Error-Shielded Register Renaming Sub-system for a Dynamically Scheduled Out-of-Order Core

Ron Gabor*, Yiannakis Sazeides‡, Arkady Bramnik*, Alexandros Andreou‡, Chrysostomos Nicopoulos‡,
Karyofyllis Patsidis†, Dimitris Konstantinou†, Giorgos Dimitrakopoulos†
*Intel Israel Design Center, Haifa, Israel
‡University of Cyprus, Nicosia, Cyprus
†Democritus University of Thrace, Xanthi, Greece

***Abstract*—Emerging mission-critical and functional safety applications require high-performance processors that meet strict reliability requirements against random hardware failures. These requirements touch even sub-systems within the core that, so far, may have been considered as low-significance contributors to the processor failure rate. This paper identifies the register renaming sub-system of an out-of-order core as a prime example of where cost-efficient and non-intrusive protection can enable future processors to meet their reliability goals. We propose two hardware schemes that guard against failures in the register renaming sub-system of a core: a technique for the detection of random hardware errors in the physical register identifiers, and a method to recover from the detected errors.**

***Index Terms*—Register renaming, Micro-architectural dependability, Mission-critical, Functional safety**

I. Introduction

Customers expect correct operation from computing hardware, with some market segments – such as large-scale compute infrastructures and automotive – having very strict functional safety requirements [1][2]. A hardware fault can cause data corruption or execution of wrong operations, which, in turn, can result in an application or system crashing, or, even worse, producing a wrong output without any warning. Random hardware failures are due to unpredictable events that follow a probability distribution [3], like Soft Errors (SE) [4] and Hard Errors (HE) [5]. Hardware vendors limit random hardware failures by employing a variety of techniques at different design levels: manufacturing, circuit, and microarchitecture.

Even though state-of-the-art processors protect against random errors within a very large fraction of the area of modern out-of-order (OOO) cores [6], it is still unknown if and how various core sub-systems (consisting of small arrays, latches, and logic) are protected. This work concentrates on one such structure: the Register Renaming sub-System (RRS) [7] of an OOO core. The RRS serves a vital role: it enforces correct dataflow, while eliminating false dependences and enabling OOO execution. An error in the RRS can devastate the program execution by causing, among other things, an instruction to hang, or execute using incorrect operands.

The RRS is smaller as compared to a Level-1 cache, but its size has been growing in recent high-end cores to increase performance [8]. Fig. 1 underlines this trend by illustrating the growth in the size of the instruction window and the number of integer and floating-point physical registers in recent cores from one hardware vendor (there is about 15% and 6% growth per generation, respectively). A bigger instruction window and physical register file translate to bigger memory arrays inside the RRS. Moreover, fault-injection analysis of the RRS shows it to have substantial sensitivity to faults. In [9], it is shown that the set of arrays and latches related to renaming is one of the largest contributors of SEs. Another study [10] observes that two renaming arrays, the register alias table and the free list, have high vulnerability to errors: 50% and 85%, respectively. This vulnerability is much higher than what is typically reported for other small core structures [11][12]. What is more, also shown in Fig. 1, is that the number of cores found in recent high-end processors is increasing exponentially. Assuming that the failure contribution per bit in the RRS is not scaling down at a faster rate than the product of the growth rates of the RRS size and the number of cores, then the combined effect of the trends in Fig. 1 is an increase in the failure contribution from the arrays in the RRS with each new processor product.

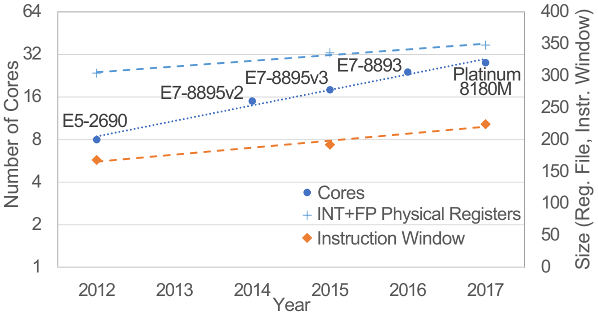

Fig. 1. Prevailing trends with technology scaling for the Intel core microarchitecture [10] and Intel Xeon processors.

Commonly known techniques, such as parity and error detection and correction codes [13] used to protect caches, TLBs, and register files, can also be employed to detect errors in small unprotected arrays, and, in some cases, correct them. However, these techniques are not free; they increase area and power and may increase cycle time or cycle latency and hurt performance. The timing issue is particularly acute for the unprotected arrays at the heart of a modern OOO pipeline, such as those in the RRS. Such small arrays often reside in time-critical paths and a large area growth, when combined with the error detection/correction logic on read/write paths, will very likely have negative effect on the processor cycle time and performance. More holistic approaches [14][15][16] can provide comprehensive coverage against errors, including the RRS, but these entail higher overheads and require per-

978-3-9819263-2-3/DATE19/©2019 EDAA

vasive changes. The choice for an error protection technique is clearly driven by Return-On-Investment (ROI) versus overhead. Consequently, there is a value for cost-effective schemes that can target the protection of a specific core sub-system, as we propose for the RRS in this paper.

Motivated by the error criticality of the RRS and its area/timing criticality, this work introduces (i) a scheme for detecting the corruption of the *physical register specifiers* (**Pdsts**) that register renaming is responsible to manage, and (ii) a method for recovery from Pdst corruption caused by random hardware errors. What is unique and novel with the proposed schemes is that they leverage fundamental RRS and design properties to enable low-cost and effective protection against errors. The paper discusses the error detection strength and recovery coverage of the proposed schemes, as well as their area, energy, and timing overheads, using RTL and micro-architectural simulation-based analysis. The results establish the advantages of the proposed techniques over traditional array protection techniques.

II. Background: Register Renaming, Faults, and Traditional Array Error Protection

A. Register Renaming

Register renaming is a technique that enables OOO execution by eliminating false register dependences between instructions. There exist several alternative implementations of register renaming [17][18]. In this work, we evaluate register renaming with a merged register file. In such implementation, the results of operations are stored in a single physical register file that combines architectural and speculative state [19].

Register renaming with a merged register file uses a large pool of physical registers and translates the logical destination register of each instruction that produces data to a physical register. Typically, each instruction consists of a *logical destination* register (i.e., an architectural register that is part of the Instruction Set Architecture), and two *logical* source (or input) registers. During register renaming, each instruction's logical register specifiers are replaced with corresponding *physical* register specifiers, i.e., **Pdst** (physical destination) and **Psrc** (physical source) specifiers. Register renaming can be performed on either a single instruction at a time (in scalar processors), or on multiple instructions simultaneously (in superscalar processors).

Fig. 2 shows the RRS assumed in this work (for simplicity the RRS of a scalar processor is drawn) that consists of the following hardware arrays:

Free List (FL): is a FIFO where Pdsts are initialized each time a core is powered on. A free Pdst is allocated to rename the logical destination register of an instruction. The Pdst is sent to the Reservation Station (RS) where the renamed instruction waits to execute. When the instruction executes, it updates the physical register pointed by its Pdst.

Register Alias Table (RAT): is a table with the most recent mapping of each logical register specifier to a Pdst. It is used to rename the input (i.e., source) logical registers of an instruction. The renamed Pdsts are forwarded to the RS of the instruction to determine when the instruction can execute.

Re-Order Buffer (ROB): is a FIFO with an entry allocated per instruction. Each ROB entry has a field to hold the Pdst that is evicted from the RAT by the instruction (if the instruction writes to a register). The Pdst is freed when the instruction retires. *Checkpoint Table (CKPT):* is used to regularly take snapshots of the RAT.

Register History Table (RHT): is a FIFO used to log the RAT changes per instruction, i.e., the logical destination register (if any) for an instruction and its allocated Pdst.

During processor operation, the CKPT and RHT buffers are useful for expediting the restoration of the RRS state following *pipeline flushes*. First, the RAT is restored with the closest previous checkpoint to the offending instruction. Then the RHT is used to perform a (positive) walk to update the RAT with information logged between the RHT entry associated with the restored checkpoint and the RHT entry of the offending instruction. The restoration process also performs another (negative) walk of the RHT to return to the FL all the Pdsts allocated after the offending instruction.

In addition to restoring the state of the arrays, the tail pointers of the RHT and ROB must be restored to the position corresponding to the flush-causing instruction. The FL head pointer is not restored, since the wrong-path Pdsts are written back to the FL during the negative walk using the FL tail.

B. Random Hardware Errors

This work considers only faults that corrupt the Pdsts while stored in the arrays of the RRS. The corruption can either be a single bit-flip (SBU), or multiple bit-flips. The multiple flips may occur in the same Pdst (SMBU) and across different Pdsts (DMBU) in different array entries. Such fault behaviors are representative of faults caused by SEs in arrays [20] with a clear trend of MBUs becoming increasingly a larger contributor to SEs with smaller manufacturing technologies. MBUs contribute about 15% of upsets at 22 nm for the circuits examined in [21]. A fault that causes a Pdst corruption can have grave consequences on functional correctness, as it can lead to dataflow violations and even data corruption.

Hard errors can also cause Pdst corruptions, but these faults tend to manifest first as intermittent and eventually as hard faults, as compared to SEs that are transient. Manufacturing faults in arrays are captured during testing and do not usually escape in the field. Corruption due to both soft and hard errors can also occur in RRS latches and logic.

C. Detecting Pdst Corruption: Traditional Approaches

One straightforward way to detect Pdst corruption is to add a parity bit per Pdst in each RRS memory array (a technique henceforth called **Traditional Parity, TP**). Each array is treated as an independently-parity-protected memory macro, without any knowledge of the sub-system in which it operates. Every time an entry is updated with a Pdst, the parity of the Pdst is computed and then written together with the Pdst in the entry. When an entry is read, the parity of its Pdst value is computed and compared against the parity stored in the entry. In the case of a mismatch, an error is detected.

This traditional parity-based approach (TP) is illustrated in Fig. 2, where 'G' and 'C' circles denote the parity generation and parity check logic, respectively, per array. Although this approach can detect a single-bit random error in any Pdst, it has some *key limitations*: (i) the parity generation logic lies

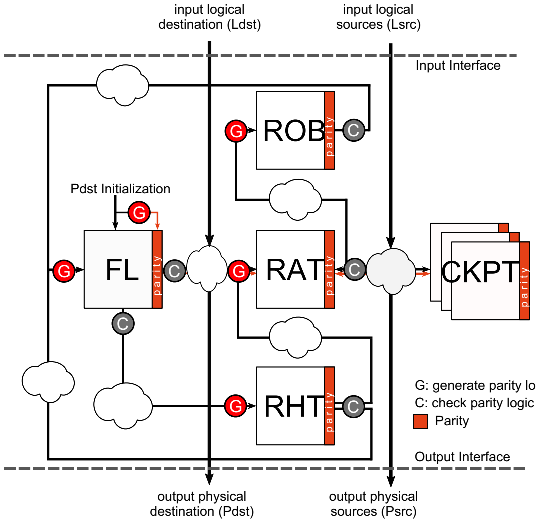

Fig. 2. Register renaming with a merged register file with Traditional Parity (TP) protection. Generate and check logic is required in all major components as indicated by the 'G' and 'C' circles.

on the write critical path, (ii) each array needs distinct parity generation and check logic modules that are used each time a write and read occurs (see the 'G'/'C' circles before/after all main components in Fig. 2), and (iii) it does not offer any detection for the non-array components of the RRS.

For the CKPT table, there is no parity generate or check logic, since a checkpoint contains the entire RAT, including the parity bits. This requires extending the bus between the RAT and CKPT by one bit per Pdst. Errors that occur while a Pdst is stored in the CKPT are detected when the corrupted Pdst is restored in the RAT, and it is subsequently read.

The TP technique can detect an odd number of bit errors, but it is unable to detect even number of errors in a Pdst. One way to overcome this is to use parity interleaving [22] and employ two parity bits per Pdst, one to protect even Pdst bit positions and the other for the odd. This guarantees the detection of all MBU bursts of up to three bits in a Pdst.

Finally, if it is possible to accommodate extra protection state and additional encoding and correction logic in the write/read paths, then an ECC code can be employed instead of parity. This enables correcting the Pdst corruption when the ECC strength is not exceeded. In this vein, an earlier work [9] proposed the use of *Register File Pointer ECC*: each register file pointer, i.e., Pdst, is accompanied with ECC to enable correction of detected Pdst errors. However, Pdsts are typically small and adding ECC incurs overhead and can put pressure on critical paths. To demonstrate this, we have designed and evaluated a variation of the technique. If ECC checking is spread in multiple cycles to avoid timing violations, performance would be certainly penalized.

III. Circular Parity: Detection

The novelty of the protection techniques proposed in this work lies in the underlying exploitation of some fundamental properties and invariant operational characteristics of the RRS. The proposed **Circular Parity (CP)** technique is a small but effective variation of the TP scheme. Parity is generated *only once*, during the initialization of Pdsts, and checked *only at the outputs* of the RRS sub-system, which returns new source and destination identifiers. The proposed CP protection scheme is exemplified with the help of Fig. 3. As compared to Fig. 2 (that illustrates the traditional TP scheme), the proposed CP mechanism removes all write-path encoding logic, 'G', except at initialization (i.e., *parity generation is performed only once*). CP also reduces the check logic, 'C', to only when a Pdst is allocated from the FL to rename a logical destination register, or when a Pdst from the RAT is used to rename a source logical register (i.e., *parity checking is performed at only two locations*). Consequently, the 'G' and 'C' circles in Fig. 3 are significantly reduced, as compared to Fig. 2.

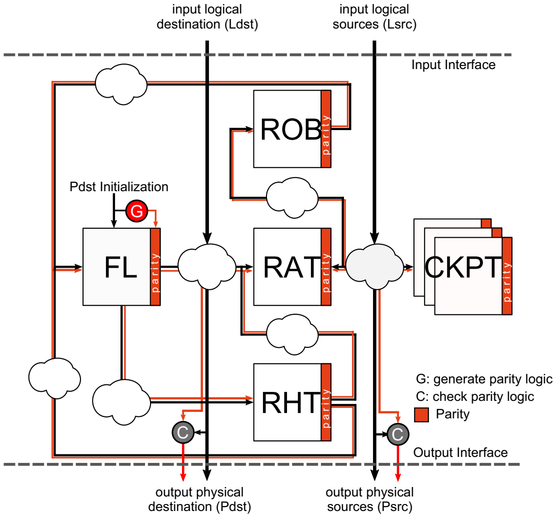

Fig. 3. The proposed Circular Parity (CP) protection scheme. Parity generation ('G') is performed at only one location (during initialization), and parity checking ('C') at two locations (at the output of the RRS).

The operation of CP leverages three basic invariances of the RRS operation to overcome some of the TP limitations:

- Pdsts come from a small, known at design-time, finite set of values.
- Pdsts circulate in the RRS during program execution.
- A Pdst affects functional correctness when it is used to rename the logical register destination of an instruction, and when it is used to rename the logical register sources (in both cases, the renamed info is sent to a RS where instructions wait to execute).

The first invariance enables to compute the parity of each Pdst only once, at the initialization time of the RRS. Thereafter, based on the second invariance, the CP will propagate and store a Pdst with its parity bit without ever generating a parity on the write path to any array. Furthermore, a Pdst gets checked only when it is used to rename a logical destination or a logical source, instead of every time a Pdst is read from an array (third invariance). Therefore, the combination of parity generation only at initialization and check-on-need help overcome limitations (i) and (ii) of TP (Section II-C). Additionally, as a Pdst flows between arrays through the various buses in the RRS, it can get corrupted. This corruption is detected by the proposed scheme when the Pdst is used again for renaming purposes, thereby alleviating limitation (iii) of the TP scheme.

CP with Interleaving (CPI): CP can be extended, like TP, with logical interleaving to enable detection of MBU in a Pdst.

All proposed detection mechanisms have 100% strength, i.e., they always detect a corrupted Pdst when it gets checked, as long as the error does not exceed the code detection strength (e.g., 1-bit error for parity, and up to 3-bit consecutive errors for 2-way logical interleaving).

IV. Recovering from Pdst Corruption (RPC)

This section describes a method that – upon the detection of a corrupted Pdst due to a soft error – can correct it for the vast majority of the time, and resumes correct execution. For the few cases where a detected error cannot be corrected, or it is a hard error, the execution will halt. The following discussion assumes a CP error detection scheme, but the scheme is also applicable to CP with interleaving. The basic idea of the recovery approach is to leverage the following RRS invariances:

- The RHT and CKPT buffers hold redundant copies of Pdsts that can be used to repair corrupted Pdsts in other arrays.
- The processor and system support an event sequence to cause initialization of the RRS with the Pdsts.

For CP, a Pdst corruption is *detected in two locations: when a Pdst is allocated from the FL, and when a Pdst is read from the RAT* to rename an input logical register (see points labeled with 'C' in Fig. 3). The proposed **RPC scheme** reacts differently depending on the location of the error detection.

If the Pdst corruption is detected when renaming a logical source from the RAT, **a pipeline** flush to the oldest instruction in the core is triggered in the hope that the redundant state in the RHT and earliest CKPT are not corrupted. When this is the case, after the pipeline flush the RAT corruption gets repaired and execution resumes normally. It is possible that an error is contained in the earliest available checkpoint, in which case error recovery may be infeasible when the corrupted checkpoint is used. This will *not* lead to silent corruption of the program output, but to many failed attempts after which the recovery procedure is aborted and an uncorrectable error signal is generated [23]. Such a scenario is unlikely to occur, because it requires a corrupted CKPT to become the oldest checkpoint, and, at that time, a recovery to be needed and the corruption not to get masked by the RHT positive walk. One other (also unlikely) scenario for which correction is not possible is when a CKPT becomes corrupted through an error that occurs in the RAT and that gets checkpointed afterwards in the CKPT buffer. Subsequently, that checkpoint becomes the oldest, and then the corrupted RAT entry gets read.

In the case where corruption is detected when a Pdst is allocated from the FL (renaming an output logical register), the offending instruction initiates an RRS initialization event sequence. It is useful to note that a corruption in the FL cannot be repaired through a pipeline flush, since the corrupted Pdst – once it enters the FL – does not have a corrupt-free copy in any other RRS array. Therefore, the only recovery option is through RRS initialization.

A possible sequence for RRS initialization that can be used is the following: (a) when an instruction detects a corruption in its allocated Pdst, it initiates an interrupt to put the core in a sleep state (e.g., C6). The interrupt prevents the corrupted Pdst from getting used to corrupt architectural state. Signals for powering off cores are already available in many processors to help reduce power. (b) The core is interrupted, the system saves the state of the process running on the core, and the core is powered off. (c) The operating system notices a core sleeping while a process is pending and wakes up the core, initializes it, and executes a pending process on it. As part of the waking up of the core, the Pdsts in the FL are initialized and, therefore, any Pdst corruptions are removed.

A subtle point about the corruptions detected when renaming from the RAT is that the initialization option is not applicable. Therefore, RAT corruptions can only be recovered with a flush. This is because the current mappings in the RAT, including the corrupted Pdst, are used to save the architectural register state (see step (b) above of RRS initialization), and, since there is a corruption in a Pdst, the system saves a corrupted program state.

The outlined technique can repair transient errors like SEs. A Pdst with an HE will persist and cause repeated interruptions that can trigger further actions [23].

Recovery from redundant state costs *10s of cycles (similarly to a pipeline flush)*, whereas *initialization* is slower (*milliseconds*). The initialization recovery is slower, but it piggybacks on an available functionality that puts cores to sleep. A slow recovery latency is acceptable, from a performance point of view, as SEs are very rare events.

V. Experimental Results

The experimental results are used to analyze the recovery coverage of RPC and to measure the hardware overhead of the proposed detection mechanisms, as compared to other traditional array error-detection approaches. Without loss of generality, the investigated RRS supports 128 physical registers, which determines the size of the RHT and FL (i.e., 128 entries each), and it includes a 96-entry ROB, a 32-entry RAT, and 4 RAT checkpoints.

A. RPC Coverage Analysis

A micro-architectural simulation-based SE fault injection campaign is performed to determine the fraction of Pdst errors detected by CP that RPC can recover from. A modern four-wide OOO core is evaluated. The simulator is an augmented version of an execution-driven simulator [24] extended to model a merged register file and OOO branch resolution. The analysis is performed in representative regions of the 19 SPEC CPU 2006 benchmarks that exhibit deterministic behavior (repeated fault-free runs produce exactly the same memory and register state at the end of the simulation). The benchmark determinism is a requirement for the analysis, since the architectural state of a fault-free run is used as a golden reference to classify error behavior.

For each injection campaign, one of the RRS arrays is injected with a *single transient bit-flip* in a random entry at a random Pdst bit position. Note that multi-bit upsets are not evaluated in this paper – they are left as future work. The single bit-flip injection is performed in a random cycle in a window of 0.5 million cycles, after 0.5 million cycles of warm-up have elapsed. Each benchmark is simulated for 2 million instructions (unless some anomalous behavior, defined below, is detected that ends the simulation prematurely). Two hundred injections are performed for each of the RRS arrays per benchmark (close to four thousand injections per RRS array).

The simulation analysis tracks whether a system call is performed between the error injection cycle and the time an error is detected. Such event is classified as causing a Silent Data Corruption (SDC), since it is possible for the error to corrupt system state and become user-visible [25].

The simulator monitors for anomalous behavior that leads to early termination [26]. This includes illegal load and store addresses and a watchdog timeout (30,000 cycles elapsing without an instruction getting committed). We classify all such events as Detectable Unrecoverable Errors (DUE) [25]. If, between the error injection cycle and the DUE detection, there is a system call, the error is, instead, classified as SDC.

All runs that do not terminate early, but have a corrupted state, are classified as SDCs, otherwise they are classified as masked (MSK). For the state comparison, the CRCs of the memory, register content, and Pdsts are compared with the corresponding normal-exit CRCs. Our SDC classification is pessimistic since it is possible for an error to get masked after the end of our 2 million instruction simulation window.

Without any protection, we observed that the fault-injection experiments result in SDC, DUE, and MSK outcomes. When CP is used, we never observed SDC or DUE. Recall that CP detects an error in a Pdst when it is read from the FL and RAT, and this is sufficient to prevent corruptions that happen in any RRS array to propagate to the output as SDC or DUE.

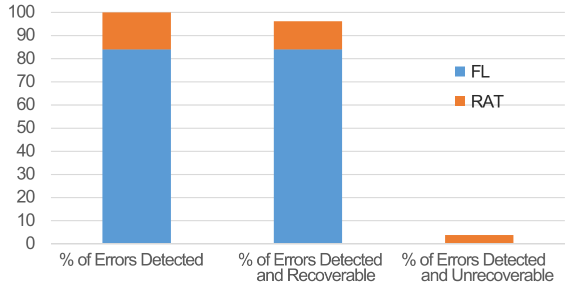

Fig. 4. Detection and recovery coverage of the proposed CP technique.

Analysis of the RPC – shown in Fig. 4 – reveals that the scheme can recover correctly from 96.2% of the CP error detections. The remaining 3.8% detections are uncorrectable and terminate execution abruptly. More specifically, the results show that 16% of the error detections occur when renaming from the RAT, and the remaining 84% when allocating a Pdst from the FL. As explained in Section IV, all FL detections are correctable through initialization. However, not all errors detected from the RAT are correctable. The results show that 76% of the errors detected from RAT renaming are correctable. The remaining 24% are uncorrectable, due to corruption in the earliest checkpoint used to recover the RAT.

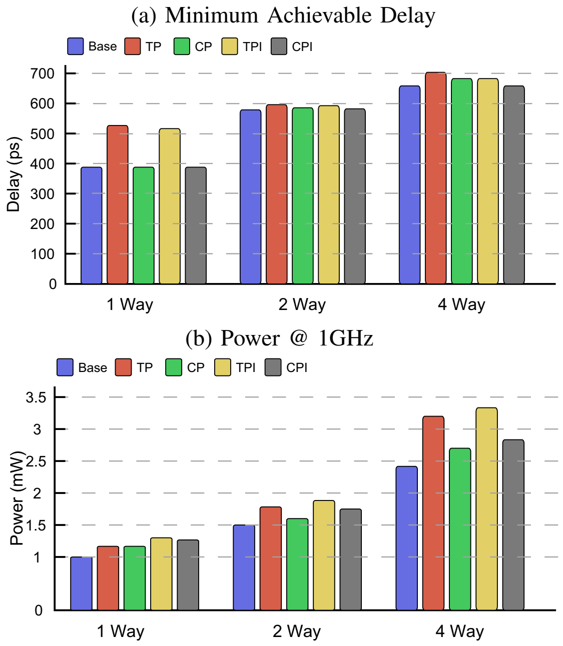

Fig. 5. Delay and power consumption for 1-way, 2-way, and 4-way register renaming, i.e., renaming and retiring 1, 2, and 4 instructions per clock cycle, respectively.

B. Hardware Analysis

All discussed proposed techniques are evaluated in terms of hardware cost, i.e., their impact on the salient metrics of delay (timing), area, and power. We investigate 1-wide, 2-wide, and 4-wide register renaming, i.e., 1, 2, or 4 instructions can be renamed and retired per clock cycle, respectively. In this way, we cover both scalar (single-issue) and superscalar (multiple-issue) microprocessor pipelines. All examined protection techniques are implemented in SystemVerilog and incorporated in a RRS. The RRS designs were synthesized to a commercial low-power 45 nm standard-cell library under worst-case conditions (low voltage 0.8 V, 125 ^{o}C), and placed-and-routed using the Cadence digital implementation flow. The RRS arrays are implemented as standard-cell-based memories, using flip- flops, following an internal clock-gated organization similar to [27]. The post-place-and-route results pertaining to the hardware cost/complexity of all designs under comparison are depicted in Fig. 5.

Fig. 5(a) reports the worst-case delay achieved by each design, including a baseline RRS without protection (Base), traditional parity protection schemes (TP and TP with Interleaving, TPI), as well as the proposed CP and CP with Interleaving (CPI) structures. Evidently, both TP schemes (simple and interleaved) incur the worst delay overhead relative to a baseline (no protection) design. On the contrary, the minimum delay achieved by CP schemes is either the same, or slightly higher, than the baseline, but always better than corresponding traditional schemes. The CP technique hides effectively the delay of parity checking, by performing the checks in parallel to other operations in the RRS.

Parity interleaving helps reduce the delay overhead both for the traditional and proposed schemes, due to the reduced number of inputs in the error-checking logic (each of the two parity tree checks half the bits). However, this delay gain comes at the expense of additional area and power overhead, due to the extra parity bits stored in each memory array. This behavior is highlighted in Fig. 5(b), which reports the average power consumption of each design, including both dynamic and static power consumption. Area follows a similar trend. Power measurements are taken when each design is optimized for a 1 GHz clock frequency, which can be reached by all designs under investigation.

CP and CPI provide better energy efficiency compared to traditional approaches. Even though in the case of 1-way renaming, the difference is marginal, in more realistic architectures the efficiency increases with the increase in complexity. In fact, CP uses 8% less energy than the TP scheme in the 2-way renaming scenario and 18% less energy in the 4-way scenario. The corresponding numbers for the CPI and TPI are 8% and 13%, respectively. All these indicate that the efficiency of the proposed techniques will continue to increase as the dispatch window grows in future architectures.

The hardware results – not shown in a graph – for the *Register File Pointer ECC* described at the end of Section II reveal that the error correction functionality provided by adding 5 extra ECC bits per 7 Pdst bits and associated ECC logic adversely affects all salient metrics (delay, area, and power). For instance, the parity-based designs (TP and the proposed CP technique) can achieve a maximum operating frequency that is as much as 50% higher than that of the ECC-based design, highlighting the impact of ECC on delay.

VI. Conclusions

This work motivates the protection from random hardware errors in the register renaming sub-system, as well as other core sub-systems that are small contributors to failure, by considering the strict reliability requirements from high-core-count processors. The paper presents an RRS protection technique for detecting random corruption in Pdsts, and another to recover from such errors. Micro-architectural simulation-based fault injection analysis shows that the proposed recovery scheme successfully repairs corruptions 96% of the time, and, in the rest of the times, can report an unrecoverable error. Hardware synthesis analysis is used to evaluate and compare different schemes, and reveals the benefits and minimal overhead of the proposed approaches. As part of future work, we will quanitfy the significance of protecting small core subsystems against errors and analyze the sensitivity of the proposed RRS protection technique to more checkers, bigger instruction window and larger physical register files.

Acknowledgments

Part of this work has been performed during a sabbatical of the second author at Intel Israel Design Center in Haifa, Israel. The authors like to acknowledge Alex Gerber for his valuable insights during the early stages of this work. The work is partially supported by the EU Horizon 2020 project Uniserver grant no. 688540 and the University of Cyprus.

References

[1] *Road vehicles - Functional safety (ISO 26262).* International Organization for Standardization, 2011.

[2] S. Siceloff, "Shuttle computers navigate record of reliability," NASA, Tech. Rep., June 2010.

[3] "Random failure vs. systematic failure: Through the looking glass," https://www.kvausa.com/random-failure-vs-systematic-failure, accessed: 1-9-2018.

[4] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sept 2005.

[5] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[6] K. T. Nguyen, "New reliability, availability and serviceability (ras) features in the intel xeon processor family," Tech. Rep., 2017.

[7] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Dev.*, vol. 11, no. 1, pp. 25–33, Jan. 1967.

[8] Intel, "Intel's next generation microarchitecture code name skylake," Tech. Rep., 2015.

[9] 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 *Intern, Conf, on Dependable Systems and Networks, 2004*, June 2004, pp. 61–70.

[10] V. Reddy and E. Rotenberg, "Coverage of a microarchitecture-level fault check regimen in a superscalar processor," in *IEEE Intern, Conf, on Dependable Systems and Networks With FTCS and DCC (DSN)*, June 2008, pp. 1–10.

[11] X. Fu, T. Li, and J. A B Fortes, "Sim-soda: A unified framework for architectural level software reliability analysis," in *Intern. Symp. on Computer Architecture - Benchmarking and Simulation: Workshop on Modeling*, June 2006.

[12] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, and D. Gizopoulos, "Accelerated microarchitectural fault injection-based reliability assessment," in *Intern. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2015, pp. 47–52.

[13] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, April 1950.

[14] T. M. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *Proc, of the ACM/IEEE International Symposium on Microarchitecture*, Nov 1999, pp. 196–207.

[15] J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. González, "End-to-end register data-flow continuous self-test," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 105–115, Jun. 2009.

[16] R. Nathan and D. J. Sorin, "Nostradamus: Low-cost hardware-only error detection for processor cores," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014.

[17] F. L. A. Gonzalez and G. Magklis, *Processor Microarchitecture. An Implementation Approach.* Morgan & Claypool Publishers, 2011.

[18] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Transactions on Computers*, vol. 37, no. 5, pp. 562–573, May 1988.

[19] K. C. Yeager, "The mips r10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–41, April 1996.

[20] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *Intern, Reliability Physics Symposium*, April 2011, pp. 5B.4.1–5B.4.7.

[21] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, and A. Bramnik, "Soft error susceptibilities of 22 nm tri-gate devices," *IEEE Transactions on Nuclear Science*, vol. 59, no. 6, pp. 2666–2673, Dec 2012.

[22] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced srams," in *IEEE International Electron Devices Meeting*, Dec 2003, pp. 21.4.1–21.4.4.

[23] Intel, "Mca enhancements in future intel xeon processors," NASA, Tech. Rep., 2013.

[24] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proc. of the Intern. Symp. on Computer Architecture*, ser. ISCA '01, 2001, pp. 266–277.

[25] 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 *Proc. of IEEE/ACM Intern. Symp. on Microarchitecture*, ser. MICRO 36, 2003, pp. 29–42.

[26] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 265–276, Mar. 2008.

[27] P. Meinerzhagen, S. M. Y. Sherazi, A. Burg, and J. N. Rodrigues, "Benchmarking of standard-cell based memories in the sub-$v_t$domain in 65-nm cmos technology," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 2, pp. 173–182, June 2011.