ECC Memory for Fault Tolerant RISC-V Processors

Alexander Dörflinger[1(✉)], Yejun Guan[1], Sören Michalik[2], Sönke Michalik[2], Jamin Naghmouchi[2], and Harald Michalik[1]

[1] Institute of Computer and Network Engineering (IDA), Technische Universität Braunschweig, Braunschweig, Germany
{doerflinger,guan,michalik}@ida.ing-tu-bs.de
[2] Institute for Robotics and Process Control (IRP), Technische Universität Braunschweig, Braunschweig, Germany
{soeren.michalik,so.michalik,naghmouchi}@tu-braunschweig.de

Abstract. Numerous processor cores based on the popular RISC-V Instruction Set Architecture have been developed in the past few years and are freely available. The same applies for RISC-V ecosystems that allow to implement System-on-Chips with RISC-V processors on ASICs or FPGAs. However, so far only very little concepts and implementations for fault tolerant RISC-V processors are existing. This inhibits the use of RISC-V for safety-critical applications (as in the automotive domain) or within radiation environments (as in the aerospace domain). This work enhances the existing implementations Rocket and BOOM with a generic Error Correction Code (ECC) protected memory as a first step towards fault tolerance. The impact of the ECC additions on performance and resource utilization are discussed.

Keywords: BOOM · Cache · Error correction code · Fault injection · RISC-V · Rocket · Scrubbing · Single Event Effects

1 Introduction

The free and open RISC-V Instruction Set Architecture (ISA) has attracted an active community building processor cores and ecosystems, which makes it competitive to established processor designs. There is a strong growth forecast for the number of RISC-V cores in industrial-, consumer-, and other areas [13]. However, there are only a few approaches of fault-tolerant RISC-V designs for safety-critical and radiation-tolerant applications, which would open its use for the automotive and areospace domain. An exploitation of this market potential requires compliance with corresponding safety standards.

Mitigation of transient faults is one important mechanism for fault-tolerant electronics. ISO26262 [10] names error detection to increase the diagnostic coverage, which is required for electronics of higher safety levels. Furthermore, aerospace systems operating in environments with increased radiation levels are

© Springer Nature Switzerland AG 2020
A. Brinkmann et al. (Eds.): ARCS 2020, LNCS 12155, pp. 44–55, 2020.
https://doi.org/10.1007/978-3-030-52794-5_4

subject to non-destructive Single Event Effects (SEEs). An effective mitigation technique for hereby caused soft errors in memories are again error correction (and detection) codes [8]. Therefore, this paper will present how existing RISC-V implementations can be enhanced with Error Correction Codes (ECCs).

Contribution: This work devises and implements a highly configurable ECC protection for arbitrary memory structures and applies it to two different RISC-V processor systems. Some ECC implementations are already existing for RISC-V designs. However, they cover only parts of the memory structures of a processor core and/or are limited to small low-power solutions with processing power restrictions. The generic and configurable ECC approach of this work targets also large RISC-V cores for high performance computing and fully covers all memory structures. This prepares RISC-V for its use in safety-critical applications and radiation-intense environments. Together with further fault tolerance mechanisms (e.g., lockstep operation or other redundancy schemes), high performance RISC-V systems could be made available for the automotive and aerospace domain.

The rest of this paper is organized as follows: Sect. 2 presents existing fault tolerance concepts for RISC-V processors and Sect. 3 gives an introduction to the Chipyard[1] framework used within this work. A detailed description of the new ECC concept follows in Sect. 4. Results of its implementation are presented in Sect. 5.

2 Related Work

The SHAKTI-F design [9] mitigates SEEs by combining ECC with recomputation techniques. It features a relatively small 5-stage in-order microprocessor. However, its development has been discontinued and it is not maintained within the current SHAKTI-C class core anymore. Fault tolerance of caches, typically representing the largest and hence most susceptible memory structures within a processor system, is not addressed. The Klessydra microprocessor [4] based on PULPino[2] is a configurable 2 to 4-stage RISC-V implementation. Several time- and space redundancy techniques have been applied for fault tolerance. Again, error protection for larger memory structures has not been addressed yet.

Apart from SHAKTI-F and Klessydra (being free and open), some proprietary implementations are available targeting space applications. Microsemi (Microchip Technology Inc.) offers the Mi-V[3] ecosystem, which allows to instantiate RISC-V cores on their radiation tolerant FPGAs. Cobham Gaisler released the 64 bit NOEL-V[4] soft-core recently. However, just as the LEON3/4 processor, it is not fault tolerant by design; fault tolerant versions are built from radiation hardened standard cell libraries and are hence bound to specific technologies.

[1] https://chipyard.readthedocs.io/en/latest, UC Berkeley.
[2] https://pulp-platform.org.
[3] https://www.microsemi.com/product-directory/fpga-soc/5210-mi-v-embedded-ecosystem.
[4] https://www.gaisler.com/index.php/products/processors/noel-v.

Just as the LEON3/4 based System-on-Chips (SoCs) GR712 [5] and GR740 [6], NOEL-V uses *write-through* and *no-write allocate* cache policies. This guarantees that an erroneous cache line can be corrected by fetching its copy from a higher memory hierarchy level at any time. It makes an error correction code dispensable, because an error detection (e.g., parity bit) suffices. However, the hereby utilized write policies typically yield lower performance than *write-back* and *write allocate*.

The Rocket and BOOM RISC-V cores by UC Berkeley implement *write-back* and *write allocate* cache policies and are partly equipped with optional ECC. A Single Error Correction Double Error Detection (SEC-DED) code protects the caches of the SiFive U-series IPs (U54, U74) and SoC (FU540), which utilize the 5-stage in-order Rocket processor core. The BROOM tapeout [2] adds resilience methods to the 7-stage out-of-order BOOMv2 processor. Several techniques tolerate hard bit errors in L1 and L2 caches, which allows an aggressive reduction of the core voltage. However, the approach requires to know the position of erroneous bits beforehand (e.g. by running a built-in self-test). Hence, it cannot correct soft errors at arbitrary bit positions and does not increase fault tolerance.

3 Rocket and BOOM Processor Cores Within Chipyard

The Chipyard framework developed by UC Berkeley bundles RISC-V cores, peripherals, software compilers, simulators, and further tools for SoC development. It targets both FPGA implementations and ASIC design. Hardware components are programmed in the Chisel hardware description language (HDL). Chisel is based on object-oriented Scala and adds hardware construction primitives. Frequently utilized hardware elements are collected in a Chisel standard library (e.g., multiplexers, arbiters, counters, FIFO queues, etc.). As a modern programming language, it offers high abstraction, re-usability, and parameterization options. Compared to well-established HDLs such as Verilog and VHDL, the increased abstraction level results in a higher line of code efficiency and speeds up development times. However, it also adds complexity to simulation and netlist generation: Chisel code has to be compiled into an intermediate circuit representation (FIRRTL) before it is transformed into synthesizable Verilog.

Chipyard integrates two RISC-V implementations, which are both highly configurable. The 5-stage in-order Rocket core [1] offers both 32 and 64 bit register file widths, several branch prediction options, arbitrary cache sizes, and optional ISA extensions (MAFD). The core provides three privilege levels, addresses virtual memory, and is capable to boot Linux. Rocket is already equipped with the ECC options *Parity*, *SEC*, and *SEC-DED* for both L1I$ and L1D$ (tag and data each) which can be activated with limitations. Rocket provides blocking and non-blocking versions for the L1D$. The non-blocking version allows hit-under-miss requests, which enables the in-order processor to execute further instructions until the load data is used. However, this powerful non-blocking L1D$ variant does not support ECC in its tag field at all, and its implementation in the data

field results in compile errors (several versions up to the current v1.3 have been tested without success).

Chipyard allows an easy replacement of the Rocket core with the 7-stage superscalar out-of-order BOOM core [3]. The instruction fetch unit is equipped with complex predictors (e.g., GShare and TAGE). A tapeout in TSMC 28 nm achieved 1.0 GHz and a Coremark of 3.77 per MHz [2], which makes BOOM one of the best performing RISC-V implementations. The BOOM utilizes the non-blocking L1D$ version of the Rocket, hence it is afflicted with the same ECC problems as described above. The L1I$ does not support any ECC implementation. Further resilience methods have been applied to the BOOM implementation [2]. However, they only target static hard errors and cannot mitigate arbitrary soft errors.

The cache resilience works on Rocket and BOOM are promising but not complete. ECC has been successfully applied only to the Rocket core, with restrictions. Memory structures apart from caches such as Branch Prediction Unit (BPU) tables and the Page Table Walker (PTW) are not protected. So far, the BOOM core lacks ECC protection for memories completely. Those gaps are closed in this work using a generic ECC memory described in the following section. The generic design makes it easy to apply it to all memory structures and is not limited to L1 caches. This work concentrates on the ECC integration in Rocket and BOOM; however, the parameterizable ECC memory interface allows to migrate the approach to other processor implementations as well.

4 Generic Error Correcting Memory Component

4.1 ECC Memory Requirements

Within the Rocket core only caches and one BPU table are implemented as memory arrays; all other buffers are mapped to registers due to their small size. This results in a small number of memory arrays ranging from 4 to 11, depending on the Rocket core configuration (Table 1). The more complex BOOM core additionally implements several buffers of the BPU and the PTW as memory arrays, due to their increased size. This results in 18 to 38 memory arrays within the BOOM core, depending on its configuration. The BOOM *Small* and *Medium* configurations differ mainly by their issue width; however, memory sizes and organization are very similar. The same applies for differences between *Large* and *Mega* configurations.

Enhancing all those memory arrays with ECC protection separately requires multiple and far-reaching code changes. For the Rocket core with FIRRTL transformation and simulation times ranging from 10 to 21 min[5], this would be still feasible. However, the BOOM core generation and simulation takes up to 185 min, making a custom ECC adaption of each memory array very laborious.

[5] Depending on its configuration; measured for `run-bmark-tests` on Intel i5-6500 3.20 GHz, 48 GB RAM.

Table 1. Number of memory arrays for selected Rocket and BOOM configurations

Core configuration	L1I$	L1D$	BPU	PTW	Sum
Rocket (Tiny)	2	2	0	0	4
Rocket (Big)	5	5	1	0	11
BOOM (Small)	5	5	7	1	18
BOOM (Medium)	5	5	7	1	18
BOOM (Large)	17	9	11	1	38
BOOM (Mega)	17	9	11	1	38

Hence, a generic ECC memory component has been developed separately, which can simply replace existing arrays and keeps the integration effort minimal.

The access scheme (e.g., single/dual-ported) and array organization (e.g., row of words) differs for each array, which has been considered during the development of the generic ECC protected memory called ECCmem. Hereby the newly created ECCmem component goes beyond existing IP such as Synopsis DesignWare STAR ECC IP, the ARM Artisan embedded memory IP, and Xilinx ECC IP [14]. It is technology independent, i.e. not bound to any FPGA family or ASIC process, and additionally mitigates error accumulation, which is not addressed in any of the existing solutions.

4.2 ECCmem Component

Figure 1 depicts the overall ECCmem architecture. Dashed blocks are instantiated depending on configuration settings. The IOs `read/write request` and `response` make use of Chisel's Decoupled interface, wrapping the data vectors with a ready-valid pair. This interface abstraction allows a simple replacement of existing memories with the ECCmem module.

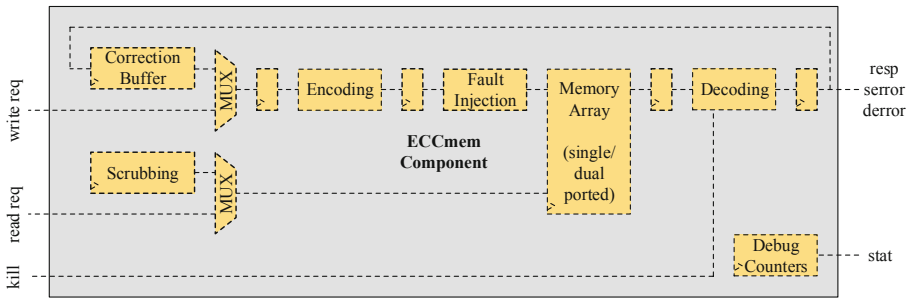

Fig. 1. Configurable ECCmem component

Depending on the capabilities of the selected ECC option, single and double errors are signaled through dedicated outputs (`serror`, `derror`) and tracked in

error counter registers. Statistics on soft errors can be retrieved from further debug counters containing the number of read/write accesses, fault injection-, and error correction events. When relying on this error statistic information, reads from uninitialized data have to be precluded as they may result in inadvertent error events. One solution is to initialize the complete memory at boot time (applied e.g. to the BOOM data cache tag array). Another option is to set the `kill` signal, canceling read accesses to uninitialized data in subsequent clock cycles. This is a feasible solution for e.g. cache data arrays, because the initialization information can be retrieved from the coherency flags one clock cycle after issuing the read access.

Listing 1.1 gives an overview of the parameterization options of the ECCmem module, which satisfy the diverse requirements of memory arrays within the Rocket and BOOM implementations. The object oriented Chisel programming language makes it easy to handle the parameterization. Some ECCmem ports are conditional (depending on the configuration), which is not supported by other HDLs.

Listing 1.1. ECCmem Parameterization

```
class ECCmemParams(
  ecc_code: Code = SECDEDCode,
  reg_enc_input: Boolean = false,
  reg_enc_output: Boolen = false,
  reg_dec_input: Boolean = false,
  reg_dec_output: Boolean = false,
  depth: UInt = 1024,
  row_format: Vec[UInt],
  block_size: Int = 8,
  interleaving: Boolean = true,
  single_ported: Boolean = true,
  correction_buffer: UInt = 1,
  scrubbing: Boolean = true,
  scrubbing_interval: UInt = 4,
  fault_injection: Boolean = true,
  name: String
)
```

Encoding and Decoding. The `ecc_code` parameter allows to select different detection/correction codes. The current implementation supports the algorithms none, parity, and hamming codes (SEC, SEC-DED). Hsiao codes could be added in future for reduced area and delay overheads. Several `reg_*` options allow to insert registers at encoder/decoder inputs and outputs, which can be used to relax timing. In particular, the decoding path can result in long signal latencies, which may require corresponding register insertions.

Array Organization. The parameters `depth`, `row_format`, and `block_size` define the array organization (Fig. 2). The read/write data may be partitioned into several words within a row. Individual words may be accessed using a read-/write mask. A word can be further divided into blocks, which allows arbitrary ECC widths. This facilitates a fine-grained balancing of area overhead and encoding/decoding latency: the smaller the block size, the smaller its encoding/decoding latencies, but the higher its area overhead. Each block contains the original data and ECC bits being grouped together. The `interleaving` option shuffles bits of different blocks. It mitigates SEEs causing multi-bit errors in neighboring cells, because the erroneous bits will be spread across different blocks. With the `single_ported` option, the memory type can be selected. By default, a dual ported memory will be generated (e.g. required for BOOM data cache). Arrays with exclusive read/write access (e.g. BOOM instruction cache) benefit from the optimized resource utilization of single ported memory.

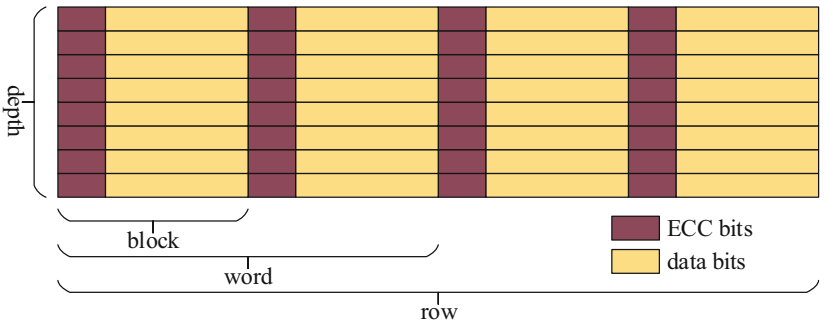

Fig. 2. ECCmem array organization

Correction Buffer. With the current ECC implementations (parity, SEC, SEC-DED), two or more accumulated errors cannot be corrected, and depending on the selected ECC algorithm, not even detected. When using codes with single error correction capabilities, the corrected data can be written back to memory. Any `correction_buffer` size greater 0 enables this error correction option. It mitigates error accumulation, because a single error typically gets corrected before a second SEE strikes the same block. In order to minimize the impact on the overall system, write back accesses are assigned with a lower priority than read and write requests. The corrected word is stored within a correction buffer until there is no concurrent write access (and in case of single-ported memory no concurrent read access). Corrected data must not overwrite updated data. Therefore, an entry within the correction buffer gets cleared once it senses a regular write access to the same memory address as the destination of corrected data. Once the correction buffer writes its content back to memory,

the soft error has been removed. This error correction process typically completes before detection of a second error in another arbitrary word being read. However, systems with high read/write loads (i.e. long retention times in the correction buffer) and high expected error rates, may use a `correction_buffer` depth of >1.

Scrubbing. In the past, only very small numbers of SEE-caused soft errors were expected in on-chip memories such as caches of earth-bound applications, hence error accumulation has not been an issue [12]. However, the soft error rate increases exponentially with voltage decrease, and error accumulation has to be considered when relying on new technologies [7]. Furthermore, space applications can be exposed to multiple SEEs within minutes [11], depending on the FPGA or ASIC technology and the mission region. When operating under such conditions, the interval of system read accesses to memory arrays is not sufficient for preventing error accumulation. This applies for caches in particular: Cache line access patterns are hardly predictable, which increases the probability of error accumulation for less frequently accessed data regions.

To overcome this problem, the optional `scrubbing` option regularly reads the complete memory array, and guarantees a minimum interval of single bit error corrections. Again, the scrubbing mechanism is assigned with a lower priority than read requests (and write requests in case of single-ported memory) to eliminate any negative performance impact on the overall system. As depicted in Fig. 1, the scrubbing block generates continuous read accesses to memory. Once the decoding block detects a correctable error in one of the reads triggered by the scrubbing block, the corrected data will be passed to the correction buffer which handles the write back to memory. In order to prevent an overflow, a full correction buffer forces the scrubbing process to pause. Scrubbing adds high load on the read port of the memory, which can increase the power consumption. This effect can be limited by setting the `scrubbing_interval`, which defines together with the operating frequency and memory `depth` the scrubbing period (Eq. 1). When setting the `scrubbing_interval` to 0, the ECCmem component attempts to scrub the memory as fast as possible. In this case Eq. 1 gives only a lower bound of the scrubbing period, because any other regular read (and write) request stalls a scrubbing access.

$$\text{scrubbing period} = \frac{1}{f} \cdot \texttt{depth} \cdot (\texttt{scrubbing\_interval} + 1) \tag{1}$$

Scrubbing accesses are distributed evenly in time for a balanced load distribution. To achieve this, a scrubbing counter decrements by 1 each clock cycle and triggers a scrubbing access to the next memory row once it hits 0 (① in Fig. 3). A scrubbing access increments the counter again by the defined `scrubbing_interval`. Higher priority accesses (read/write requests) delay the scrubbing access ②. Multiple high priority accesses could cause the scrubbing counter to underflow, which is prevented by stalling the scrubbing access ③.

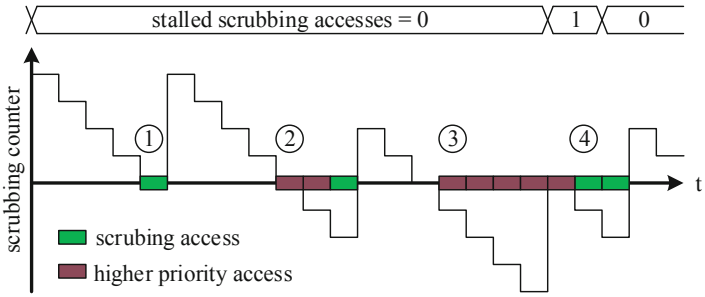

Fig. 3. Distributing scrubbing accesses in time

Stalled accesses are executed as soon as no other high priority access blocks the memory port ④.

Fault Injection. The `fault_injection` option allows sporadic injection of 1-bit and 2-bit errors with a user-defined probability into already encoded data (containing both data- and ECC bits) when writing to memory. This feature is used to test the functionality of the ECC, correction buffer, scrubbing, and debug counters. It further allows to simulate the processor behavior under SEEs, which can replace expensive radiation tests to some extent.

5 Evaluation

All memory arrays of Rocket and BOOM have been replaced with the ECCmem component described in Sect. 4. The integration did not require any far reaching changes, because the `read/write request` and `response` interfaces allowed a simple mapping to existing memories. The `kill` signal (compare Sect. 4.2) is generated correctly for all memories by determining the status of the memory content (initialized/uninitialized). The ECCmem is designed to have no effect on system performance (except when inserting additional register stages with a `reg_*` option). Both the write back of corrected data and the scrubbing mechanism are low prioritized, preventing to thwart read/write accesses. This has been verified running benchmark tests in a Verilator simulation for various Rocket and BOOM configurations (Table 2). Results for the Dhrystone benchmark are identical before and after integration of the ECCmem component.

Due to the similarity of the *Small*/*Medium* and *Large*/*Mega* variants regarding memory size and organization (Table 1), resource utilization results will be discussed for the *Small* and *Mega* configurations only, but apply for the *Medium* and *Large* variants respectively. Figure 4 (left) summarizes the resource utilization of BOOM implementations on the Xilinx Virtex UltraScale+ VCU118 evaluation board and the respective overhead for ECC protection. Figure 4 (right) plots the results for an ASIC synthesis in the GlobalFoundries 22 nm FDX technology (12 T), whereas area is reported for combinatorial cells, flip-flops, and

Table 2. Dhrystone results for different Rocket and BOOM configurations

Core configuration	Dhrystones/s
Rocket (Big)	1912
BOOM (Small)	1920
BOOM (Medium)	2526
BOOM (Large)	3521
BOOM (Mega)	3700

memory macros separately. The ECCmem components have been configured with default parameters, except for array specific attributes such as width, depth, and single/dual ported variants. Hence, a SEC-DED code is applied; scrubbing, error correction, and fault injection are activated.

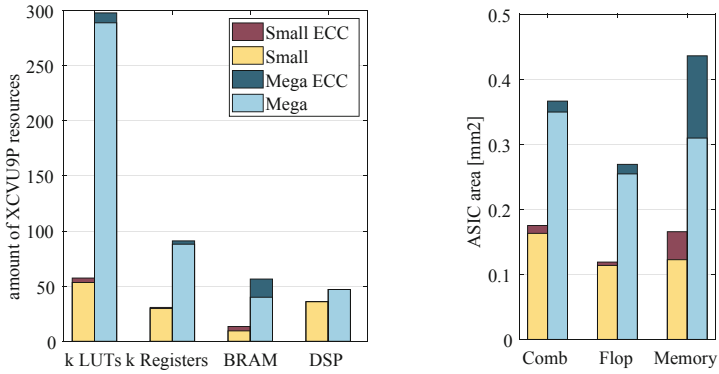

Fig. 4. Resource utilization of Small- and Mega BOOM configurations with and without ECC protection. Left: Xilinx Virtex UltraScale+ XCVU9P FPGA resources. Right: Area for GF 22 nm FDX technology after synthesis.

The FPGA resource overhead for ECC protected BOOM variants compared to original BOOM implementations is calculated in Table 3. It shows moderate overhead for logic (5.31%) and registers (3.44%) on average, but a large increase of RAM resources (41.68%). The ECC memory protection has no effect on DSP utilization. The area increase of the ASIC synthesis yields similar results.

For further evaluation of the increased RAM utilization, Fig. 5 depicts the RAM size for caches and other memory arrays within BOOM implementations with and without ECC protection. The overhead of RAM resources depends on the selected ECC `block_size`. Here the block size has been limited to 26 bits for all memories, which adds a maximum of 6 parity bits to each block. The block size can be only as large as the memory word size. Hence, very small word sizes result in high area overheads, as it is the case for e.g. the Branch Target Buffer

Table 3. Resource/area overhead of ECC protection for BOOM cores

Core config	FPGA resource overhead				ASIC area overhead		
	LUT	Regs	BRAM	DSP	Comb	Flop	Mem
Small BOOM	7.50%	3.47%	42.11%	0%	7.40%	4.52%	38.89%
Mega BOOM	3.11%	3.41%	41.25%	0%	4.82%	5.77%	40.85%
Average	5.31%	3.44%	41.68%	0%	6.11%	5.15%	39.87%

Bimodal Predictor table (BTB bim) with 1 bit words. In this case, applying TMR to this array is a more area efficient protection against SEEs.

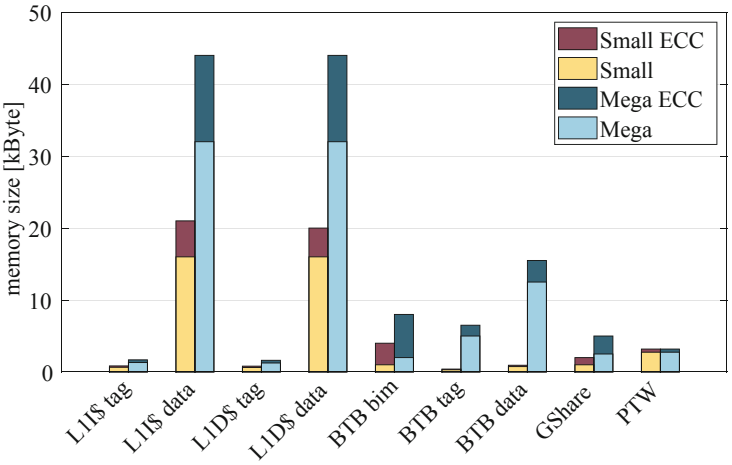

Fig. 5. ECC BRAM overhead for individual memories

6 Conclusion

In this paper we presented a generic solution to enhance existing RISC-V processor core implementations with ECC protected memory. When selecting codes with error correction capabilities, error accumulation can be mitigated by writing corrected data back to memory. Applying a scrubbing mechanism further reduces probabilities of error accumulation. As a reference implementation, all memory structures within the Rocket and BOOM cores have been replaced by the newly developed ECC protected memory. Logic and register overheads for the ECC protection are small, while RAM resource usage increases as expected for the applied hamming codes. Future work will complete the fault tolerance mechanisms for RISC-V processors by applying further redundancy techniques,

which enables the use of RISC-V for safety-critical applications and the aerospace domain. Hereby, the remaining processor logic could be protected using TMR or lockstep techniques.

Acknowledgment. This work has been funded by BMWI under grant number 50 RK 1820 and is part of the DLR Raumfahrtmanagement Komponenteninitiative.

References

1. Asanović, K., et al.: The rocket chip generator. Technical report. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, April 2016
2. Celio, C., Chiu, P., Asanović, K., Nikolić, B., Patterson, D.: Broom: an open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos. IEEE Micro **39**(2), 52–60 (2019). https://doi.org/10.1109/MM.2019.2897782
3. Celio, C., Chiu, P.F., Nikolic, B., Patterson, D.A., Asanović, K.: Boom v2: an open-source out-of-order RISC-V core. Technical report. UCB/EECS-2017-157, EECS Department, University of California, Berkeley, September 2017
4. Cheikh, A., Cerutti, G., Mastrandrea, A., Menichelli, F., Olivieri, M.: The microarchitecture of a multi-threaded RISC-V compliant processing core family for IoT end-nodes. In: De Gloria, A. (ed.) ApplePies 2017. LNEE, vol. 512, pp. 89–97. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-93082-4_12
5. Cobham Gaisler AB: GR712-UM, 2.12 edn. (2018)
6. Cobham Gaisler AB: GR740-UM-DA, 2.3 edn. (2019)
7. Dixit, A., Wood, A.: The impact of new technology on soft error rates. In: International Reliability Physics Symposium. pp. 5B.4.1–5B.4.7, April 2011. https://doi.org/10.1109/IRPS.2011.5784522
8. European Cooperation for Space Standardization - ECSS: ECSS-Q-HB-60-02A Space Product Assurance - Techniques for Radiation Effects Mitigation in ASICs and FPGAs Handbook, 1 edn., September 2016
9. Gupta, S., Gala, N., Madhusudan, G.S., Kamakoti, V.: SHAKTI-F: a fault tolerant microprocessor architecture. In: IEEE 24th Asian Test Symposium (ATS), pp. 163–168, November 2015. https://doi.org/10.1109/ATS.2015.35
10. International Organization for Standardization - ISO: ISO 26262 - Road Vehicles - Functional Safety, 2016 edn., April 2016
11. Michel, H., Guzmán-Miranda, H., Dörflinger, A., Michalik, H., Echanove, M.A.: SEU fault classification by fault injection for an FPGA in the space instrument SOPHI. In: NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 9–15, July 2017. https://doi.org/10.1109/AHS.2017.8046353
12. Mukherjee, S.S., Emer, J., Fossum, T., Reinhardt, S.K.: Cache scrubbing in microprocessors: myth or necessity? In: Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing, pp. 37–42, March 2004. https://doi.org/10.1109/PRDC.2004.1276550
13. SEMICO Research Corporation: RISC-V Market Analysis The New Kid on the Block, cc315-19 edn., November 2019
14. Xilinx Inc.: ECC LogiCORE IP Product Guide, PG092, v2.0 edn. (2017)