Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 962062, 15 pages
doi:10.1155/2011/962062

Research Article

A Self-Checking Hardware Journal for a Fault-Tolerant Processor Architecture

Mohsin Amin,[1] Abbas Ramazani,[2] Fabrice Monteiro,[1] Camille Diou,[1] and Abbas Dandache[1]

[1] *LICM Laboratory, University Paul Verlaine, Metz, 7 rue Marconi, 57070 Metz, France*
[2] *Electrical Engineering Department, Engineering Faculty Lorestan, University Khorramabad, Iran*

Correspondence should be addressed to Fabrice Monteiro, fabrice.monteiro@ieee.org

Received 1 August 2010; Revised 10 February 2011; Accepted 24 March 2011

Academic Editor: Gilles Sassatelli

We introduce a specialized self-checking hardware journal being used as a centerpiece in our design strategy to build a processor tolerant to transient faults. Fault tolerance here relies on the use of error detection techniques in the processor core together with journalization and rollback execution to recover from erroneous situations. Effective rollback recovery is possible thanks to using a hardware journal and chosing a stack computing architecture for the processor core instead of the usual RISC or CISC. The main objective of the journalization and the hardware self-checking journal is to prevent data not yet validated to be sent to the main memory, and allow to fast rollback execution on faulty situations. The main memory, supposed to be fault secure in our model, only contains valid (uncorrupted) data obtained from fault-free computations. Error control coding techniques are used both in the processor core to detect errors and in the HW journal to protect the temporarily stored data from possible changes induced by transient faults. Implementation results on an FPGA of the Altera Stratix-II family show clearly the relevance of the approach, both in terms of performance/area tradeoff and fault tolerance effectiveness, even for high error rates.

1. Introduction

For years, a substantial research effort has been successfully devoted to increase the performance of processor architectures, while making the best profit of the technological improvements predicted by Moore's law. However, the long-followed approach is reaching its limits. Indeed, the current technological boundaries are raising major constraints on future architectures, particularly in terms of reliability and fault tolerance, given the enlarging rates of physical defects and the increased sensitivity to external disturbances.

Fault tolerance aims at building systems that behave satisfactorily even in the presence of faults. The tolerance of a system is devised to some predefined set of fault types that may include transient, intermittent, or permanent faults, depending on the fault causes being addressed. What can be considered as a satisfactory behavior can vary according to the application domains: a simple error detection with an error alarm indication may be acceptable in some cases while in other cases, the system must ensure operation continuity with no visible impact on the service being delivered.

Transient faults, long while considered as a problem only in space applications and some other critical domains such as nuclear plants, are now becoming a significant threat at sea level. They are now a common source of errors in digital circuits, and their occurrence rate can be rather high compared to permanent faults. Actually, in the search of higher performance, transistors dimensions in each new technological generation have been continuously reduced, allowing higher integration and higher clock rates. However, smaller dimensions lead to lower charges being dealt with, particularly to hold logical states in registers, making new devices much more sensitive to external disturbances. This trend is leading towards an increasing number of soft errors in logic circuits [1, 2], affecting the reliability of those systems. The transient error rates will be far more in future [3].

Sources of disturbance include high energy particles coming from deep space (e.g., cosmic rays and solar wind), natural radioactivity (alpha particles produced by disintegration of radioactive isotopes), electromagnetic fields (e.g., electrical engines, radio frequency sources), and many others. Ratios of transient-to-permanent faults can vary from 2 : 1

to 100 : 1 or even higher [4]. This ratio has been continuously increasing due to higher complexity, smaller transistor sizes, higher operational frequency, and lower voltage levels [5]. This trend may possibly change in the future due to the rapidly increasing rates of physical defects in circuit foundry processes. Still, the transient fault rates are not likely to diminish in the future. Single Event Upsets (SEUs), consisting in a bit state change caused by ions or electromagnetic radiation striking a memorization node, have been frequently addressed [2], using in particular techniques based on the use of error detecting and correcting codes (EDC).

Related work has been dedicated to improve the dependability of digital systems and, in particular, to design digital circuits displaying fault-tolerant characteristics. Hereafter, we reference some prior research that addresses tolerance to transient faults and may be used in processor architectures [6]. The approaches being employed usually rely on the capacity to detect and possibly correct the errors induced by the faults. Actually, one of the main problems that must be faced is to achieve the required levels of tolerance while keeping under acceptable limits the penalty in terms of area, cost, performance, and also power consumption, particularly in embedded systems. On real-time systems, this may be a rather difficult challenge to address.

Processor replication has been used since long as a fault tolerance technique against transient faults [7]. However, this is a costly solution requiring more than 100% of area overhead (and also power overhead), since duplication at least is required for error detection (triplication at least for error correction/masking) and additional voting circuitry. Practically, it is often a more effective compromise to detect errors at register level, specially when SEU are being considered. Triple Modular Redundancy (TMR) is such a classical hardware approach, in which all registers (or the critical ones) are triplicated in order to mask the effect of transient errors affecting one register among the three. Well-known fault-tolerant machines like the LEON FT [8] and the IBM S/390 mainframe processor [9] are TMR-based machines. Double Modular Redundancy (DMR), offering only error detection capability as been used also, for example in the DMR-based server developed by Tandem computers and presented in [10]. Razor [11] is another circuit-level correction mechanism addressing timing errors using supplementary shadow latches to detect delay errors. The DIVA [12] approach is tailored at the micro architecture level.

Software replication recovery [13, 14] with checkpointing [15–17] and re-execution [18] offers much cheaper solutions. However, these techniques tend to induce significant time overheads making severe time constraints hard to match in real-time designs. In redundant multithreaded architectures [19, 20], all instructions are executed twice to detect transient errors. The performance overhead of dual redundancy threads on the AR-SMT (Activestream/Redundant-stream Simultaneous Multi-Threading) pipeline can be significant due to resource contention and checking bandwidth. Dependence-based checking Elision [21, 22] allows to reduce the number of checks and, hence, to improve the overall performance.

A mixed hardware/software N-modular redundant approach is to run virtual machines on separate processors and compare the outputs [23, 24]. Another hardware/software codesign technique is addressed in [25] where little supplementary hardware is used to achieve fault tolerance.

Information redundancy is another classical approach to implement fault tolerance. It relies on the use of error detecting and correcting codes and requires extra circuitry to handle the later (encoders, decoders, and additional storage room). It is largely employed to protect memory devices [26, 27].

In this paper, we are proposing an alternative technique based on the design of a self-checking hardware journal (SCHJ) being used as a centerpiece in our strategy to devise a fault-tolerant processor against transient faults. This strategy relies on two main choices: having built-in hardware error detection capacity in the processor core and using software rollback execution for error recovery. Hardware implementation is the best choice for error detection as it needs to be done permanently and concurrently. Conversely, rollback error recovery is only required on error occurrence. It can be realized through software means if the time penalty remains acceptable, that is, the case when either one or both of the following are true: error rate is low; rollback recovery mechanism is fast. Error rate is an external parameter not under control of the designer that can possibly be high in harsh environments. Consequently, the rollback mechanism should be as fast as possible. Architectural choices for the processor core, and overall, the journalization mechanism and related SCHJ focused in this paper are all devised with this objective in mind.

Section 2 introduces the background motivation and the main architectural choices for the overall processor design. Section 3 presents the principles behind the journalization and fully discusses the architecture and operation modes of the SCHJ. Section 4 presents the main implementation results.

2. Overall Architectural Approach

In this section, we briefly present the basic principles on which the architectural design and the operation of our fault-tolerant processor are built. Actually, the approach we have chosen is largely dictated by our long-term objective: devising a new fault-tolerant massively parallel MPSoC (Multi-Processor System-on-Chip) architecture in which the current fault-tolerant processor design will be used as a building block (as a MPSoC processing node). It was clear from the beginning that severe design constraints concerning the processing node area should apply in order to match the massively parallel objective, yet preserving as much as possible the individual node performance.

Our basic choice for the core processor was to select an architecture in the MISC family (Minimal Instruction Set Computer) instead of the more classic RISC (Reduced Instruction Set Computer) or CISC (Complex Instruction Set Computer) ones. Our architecture [28, 29], inspired from that of the canonical stack processor [30], is able to offer a rather good level of performance with only a limited

amount of hardware being required. Another interesting characteristic is the great compactness of the code running on this kind of processor. On control-oriented applications, in which the quantity of data being manipulated is low and only a small amount of memory is required, it is possible to implement all the required storage in fast memory devices and avoid complex memory cache structures.

The other major reason for choosing an MISC stack computer architecture is related to our approach on how to provide transient fault tolerance to the system. Actually, the very little amount of logical resources and internal storage keeping the state of the processor core allows:

(i) little hardware area overhead to be used to concurrently check its correct operation and, hence, to make the processor core self-checking,

(ii) little time overhead to be used to periodically backup the internal state and to restore it when required on error recovery.

Among other underlying hypotheses, we are supposing that the processor core is to be connected to a dependable memory in which data is supposed to be kept safe fully without any risk of corruption. Actually, a lot of work has been dedicated in the past to the protection of memory devices [26, 27] making this hypothesis pertinent.

Having a self-checking processor core (SCPC) and a dependable memory (DM) device able to safe-fully store data is not enough to build an effective fault-tolerant system. Indeed, the processor may generate undue or corrupted data in sequence to a transient fault. The built-in hardware means will detect the error but will not correct it. Hence, erroneous data may flow into the DM (see Figure 1), and this later can no more be considered as a place where data is to be trusted. In this case, implementing a software recovery mechanism can be a rather painful task with a lot of data redundancy being necessary in the memory device.

The underlying idea behind the journalization mechanism presented in this paper is to prevent untrustable data to flow into the DM and to allow an easy recovery from faulty situations. The basic idea is to implement some hardware device on the path between the SCPC and the DM controlling the way data flows from one side to the other and preventing untrustable data to end up in the DM, as suggested in Figure 2.

Our strategy to implement fault recovery is based on rollback execution, a classic software technique employed in real-time embedded systems [31–33], and relies on the following usual behavior:

(i) program (or thread) execution is split in sequences of fixed maximal length;

(ii) each sequence must reach its end without any error being detected to be validated;

(iii) data generated during a faulty sequence must be discarded and execution restarted from the beginning of the faulty sequence.

For an effective implementation of the above scenario, several points must be taken into consideration. Discarding

Figure 1: Untrusted data flowing into DM.

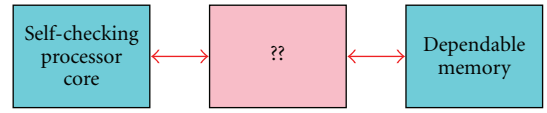


Figure 2: Principle of the overall organization scheme.

data from a faulty sequence, requires the state (data) prevailing at the end of the previous sequence being available somewhere. The DM, as a trustable location, seems to be the ideal location. But there is the need to store the data not yet validated from the current sequence under execution. DM is not the better location as in that case, it would contain untrustable data. Thus, a temporary location for not-yet-validated data is required.

Data stored inside this temporary location can also be corrupted in the consequence of transient faults affecting it, such as SEUs (see Figure 3). Hence, an error detecting and correcting mechanism is required to ensure the reliable operation of this temporary data storage.

In this regard, we are proposing a Self-Checking Hardware Journal (SCHJ) as the dependable temporary storage (DTS) location (see Figure 4). This SCHJ has built-in mechanisms to detect and correct transients errors affecting its content, allowing an effective protection of data during its temporary stay. The basic role of this SCHJ is to hold the new data being generated during the sequence being currently executed until it can be validated (at the end of the current sequence). If sequence is validated, this data can be transfered to the DM. Otherwise, in case of error detection during the current sequence, this data is simply dismissed and the current sequence can restart from the beginning using the trustable data hold in the DM and corresponding to the state prevailing at the end of the previous sequence.

Apart from the data generated by the program/thread code being executed during the current sequence, additional data is to be saved at the end of the sequence and corresponding to the internal state of the processor core, that is, to its internal registers or State determining Elements (SE). The interesting point in choosing an MISC stack computer architecture is the very few bytes of data (corresponding to the SE) that need to be saved. This makes our strategy very effective on control flow applications compared to other fault-tolerant techniques [34]. Indeed, the incurred time penalty being very low, most of the sequence duration (SD) is used for active program/thread instruction execution. Consequently, SD can be reduced and hence the journal size be smaller, leading to a simplified hardware and a shorter average time for not-yet-validated data to remain in the SCHJ. This limits the risk that errors cumulate and become undetectable in the SCHJ. To conclude this analysis, rollback

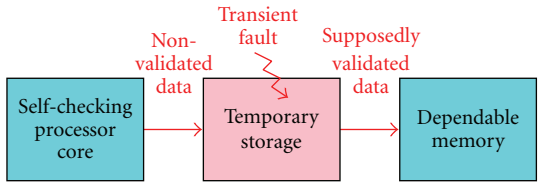


Figure 3: Data corruption in temporary storage.

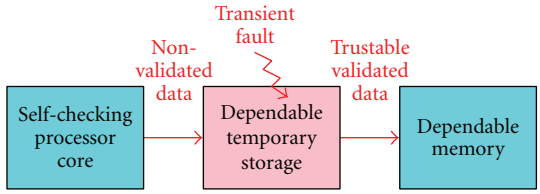


Figure 4: DTS protecting DM from contamination.

error recovery is a favorable choice to be made with our combination of a self-checking MISC stack processor core and a self-checking hardware journal.

Section 3 describes in detail the architecture and operation of the SCHJ in its environment (between the SCPC and the DM).

3. Journal Architecture and Operation

The journal storage space is internally split into two parts, as shown in Figure 6. The upper part contains the not-yet-validated data being generated in the current sequence. This data will be called unvalidated data (UVD) in the remaining of this paper. The lower part holds the data validated at the end of the previous sequence. This validated data (VD) is being transfered to the DM during the execution of the current sequence. At the end of the current sequence, if the sequence is validated UVD turns into VD and, thus, the virtual line separating the upper part from the lower part shifts up to denote the new situation.

Each row in the SCHJ is 41 bits long as shown in Figure 5. The v and w bits will be discussed later. Together with the 16 address bits and the 16 data bits, they represent the information corresponding to a single element being stored in the SCHJ. In order to trust the data temporarily stored in SCHJ, we need a built-in mechanism to detect and correct errors that may occur due to transient faults. Here, we have chosen to rely on error control coding, a classic and effective approach to protect storage devices [27]. The remaining bits in a row, that is, the parity bits, represent the information redundancy related to the error correcting code and protecting the other bits. In the present case, we selected a Hsiao (41, 34) code, a systematic single-error-correction and double-error-detection (SEC-DED) code, Hsiao codes being more effective than Hamming codes in terms of cost and reliability [26].

The overall organization of system composed of the SCPC, the SCHJ, and the DM is depicted in Figure 6. Thanks to the parallel access on reading to the SCHJ and the DM, data can be simultaneously checked in both places. If the data corresponding to a given address is found in the SCHJ, it will be preferred to that present in the DM as it is the most recently written data. If the address is not found in the SCHJ, the data will be read from the DM. As both accesses to SCHJ and DM are done at the same time, there is no time overhead due to an MISS in finding the data in the SCHJ. Actually, whether data comes from the SCHJ or the DM is totally transparent to the processor.

Data is not written directly to the DM in order to insure the trustability of DM: data to be written is always written first in the SCHJ. The corresponding address is always searched in unvalidated area so no two data elements in this area correspond to the same address. If the address is found, the data element is updated. Else, a new row is initialized in the unvalidated area with $w = 1$ and $v = 0$ and the address, data and parity-bit fields filled with the adequate values.

Before transferring to the DM, data awaits for the validation of the current sequence at the validation point (VP). The waiting delay depends on the number of instructions being executed in a sequence, that is, the sequence duration (SD). If no error is found at the end of the current sequence, the processor validates the sequence (sending the validation signal to the SCHJ). All the UVD in the SCHJ is validated by switching the corresponding v bit to 1. Otherwise, if any error is detected, the sequence is not validated and the UVD data in the SCHJ is disclosed by switching the corresponding w bits to 0. As one can easily imagine, the w and v bits are used to denote written and validated data, respectively. Only data having $v = 1$ can be transferred to the DM.

The last instructions in a sequence are used to write the SE to the SCHJ. In our processor core, SE includes the following internal registers: Program Counter (PC), Data Stack Pointer (DSP), Return Stack Pointer (RSP), Top of Stack (TOS), Next of Stack (NOS), and Top of Return Stack (TORS), which is few compared to modern RISC and CISC based processors. On sequence validation, all the data written to the SCHJ during the current sequence get the v bit set to 1 and are consequently sent to the DM. In case the sequence is not validated (see Figure 7), the SE data is restored from memory on rollback as the UVD in the SCHJ is dismissed, and execution is restarted from previous VP. Further explanation on the rollback operation can be found in [28].

As stated before, the on-chip DM is supposed to be fast enough to fulfill the performance requirements of our SCPC. Our strategy of using a SCHJ aims not only to improve fault tolerance but also to allow the rollback mechanism to be used with very little time penalty, and much less area overhead compared to a full hardware approach.

Each row in the SCHJ is protected by a SEC-DEC code as shown in Figure 8. This protection is used the following way:

(i) any error detected in the UVD area will result in the sequence invalidation. One can argue that having a strategy to correct more than a single error is possible but, practically, the hardware overhead necessary to implement it would be too high as all the rows must be checked concurrently;

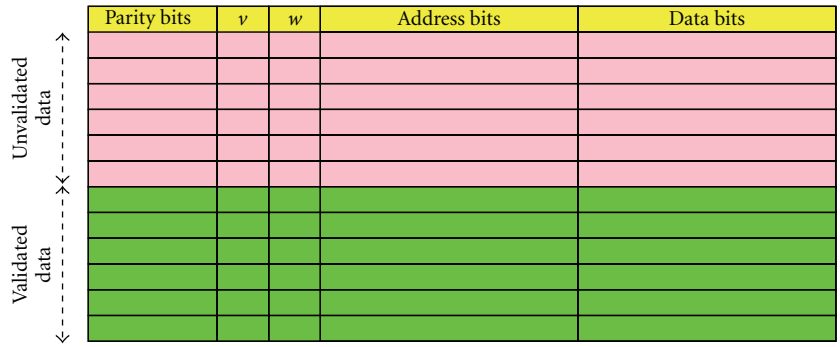


Figure 5: SCHJ structure.

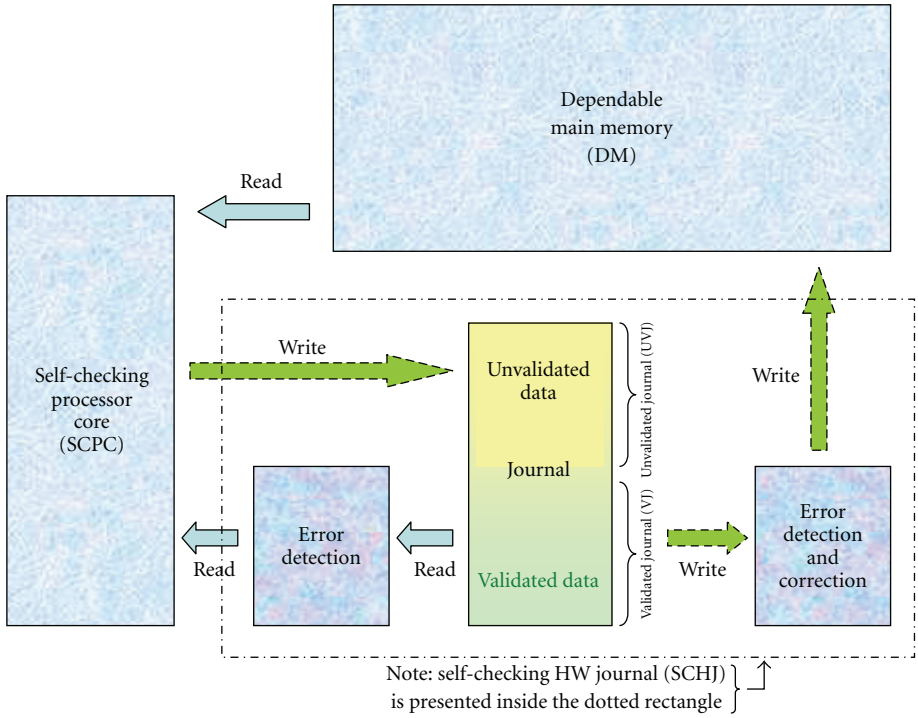
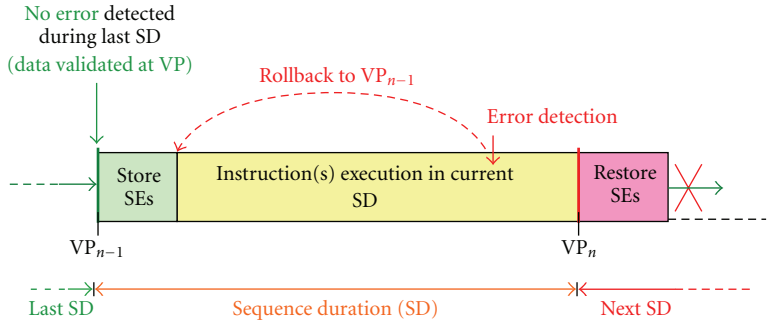


Figure 6: Overall Architecture.



Note: VP: validation point
SE: state-determining element(s) of the processor

Figure 7: Rollback mechanism on error detection.

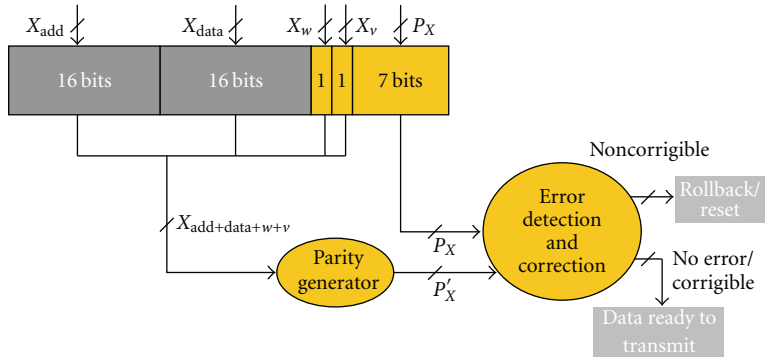


Figure 8: Error detection and correction in journal.

(ii) VD area is written row by row to the DM. The overhead related to the hardware mechanism necessary to correct a single VD row before it flows to the DM is thus acceptable. It is to be noticed that the VD still in the SCHJ is the only copy of the latest validated sequence. Thus, throwing away this data would avoid correct completion of the program/thread execution and require a system reset. This may still happen if an error is detected that overpasses the correction capacity of the code (e.g., a double error in a single VD row).

The overall operation of the SCHJ is depicted in the self-explanatory flow chart of Figure 9. Four modes of operation can be distinguished which are summarized in Table 1. The traffic signals in Figures 10, 12, 14, 15 are with respect to write-operation.

Mode 00. This mode is active on start of program or whenever a noncorrigible error is detected in a VD of the SCHJ. In this mode, the processor is reset, the execution starting (or restarting) from default initial values. All the data stored in the journal is cleared, with all the w and v bits being set to 0. There is no data exchange between the SCPC, the SCHJ, or the DM, as shown in Figure 10.

Mode 01. This is a normal read or write mode depending on the active instruction in the SCPC (rd = 1 or wr = 1). In this mode, the SCPC can write directly into the SCHJ but not into the DM, in order to avoid any risk of data contamination in the DM. Read access is allowed both from the SCHJ and the DM (not shown in Figure 12 to avoid complexity). The data read from the SCHJ is checked for possible errors. On error detection, the processor enters mode 11 in which rollback mechanism is activated without waiting for the VP of the current sequence.

A deeper analysis of this mode is useful because most of the execution time corresponds to this mode (in average, more than 80% execution time is in mode 01). As shown in Figure 11, when the processor wants to read from the SCHJ, the address tags are checked to match the requested data element (arrow *a* in Figure 11). If the corresponding address is found, the related data is checked for possible errors before its transfer towards the SCPC. The checking is done comparing the stored parity bits (of the employed Hsiao code) to regenerated parity bits in the error detection unit (shown in Figure 11).

Table 1: SCHJ operating modes.

Modes	Operation
00	Initialization
01	Read/Write
10	Validate ($v = 1$)
11	Invalidate (rollback)

(i) If an error is detected (shown in Figure 11), the rollback mechanism is invoked because data contents in UVJ contains data generated during the current sequence (denoted by the v field set to 0). The *Enable* signal on the data bus is then set to 0 to forbid further data transfers from the SCHJ to the SCPC. All the data contents written during this sequence are considered as garbage values ($w \leq 0$).

(ii) If no error is detected, the requested data is sent to the SCPC. The data bus between the SCHJ and the SCPC is activated while the data bus between the SCHJ and DM is temporary unactivated. Note in Figure 11: the arrow 1 shows the $w = 1$ which indicates the data written in SCHJ while the arrow 2 shows the data validated during the previous sequence.

Mode 10. This mode relates to the transfer of validated data (VD) from the SCHJ to the DM (see Figure 14). This mode is active at VP when no error has been detected during the current sequence. The sequence is thus to be validated, and all the data written into the SCHJ during this sequence must now have its v bit set to 1 ($v \leq 1$). In consequence, the transfer of this freshly validated data to the DM is allowed, and will be processed on following clock cycles, based on the availability of the data bus.

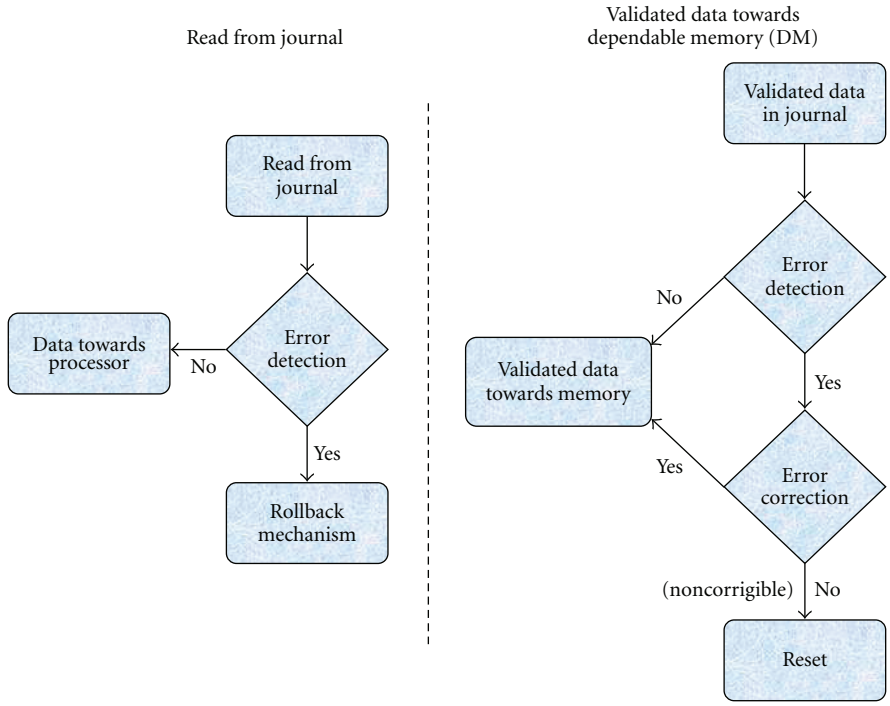


Figure 9: SCHJ operation flow chart.

The VD elements being written to DM are checked for errors affecting them as a consequence of transient faults that may have occurred during their stay in the SCHJ.

(i) If a corrigible error is detected, then data is send to the DM after correction;

(ii) if a noncorrigible error is detected (the probability for this to happen is very low normally), for example, a double error affecting the same memory block (see arrow b in Figure 13), then an unrecoverable situation arises as there is no way to rollback to a previous VP (the corresponding data being no more available in the DM). Resetting the SCPC and switching to mode 00 (and possibly raising some alarm indicator) is the usual behavior in this situation.

Mode 11. This mode is invoked when an error is detected during the read/write-operation as shown in Figure 15, and it has been partially discussed with mode 01. In this mode, all the data written in UVJ of SCHJ (i.e. all the data generated during the current sequence) is invalid and discarded ($w \leq 0$) as shown by the cross in Figure 11. Rollback is invoked to restart execution of the current sequence from the begin (after restoring the last valid SE). On successful restore, the mode 01 (read/write-mode) is activated.

4. Results

The SCPC and SCHJ have been modeled in VHDL at the Register Transfer Level (RTL) and implemented on a Altera Stratix II EP2S15F484C3 device and the Altera QuartusII.

The area dedicated to the SCHJ depends on the number of writes to different addresses that can be supported in a single sequence. This depends both on the sequence duration and on the rate of "writes" in the sequence. The later depends largely on the program being executed. Figure 16 depicts two extreme cases:

(i) a sequence of Nb "DUP" instructions will generate Nb "writes" to different addresses ("DUP" duplicates the top stack data element). Indeed, on each "DUP," the stack grows of one additional position generating a "write" to a new address in the SCHJ ("TOS" → "NOS" → SCHJ shift). Hence, Nb positions are necessary to hold the UVD of the sequence in the SCHJ. In the worst case, where two sequences of this kind follows one another, $2 * Nb$ positions could be required to hold the UVD and VD together. The practical value is closer to Nb than $2 * Nb$, as VD will be progressively written to the DM;

(ii) a sequence of any length of "SWAP" instruction will not generate any "write" to the SCHJ. Indeed, on each "SWAP", only "TOS" and "NOS" contents are swapped without any occurrence of a "write to" or "read from" the SCHJ.

In practice, considering the worst case is not very realistic and an average case is more indicated, which can be evaluated from representative benchmarks of the target applications to be executed on the processor.

Finding the optimal depth for the journal is a key issue in obtaining an acceptable node size for the future MPSoC architecture. Hence, we have investigated the impact of the

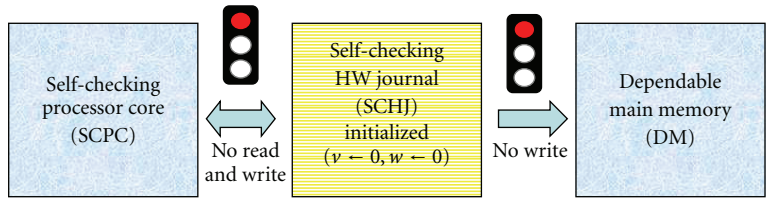


Figure 10: SCHJ relations in mode 00.

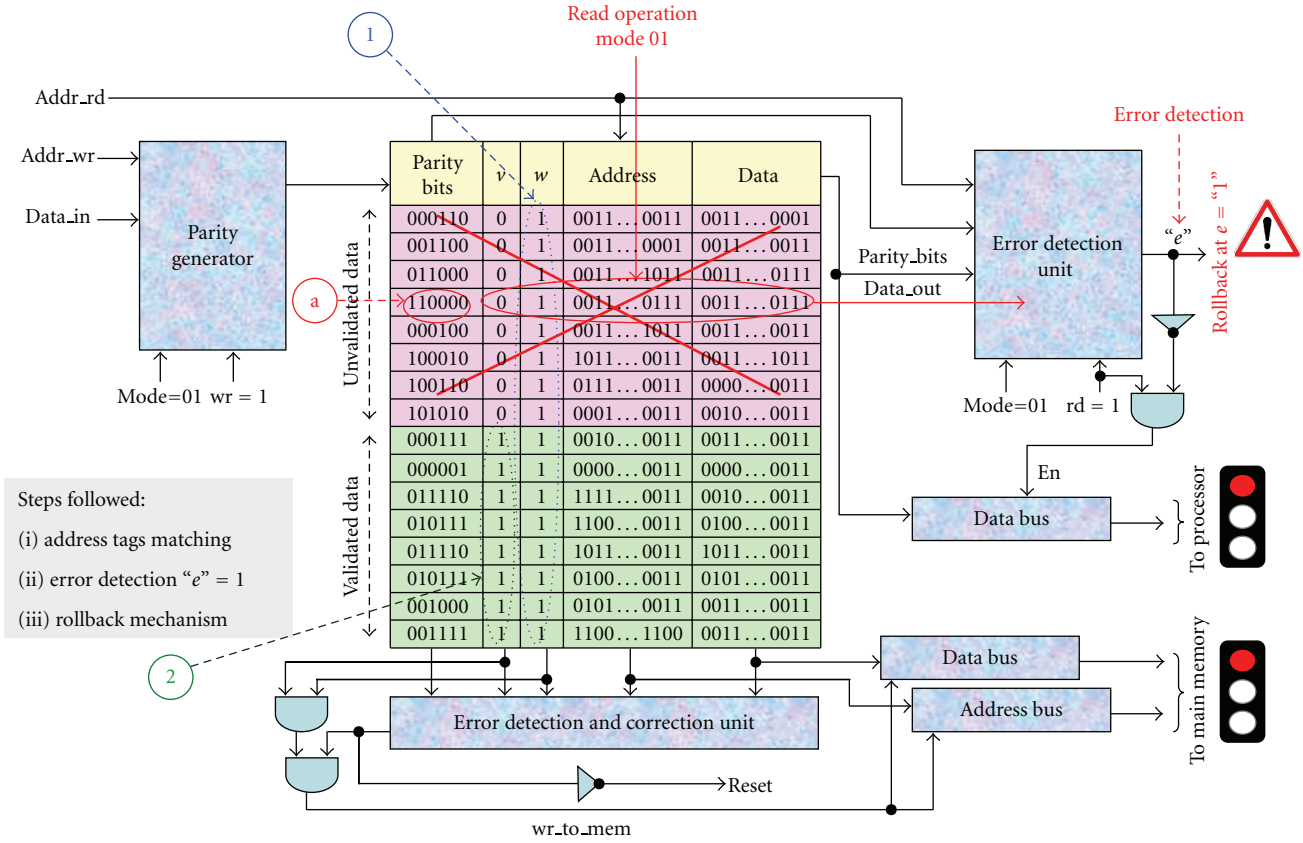


Figure 11: Reading UVD from SCHJ in mode 01.

SCHJ depth on the total area (SCHJ + SCPC). The results are reported in Figure 17 where vertical axes scale is in hundreds of combinatorial ALUTs.

The case of a journal depth of 50 is analyzed in Table 2. Here, the area dedicated to the SCHJ is about 54%. However, the total area required for the SCPC and the SCHJ together is reasonably small making it suitable for integration as an active node of an MPSoC. The analysis of varied algorithms shows that, most of the time, a depth of 50 is more than enough, due to data locality. Using a stack computing-oriented programming style enforces even more data locality on the stack, which in turn results in a smaller journal depth being required.

4.1. Validation by Error Injection. The overall architecture operation has been checked with fault injection techniques.

Table 2: Implementation area.

	Comb. ALUTS	(Ded. Logic)
SCPC and SCHJ	2400	(1295)
SCHJ alone	1305	(726)

Fault injection is a method to observe and evaluate the behavior of a system in a controlled experimental environment in which faults are introduced voluntary in the system [35]. In our case, we are injecting transient faults by simulation, that is, the faults are injected altering the logical values during the simulation. Simulation-base injection is as special case of error injection that has been widely used at different levels of abstraction such as logic, architectural, or functional [36].

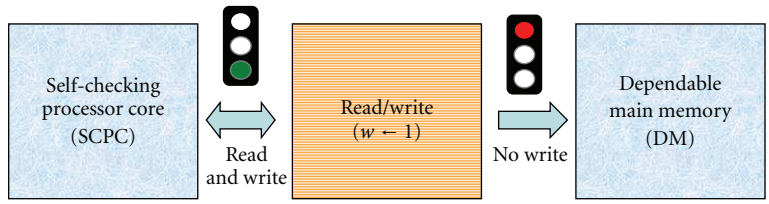


Figure 12: SCHJ relations in mode 01.

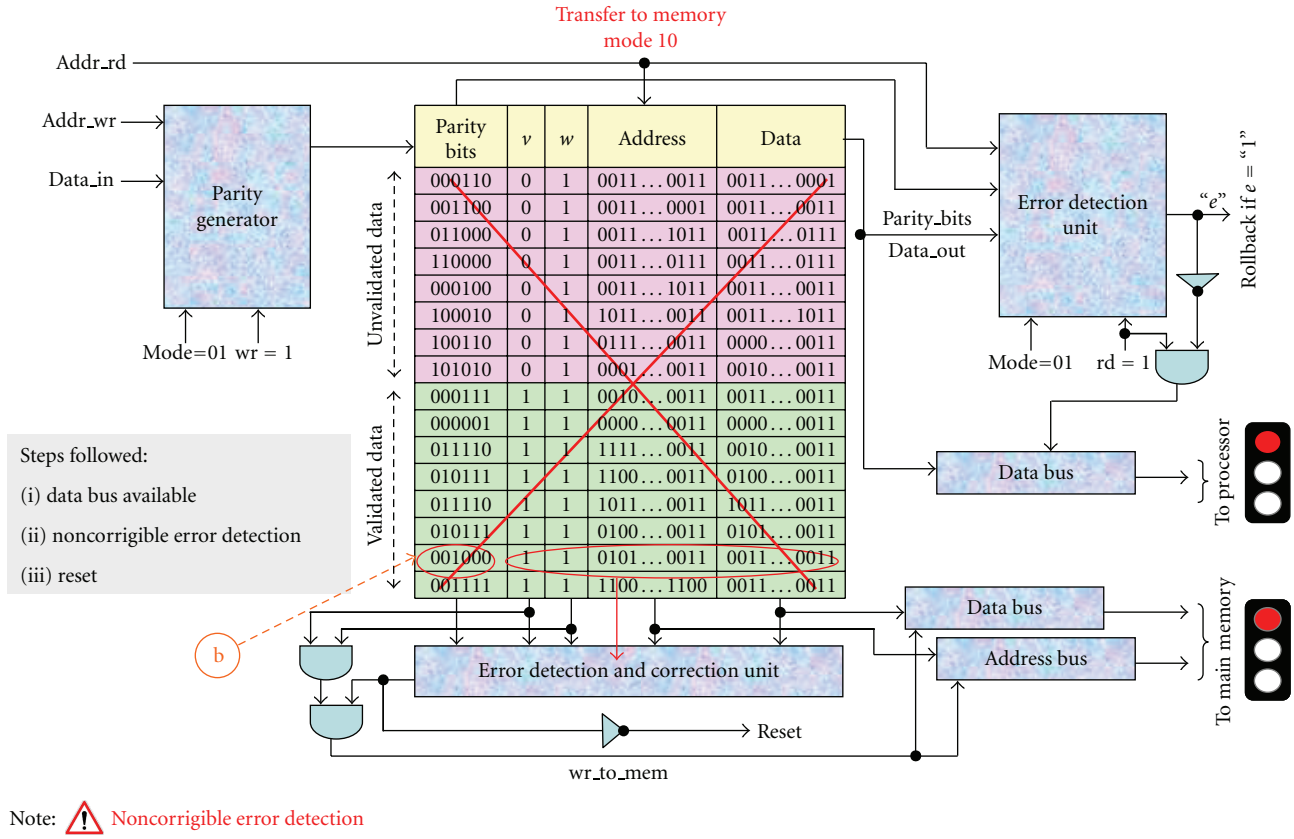


Figure 13: Mode 10 of SCHJ operation.

In order to allow very fast simulation (and hence, allow a large number of simulation campaigns to be conducted in a reasonable delay), dedicated C++ tools have been developed to replace the usual "discrete event driven" simulation model on which VHDL relies, by the faster "cycle driven" simulation model that fits very well synchronous designs [37]. For the simulation, strictly equivalent C++ "cycle drive models" replaced the original VHDL models at RTL level used for synthesis.

Our goal in this section is to check the fault tolerance properties of the full system, that is, to verify its correct operation in the presence of faults, as expected from the specifications.

Figures 18 and 19 show two different simulation situations: a recoverable and a unrecoverable error, respectively. In the first case, a detectable error occurrence in the unvalidated part of the journal (UVD). As seen on Figure 18, on error detection, the system rollbacks to the previous saved sure state. In Figure 19, a noncorrigible error is detected in the validated part of the journal (VD). Rollback cannot occur as it cannot recover the error, the data to recover being no more present in any place (DM or SCHJ). The only choice is hence to reset the processor and clear the SCHJ.

The faults considered are SEUs (Single Event Upsets: only one bit changes in a single register) and MBUs (multiple bits change at once in one register), injected at a randomly chosen moment. The corresponding error patterns are shown in Figure 20: (a) SEU (single bit error); (b) 1 up to 2-bit MBU; (c) 1 up to 3-bit MBU; (d) 1 up to 8-bit MBU. These fault patterns are commonly used ones in RTL models [35, 38].

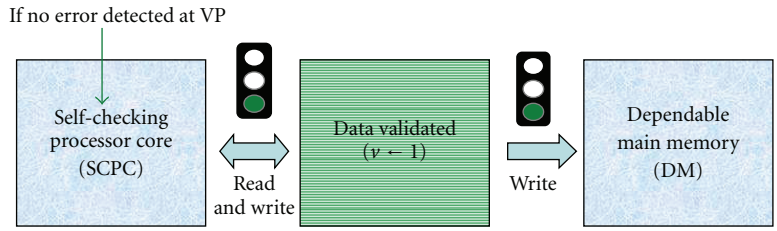


Figure 14: SCHJ relations in mode 10.

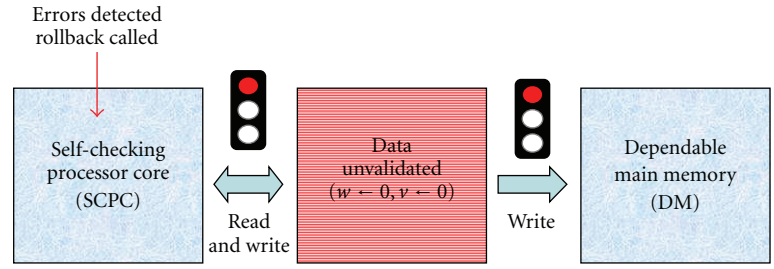


Figure 15: SCHJ relations in mode 11.

Note: the affected bits can be any, not necessarily those shown in the figure.

Different campaigns were conducted for each of the different error patterns. The results are presented in Figures 21, 22, 23, and 24, respectively. On error injection, the simulation is tracked to check whether the error is detected or not within a maximum of 2 cycles (the maximal detection latency of the processor). The simulation is hence stopped after two cycles in either cases, whether the error is detected or not. Then, a new simulation starts. The figures compile the cumulated results for each kind of error pattern.

The graph in Figure 21 shows that the proposed journalized processor architecture is 100% effective in detecting single bit random errors. The results remain reasonable, being higher than 60% and 78% for 2-bit and 3-bit errors, as shown in Figures 22 and 21. Despite the hard job the architecture must face with the last experimental conditions, using wide (1 to 8 bit) random error patterns, the error detection rate remains reasonable, with values greater than 36% for all the configurations. This clearly shows the effectiveness of the architectural approach being used to protect the processor against transient errors.

4.2. Performance Evaluation. There are two main limiting factors to the speed performance of the proposed fault-tolerant processor architecture.

(i) The encoding and decoding blocks in the journal, limiting the maximal operation frequency, as they are located in the critical path of the data incoming and outcoming the SCHJ. Some additional optimization is actually possible using some pipelining strategies in the journal (particularly on the "write to DM" path') and some advanced techniques for the error control coding implementation.

(ii) The delay induced by the rollback mechanism when triggered by the errors being detected. This delay is directly related to the amount of code requiring to be executed again. This in turn, depends on the injected error rate and on the length of sequences. Indeed, longer sequences are more prone to errors than shorter ones. Thus, for a given error rate, sequences are more likely to be faulty and require re-execution. It is to be noticed, however, that the journalization mechanism requires some execution overhead to backup the SCPC internal state. Thus, reducing too much the length of sequences will also have a negative impact. Theoretically, it is possible to determine the optimal sequence duration minimizing the delay due to re-execution [39]. However, in real-life conditions, it is generally not easy to estimate the the actual error rate.

To evaluate the impact of transient errors on speed performance, we have measured the degradation of execution time for different error injection rates (using the previously indicated error patterns) and different sequence duration on 3 different sets of benchmarks. In the corresponding graphs (cf. Figures 25, 26, and 27), time is measured as "number of Clock Per Operation" (CPO) and is plotted against the "Error Injection Rate" (EIR). The "no injected error" cases are the reference cases for each sequence duration SD, and match the fixed CPO value of "1."

The 3 sets of benchmarks are the following: group I contains algorithms where logic and arithmetic operations dominate; group II addresses memory manipulation algorithms (e.g., permutation of memory elements); group III is representative of typical control dominated algorithms. Table 3 summarizes, for each group of benchmark, the percentage profiles of "read from"/"write to" memory (SCHJ

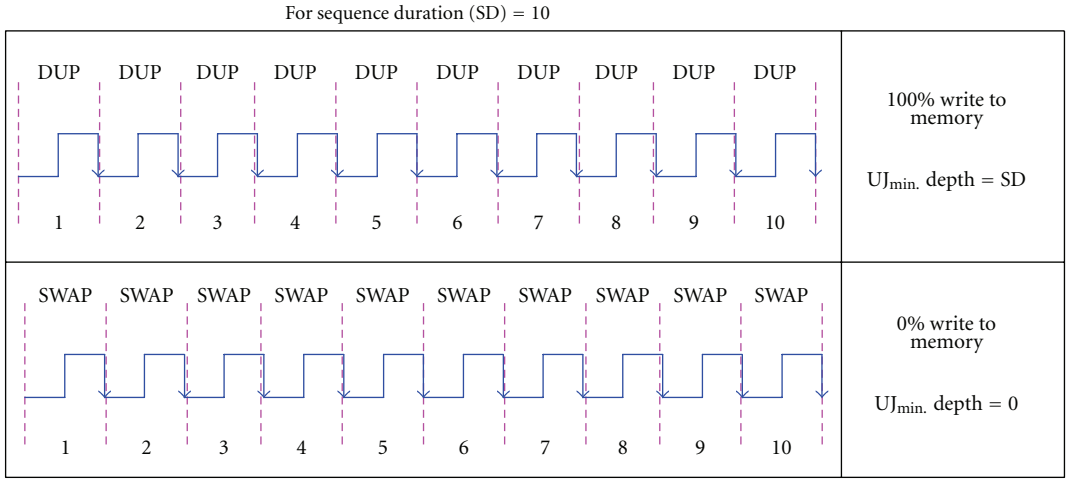


Figure 16: "Writes" rate for 2 different benchmarks.

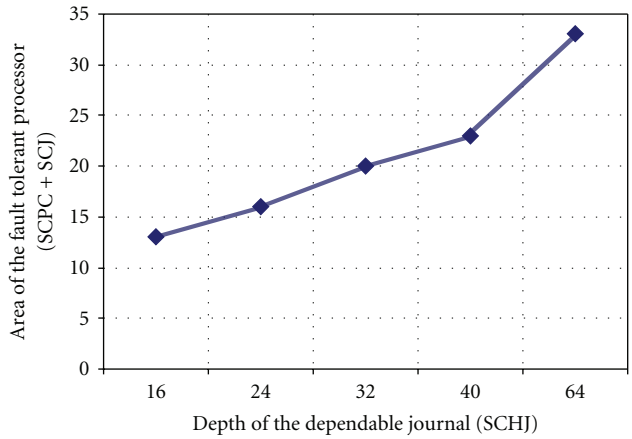


Figure 17: SCHJ/SCPC area ratio to SCHJ depth.

Table 3: Read/Write profiles in benchmarks groups.

Group	Read	Write
I	45%	39%
II	57%	38%
III	50%	38%

and DM) induced by the instructions running on the SCPC. Note that the instruction set of the SCPC has 36 instructions among which a majority (23 instructions) involve reading or writing from/to the memory.

Figures 25, 26, and 27 summarize the results for the benchmarks in Group I, Group II, and Group III, respectively. The analysis of the graphs show that the curves tend to overlap for the lower values of EIR. This is logic as, in absence of error, no time penalty due to rollback is induced whatever the benchmark being used.

In Figure 25, moving from point A to B incurs an increase of 10× in the error rate. The corresponding increase in CPO remains negligible for SD = 10 and 20. It remains low for SD = 50 with only a 10% overhead. The overhead becomes relatively important for SD = 100 with a value of about 60%. Moving from B to point C, a new error rate increase of 10× is applied. Now, the overhead is more important. Yet, it remains lower than 100% for SD = 20 and SD = 50, and still negligible for SD = 10. Similar observations can be seen in Figures 26 and 27.

In order to observe more directly, the effect of the error rate on the rollback mechanism, we have compiled the results for Group III in the graph of Figure 28, where rollback re-execution rate is measured for different error rates and different sequence durations. As expected, the rollback rate (and hence, the rollback penalty) is lower for the lower error rates. Furthermore, the impact of higher error rates is much larger for longest sequences than for the shortest ones.

With higher EIR, the smaller SD are the ones that denote the lower time penalty being incurred. This is also coherent with predicted results. Indeed, for a given error rate, the probability for a sequence to be invalidated is higher for a longer SD, hence leading to a higher rollback rate.

Taking into account that the architecture chosen for the SCPC requires little time to save the SE, it is possible to select a short SD and still have a good level of performance. Furthermore, this allows a lower SCHJ depth to be chosen with a reduced area consumption and diminishes the risk that errors cumulate in the SCHJ and induce a nonrecoverable error.

4.3. Comparison with FT-LEON-3. The LEON-FT is one of the most popular fault-tolerant processors of the last decade. It uses TMR (triple modular redundancy) to implement fault tolerance. In this section, we have chosen LEON FT-3 (the latest FT-version from LEON) to make some comparision with the approach proposed in this paper. It is the successor of the ERC32 and LEON processors, developed for the European Space Agency (ESA). The LEON FT-3 has been designed for operation in the harsh space environment and includes functionality to detect and correct (SEU) errors

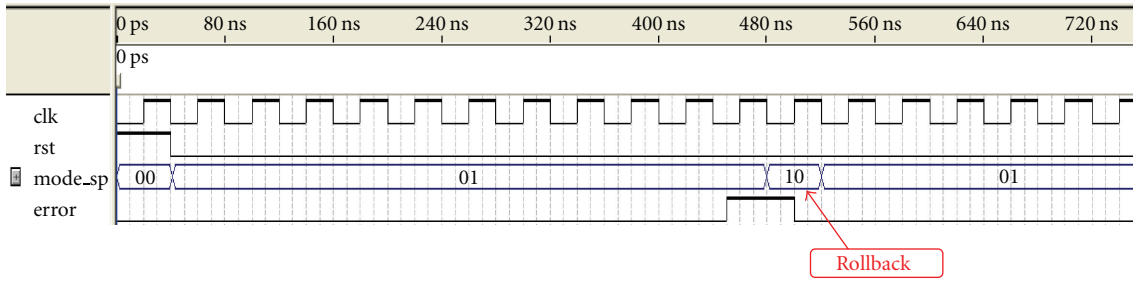


Figure 18: Recoverable error detection.

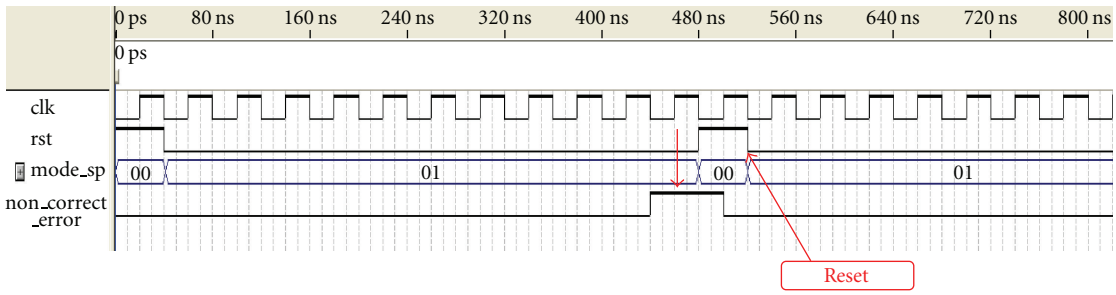


Figure 19: Unrecoverable error detection.

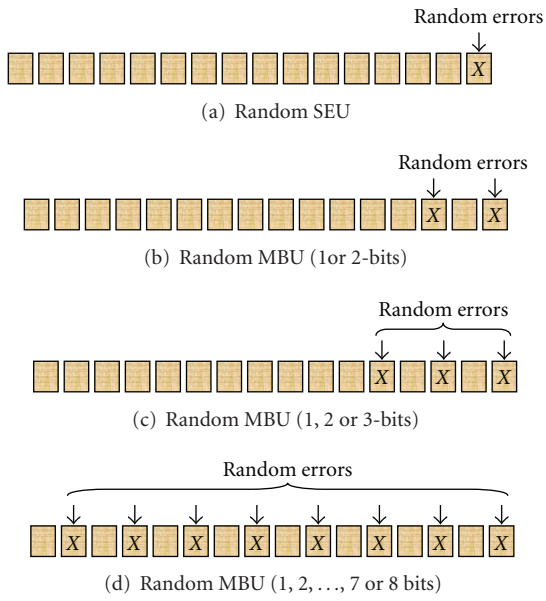


Figure 20: Error patterns for fault injection.

in all on-chip RAM memories [40]. We have chosen this processor for comparison because: (i) it is a well-established FT-processor, (ii) it has been used in MPSoC designs, and (iii) it is an open source machine.

Although it is difficult to compare the fault-tolerant strategies employed for these two very different processor architectures (a simple "low-budget" MISC stack processor and a rather complex and sophisticated RISC processor), it is interesting to check the validity of our approach compared to the well established approach of the LEON FT-3.

Table 4: Comparison of LEON-3 FT and Journalized Stack Processor.

	LEON-3 FT	Journalized Stack
Max. MHz	150	110
Pipelining	7-Stages	2-Stages
Register size	32-bits	16-bits
FT Method	Triple Modular Redundancy	Error Correcting Codes
100% Error Detection	4 bits out of 32 bits	1 bit out of 16 bits
Area	3500	1151

For fair comparison, both were implemented on Altera Stratix-III. The implementation results are reported in the Table 4. The parameters chosen for the comparison are the maximum clock frequency, the area, the error detection/correction approach, and the number of pipeline stages. Note that the area overhead related to the journal has not been included. Actually, this overhead can remain rather low for a journal tailored for a sequence duration of less than 16 (which is actually very effective).

The comparison shows that the LEON 3-FT requires more than three times the area of our processor. On the other hand, the maximum frequency of LEON3 FT is higher. Practically, there is still a margin of optimization available in our design, particularly in the journal, where additional pipelining can be used to reduce the critical path in the encoding and decoding circuitry for the error correcting code.

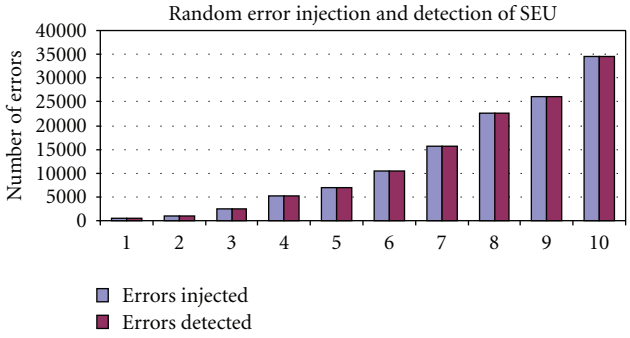


Figure 21: Results for single bit error injection.

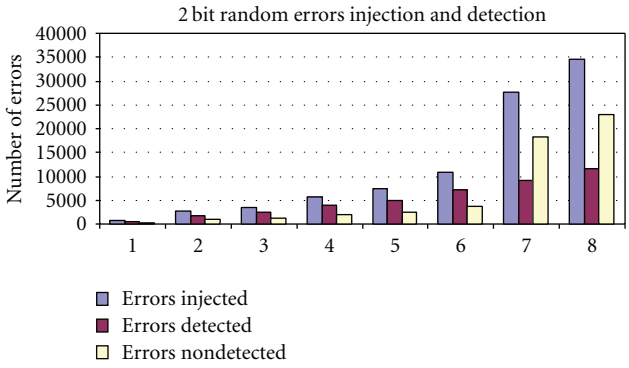


Figure 22: Results for 2 bit error injection.

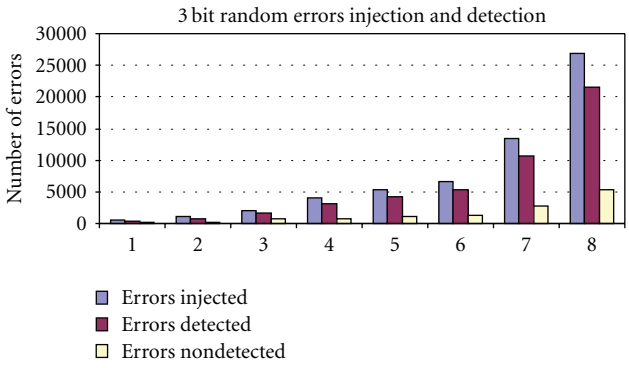


Figure 23: Results for 3 bit error injection.

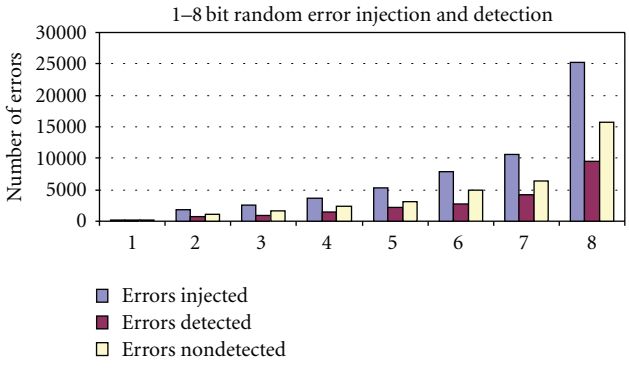


Figure 24: Results for 1–8 bit error injection.

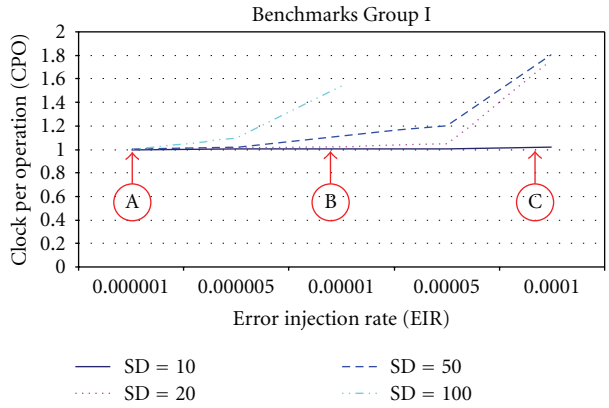


Figure 25: Simulation Curves for Group I.

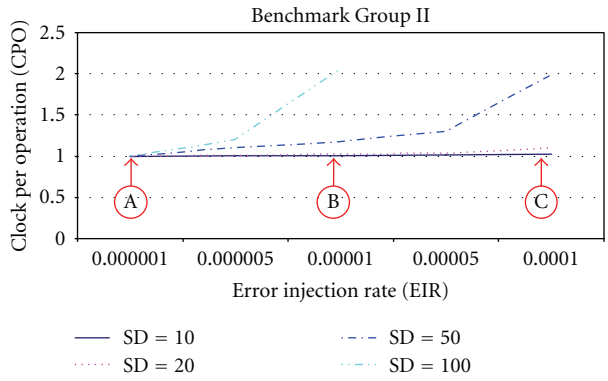


Figure 26: Simulation Curves for Group II.

LEON 3-FT is able to detect all error patterns up to 4-bits whereas the capability of the present version of our processor is only guarantied to single bits in each SCPC's internal register, and up to 2-bits in each word of the SCHJ.

However, this short comparison demonstrates that the principle of journalization can be rather effective on a stack computing-based processor core architecture and deserves more research effort to enhance the performances and protection capability.

5. Conclusion

In this paper, we present a specialized self-checking hardware journal being used as a centerpiece in a design strategy to build a transient fault-tolerant processor later to be used as a building block in massively parallel fault-tolerant MPSoC architecture. Together with the choice of the MISC stack computer architecture for the processor core (instead of RISC or CISC), it allows the combination of hardware error detecting techniques and error recovery through software rollback recovery to be a very effective approach to fault tolerance. The self-checking hardware journal is central to the journalization, the key functionality for fast rollback. This journalisation scheme is made possible thanks to the simple memory organization permitted by the processor core architecture choice.

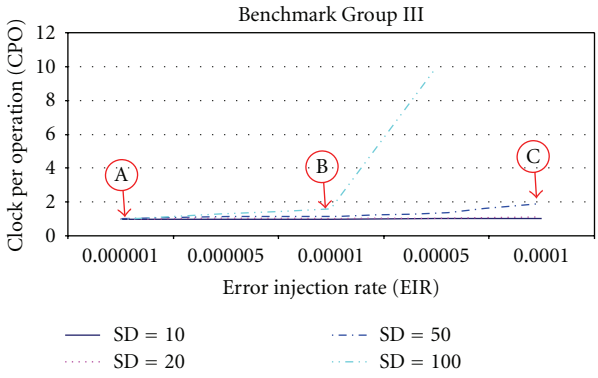


Figure 27: Simulation Curves for Group III.

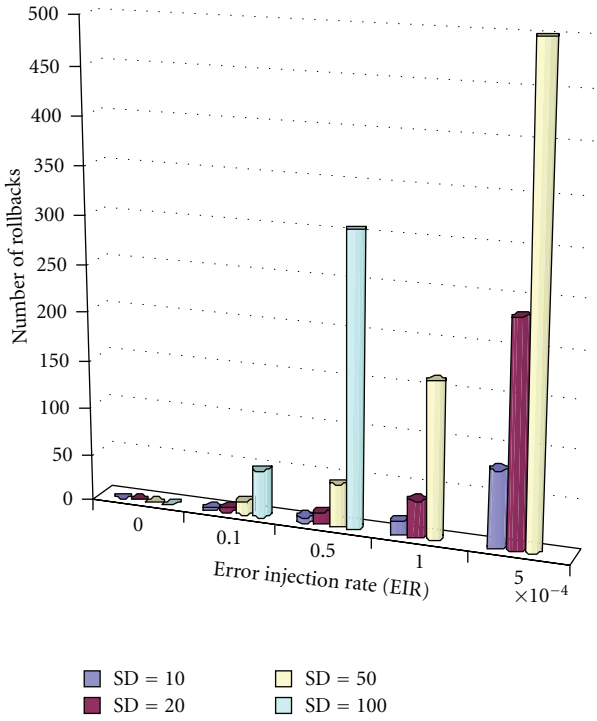


Figure 28: E.I.R effect on rollback (benckmarks of group III).

Implementation results show that area requirements are small and that speed performance degradation remains low under transient error injection, even for high error rates. For injection of simple errors, about 100% of the injected errors are detected and recovered for several experimental configurations. Similarly, for a double and triple bit error pattern injection, recovery capacity is about 60% and 78%. According to the results, the recovery is still possible, even for error patterns of up to 8 bits where recovery goes up to 36%. Therefore, the proposed approach can be effectively used in applications requiring a reasonable level of protection against transient errors at low HW cost.

In summary, the experiments presented in this section demonstrate beyond doubt that the proposed architecture is an interesting alternative approach to implement fault tolerance in processor architecture.

References

[1] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld et al., "IBM experiments in soft fails in computer electronics," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 3–16, 1996.

[2] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–315, 2005.

[3] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 389–398, June 2002.

[4] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri, "From a federated to an integrated architecture for dependable embedded real-time systems," Tech. Rep. 22, TU, Vienna, Austria, 2003.

[5] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[6] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[7] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating transient faults in MARS," in *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS '90)*, pp. 466–473, June 1990.

[8] J. Gaisler, "Preparations for next-generation SPARC processor," in *Proceedings of the Workshop on Spacecraft Data Systems*, May 2003.

[9] T. J. Slegel, "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, 1997.

[10] D. McEvoy, "The architecture of tandem's nonstop system," in *Proceedings of the Association for Computing Machinery Conference (ACM '81)*, p. 245, 1981.

[11] D. Ernst, S. Das, S. Lee et al., "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[12] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '99)*, pp. 196–207, November 1999.

[13] P. Chevochot and I. Puaut, "Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies," in *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '99)*, pp. 356–363, 1999.

[14] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Reliability-aware co-synthesis for embedded systems," in *Proceedings of the 15th IEEE International Conference on Applications-Specific Systems, Architectures and Processors*, pp. 41–50, September 2004.

[15] J. Xu and B. Randell, "Roll-forward error recovery in embedded real-time systems," in *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS '96)*, pp. 414–421, June 1996.

[16] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Systems*, vol. 20, no. 1, pp. 83–102, 2001.

[17] Y. Zhang and K. Chakrabarty, "A unified approach for fault tolerance and dynamic power management in fixed-priority

real-time embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 1, pp. 111–125, 2006.

[18] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Transparent recovery from intermittent faults in time-triggered distributed systems," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 113–125, 2003.

[19] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," in *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS '99)*, pp. 84–91, June 1999.

[20] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pp. 25–36, June 2000.

[21] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 87–98, May 2002.

[22] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Proceedings of the30th Annual International Symposium on Computer Architecture*, pp. 98–109, June 2003.

[23] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault-tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 80–107, 1996.

[24] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pp. 297–306, June 2007.

[25] B. P. Dave and N. K. Jha, "COFTA: hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance," *IEEE Transactions on Computers*, vol. 48, no. 4, pp. 417–441, 1999.

[26] M. Y. Hsiao, "A class of optimal minimum oddweight-column SEC-DED codes," *IBM Journal of Research and Development*, vol. 25, no. 5, 1970.

[27] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: a state-of-the-art review," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, 1984.

[28] A. Ramazani, M. Amin, F. Monteiro, C. Diou, and A. Dandache, "A fault tolerant journalized stack processor architecture," in *Proceedings of the15th IEEE International On-Line Testing Symposium (IOLTS '09)*, pp. 201–202, June 2009.

[29] M. Amin, F. Monteiro, C. Diou, A. Ramazani, and A. Dandache, "A HW/SW mixed mechanism to improv. The dependability of a stack processor," in *Proceedings of the 16th IEEE International Conference on Electronics, Circuits and Systems (ICECS '09)*, pp. 976–979, December 2009.

[30] P. H. J. Koopman, *Stack Computers: The New Wave*, Mountain View Press, La Honda, Calif, USA, 1989.

[31] D. B. Hunt and P. N. Marinos, "A general purpose cache-aided rollback error recovery (CARER) technique," in *Proceedings of the 17th Annual Symposium on Fault-Tolerant Computing*, pp. 170–175, 1987.

[32] N. S. Bowen and D. K. Pradhan, "Virtual checkpoints: architecture and performance," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 516–525, 1992.

[33] F. Liberato, R. Melhem, and D. Mossé, "Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems," *IEEE Transactions on Computers*, vol. 49, no. 9, pp. 906–914, 2000.

[34] D. K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall, New York, NY, USA, 1996.

[35] J. Arlat, A. Costes, Y. Crouzet, J. C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 913–923, 1993.

[36] J. A. Clark and D. K. Pradhan, "Fault injection: a method for validating computer-system dependability," *Computer*, vol. 28, no. 6, pp. 47–56, 1995.

[37] W. T. Chang, S. Ha, and E. A. Lee, "Heterogeneous simulation—mixing discrete-event models with dataflow," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 15, no. 1-2, pp. 127–144, 1997.

[38] P. Vanhauwaert, *Fault-injection based dependability analysis in a FPGA-based environment*, Ph.D. thesis, Institut National Polytechnique de Grenoble, 2008.

[39] M. Väyrynen, V. Singh, and E. Larsson, "Fault-tolerant average execution time optimization for general-purpose multiprocessor system-on-chips," in *Proceedings of the Conference on Design Automationand Test in Europe*, pp. 484–4899, April 2009.

[40] http://www.gaisler.com/doc/leon3ft-rtax.pdf.