ACE Bit Count (ABC)* is expressed as:

$$ABC = \sum_{i=1}^{N} ACE_i \tag{1}$$

where ACE_i represents the ACE cycles for bit i.

Architectural Vulnerability Factor (AVF) is the fraction of the total number of processor bits exposed by correct-path instructions. AVF can be expressed as:

$$AVF = \frac{ABC}{N \times T} \tag{2}$$

with T the execution time of the workload. Soft Error Rate (SER) is closely related to AVF, and is defined as the total number of errors on ACE bits encountered by a program per unit of time.

Mean Time To Failure (MTTF) and *Failure In Time (FIT) rate* are widely used metrics in the reliability literature. *Mean Time To Failure (MTTF)* is the inverse of FIT rate:

$$MTTF = \frac{1}{FIT}. \tag{3}$$

[1] As an alternative to ACE analysis, an elaborate fault injection campaign might report lower absolute vulnerability numbers [52], but we believe that the overall conclusions and insights would be similar.

TABLE III: Number of bits per entry in the ROB, issue queue, load queue and store queue.

Structure	*Details*	*Bits/entry*
ROB	PC index: 12 bits; mapping: arch(8), phy(8), oldphy(8), 2 src, 1 dest, total = 24 × 3 = 72 bits; LQ and SQ index: 14 bits; ld, st, int, fp completion status, exception bits, marker bits; other control info	120
Issue queue	2 src, 1 dest reg tags: 24 bits; LQ and SQ index for address generation: 14 bits; micro-op: 32 bits; other control info	80
Load queue	VA and PA for memory-ordering violations: 96 bits; ROB ID: 8 bits; SQ index: 7 bits; fault bits; other control info	120
Store queue	Everything in load queue plus 64-bit data	184

FIT rate is defined as the total number of errors experienced in a billion device hours. The relation between FIT rate and AVF is defined as [39, 80]:

$$FIT = AVF \times raw\ error\ rate \quad (4)$$

where *raw error rate* is determined by the circuit technology and environment. In other words, the effective number of errors is the raw error rate derated by AVF.

In this paper, we quantify soft-error vulnerability using MTTF and ABC. The reason for reporting both metrics is because they report reliability from a system and architecture perspective, respectively. MTTF is a system-level metric that quantifies the average time between two failures due to soft errors. MTTF is a higher-is-better metric. ABC is an architecture-level metric that quantifies the total number of bits exposed by the architecture during the execution of a particular workload (fixed unit of work). ABC is a lower-is-better metric. We report normalized MTTF and ABC metrics relative to our baseline OoO core, which makes the reliability analysis independent of a particular technology and environmental setting.

V. Evaluation

We compare the following five configurations:

1) OoO: Our baseline out-of-order core from Table II.
2) FLUSH: The *Flushing* mechanism proposed by Weaver et al. [80] flushes the pipeline when a memory access blocks the head of the reorder buffer. The pipeline is refilled when the blocking memory access returns.
3) PRE: *Precise Runahead Execution* proposed by Naithani et al. [48] initiates runahead execution upon a full-window stall, and executes only useful instructions for generating future memory accesses without flushing the ROB and other back-end structures.
4) RAR-LATE: *Reliability-Aware Runahead with Late Start* initiates runahead execution upon a full-ROB stall, just like PRE. The pipeline is flushed when the memory access that blocks commit returns.
5) RAR: *Reliability-Aware Runahead* is the microarchitecture mechanism proposed in this work. RAR initiates (PRE-style) runahead execution when a memory access blocks the head of the ROB for at least 15 cycles, and the instructions in the ROB and back-end structures during the runahead interval are flushed when the blocking memory access returns.

We use Mean-Time-To-Failure (MTTF) and ACE Bit Count (ABC) to quantify soft-error reliability. We use useful Instructions committed Per Cycle (IPC) to quantify performance. We use arithmetic mean to compute average ABC and MLP numbers; harmonic mean for IPC; and geomean for MTTF, following the methodology by John [30].

A. Overall Reliability and Performance

We first provide a general overview of RAR against our baseline in terms of reliability and performance. Figure 7 quantifies the soft error reliability for all techniques across our set of memory- and compute-intensive benchmarks. Overall, RAR improves MTTF and ABC by 2.5× and 51.5%, respectively, relative to the baseline OoO core. The memory-intensive benchmarks benefit most and witness a substantially higher improvement in reliability: RAR improves MTTF and ABC by 4.8× and 81.4%, respectively. The compute-intensive benchmarks do not frequently access memory, and therefore the gain in reliability for the compute-intensive benchmarks is, while still significant, more modest: RAR improves MTTF and ABC by 1.5× and 28.7% on average, respectively.

Figure 8(a) evaluates performance for all techniques. Across the complete set of benchmarks, RAR improves performance by 11.9%. Performance improves significantly for the memory-intensive benchmarks, by 33.5% on average, while the compute-intensive benchmarks are minimally affected (0.4% performance improvement). The key takeaway message is that RAR significantly improves both reliability and performance for the memory-intensive benchmarks.

We now dive deeper and compare the various runahead techniques in more detail. We do so while focusing on the memory-intensive benchmarks as they are the primary beneficiary of runahead execution.

B. Reliability Analysis

Flushing is an effective technique to improve reliability, see Figure 7: MTTF and ABC improve by 3.2× and 61.4% on average relative to the OoO baseline, respectively. Flushing removes all vulnerable state from the processor back-end as soon as a memory access blocks commit. Unfortunately, flushing substantially degrades performance, as we will quantify later, making it a sub-optimal design point.

PRE also improves ABC compared to an OoO core, however, less so than Flushing, i.e., PRE improves ABC by 28.3% on average. PRE maintains the microarchitecture state in the processor back-end during runahead mode, thereby exposing vulnerable state. However, by speculating into the future instruction stream, PRE also improves performance and instructions speculatively executed during runahead mode do not expose vulnerable state. As a result of improving both ABC and performance, there is no net improvement in MTTF for PRE.

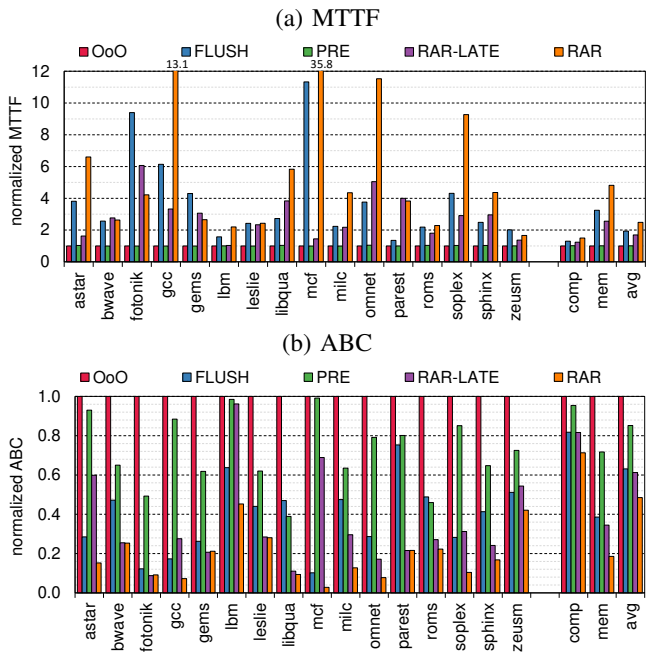

Fig. 7: Evaluating reliability: (a) MTTF and (b) ABC. *Reliability-Aware Runahead significantly improves reliability.*

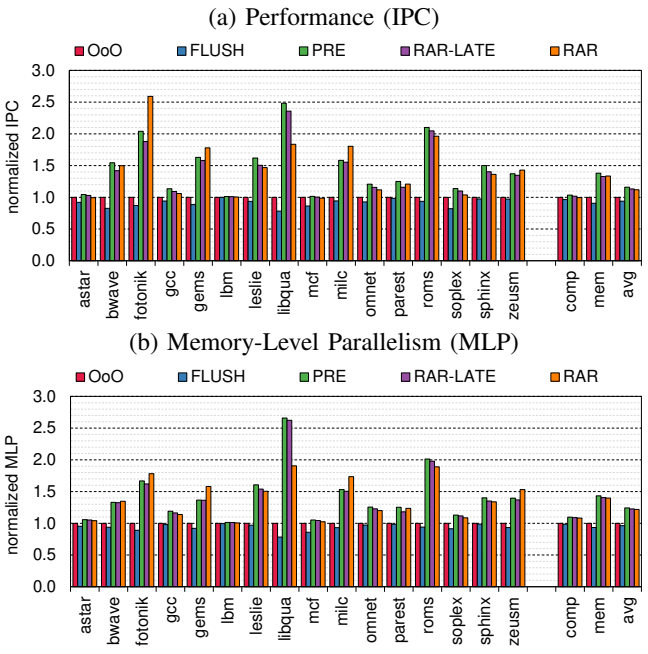

Fig. 8: Evaluating performance: (a) IPC and (b) MLP. *Reliability-Aware Runahead improves performance and MLP to a level that is comparable to the best performing PRE technique.*

Reliability-Aware Runahead with Late Start (RAR-LATE) flushes the back-end pipeline when returning from runahead mode to normal mode, which removes the vulnerable state in the pipeline. This leads to an average 2.5× and 51.9% improvement in MTTF and ABC compared to PRE, respectively. Note that RAR-LATE outperforms Flushing in terms of reliability: RAR-LATE improves MTTF and ABC by 2.5× and 65.5% on average compared to the baseline OoO core, respectively. Flushing the back-end pipeline when exiting the runahead interval makes all the state exposed during the runahead interval non-vulnerable to soft errors.

Reliability-Aware Runahead (RAR) further improves reliability by entering runahead mode earlier. As mentioned before, even when a memory access does not lead to a full ROB, a significant amount of vulnerable state is accumulated and exposed to soft errors. RAR eliminates the exposition of vulnerable state in these cases. Relative to PRE, RAR triggers runahead mode 2.3× more often. Overall, RAR is the best performing mechanism, improving MTTF and ABC by 4.8× (and up to 35.8× for mcf) and 81.4% on average, respectively, compared to the baseline.

C. Performance Analysis

Flushing leads to a significant performance degradation by 9.3% on average and up to 21.9% (libquantum), see Figure 8(a). The reason is that Flushing significantly reduces the amount of MLP exposed compared to an OoO core, see Figure 8(b). An OoO core can find independent memory accesses within the entire ROB. Flushing on the other hand reduces the opportunity to exploit MLP by flushing the pipeline.

PRE is the best performing mechanism as it speculatively searches for distant MLP beyond the ROB upon a full-ROB stall. At the same time, PRE maintains the microarchitecture state in the processor's back-end structures during the runahead interval, which limits the overhead when transitioning from runahead mode to normal mode. We report an average 38% performance improvement over the OoO baseline, and up to 2.5× (libquantum), in line with prior work [48].

RAR-LATE leads to a slight and consistent performance degradation compared to PRE. The reason is the overhead incurred by flushing the microarchitecture state when transitioning from runahead mode to normal mode. This overhead is limited to an average 4.2% performance hit, and at most 8.0% (leslie3d) relative to the PRE. Overall, RAR-LATE provides an average 32.7% performance improvement over the baseline, and up to 2.4× (libquantum).

RAR degrades performance for some benchmarks compared to RAR-LATE, while improving performance for others. RAR achieves higher performance than RAR-LATE for benchmarks for which initiating runahead early is beneficial by exploiting more MLP, see for example fotonik, gems and milc. RAR degrades performance relative to RAR-LATE for benchmarks for which a memory access does not lead to a full-ROB stall, see for example libquantum and roms. On average though, we find that RAR only slightly degrades performance compared to the state-of-the-art PRE. Overall, RAR improves performance by 33.5% on average and up to 2.6× (fotonik) compared to the baseline.

D. Runahead Variants

We now perform a systematic exploration of the runahead design space. To do so, we consider three more configurations

TABLE IV: Runahead variants.

	Early	*Flush*	*Lean*
TR		✓	
TR-EARLY	✓	✓	
PRE			✓
PRE-EARLY	✓		✓
RAR-LATE		✓	✓
RAR	✓	✓	✓

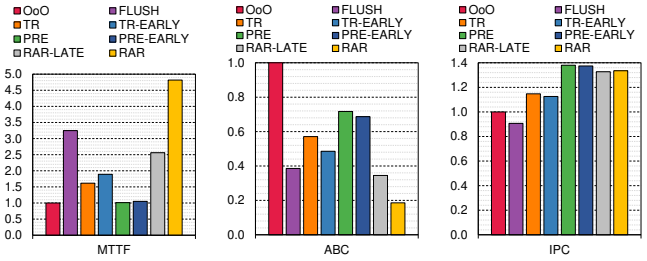

Fig. 9: MTTF, ABC and IPC for the various runahead variants normalized to the OoO core. *RAR achieves the highest improvement in soft-error reliability while achieving performance similar to the state-of-the-art PRE runahead technique.*

beyond the ones already included in the analysis:

- TR: Traditional runahead execution, as proposed by Mutlu et al. [43], triggers runahead execution upon a full-ROB stall. TR flushes the pipeline when returning from runahead mode to normal mode. We include the following two enhancements:
  - There are no overlapping runahead intervals.
  - Runahead is only triggered for load instructions issued to the memory hierarchy less than 250 cycles before a full-window stall; this condition ensures that the runahead is not triggered for short runahead intervals.
- TR-EARLY: This is similar to TR except that runahead mode is triggered when long-latency load blocks the head of the ROB, instead of waiting for a full-ROB stall.
- PRE-EARLY: Similar to PRE but the runahead mode is initiated as soon as a memory access blocks the ROB head. The instructions already present in the ROB are not flushed, as in PRE.

Table IV summarizes the six runahead variants in terms of (i) whether runahead mode is initiated early or not, i.e., early means as soon as a memory access blocks commit versus when the ROB completely fills up; (ii) whether microarchitecture state is flushed or not at the end of a runahead interval; and (iii) whether runahead execution is lean or not, i.e., whether it executes all instructions in the future stream or only those instructions that are useful to generate future memory accesses. PRE executes only useful instructions in runahead mode, in contrast to the traditional runahead techniques. The traditional runahead techniques flush the pipeline in contrast to PRE. RAR initiates runahead early, flushes the pipeline upon its return, while executing only useful instructions.

Figure 9 compares these six runahead variants against our baseline OoO core as well as Flushing, in terms of their average MTTF, ABC and IPC. This result confirms that RAR

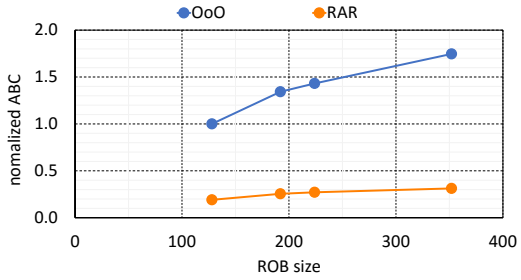

Fig. 10: Impact of back-end resource scaling on ABC. *RAR closes the widening reliability gap with increasing back-end structure sizes.*

is indeed the most compelling design point. The traditional runahead techniques (TR and TR-EARLY) are suboptimal compared to RAR. Performance is inferior because it executes all instructions as opposed to only useful instructions under PRE during runahead mode. Moreover, flushing microarchitecture state at the end of a runahead interval incurs runtime overhead. Reliability is comparable to Flushing as the TR-techniques also flush the pipeline at the end of the runahead interval. Finally, PRE-EARLY does not noticeably improve reliability over PRE because, although it initiates runahead early, it does not flush vulnerable state upon a long-latency memory access. Overall, we conclude that RAR achieves the highest reliability — significantly surpassing Flushing — while achieving a level of performance that is close to the best performing runahead technique which is PRE.

E. Sensitivity Analysis

As mentioned in Section II-B, the amount of vulnerable state exposed in an out-of-order core increases with increasing back-end structure sizes. We now explore how effective RAR is as we increase the ROB, issue queue, load/store queue and physical register file sizes. To this end, we consider the four core configurations from Table I and report ABC as a function of ROB size, see Figure 10. The OoO curve is the same as the average bars in Figure 4; all data points are normalized to the Core-1 OoO baseline. This result clearly shows that RAR closes the widening reliability gap with increasing back-end structure size, making RAR an effective microarchitecture technique for future high-reliability high-performance microprocessors.

F. Hardware Prefetching

We now evaluate RAR on architectures that feature an aggressive stride-based hardware prefetcher with up to 16 streams at the LLC level or across all three cache levels. Such enhanced architectures eliminate some of the LLC misses that RAR speculates upon, thereby possibly reducing the opportunity for RAR to improve reliability and performance. Figure 11 reports MTTF, ABC and IPC for the OoO baseline, PRE and RAR with hardware prefetching at the LLC level (i.e., '+L3') and at all cache levels (i.e., '+ALL'), relative to our baseline. The overall conclusion is that RAR leads to a significant improvement in soft-error reliability and performance even for architectures that feature aggressive hardware prefetching.

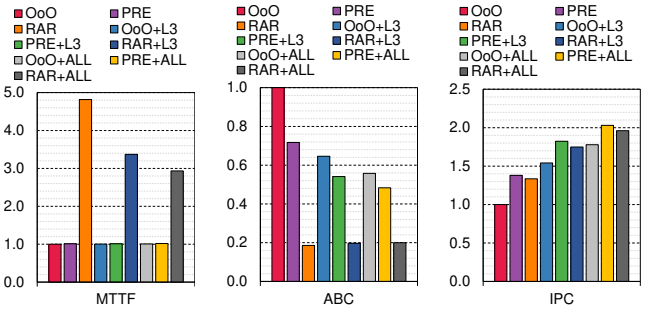

Fig. 11: Evaluating RAR under hardware prefetching. *RAR improves reliability and performance for a baseline architecture with hardware prefetching.*

VI. Related Work

A large body of work over more than two decades has targeted soft error reliability, focusing on soft error estimation [34, 41, 44], modeling [9, 45, 52, 61, 69, 75, 78, 79, 81] and optimization [11, 46, 50, 68, 74, 80]. We now discuss the most closely related work in more detail.

A. Error Detection and Correction

Error detection and correction techniques, e.g., ECC and parity, are widely employed in address-based memory structures. L2/LLC and main memory are commonly protected with ECC, while L1 caches, TLBs and BTBs are protected with EDC (e.g., parity) or ECC [6, 38, 67]. For example, the L1-D, L1-I and TLBs of the Intel Xeon Phi chip are protected with parity; the L2 is protected with ECC [28].

Core structures are not as easily protected against soft errors. In particular, coding techniques such as ECC cannot be applied to latency-sensitive pipeline structures such as the ROB, IQ, etc., as they add additional latency to each cycle [10, 14, 72, 78]. Parity can bring significant reliability improvements, however, the area, power and energy overheads amount to 14% for an OoO core, and are even higher for an in-order core [14].

B. Redundant Execution

Redundant execution techniques exploit multiple hardware contexts supported by a simultaneous multithreading (SMT) processor for improving reliability by generating identical program threads and comparing their outputs. Such techniques either use full redundancy [40, 53, 57, 77] or partial redundancy [19, 54, 56, 73]. Slipstream processors [73] feature a speculative scouting A-thread to run ahead of a redundant R-thread on separate cores in a multi-core processor to improve both performance and reliability. Earlier techniques also used radiation-hardened circuits to recover from soft errors [11]. DIVA [7] checks the correctness of a superscalar OoO core using a more reliable superscalar in-order core. However, all these techniques incur significant performance, area and power overheads [40, 68, 72], in contrast to RAR. More specifically, redundant multithreading can cause up to 32% performance degradation, in addition to consuming one hardware context [40]. Qureshi et al. [55] detect soft errors by performing redundant execution upon a long-latency cache miss; however, their approach leads to a 7.1% IPC degradation for memory-intensive applications. Fu et al. [16] propose ORBIT, a mechanism that exploits operand readiness-based instruction dispatch to mitigate high vulnerability of the issue queue in SMT processors. However, their approach focuses only on issue queue vulnerability with a 3% reduction in performance. SafetyNet [66] reduces the overhead for restoring the correct state when an error is detected. However, it does not fundamentally reduce the likelihood for a soft error to occur, in contrast to RAR which substantially decreases the amount of exposed vulnerable state.

C. Dispatch Throttling

Soundararajan et al. [68] propose dispatch throttling and selective redundancy for vulnerability control by trading off performance and reliability at runtime. Throttling decreases the rate of instruction dispatch when AVF of the ROB exceeds a preset bound. Their results report substantial performance degradation of 9% on average, even for high-AVF bounds. For low-AVF bounds, throttling performs even worse with average performance degradation up to 80%. Selective redundancy builds upon simultaneous redundant threading [57] to maintain hard vulnerability bounds at all times; unfortunately, it still suffers a performance degradation of 7% on average. In contrast to dispatch throttling and selective redundancy, RAR improves performance (while also improving reliability).

D. Latency-Tolerant Execution

Proposals aimed at improving performance or energy-efficiency also indirectly improve reliability. Runahead techniques other than PRE and TR, such as runahead buffer [25], continuous runahead [26], and vector runahead [49], also improve reliability by pre-executing the future instruction stream to target distant MLP. Waiting Instruction Buffer (WIB) [32] and continual flow pipelines [71] release microarchitectural resources occupied by miss-dependent instructions to execute more future instructions. Long-term parking [60] allocates back-end pipeline resources as late as possible for saving power, but still allocates ROB entries for all instructions past a long-latency load miss. Fetch halting [36] improves power by reducing occupancy in the issue queue and reorder buffer, while degrading performance by 6.5%. Overall, these prior proposals (still) expose a major portion of the pipeline to soft errors under an LLC miss. RAR, on the other hand, turns the pipeline into a fully speculative engine upon an LLC miss and significantly improves both reliability and performance.

E. Heterogeneous Systems

Recent work explores how to improve reliability in heterogeneous systems. Naithani et al. [46, 47] improve system reliability in a heterogeneous multicore. Ainsworth et al. [1, 2, 3] propose hardware-only techniques that use parallel heterogeneous cores to detect and correct errors. Gupta et al. [22] propose reliability-aware data placement to improve reliability in heterogeneous memory architectures. Liu et al. [35] propose reliability-aware garbage collection in hybrid memory systems.

Leng et al. [33] introduce 'asymmetric resilience' for efficiently handling transient errors in accelerator-based systems. RAR is orthogonal to these mechanisms, i.e., deploying RAR in the OoO cores will further enhance soft-error reliability of the overall (heterogeneous) system.

VII. Conclusion

Transient faults lead to a major reliability challenge in modern-day computer systems because of technology scaling, reduced operating voltages and increased core microarchitecture structure sizes. This is particularly problematic for memory-intensive workloads as a large amount of vulnerable state is exposed in the processor's back-end while waiting for a long-latency memory access to return. This paper makes the observation that runahead execution, as an unintented side effect, improves soft-error reliability while improving performance. This paper advances the state-of-the-art by proposing Reliability-Aware Runahead which renders the microarchitecture state non-vulnerable during runahead execution and which initiates runahead execution early. Across our set of compute- and memory-intensive benchmarks, we find that RAR improves MTTF by on average 2.5× (and up to 35.8×) compared to an out-of-order baseline while at the same time improving performance by on average 11.2% and up to 2.6×.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. This work is supported by the European Research Council (ERC) under Grant 741097, and the Research Foundation Flanders (FWO) under Grant G.0144.17N.

References

[1] S. Ainsworth and T. M. Jones. Parallel error detection using heterogeneous cores. In *DSN*, 2018.

[2] S. Ainsworth and T. M. Jones. Paramedic: Heterogeneous parallel error correction. In *DSN*, 2019.

[3] S. Ainsworth, L. Zoubritzky, A. Mycroft, and T. M. Jones. Paradox: Eliminating voltage margins via heterogeneous fault tolerance. In *HPCA*, 2021.

[4] AnandTech. Examining intels ice lake processors: Taking a bite of the sunny cove microarchitecture, 2019. URL https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3.

[5] AnandTech. Apple announces the apple silicon M1: ditching x86 – what to expect? based on Al4, 2020. URL https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2.

[6] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3 GHz fifth generation SPARC64 microprocessor. *ISSCC*, 2003.

[7] T. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *MICRO*, 1999.

[8] R. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. *IEEE Reliability Physics Tutorial Notes, Reliability Fundamentals*, 2002.

[9] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *ISCA*, 2005.

[10] S. Borkar, N. P. Jouppi, and P. Stenstrom. Microprocessors in the era of terascale integration. In *DATE*, 2007.

[11] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron CMOS technology. *IEEE TNS*, 1996.

[12] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *IJHPCA*, 2009.

[13] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM TACO*, 2014.

[14] E. Cheng, S. Mirkhani, L. G. Szafaryn, C. Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra. Tolerating soft errors in processor cores using CLEAR (Cross-Layer Exploration for Architecting Resilience). *IEEE TCAD*, 2017.

[15] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS*, 1997.

[16] X. Fu, T. Li, and J. Fortes. ORBIT: Effective issue queue soft-error vulnerability mitigation on simultaneous multithreaded architectures using operand readiness-based instruction dispatch. In *SBAC-PAD*, 2008.

[17] R. Gabor, Y. Sazeides, A. Bramnik, A. Andreou, C. Nicopoulos, K. Patsidis, D. Konstantinou, and G. Dimitrakopoulos. Error-shielded register renaming subsystem for a dynamically scheduled out-of-order core. In *DATE*, 2019.

[18] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *ISCA*, 2003.

[19] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *ISCA*, 2005.

[20] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *S & P*, 2003.

[21] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta. Compiler techniques to reduce the synchronization overhead of GPU redundant multithreading. In *DAC*, 2017.

[22] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta. Reliability-aware data placement for heterogeneous memory architecture. In *HPCA*, 2018.

[23] S. Gupta, N. Soundararajan, R. Natarajan, and S. Subramoney. Opportunistic early pipeline re-steering for data-dependent branches. In *PACT*, 2020.

[24] I. S. Haque and V. S. Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In *CCGRID*, 2010.

[25] M. Hashemi and Y. N. Patt. Filtered runahead execution with a runahead buffer. In *MICRO*, 2015.

[26] M. Hashemi, O. Mutlu, and Y. N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *MICRO*, 2016.
[27] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *DAC*, 2013.
[28] Intel. Intel® Xeon Phi™ coprocessor system software developers guide, 2014. URL https://software.intel.com/sites/default/files/managed/09/07/xeon-phi-coprocessor-system-software-developers-guide.pdf.
[29] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, 2016.
[30] L. K. John. Aggregating performance metrics over a benchmark suite. In *Performance Evaluation and Benchmarking*. 2006.
[31] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.
[32] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *ISCA*, 2002.
[33] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, Q. Chen, M. Guo, and V. Janapa Reddi. Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems. In *HPCA*, 2020.
[34] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Online estimation of architectural vulnerability factor for soft errors. In *ISCA*, 2008.
[35] W. Liu, S. Akram, J. B. Sartor, and L. Eeckhout. Reliability-aware garbage collection for hybrid HBM-DRAM memories. *ACM TACO*, 2021.
[36] N. Mehta, B. Singer, R. I. Bahar, M. Leuchtenburg, and R. Weiss. Fetch halting on critical load misses. In *ICCD*, 2004.
[37] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *IEEE TDMR*, 2005.
[38] S. E. Michalak, A. J. DuBois, C. B. Storlie, H. M. Quinn, W. N. Rust, D. H. DuBois, D. G. Modl, A. Manuzzato, and S. P. Blanchard. Assessment of the impact of cosmic-ray-induced neutrons on hardware in the Roadrunner supercomputer. *IEEE TDMR*, 2012.
[39] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers, 2008.
[40] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA*, 2002.
[41] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO*, 2003.
[42] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA*, 2003.
[43] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA*, 2005.
[44] A. A. Nair, L. K. John, and L. Eeckhout. AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors. In *MICRO*, 2010.
[45] A. A. Nair, S. Eyerman, L. Eeckhout, and L. K. John. A first-order mechanistic model for architectural vulnerability factor. In *ISCA*, 2012.
[46] A. Naithani, S. Eyerman, and L. Eeckhout. Reliability-aware scheduling on heterogeneous multicore processors. In *HPCA*, 2017.
[47] A. Naithani, S. Eyerman, and L. Eeckhout. Optimizing soft error reliability through scheduling on heterogeneous multicore processors. *IEEE TC*, 2018.
[48] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. Precise runahead execution. In *HPCA*, 2020.
[49] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout. Vector runahead. In *ISCA*, 2021.
[50] M. Nicolaidis. Design for soft error mitigation. *IEEE TDMR*, 2005.
[51] E. Normand. Single event upset at ground level. *IEEE TNS*, 1996.
[52] G. Papadimitriou and D. Gizopoulos. Demystifying the system vulnerability stack: Transient fault effects across the layers. In *ISCA*, 2021.
[53] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A complexity-effective approach to alu bandwidth enhancement for instruction-level temporal redundancy. In *ISCA*, 2004.
[54] A. Parashar, A. Sivasubramaniam, and S. Gurumurthi. Slick: Slice-based locality exploitation for efficient redundant multithreading. In *ASPLOS*, 2006.
[55] M. K. Qureshi, O. Mutlu, and Y. N. Patt. Microarchitecture-based introspection: a technique for transient-fault tolerance in microprocessors. In *DSN*, 2005.
[56] V. K. Reddy, E. Rotenberg, and S. Parthasarathy. Understanding prediction-based partial redundant threading for low-overhead, high- coverage fault tolerance. In *ASPLOS*, 2006.
[57] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, 2000.
[58] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *FTCS*, 1999.
[59] S. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke. IBM Power9 processor architecture. *IEEE Micro*, 2017.
[60] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud. Long term parking (LTP): Criticality-aware resource allocation in ooo processors. In *MICRO*, 2015.
[61] Seongwoo Kim and A. K. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *DSN*, 2002.
[62] A. Seznec. TAGE-SC-L branch predictors again. In *CBP*,

2016.

[63] S. Z. Shazli, M. Abdul-Aziz, M. B. Tahoori, and D. R. Kaeli. A field analysis of system-level effects of soft errors occurring in microprocessors used in information systems. In *ITC*, 2008.

[64] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[65] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *ASPLOS*, 2004.

[66] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA*, 2002.

[67] D. J. Sorin. *Fault Tolerant Computer Architecture*. Morgan and Claypool Publishers, 2009.

[68] N. K. Soundararajan, A. Parashar, and A. Sivasubramaniam. Mechanisms for bounding vulnerabilities of processor structures. In *ISCA*, 2007.

[69] V. Sridharan and D. R. Kaeli. Using hardware vulnerability factors to enhance AVF analysis. In *ISCA*, 2010.

[70] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi. Feng shui of supercomputer memory positional effects in dram and sram faults. In *SC*, 2013.

[71] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS*, 2004.

[72] V. Stojanovic, R. I. Bahar, J. Dworak, and R. Weiss. A cost-effective implementation of an ecc-protected instruction queue for out-of-order microprocessors. In *DAC*, 2006.

[73] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: Improving both performance and fault tolerance. In *ASPLOS*, 2000.

[74] K. Swaminathan, N. Chandramoorthy, C. Cher, R. Bertran, A. Buyuktosunoglu, and P. Bose. BRAVO: Balanced reliability-aware voltage optimization. In *HPCA*, 2017.

[75] K. Swaminathan, R. Bertran, H. Jacobson, P. Kudva, and P. Bose. Generation of stressmarks for early stage soft-error modeling. In *DSN*, 2019.

[76] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO*, 2001.

[77] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *ISCA*, 2002.

[78] K. R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *ISCA*, 2007.

[79] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *DSN*, 2004.

[80] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *ISCA*, 2004.

[81] Xin Fu, J. Poe, Tao Li, and J. A. B. Fortes. Characterizing microarchitecture soft error vulnerability phase behavior. In *MASCOTS*, 2006.

[82] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. In *DSN*, 2001.

[83] J. Ziegler, H. Curtis, H. Muhlfeld, C. Montrose, B. Chin, M. Nicewicz, C. Russell, W. Wang, L. Freeman, P. Hosier, L. LaFave, J. Walsh, J. Orro, G. Unger, J. Ross, T. O'Gorman, B. Messina, T. Sullivan, A. Sykes, H. Yourke, T. Enger, V. Tolat, T. Scott, A. Taber, R. Sussman, W. Klein, and C. Wahaus. IBM experiments in soft fails in computer electronics. *IBM Journal of Research and Development*, 1996.