Exploiting Idle Hardware to Provide Low Overhead Fault Tolerance for VLIW Processors

ANDERSON L. SARTOR, ARTHUR F. LORENZON, LUIGI CARRO, and FERNANDA KASTENSMIDT, Federal University of Rio Grande do Sul
STEPHAN WONG, Delft University of Technology
ANTONIO C. S. BECK, Federal University of Rio Grande do Sul

Because of technology scaling, the soft error rate has been increasing in digital circuits, which affects system reliability. Therefore, modern processors, including VLIW architectures, must have means to mitigate such effects to guarantee reliable computing. In this scenario, our work proposes three low overhead fault tolerance approaches based on instruction duplication with zero latency detection, which uses a rollback mechanism to correct soft errors in the pipelanes of a configurable VLIW processor. The first uses idle issue slots within a period of time to execute extra instructions considering distinct application phases. The second works at a finer grain, adaptively exploiting idle functional units at run-time. However, some applications present high instruction-level parallelism (ILP), so the ability to provide fault tolerance is reduced: less functional units will be idle, decreasing the number of potential duplicated instructions. The third approach attacks this issue by dynamically reducing ILP according to a configurable threshold, increasing fault tolerance at the cost of performance. While the first two approaches achieve significant fault coverage with minimal area and power overhead for applications with low ILP, the latter improves fault tolerance with low performance degradation. All approaches are evaluated considering area, performance, power dissipation, and error coverage.

CCS Concepts: • **Computer systems organization** → **Redundancy**; *Reliability*; • **Hardware** → **Error detection and error correction**

Additional Key Words and Phrases: Fault tolerance, VLIW, soft errors, adaptive processor

ACM Reference Format:
Anderson L. Sartor, Arthur F. Lorenzon, Luigi Carro, Fernanda Kastensmidt, Stephan Wong, and Antonio C. S. Beck. 2017. Exploiting idle hardware to provide low overhead fault tolerance for VLIW processors. J. Emerg. Technol. Comput. Syst. 13, 2, Article 13 (January 2017), 21 pages.
DOI: http://dx.doi.org/10.1145/3001935

1. INTRODUCTION

Technology scaling has been allowing increased logic integration and performance improvements in processors, as higher frequencies can be achieved. However, as the feature size of transistors decreases, their reliability is also compromised, so they get more susceptible to soft errors [Shivakumar et al. 2002]. Soft errors affect processors by modifying values stored in memory elements (such as pipeline registers, register files, and control registers) and are caused by numerous energetic particles such as protons and heavy ions from space or neutron and alpha particles at the ground level. To harden the processor against such errors, fault-tolerant techniques are mandatory

Authors' addresses: A. L. Sartor, A. F. Lorenzon, L. Carro, F. Kastensmidt, and A. C. S. Beck, Institute of informatics, Federal University of Rio Grande do Sul, Av. Bento Gonçalves 9500, 91501-970 Porto Alegre, Brazil; emails: {alsartor, aflorenzon, carro, fglima, caco}@inf.ufrgs.br; S. Wong, Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands; email: j.s.s.m.wong@tudelft.nl.

DOI: http://dx.doi.org/10.1145/3001935

for detection and correction before a failure in the system can be observed [Beck et al. 2012].

Very long instruction word (VLIW) processors are representative examples of current architectures that may suffer from the aforementioned issues (e.g., Intel Itanium [Sharangpani and Arora 2000] and Trimedia CPU64 [van Eijndhoven et al. 1999]). VLIW processors exploit instruction-level parallelism (ILP) by means of a compiler, executing several operations (instructions) per cycle depending on the processor's issue-width and the intrinsic ILP available in the application. These instructions are organized into words (*bundles*), and all instructions in a bundle are executed in parallel. VLIW processors occupy less area and dissipate less power when compared to traditional superscalar processors, since the process of scheduling instructions is statically done by a compiler. Therefore, the hardware of a VLIW processor is much simpler: the instruction queue, reorder buffer, dependency-checking, and many other hardware components are not needed.

Even though VLIW processors are also used in space missions [Villalpando et al. 2011], the focus of this work is to provide the best trade-off when it comes to area, performance, fault tolerance, and power dissipation. Therefore, providing fault tolerance at a low cost, as processors are getting more susceptible to failures at lower altitudes, due to the technology scaling; instead of providing a bulletproof, and expensive, processor against faults. In addition, the pipelanes of a VLIW processor occupy about 45% of the core total area and the register file (which occupies the rest) can be protected with parity [Gaisler 1997; McNairy and Bhatia 2005] or error correction codes (ECC) [Slegel et al. 1999].

In several cases, however, the compiler is not able to fill all slots of the bundle with independent instructions [Aditya et al. 2000]. The solution is filling the unused slots with no operations (NOPs). These NOPs require memory bandwidth to be fetched, potentially increasing cache misses, which would result in performance degradation and extra energy consumption. In order to amortize such costs, several techniques have been proposed to remove these NOPs [Tremblay et al. 2000; Colwell et al. 1991; Conte et al. 1996; Jee and Palaniappan 2002; Sharangpani and Arora 2000; de Waerdt et al. 2005; Fisher et al. 2005; Raje and Siu 1999; Suga and Matsunami 2000; Hubener et al. 2014; Brandon et al. 2015]. Even so, the functional units of the issue slot responsible for executing the NOP (whether it was removed from code or not) will still be idle.

Therefore, we propose three approaches for detecting and correcting soft errors in VLIW issue slots (*pipelanes*) that will exploit this idle hardware to provide fault tolerance at a low cost. All approaches are based on a modified dual modular redundancy (DMR) approach with an instruction rollback mechanism, and they are implemented in an 8-issue VLIW processor. The first is a configurable mechanism implemented in hardware that uses idle issue slots in a given program phase (i.e., a sequence of bundles with similar number of instructions) to execute duplicated instructions. It has a variable number of duplicated instructions depending on the application phase: between zero (no fault tolerance) and four (full duplication). The second duplicates instructions whenever there is a free pipelane at a given cycle. In this case, the idle functional unit will execute a duplicated instruction to increase fault tolerance.

However, applications with high ILP will have a reduced number of NOPs. This lack of NOPs would reduce the opportunities for instruction duplication, which might not deliver the necessary protection against faults. The third approach goes one step further by attacking this issue. It allows the tuning of how much fault tolerance is needed for a given application by reducing the ILP at runtime (i.e., some issue slots are artificially freed by moving instructions to the next cycle) to increase duplication. For this process to occur, an ILP threshold, which is configured before application's execution, is used. When the average ILP of the application reaches such threshold, the instructions that

follow and use more than half of the issue slots are split into two, and executed in two cycles, providing full duplication for both halves. By changing the value of the aforementioned threshold, it is possible to configure how many instructions will be split throughout program execution, changing the level of fault tolerance provided and the incurred performance overhead.

The details of the fault tolerance techniques proposed by this work are presented in Section 2. Next, we describe the implementation and show the results. For that, a fault injection campaign was performed in several benchmarks on different configurations of the VLIW processor. We evaluate error coverage, area, power dissipation, and performance. Section 4 discusses related works and compare the proposed approaches with several others, considering many factors. Finally, Section 5 concludes this work and discusses future directions.

2. VLIW PROCESSOR AND PROPOSED FAULT TOLERANCE TECHNIQUES

The VLIW processor used in this work is the ρ-VEX softcore VLIW processor [Wong et al. 2008], implemented in VHDL. The ρ-VEX core has a five-stage pipeline, and it can be configured to have a different number of issue slots (e.g., 2, 4, or 8). Each pipelane (issue slot) may contain different functional units from the following set: ALU (always present), multiplier, memory, and branch units. For the three proposed approaches, which will be further explained in the next subsections, the 8-issue version (i.e., eight pipelanes) configuration is considered. It has the following organization: eight ALUs, four multipliers, two memory units, and two branch units (one branch and one memory unit only execute duplicated instructions), which is similar to other VLIW processors (e.g., Intel Itanium [Sharangpani and Arora 2000] and TMS320C6745 [Instruments 2011]).

When there are idle pipelanes (and how these idle issue slots are detected/configured will vary according to the different approaches implemented in this work), they are used to execute duplicated instructions (e.g., arithmetic operations, jump address of a branch, or the values of a memory operation) from other pipelanes. Then, their results (i.e., all output signals) are compared by a checker. Consequently, it may have none (eight issue slots without duplication) to full duplication (four main issue slots and four duplicated ones), depending on the available resources. The destination register, the register file's and memory's write enable signals are also compared by a checker.

The fault-tolerant implementation of the ρ-VEX is depicted in Figure 1. The pipelanes are numbered from P0 to P7, *Dec* stands for the decode stage, *Exe* for the execution (two cycles), and *WB* for the write-back stage. In order to keep the low overhead (area and delay), the duplication pairs are statically placed, that is, pipelane 0 with pipelane 4, pipelane 1 with pipelane 5, and so on. Therefore, the pipelanes are combined in a way that the first four are compared with the four last ones. For example, if pipelane 6 was going to execute a NOP, then it will execute the duplicated instruction from the pipelane 2 instead. Every functional unit is capable of executing both main program instructions and, in the case of pipelanes P4, P5, P6, and P7, duplicated instructions from its correspondent pipelane. The exceptions are the memory unit in pipelane 4 and the branch unit in pipelane 7: these units execute only duplicated instructions, since the ρ-VEX does not support more than one memory or branch operation per cycle. The compiler used in this work (HP VEX compiler) schedules the instructions starting from the lower issue slots (from 0 to 7), considering the availability of the functional units. Thus, our approach efficiently exploits this scheduling mechanism. For the sake of comparison, a fault-tolerant 4-issue version was implemented: it has full duplication (i.e., all pipelanes are duplicated in hardware), so it is also composed of eight pipelanes, as depicted in Figure 2. Each duplicated pipelane executes the same instructions than its regular counterpart with the full duplication approach.

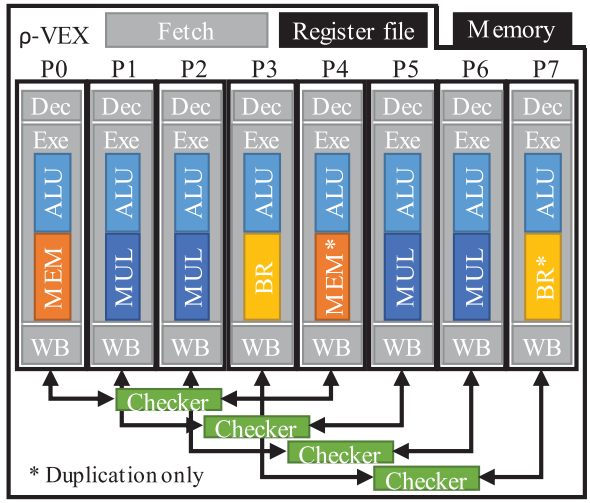


Fig. 1. Phase-configurable and adaptive duplication.

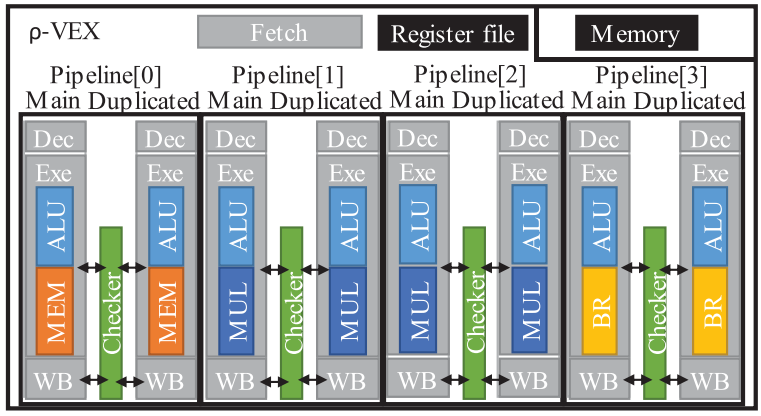


Fig. 2. Full duplication configuration for the 4-issue VLIW processor.

In order to not only detect an error, but also correct it, a rollback mechanism is used. When a mismatch is found in any of the compared signals, the rollback executes the last instruction again. The PC for the rollback is stored in a register, in case of an error, this stored PC overwrites the current PC (rolling back the execution). As no memory or register file was modified in the meantime (between the rollback PC and the current PC), the pipeline is simply flushed and the writing to the memory and register file are blocked, avoiding memory corruption. Once the rollback PC is loaded, the instruction corresponding to that PC is fetched again and the execution resumes from that point. Thus, having a fixed cost of 5 cycles to refill the pipeline, which is negligible considering the total number of cycles of an application and that this cost is only paid in case of an error.

As the checker has zero latency error detection, the memory and the register file will not be corrupted in case of an error, because the writing to these components will be disabled in time. Both the checkers and the rollback mechanism do not affect the critical path of the processor, as they operate in parallel to the pipelanes. In addition, the application does not have to be modified at all, as all the proposed techniques were

implemented in hardware. Modifying and recompiling the binary code may not be a trivial task, leading to incompatibility with future processors and losing backward compatibility. Hence, any compiler that supports the VEX instruction set architecture may be used to compile the applications (e.g., HP VEX compiler, GCC VEX, and others). The HP VEX compiler was chosen because it is more stable and robust than the GCC VEX.

The benchmark set chosen is composed of the following 10 applications, which comprises a subset of the WCET benchmark suite [Gustafsson et al. 2010]: adaptive differential pulse-code modulation (ADPCM), compress JPEG (CJPEG), cyclic redundancy code (CRC), discrete Fourier transform (DFT), Expint, finite impulse response (FIR), Matrix Multiplication, NDES (bit manipulation, shifts, array and matrix calculations), sums (recursively executes multiple additions on an array), and x264. From this point on, we will use only the benchmark's acronyms.

Each of the three proposed methods is suitable for different system requirements, presenting a trade-off between area, performance and power dissipation. Next, each one will be discussed.

2.1. Phase-Configurable Duplication

In this first approach, idle pipelanes during a whole given program phase (i.e., a sequence of instructions words that always have NOPs in specific issue slots) are used to execute duplicated instructions from other pipelanes. The first step for this approach is to profile the application, in order to detect the phases. This was done with the Mentor Graphic's ModelSim. After that, a table indexed by the program counter (32 bits) and containing the configuration of each application's phase (4 bits) is created. The phase configuration represents the function of each pipelane in a given phase, informing whether each issue slot will execute regular instructions of the application or execute duplicated instructions from another pipelane. Based on this table, the processor will dynamically change the function of the pipelanes and will enable or disable the checkers in each phase. The maximum number of phases for the considered benchmarks was 5, which results in a table of only 180 bits.

The profiling was performed for all applications from our benchmark set. The results for five benchmarks are depicted in Figure 3. The dots demonstrate when a given pipelane, identified by its ID (y axis), is being used (i.e., executing program instructions) in a given moment of the application's execution (x axis). The profiling for the other five benchmarks has a similar behavior to the one from the Matrix Multiplication benchmark (i.e., there are no idle phases). The idle phases that were used to execute duplicated instructions are highlighted in Figure 3 (empty blocks in Figure 3(a, b, d, e). The blank areas of a given pipelane ID represent a period of time in which this pipelane is idle (executing NOPs only).

Figure 4 depicts each phase for the ADPCM benchmark: the *P0-P7* represent pipelanes 0 to 7. The pipelanes in white background are executing duplicated instructions from the other pipelane, according to their respective pairs (as discussed in Figure 1). The pipelanes in black background are executing main program instructions. In this example, there are phases with full duplication (phase 4), partial duplication (phases 2 and 5), and no duplication (phases 1 and 3).

As it can be noticed, the ADPCM, CRC, Sums, and x264 benchmarks have phases when some issue slots are not utilized. On the other hand, as the Matrix Multiplication, CJPEG, DFT, Expint, FIR, and NDES benchmarks do not have such phases, they cannot take advantage of the phase-configurable approach, because the modified processor would have the same behavior as the unprotected version. Therefore, even though this approach has no costs in terms of performance and negligible power overhead, it can be only used when the application has phases with lower ILP than the processor supports. This approach may also be used with power gating [Hu et al. 2004;

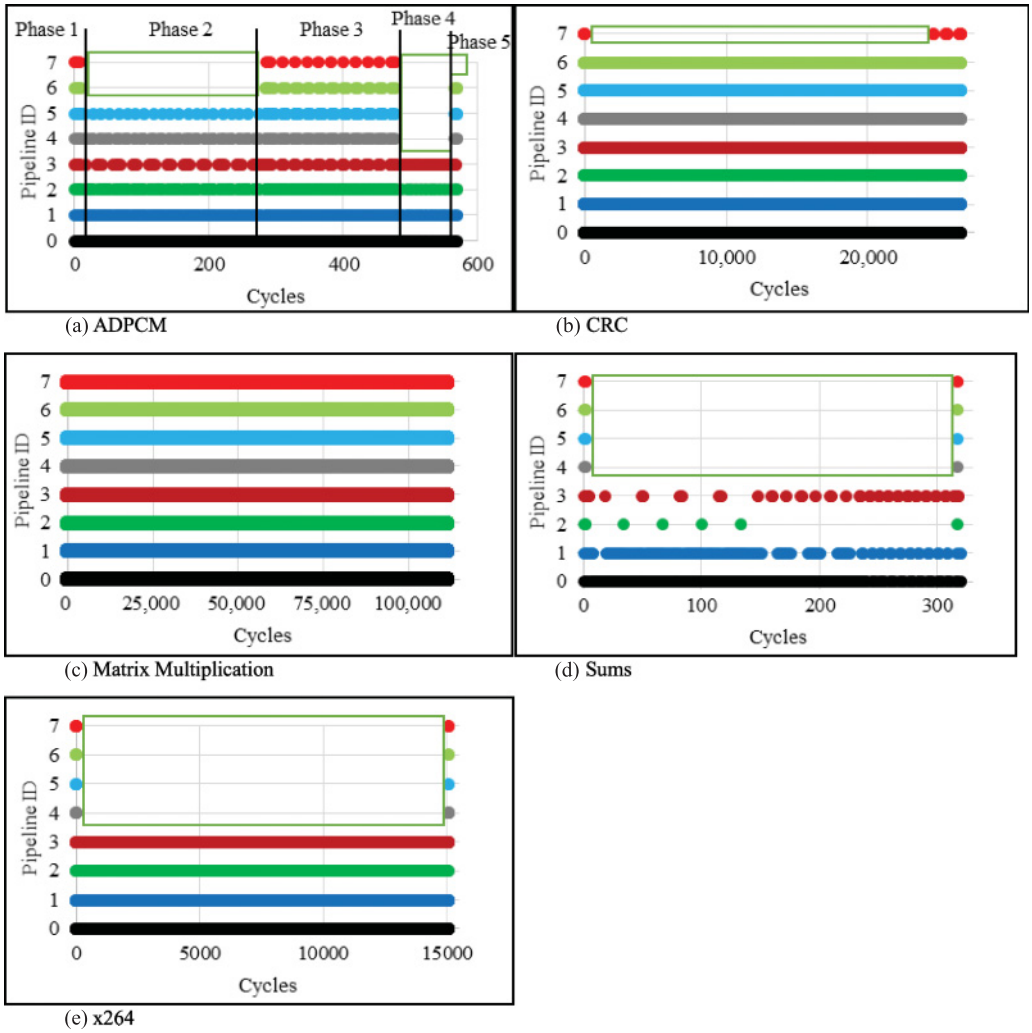


Fig. 3. Issue utilization and configurable duplication.

Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
P7	P7	P7	P7	P7
P6	P6	P6	P6	P6
P5	P5	P5	P5	P5
P4	P4	P4	P4	P4
P3	P3	P3	P3	P3
P2	P2	P2	P2	P2
P1	P1	P1	P1	P1
P0	P0	P0	P0	P0

Fig. 4. Phase-configurable duplication for the ADPCM benchmark.

Giraldo et al. 2015], for instance, allowing idle hardware to be shut down on noncritical parts of the program, consequently reducing the energy consumption of the application.

2.2. Adaptive Duplication

In this technique, the idle pipelanes are used to execute duplicated instructions when possible (i.e., when there are NOPs). Therefore, the verification is done on a per-cycle basis and not by phases as the previous approach. After fetching an instruction word, each pipelane receives one instruction, for decoding and further execution. We have modified this process so that the pipelane receives the program instruction (no duplication) or the instruction from another pipelane (when a NOP is found, it is replaced for a duplicated instruction). Therefore, the duplicated instruction is obtained simply by choosing between one signal or the other with a multiplexer. No additional accesses to the memory are required. Therefore, this approach is completely dynamic, providing fault tolerance adaptively. For this reason, its costs in area and power are higher than the phase-configurable configuration, as it will be presented later.

An example of code execution comparing the original (unprotected) 8-issue version with the adaptive duplication approach is presented in Figure 5(a) and Figure 5(b). P stands for the pipelanes, in which x corresponds to the pipelane number (0 to 7), tx represents the time, and the Ix the program instructions that are being executed. This will not affect performance, as all instructions of a VLIW bundle are executed in parallel, but it will increase fault tolerance when there are NOPs available.

2.3. Adaptive Duplication with ILP Reduction

As previously explained, the adaptive duplication exploits idle hardware to provide fault tolerance. However, when the VLIW bundle has more than half of the issue-width filled with instructions, the duplication will not be full, as depicted in Figure 5(b) at t2, t3, and t5.

The third and last method is able to perform the trade-off between performance and fault tolerance using an ILP threshold. Therefore, if the ILP in a given moment is high and the application still needs more fault tolerance, this method will reduce the ILP for that purpose. On the other hand, the first two approaches only exploited idle cycles, not being able to guarantee fault tolerance for high ILP phases. This flexibility comes at a cost in area and power; however, it is still low when compared to other techniques.

This process can be tuned by configuring the threshold that will activate the ILP reduction, offering a trade-off between fault tolerance and performance. A "utilization value" is calculated at every bundle and changed according to the ILP available in the current bundle. A dedicated hardware is used to calculate this value. When the utilization value reaches the threshold, the current bundle (if it has more than half of the issue-width occupied) is divided into two, so it is possible to apply full duplication to each half of the bundle.

The utilization value is calculated from the ratio between the sum of the number of used issue slots on the high part of each bundle (varying from zero to half of the issue-width) and the number of executed bundles that use more than half of the issue-width. Hence, this value represents the average utilization of the issue slots considering the bundles on which full duplication without the ILP reduction cannot be applied. Note that only bundles that have some instruction at the high part will change the utilization value; otherwise, the full duplication will be automatically applied, since it incurs in no performance penalties.

Examples of code execution using different thresholds (1 and 2) are depicted in Figure 5(c) and Figure 5(d), respectively. The instructions that at broken into two cycles (allowing full duplication) are highlighted by the arrows on the right side of the instruction word. When the threshold is equal to 1, every bundle that has more

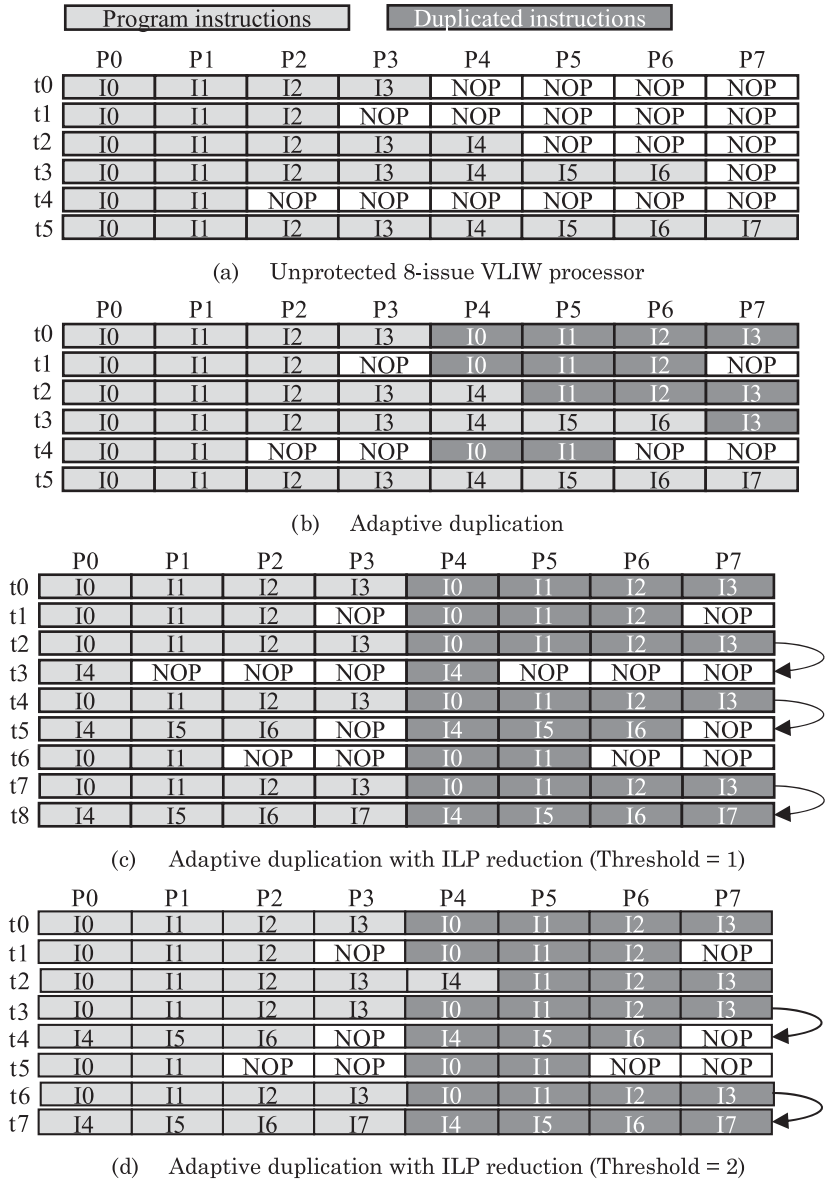


Fig. 5. Code execution example.

instructions than the half of the issue-width is split into two, because the utilization value will always be at least 1 for those bundles (e.g., t2, t4, and t7). When setting the threshold to 2, the bundle at time t2 will not be divided because the average utilization value will be equal to 1, which is below the threshold. The instruction bundle at t3 will be divided because the utilization value will be equal to 2 (4 used issue slots/2 bundles). The same reasoning goes to the instruction at time t6, which has a value above the threshold.

Let us analyze Figure 5 again. As it can be observed, there is no performance overhead when the adaptive duplication without ILP reduction is used (Figure 5(b)). However,

eight instructions would not be duplicated. By using ILP reduction with threshold equal to 1 (Figure 5(c)), we would have 50% of performance degradation with the ability to duplicate all instructions. If a threshold equal to 2 (Figure 5(d)) is chosen, there would be 33% of performance degradation, and one instruction would not be duplicated. Hence, either fault tolerance or performance can be prioritized for a given application by changing the technique and/or the threshold value.

3. RESULTS

3.1. Methodology

An extended fault injection campaign was performed to inject soft errors in the pipelanes and checkers of the VHDL version of the processor, making it possible to evaluate the failure rate. Next, the fault model is described:

Fault type: The injected faults are transient and comprise a single event transient (SET) that will affect a signal from the design. Note that a single SET may cause single bit upsets (SBUs) or multiple bit upsets (MBUs).

Injection place: The faults are injected in any atomic signal of the target module. All internal and low-level signals from the processor core are considered (the memory and the register file are considered to be ECC protected).

Injection instant: Follows a uniform probability function in the range between zero and t equal to the expected execution time from the application without faults.

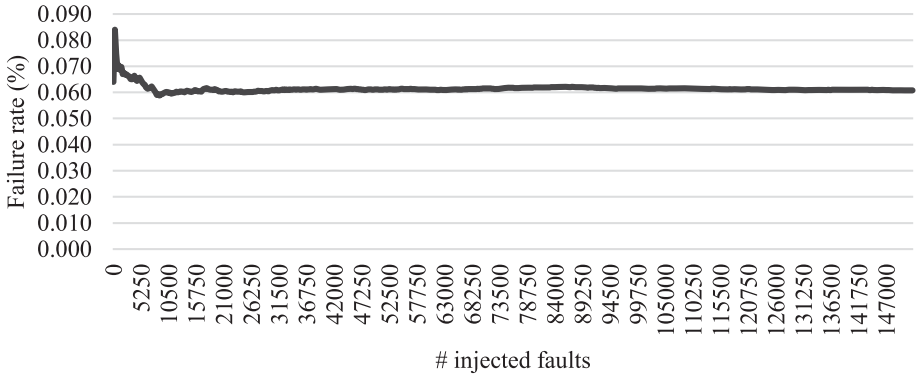
Fault duration: To increase the likelihood of the SET to be captured by a flip-flop, the signal is forced for the duration of one clock cycle.

The faults are injected via tool command language (TCL) scripts, which were developed in the context of this work, and the design is simulated using the Mentor Graphics' Modelsim simulator. One fault is injected per application's execution, due to the extremely low probability of more than one fault affect the same execution of a given application.

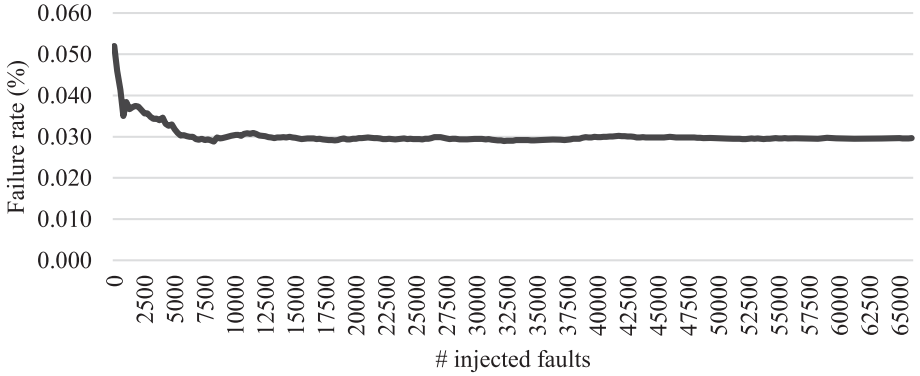
The total number of injected faults was 5.5 million (so there was the same number of application executions). The failure rate distribution as the number of injected faults increases is depicted in Figure 6 for three benchmarks, demonstrating that the failure rate stabilizes after a certain number of fault injections.

The aforementioned approach for injecting faults considers low-level signals of the processor; hence, it allows a controllable and precise injection of the faults. On the other hand, several other approaches rely on higher-level fault injectors, which are not able to precisely estimate the failure rate when the circuit area distribution is considered. Examples of such approaches are high-level simulators [Nakka et al. 2007; Sanchez and Reorda 2015], code instrumentation [Reis et al. 2005], assumption of 100% coverage [Ray et al. 2001], or works that propose fault tolerance techniques but do not evaluate the coverage of such techniques [Austin 1999; Subramanyan et al. 2010]. One approach to statistically determine the reliability of a structure is to compute the architectural vulnerability factor (AVF) [Mukherjee et al. 2003], which is the probability that a fault in a particular structure will result in an error. However, computing AVF for complex structures and processors requires knowledge of the synthesized components, which results in a loss of controllability regarding the target components that are under test. Exposing the circuit to high-energy particle accelerators suffers from the lack of controllability of which parts of the circuit that will be exposed to the radiation beam. In addition, as the ASIC chip of the ρ-VEX processor is not available, it is not possible to test its behavior under a source of energetic particles.

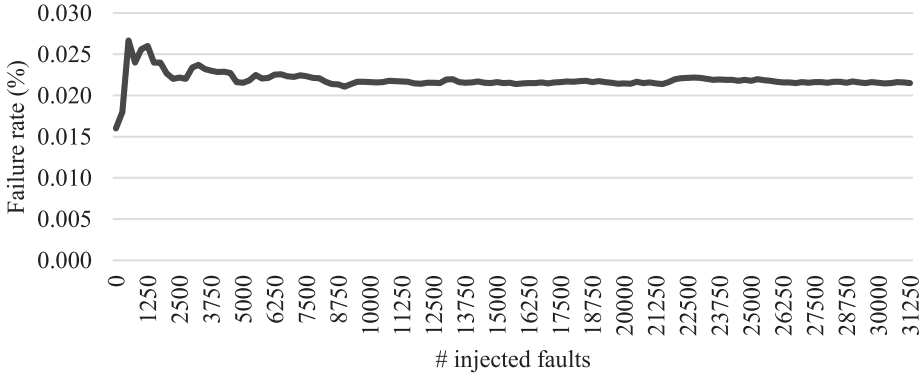
The synthesis tools used were the Xilinx ISE synthesis tool to obtain the FPGA area and frequency using the Virtex 6 - XC6VLX240T FPGA, and the Cadence Encounter



(a) CJPEG



(b) CRC



(c) NDES

Fig. 6. Failure rate behavior as more faults are injected (unprotected processor).

RTL compiler to obtain power dissipation and ASIC area, using a 65nm CMOS cell library from STMicroeletronics. There are three possible reasons for failures in the system (the distribution of these failures types is evaluated in Sartor et al. [2015]):

- *Data failure*: there is a mismatch between the memory dump from the application and the golden memory dump. The dumps are compared once the application ends its execution.
- *Data flow failure*: the application does not stop within the number of cycles that it should (i.e., number of cycles to execute the application without any failures).
- *Simulation failure*: some specific signals at specific times are flipped and crashes the simulation (i.e., ModelSim's simulation is aborted without finishing the execution of the application).

3.2. Failure Rate and Performance Degradation Analysis

Table I presents the failure rate and performance of the chosen applications in all proposed configurations (4-issue full duplication; phase-configurable; Adaptive only, without ILP reduction; and with ILP reduction, using Threshold = 1.75, 2, or 2.5 and Threshold = 1), and unprotected versions (4- and 8-issue). Note that, in some benchmarks, results of the adaptive version with ILP reduction are not shown for a threshold greater than 1 because the failure rate does not decrease significantly. On average, the unprotected processors have a failure rate of 6.61% and 3.73% for the 4- and 8-issue, respectively, while the protected versions present the following failure rates: 0.05% for the full duplication, 0.45% for the phase-configurable (considering only benchmarks with phases), 0.75% for the adaptive only, and 0.4% (threshold = 1) for the adaptive with ILP reduction. The unprotected 8-issue has a lower failure rate than the unprotected 4-issue due to the elevated number of NOPs in the VLIW instruction; therefore, the probability of a flipping bit affecting the result of an instruction is lower than on the 4-issue configuration. The failure rate comprises the detection and correction; all errors are detected, but not all can be corrected in time. Even though there is no latency for the fault detection, the circuit delay may prevent the memory or the register file to be blocked for writing in time. In these specific cases, the memory and register file are blocked a moment after the incorrect data began to be written, hence, generating wrong results in some cases.

The only approach that affects the performance of the applications is the adaptive with ILP reduction; all others have no performance overhead. Figure 7 presents the performance degradation (y axis) according to the threshold (x axis). It varies from zero to 27.25% with the threshold equal to 1 (the lowest possible value). As we increase the threshold, the performance degradation is reduced, being negligible (less than 1%) at 3.5. Therefore, performance degrades as the threshold reduces; on the other hand, fault tolerance increases.

Figure 8 depicts the trade-off between failure rate and performance of the adaptive approach without and with ILP reduction and different thresholds, normalized to the unprotected 8-issue version. "*Adpt.*" stands for "Adaptive Only" (without ILP reduction), and T stands for "Threshold" for the Adaptive with ILP reduction. The failure rate reduction varies from 61.68% (CJPEG executing on the Adaptive Only version) to 97.09% (Matrix multiplication on Adaptive with T = 1), while performance degradation reaches up to 27.25% (CJPEG on Adaptive with T = 1), when compared to the unprotected version.

In the CJPEG benchmark, for instance, when switching from threshold 2.5 to 1, the failure rate is further reduced from 65.65% to 86.96% (when compared to the unprotected 8-issue), and the performance degrades from 3.65% to 27.25% (also compared to the unprotected version). Therefore, for this benchmark, there is a large improvement

Table I. Failure Rate and Performance Degradation

		Unprot.	Prot.	Unprot.	Protected							
		4-issue	Full dup.	8-issue	Phase-config.	Adaptive	Threshold					
ADPCM	Failure rate (%)	6.93	0.06	3.66	0.99	0.66	T=1	0.59	T=1.75	0.65	T=2	0.66
	Exec. Cycles	571	571	568	568	568		633		621		574
CJPEG	Failure rate (%)	9.55	0.02	6.07	6.07	2.33	T=1	0.79	T=2.5	2.12		
	Exec. Cycles	508	508	411	411	411		523		426		
CRC	Failure rate (%)	5.20	0.06	2.95	0.64	0.33	T=1	0.32				
	Exec. Cycles	13,289	13,289	13,270	13,270	13,270		13,616				
DFT	Failure rate (%)	4.63	0.07	2.68	2.68	0.38	T=1	0.15				
	Exec. Cycles	35,072	35,072	32,575	32,575	32,575		32,979				
Expint	Failure rate (%)	4.21	0.05	2.37	2.37	0.13	T=1	0.13				
	Exec. Cycles	9,341	9,341	9,097	9,097	9,097		9,257				
FIR	Failure rate (%)	10.94	0.04	5.93	5.93	1.21	T=1	0.93				
	Exec. Cycles	119,392	119,392	111,769	111,769	111,769		120,095				
Matrix Mult.	Failure rate (%)	9.91	0.08	5.68	5.68	1.30	T=1	0.17	T=2	0.53		
	Exec. Cycles	111,050	111,050	111,025	111,025	111,025		113,929		112,547		
NDES	Failure rate (%)	3.99	0.04	2.09	2.09	0.42	T=1	0.24				
	Exec. Cycles	28,527	28,527	27,499	27,499	27,499		28,667				
Sums	Failure rate (%)	5.52	0.04	2.96	0.11	0.37	T=1	0.37				
	Exec. Cycles	332	332	319	319	319		319				
x264	Failure rate (%)	5.21	0.09	2.94	0.07	0.33	T=1	0.33				
	Exec. Cycles	15,102	15,102	15,089	15,089	15,089		15,090				

in fault tolerance, which comes at the high cost of performance. For benchmarks such as the ADPCM, the performance degradation of changing the threshold from 2 to 1 is greatly increased (1.06% to 11.44%), while the fault tolerance improvement is minimal (81.96% to 83.98%). On the other hand, other benchmarks, such as the Matrix multiplication, present high fault tolerance improvements with low impact on performance: with 2.62% of performance degradation, the failure rate reduction goes from 77.08% (Adaptive Only) to 97.09% (Threshold = 1).

3.3. Dynamic Threshold Adaptation

In this section, the dynamic threshold adaptation is exploited when executing a given application. This approach will adapt the threshold in order to cope with a given acceptable failure rate variation (AFRV) defined a priori by the designer before execution. In this experiment, the threshold starts at its lowest value and will gradually be increased

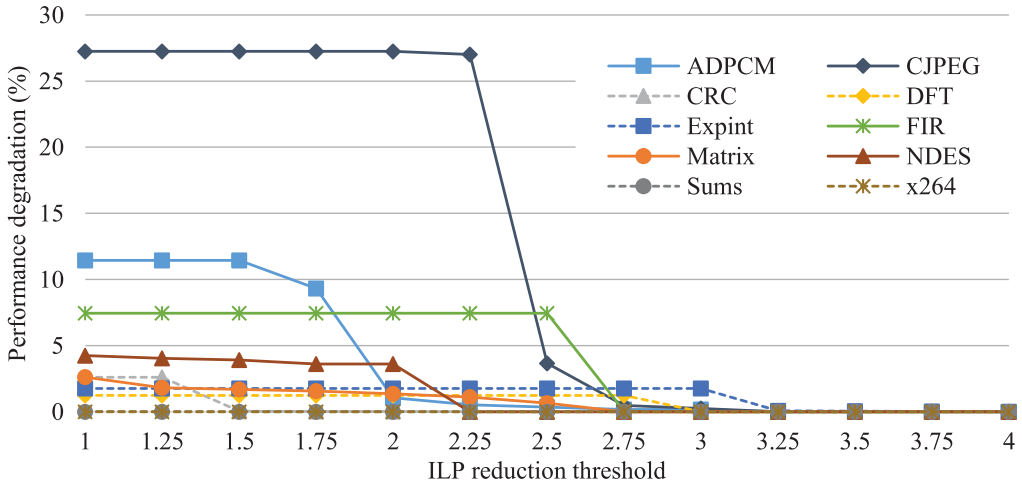


Fig. 7. Performance degradation when varying the ILP reduction threshold.

in order to reduce the performance degradation according to the AFRV: If the failure rate increases more than the AFRV, the threshold will be restored to its last value in order to maintain the failure rate within the bounds defined by the user.

Figure 9 depicts this approach being applied to three benchmarks. The application is executed in batches of 250 times, and for each batch, the threshold is gradually increased if the failure rate does not increase more than the AFRV (in this example, AFRV = 0.5%). Figure 9(a) presents the matrix multiplication benchmark, in which the threshold is gradually increased from 1 to 3.75 without reaching the AFRV value. However, when the threshold is increased to 4, the failure rate surpasses the AFRV, which triggers the threshold reduction back to 3.75 and restore the acceptable failure rate defined by the user. In Figure 9(b) (ADPCM), the threshold is increased from 1 to 4 without reaching the AFRV limit, and Figure 9(c) (CJPEG) reaches the AFRV value with a threshold equal to 2.5, which is reduced back to 2.25 for the next executions.

Figure 10 presents the performance improvement and the failure rate variation that the dynamic threshold provides when compared to the Threshold = 1. In the ADPCM benchmark, the dynamic threshold is able to improve the performance by 11.44% with a failure rate that varies from 0.59 (T = 1) to 0.66; for the matrix multiplication: 2.61% speed-up with 0.17 to 0.53% failure rate variation; finally, for CJPEG, 0.2% speed-up and no failure rate variation. Therefore, the dynamic threshold approach can be used to reduce the performance overhead and still maintain the failure rate within the bounds defined by the user.

3.4. Area and Power Dissipation

Table II presents the area (both FPGA and ASIC versions) and power consumption (ASIC only) for all VLIW configurations. The operating frequency for these configurations was set to 65MHz. As it can be observed, the overhead for the 4-issue full duplication is small in terms of area and power dissipation when compared to the unprotected 4-issue, even though the pipelines are duplicated (the area for each checker is less than 1%). The area overhead for the FPGA is 30% in LUTs and 35% in registers; for the ASIC is 50%, while the power dissipation overhead is 35%. The area overhead between ASIC and FPGA is not comparable with each other as different technology and synthesis tools are used. The overhead is almost negligible when one compares the phase-configurable approach with the base 8-issue configuration:

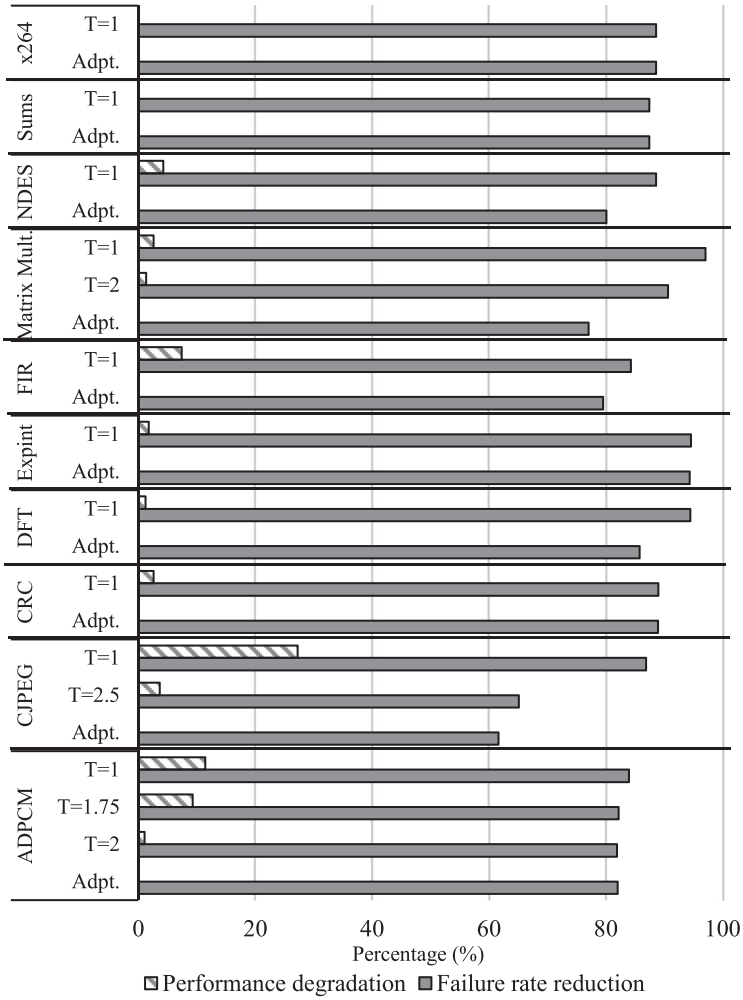
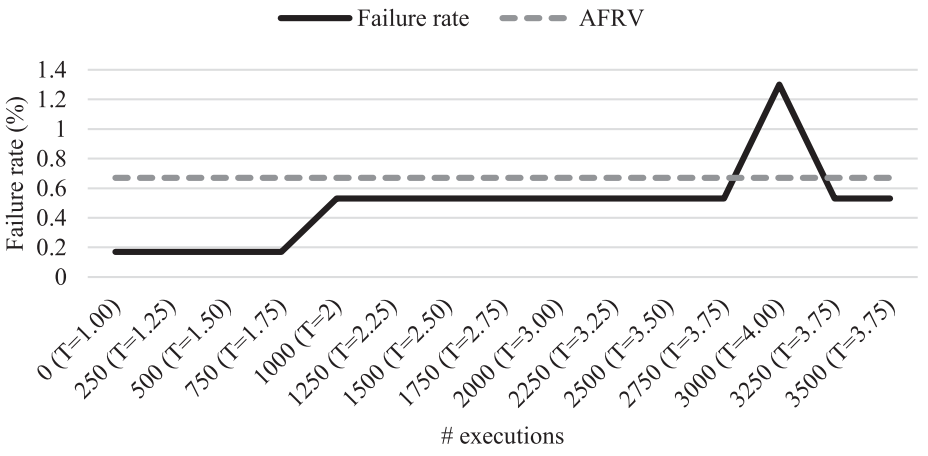


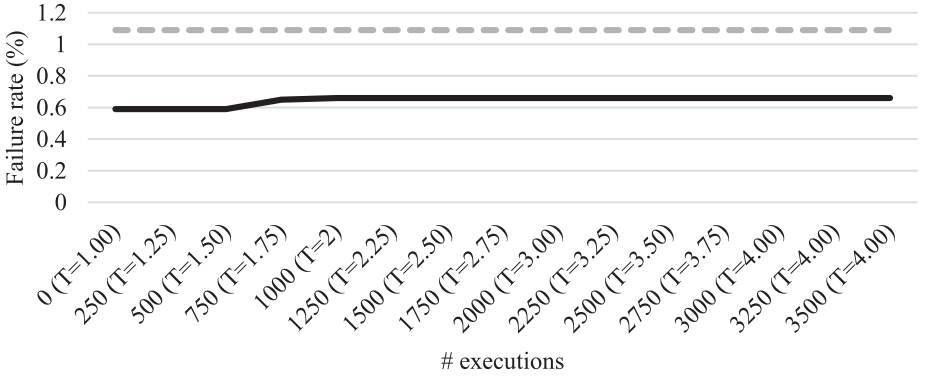
Fig. 8. Adaptive with ILP reduction normalized to the unprotected version.

2.6% for the FPGA (in LUTs) and 2.8% for the ASIC, with 3.8% overhead in power dissipation. For the adaptive version without ILP reduction, the overhead is also extremely low: 4.6% for the FPGA and 3.5% for the ASIC, while the power dissipation overhead is 5.3%. The overhead for the adaptive approach with ILP reduction is higher because of extra control circuitry. However, the overhead is still low when compared to other techniques (as it will be discussed next), being 18.3% for the FPGA, 14.1% for the ASIC and 27.6% in power dissipation. The energy consumption overhead for some of these techniques is further discussed in Sartor et al. [2016].

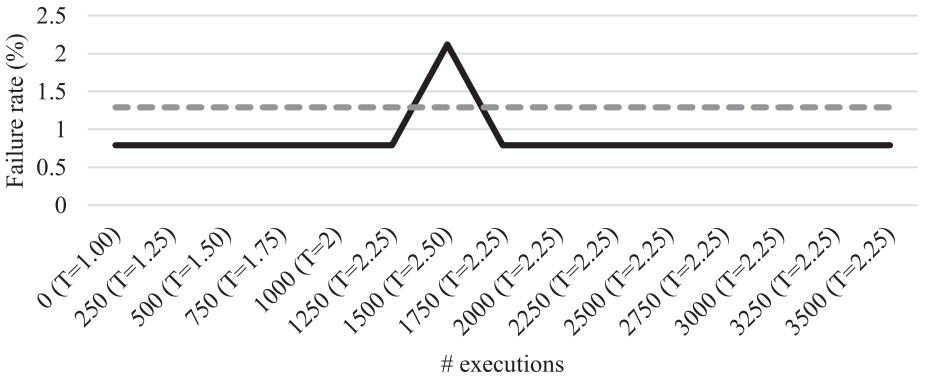
Therefore, each approach has its advantages depending on the target application and its requirements. For instance, the phase-configurable may be used with power gating, allowing idle hardware to be shut down on noncritical parts of the program. The adaptive approach is able to exploit idle hardware for low ILP applications in a completely transparent manner with extremely low overhead. For high ILP applications, the adaptive with ILP reduction can guarantee a certain amount of duplicated instructions and therefore fault tolerance.



(a) Matrix multiplication



(b) ADPCM



(c) CJPEG

Fig. 9. Dynamic threshold adaptation

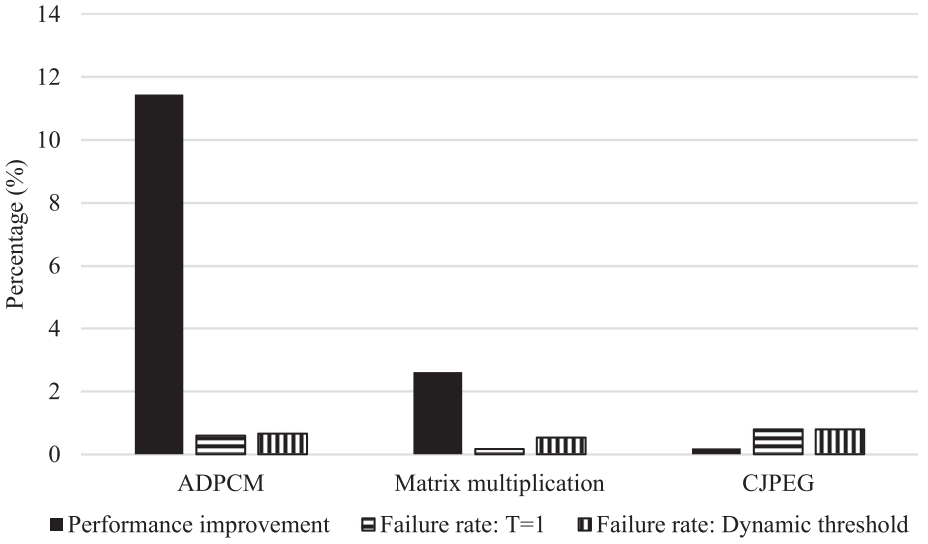


Fig. 10. Performance improvement and failure rate variation for the dynamic duplication approach when compared to the Threshold = 1.

Table II. Area and Power Dissipation Comparison

		FPGA		ASIC	
		Registers	LUTs	Cells	Power dissipation (nW)
Unprotected	4-issue	3,058	16,006	28,041	2,298,962.51
	8-issue	3,974	35,075	66,967	7,484,818.25
Protected	Full duplication	4,102	20,819	42,121	3,109,613.33
	Phase-configurable	4,133	35,973	68,849	7,771,568.02
	Adaptive	4,206	36,672	69,305	7,878,161.31
	Adaptive with ILP reduction	4,834	41,485	76,407	9,553,048.27

4. RELATED WORK

Several works have been proposed for the detection and correction of soft errors in VLIW and superscalar processors. These works aim to improve the fault tolerance of the target system, typically based on redundancy, which may be implemented in software, hardware, or both.

Dual modular redundancy (DMR) based on checkpoints with rollback was used by Xiaoguang et al. [2015] and Yang and Kwak [2010] to detect and correct errors. Whenever an error is detected, the state in which the execution was correct is recovered. Therefore, the latency to detect the error on these approaches will vary according to the periodicity of the checkpoints (i.e., when a new checkpoint must be made). On the other hand, the proposed duplication with rollback has zero latency detection as it compares the results at all times and executes again only the instruction word that presented the error. Therefore, in addition to the zero latency detection, the control structure of the rollback is much simpler than the ones that use checkpoints.

Another common approach is to triplicate a processor and use a majority voter (triple modular redundancy (TMR)), as implemented in Schölzel [2007] and Chen and Leu [2010]. In these cases, they only triplicate the functional units of a VLIW processor rather than the entire processor; therefore, it is possible to reduce area and power dissipation costs. Schölzel [2007] proposed the Reduced TMR, in which both hardware and software needed to be changed. If the two instructions (main and duplicated)

compute different results, the instruction is executed a third time. However, such approaches only cover errors that happen in the computation of a given operation. Therefore, errors that may occur before or after the execution stage are not detected. Moreover, the proposed duplication with rollback occupies less area and dissipates less power than Schölzel [2007] and Chen and Leu [2010] and does not change the binary code of the application.

Hu et al. [2005], propose a similar approach to Schölzel [2007]. However, instruction replication is done in software, so the binary code is changed, even though there is no area overhead. In the same way, replication is done partially to some instructions to amortize the costs in performance (although it also affects the capacity of providing fault tolerance). Anjam and Wong [2013] propose a TMR approach to be applied on the synchronous flip-flops. However, the area and power dissipation overheads are higher than the proposed technique.

Bolchini [2003] and Hu et al. [2009] propose a software-based redundancy based on duplication with comparison (DWC) for VLIW data paths aiming to reduce the performance overhead by using the idle functional units. However, these techniques still present huge performance degradation and increase code size, as they are implemented in software. Mitropoulou et al. [2014] propose an optimization to the DWC's generated code by reducing the impact of the basic block fragmentation caused by the check instructions, having lower, but still not negligible, performance degradation than the previous two techniques.

Tan and Fu [2012] propose to exploit idle streaming processors on GPGPUs by executing replicated warps. Even though this approach is implemented in software, the hardware also requires modifications. In addition, it is only able to detect errors in the execution, not correct them, as the proposed approaches. Also, no results regarding area, power and energy are provided.

The main limitations of software-based redundancy are the increase in the code size, energy consumption, and performance overheads that come with it. On the other hand, hardware-based redundancy approaches increase area and power dissipation with little or no performance overhead. The approaches proposed in this article, even though implemented in hardware, have low overhead in area and power dissipation.

Adaptive fault tolerance: Some works exploit the previous techniques in order to provide an adaptive fault tolerance mechanism. Jacobs et al. [2012] propose an adaptive framework that switches between different fault tolerance techniques depending on a priori knowledge of the environment, external events, or application-triggered events. The supported fault tolerance modes are TMR, duplication with comparison, algorithm-based fault tolerance (ABFT), internal TMR, and high-performance (no fault tolerance). This approach is for FPGAs only, as the hardware needs to be reconfigured. On the other hand, the proposed approach is implementation independent (i.e., can be used on both FPGAs and ASICs).

An adaptive checkpoint mechanism was proposed in Zhang and Chakrabarty [2004], in which the checkpointing interval is adjusted during the execution based on the occurrence of faults and the available slack. An offline preprocessing based on linear programming is used to determine the parameters that are provided to the online checkpointing procedure. Even though the checkpointing is adaptively made, the detection latency is still greater than zero, besides the need for preprocessing. Mills et al. [2014] replicate a task and execute the replicated task in a processor with lower processor speed in order to save energy. If the main task completes successfully, the duplicated one is terminated; otherwise, the duplicated task takes over, possibly at an increased processing speed, and completes the computation. Although this technique

Table III. VLIW Fault Tolerance Techniques Comparison

Technique	Error Coverage	Area overhead	Performance degradation	Power dissipation overhead	Code size increase
Phase-configurable duplication	∼100%	2.8%	∼0%	3.8%	0%
Adaptive duplication	∼100%	3.5%	∼0%	5.3%	0%
Adaptive with ILP reduction	∼100%	14.1%	∼0%–27.25%	27.6%	0%
DMR with rollback [Xiaoguang et al. 2015; Yang and Kwak 2010]	∼100%	0%	51%–100%	0%	100%
TMR	∼100%	200%	∼0%	∼200%	0%
Partial TMR [Chen and Leu 2010]	95%–99%	100%	0.6%–34.3%	∼100%	0%
Reduced TMR [Schölzel 2007]	∼100%	100%	0%–100%	∼100%	>0%
Reduced TMR - SW [Hu et al. 2005]	∼100%	0%	30%–60%	0%	100%
Flip-flops TMR [Anjam and Wong 2013]	∼100%	200%	∼0%	∼200%	0%
DWC - SW [Bolchini 2003; Hu et al. 2009]	∼100%	0%	28%–106%	0%	109%–217%
DWC opt. - SW [Mitropoulou et al. 2014]	∼100%	0%	29%	0%	100%–150%

is able to reduce the energy consumption when compared to regular task duplication, the overhead area and power is still huge, as it needs an extra processor.

Also, some works that aim to increase the performance of VLIW processors may be used to complement the proposed techniques. For instance, Jones et al. [2006] propose to increase the ILP on VLIW processors via hardware accelerators. Even though the use of hardware accelerators often require code changing and recompilation, this approach is orthogonal and can be applied simultaneously to the techniques of this work. Therefore, hardware accelerators may be used along with the proposed techniques of this work on those phases in which the ILP can be improved without jeopardizing the fault tolerance. Adding extra hardware accelerators, naturally, would increase the area overhead.

Table III presents the comparison among the results from the proposed approaches and the other works previously discussed in this section. We consider error coverage, area, performance, power dissipation, and code size increase. As it can be noticed, the proposed approaches have the lowest area and power dissipation overheads when compared to other hardware-based techniques (in **bold**). Software-based techniques (in *italic*) naturally do not affect the area nor the power dissipation, but they create a performance overhead and increase the code size, both affecting total energy consumption of the system, as the application will take longer to execute and the memory will be more stressed.

5. CONCLUSIONS AND FUTURE WORK

In this work, fault tolerance mechanisms that exploit idle hardware and are based on duplication and instruction rollback are proposed, which are able to not only detect a fault, as conventional DMR approaches, but also correct the error by executing the faulty instruction again via rollback. The performance overhead that a rollback causes is negligible compared to the application's total number of cycles. In addition, these mechanisms are able to provide fault tolerance at a minimum cost, by using idle resources of the VLIW processor with low area and power overhead. Moreover, the ILP reduction approach can be used to improve the fault tolerance at the cost of performance when the application has high ILP.

Finally, we will evaluate the applicability of this approach in other configurations and processors: in the current processor configuration (1 branch, 1 memory unit, and 4 multipliers), we need to add one more branch and memory units to allow the execution of all duplicated instructions. In the simplest configuration of the current processor (1 branch, 1 memory, and 1 multiplier), the overhead of adding one more of those units (for duplication) would be of only 2%. Therefore, it is very likely that this approach may be applied to any VLIW configuration/processor, with more or less area overhead depending on the available functional units. This organization is similar to other commercial VLIW processors, for instance, the Intel Itanium [Sharangpani and Arora 2000] has 8 ALUs (4 integer and 4 MMX), 2 floating point multiply-add units, 2 memory units and 3 branch units; and the TMS320C6745 [Instruments 2011] from Texas Instruments has 6 ALUs, 2 multipliers, and 2 memory units. Therefore, as long as the functional units are symmetric, the duplication can be applied to any type of functional unit, in any VLIW processor.

As future work, we will consider other VLIW configurations (e.g., 2- and 4-issue versions) with different issue slot organizations, based on how critical each instruction is. In these cases, the bundle will proportionally be much more used than the ones from the 8-issue (the compiler will fill the bundles with more instructions than NOPs). Therefore, there will be less and different phases in the application, and a higher performance and fault tolerance variation when using the ILP reduction approach.

In addition, temporal redundancy techniques will be used to complement the spatial redundancy that is currently used. Temporal redundancy will potentially increase the rollback overhead and detection latency, as checkpoints will be needed to restore to an error-free state. However, this approach allows instructions to be compared with more flexibility (in different cycles); therefore, exploiting idle cycles that spatial redundancy is not able to benefit from. For instance, if a bundle has more empty slots than the number of program instructions, when applying the duplication, some of those will remain empty. Therefore, temporal redundancy allows the exploitation of these slots to improve fault tolerance by executing duplicated instructions from a previous bundle, instead of only duplicating instructions within the same bundle. A dynamic approach for dynamically detecting phase changes [Guo et al. 2016] will also be evaluated to be integrated to the fault tolerance approach.

REFERENCES

Shail Aditya, Scott A. Mahlke, and B. Ramakrishna Rau. 2000. Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats. *ACM Trans. Des. Autom. Electron. Syst.* 5, 4, 2000, 752–773.

Fakhar Anjam and Stephan Wong. 2013. Configurable fault-tolerance for a configurable VLIW processor. In *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 167–178.

Todd M. Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*. 196–207.

Antonio Carlos Schneider Beck, Carlos Arthur Lang Lisbôa, and Luigi Carro. 2012. *Adaptable Embedded Systems*. Springer Science & Business Media.

Cristiana Bolchini. 2003. A software methodology for detecting hardware faults in VLIW data paths. *IEEE Trans. Reliab.* 52, 4, 2003, 458–468.

Anthony Brandon, et al. 2015. A Sparse VLIW instruction encoding scheme compatible with generic binaries. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*.

Yung-Yuan Chen and Kuen-Long Leu. 2010. Reliable data path design of VLIW processor cores with comprehensive error-coverage assessment. *Microprocess. Microsyst.* 34, 1, 2010, 49–61.

Robert P. Colwell, John O'donnell, David B. Papworth, and Paul K. Rodman. 1991. Instruction storage method with a compressed format using a mask word, U.S. Patent 5057837.

Thomas M. Conte, Sanjeev Banerjia, Sergei Y. Larin, Kishore N. Menezes, and Sumedh W. Sathaye. 1996. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In. *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO-29)*. 201–211.

Jos T. J. van Eijndhoven, et al. 1999. TriMedia CPU64 architecture. In *Proceedings of the International Conference on Computer Design (ICCD'99)*. 586–592.

Joseph A. Fisher, Paolo Faraboschi, and Clifford Young. 2005. *Embedded Computing: a VLIW Approach to Architecture, Compilers and Tools*. Elsevier.

Jiri Gaisler. 1997. Evaluation of a 32-bit microprocessor with built-in concurrent error-detection. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS-27)*. 42–46.

J. S. P. Giraldo, A. L. Sartor, L. Carro, Stephan Wong, and A. C. S. Beck. 2015. Evaluation of energy savings on a VLIW processor through dynamic issue-width adaptation. In *Proceedings of the International Symposium on Rapid System Prototyping (RSP'15)*. 11–17.

Qi. Guo, Anderson Sartor, Anthony Brandon, Antonio C. S. Beck, Xuehai Zhou, and Stephan Wong. 2016. Run-time phase prediction for a reconfigurable VLIW processor. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1634–1639.

Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The M{ä}lardalen WCET Benchmarks: Past, Present And Future. *WCET* 15, 2010, 136–146.

Jie Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, Narayanan Vijaykrishnan, and Mary J. Irwin. 2009. Compiler-assisted soft error detection under performance and energy constraints in embedded systems. *ACM Trans. Embed. Comput. Syst.* 8, 4, 2009, 27.

Jie S. Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, Narayanan Vijaykrishnan, and Mary J. Irwin. 2005. Compiler-directed instruction duplication for soft error detection. In *Design, Automation and Test in Europe (DATE)*. 1056–1057.

Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. 2004. Microarchitectural techniques for power gating of execution units. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 32–37.

Boris Hubener, Gregor Sievers, Thorsten Jungeblut, Mario Porrmann, and Ulrich Ruckert. 2014. CoreVA: A Configurable Resource-Efficient VLIW Processor Architecture. In *Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC'14)*. 9–16.

Texas Instruments. 2011. TMS320C6745/C6747 DSP technical reference manual. SPRUH91A, Texas Instruments Inc.

Adam Jacobs, Grzegorz Cieslewski, Alan D. George, Ann Gordon-Ross, and Herman Lam. 2012. Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing. *ACM Trans. Reconfigurable Technol. Syst.* 5, 4, 2012, 21.

Sunghyun Jee and Kannappan Palaniappan. 2002. Performance evaluation for a compressed-VLIW processor. In *Proceedings of the ACM Symposium on Applied Computing*. 913–917.

Alex K. Jones, Raymond Hoare, Darag Kusic, Justin Stander, Gayatri Mehta, and Josh Fazekas. 2006. A vliw processor with hardware functions: Increasing performance while reducing power. *IEEE Trans. Circuits Syst. II Express Briefs* 53, 11, 2006, 1250–1254.

Cameron McNairy and Rohit Bhatia. 2005. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro* 2, 2005, 10–20.

Bryan Mills, Taieb Znati, and Rami Melhem. 2014. Shadow computing: An energy-aware fault tolerant computing model. In *Proceedings of the 2014 International Conference on Computing, Networking and Communications (ICNC'14)*. 73–77.

Konstantina Mitropoulou, Vasileios Porpodas, and Marcelo Cintra. 2014. DRIFT: Decoupled compiler-based instruction-level fault-tolerance. In *Languages and Compilers for Parallel Computing*. Springer, 217–233.

Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 29.

Nithin Nakka, Karthik Pattabiraman, and Ravishankar Iyer. 2007. Processor-level selective replication. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 544–553.

Prasad A. Raje and Stuart C. Siu. 1999. Method and apparatus for sequencing and decoding variable length instructions with an instruction boundary marker within each instruction. 1999.

Joydeep Ray, James C. Hoe, and Babak Falsafi. 2001. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. 214–224.

George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*. 243–254.

E. Sanchez and M. S. Reorda. 2015. On the functional test of branch prediction units. *IEEE Trans. Very Large Scale Integr. Syst.* 23, 9, 2015, 1675–1688. DOI: http://dx.doi.org/10.1109/TVLSI.2014.2356612

Anderson L. Sartor, Arthur F. Lorenzon, Luigi Carro, Fernanda Kastensmidt, Stephan Wong, and Antonio Beck. 2015. A novel phase-based low overhead fault tolerance approach for VLIW Processors. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'15)*. 485–490.

Anderson Luiz Sartor, Stephan Wong, and Antonio Carlos Schneider Beck. 2016. Adaptive ILP control to increase fault tolerance for VLIW processors. In *Proceedings of the IEEE 27th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE.

Mario Schölzel. 2007. Reduced triple modular redundancy for built-in self-repair in VLIW-processors. In *Signal Processing Algorithms, Architectures, Arrangements and Applications*. 21–26.

Harsh Sharangpani and Ken Arora. 2000. Itanium processor microarchitecture. *IEEE Micro* 20, 5, 2000, 24–43.

P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference Dependable Systems and Networks (DSN'02)*. 389–398. DOI: http://dx.doi.org/10.1109/DSN.2002.1028924

Timothy J. Slegel, et al.1999. IBM's S/390 G5 microprocessor design. *IEEE Micro* 19, 2, 1999, 12–23.

Pramod Subramanyan, Virendra Singh, Kewal K. Saluja, and Erik Larsson. 2010. Energy-efficient fault tolerance in chip multiprocessors using critical value forwarding. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10)*. 121–130.

Atsuhiro Suga and Kunihiko Matsunami. 2000. Introducing the FR500 embedded microprocessor. *IEEE Micro* 20, 4, 2000, 21–27.

Jingweijia Tan and Xin Fu. 2012. RISE: improving the streaming processors reliability against soft errors in gpgpus. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. 191–200.

Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. 2000. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro* 6, 2000, 12–25.

Carlos Villalpando, David Rennels, Raphael Some, and Manuel Cabanas-Holmen. 2011. Reliable multicore processors for NASA space missions. In *Proceedings of the 2011 IEEE Aerospace Conference*. 1–12.

Jan-Willem de Waerdt et al. 2005. The TM3270 media-processor. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. 331–342.

Stephan Wong, Thijs Van As, and Geoffrey Brown. 2008. ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In *Proceedings of the International Conference on ICECE Technology*. 369–372.

Ren Xiaoguang, Xu Xinhai, Wang Qian, Chen Juan, Wang Miao, and Yang Xuejun. 2015. GS-DMR: Low-overhead soft error detection scheme for stencil-based computation. *Parallel Comput.* 41, 2015, 50–65.

J. M. Yang and S. W. Kwak. 2010. A checkpoint scheme with task duplication considering transient and permanent faults. In *Proceedings of the IEEE International Conference on Industrial Engineering and Engineering Management (IEEM'10)*. 606–610.

Ying Zhang and Krishnendu Chakrabarty. 2004. Dynamic adaptation for fault tolerance and power management in embedded real-time systems. *ACM Trans. Embed. Comput. Syst.* 3, 2, 2004, 336–360.

Received September 2015; revised August 2016; accepted September 2016