Adaptive ILP Control to increase Fault Tolerance for VLIW Processors

Anderson L. Sartor[1], Stephan Wong[2], Antonio C. S. Beck[1]
[1] Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS), Brazil
{alsartor, caco}@inf.ufrgs.br
[2] Computer Engineering Laboratory, Faculty of EEMCS, Delft University of Technology (TU Delft), The Netherlands
j.s.s.m.wong@tudelft.nl

***Abstract*— Because of technology scaling, soft error rate has been increasing in modern processors, affecting system reliability. To mitigate such effect, we propose an adaptive fault tolerance approach that exploits, at run-time, idle functional units to execute duplicated instructions in a configurable VLIW processor. In applications with high Instruction Level Parallelism (ILP) and few functional units available for duplication, it adaptively reschedules instructions according to a configurable threshold, providing a tradeoff between performance and fault tolerance. On average, failure rate is reduced by 89.53%, performance by 5.86%; while energy consumption increases by 72% and area by 22.2%, using a fault tolerance oriented threshold.**

Keywords*— Fault tolerance; VLIW; soft errors; adaptive processor.

I. 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 and so they get more susceptible to soft errors [1]. 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 ground-level. To harden the processor against such errors, fault-tolerant techniques are mandatory to detect and correct their effects before a failure in the system is observed [2].

Very Long Instruction Word (VLIW) processors are representative examples of current architectures that may suffer from the aforementioned issues (e.g.: Intel Itanium [3] and Trimedia CPU64 [4]). 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 from a bundle are executed in parallel. Because the process of scheduling instructions is statically done by a compiler, the hardware of a VLIW processor is much simpler: the instruction queue, reorder buffer, dependency-checking and many other hardware components are not needed. Therefore, VLIW processors occupy less area and dissipate less power when compared to traditional superscalar processors.

Even though VLIW processors are also used in space missions [5], we are aiming at 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. Therefore, we seek for the best tradeoff when it comes to area, performance, fault tolerance, and power dissipation. We focus only on the pipelanes of the VLIW processor (which occupy about 45% of the core total area), since the register file (which occupies the rest) can be protected with parity [6], [7] or error correction codes (ECC)[8].

In such processors, in several cases the compiler is not able to fill all the bundle with independent instructions [9], so they are completed with No Operations (NOPs). Even though several techniques have been proposed to avoid waste of instruction memory by not storing these NOPs (e.g.: compressed encoding for VLIW instructions [10], instruction template bits [11], and stop-bits [12]), the functional units of the issue-slot responsible for executing the NOP (whether it was removed from code or not) will still be idle.

Taking advantage of this fact, we propose an approach for detecting and correcting soft errors in VLIW issue-slots, exploiting these idle functional units to provide fault tolerance at a low cost. It is based on a modified dual modular redundancy (DMR) with an instruction rollback mechanism. In this case, idle functional units will execute duplicated instructions to improve fault tolerance, whenever there are free issue-slots at a given cycle. However, applications with high ILP will have a reduced number of NOPs, which would very likely reduce the opportunities for instruction duplication and might not deliver the necessary protection against faults.

Therefore, we extend the proposed technique by also attacking this issue. It allows the tuning of how much fault tolerance is needed for a given application by reducing the ILP at run time (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 long instructions 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 technique proposed by this work are presented in Section II. Next, we describe the implementation and show the results. For that, a fault injection campaign was performed in several benchmarks using different ILP thresholds. We evaluate error coverage, area, energy consumption, and performance. Section IV discusses related works and compare the proposed approaches with several others, considering many factors. Finally, Section V concludes this work and discusses future directions.

II. PROPOSED FAULT TOLERANCE TECHNIQUE

A. *Processor configuration*

The VLIW processor used in this work is the ρ-VEX softcore VLIW processor [13], 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 issue-slot (also called pipelane) may contain different functional units from the following set: ALU (always present), multiplier, memory, and branch units. In this work, it is used the 8-issue version, which is similar to other VLIW processors (e.g., Intel Itanium [3]). It has 8 ALUs, 4 multipliers, 2 memory units, and 2 branch units (1 branch and 1 memory unit only execute duplicated instructions, as it will be further explained later).

The issue-slots are able to execute both regular and duplicated instructions, and a checker compares the results (i.e., all output signals) of the pipelanes that are executing duplicated instructions (e.g., arithmetic operations, jump address of a branch, or the values of a memory operation are checked). The destination register, the register file's and memory's write enable signals are also compared.

B. *Rollback mechanism*

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 and, in case of an error, this stored PC overwrites the current PC (rolling back the execution). As the memory and register file were not 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 any sort of memory corruption. Once the rollback PC is loaded, the instruction corresponding to that PC is fetched again and the execution resumes from that point. Therefore, the process has a fixed cost of 5 cycles to refill the pipeline, which is negligible considering the total number of cycles of an application, and this cost is only paid in case of an error. Both the checkers and the rollback mechanism do not affect the critical path of the processor, as they operate in parallel to the rest of the processor. The memory and the register file are considered to be ECC protected.

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).

C. *Duplicate when possible (DWP)*

In this technique, the idle pipelanes are used to execute duplicated instructions whenever possible (i.e., when there are NOPs). Therefore, the verification is done on a per cycle basis. 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 (so it is replaced for a duplicated instruction). No additional accesses to the memory are required for both instructions and data: instruction words are fetched (1 access), then the bundles are divided into the pipelanes (applying duplication when possible); data accesses also access the memory once, the result is divided into the pipelanes with memory units. As the whole process is dynamic, this approach is completely transparent to the application.

This approach is depicted in Fig. 1. In order to keep the overhead low (area and delay), the duplication pairs are statically placed, i.e., pipelane 0 with pipelane 4, pipelane 1 with pipelane 5, and so on. Therefore, the issue-slots are combined in a way that the first four pipelanes are compared with the four last ones. For example, if the pipelane 6 was going to execute a NOP, then it will execute the duplicated instruction from the pipelane 2 instead. As the compiler used in this work (HP VEX compiler) always schedules the instructions starting from the lower issue-slots (from 0 to 7), our approach in combining pairs of pipelanes efficiently exploits this scheduling mechanism. Every functional unit is capable of executing both main program instructions and, in

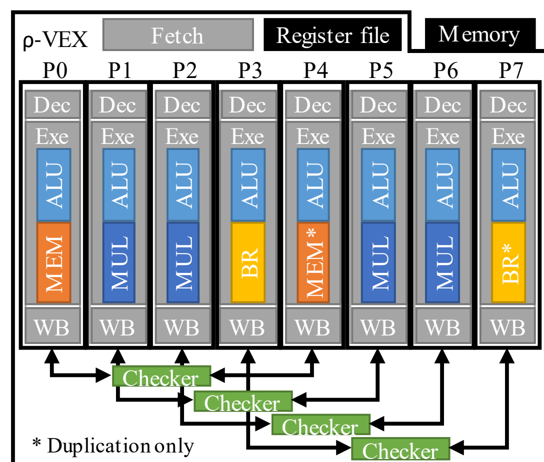

Fig. 1. DWP and Adaptive duplication design

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 the pipelane 7: these units execute only duplicated instructions, since the ρ-VEX does not support more than one memory or branch operations per cycle.

An example of code execution, comparing the original (unprotected) 8-issue version with the DWP approach, is presented in Fig. 2a and Fig. 2b. *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.

D. Adaptive duplication with ILP reduction

This approach is able to perform the tradeoff between performance and fault tolerance using an ILP threshold: 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. As previously explained, when the VLIW bundle has more than half of the issue-width filled with instructions, the duplication will not be full, as depicted in Fig. 2b at *t2*, *t3*, and *t5*. Hence, the ILP reduction mechanism is applied to allow increasing the fault tolerance for those cases in which the duplication is only partial. This process can be tuned by configuring the threshold that will activate the ILP reduction, offering a tradeoff 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 the 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 on those bundles that are not able to be fully duplicated without the ILP reduction. 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 no performance penalties.

Examples of code execution using different thresholds (1 and 2) are depicted Fig. 2c and Fig. 2d, respectively. The instructions that are broken into two cycles (allowing full duplication) are highlighted by the arrows on the right side of the instruction word. Instructions that write to a certain register in the first half of the instruction word and read from the same register on the second half will introduce a read after write hazard if it were to be split into two cycles, resulting in wrong computation [14]. These hazards are detected in hardware and these instructions are not duplicated, preserving program correction.

When the threshold is equal to 1, every bundle that has more 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 Fig. 2 again. As it can be observed, there is no performance overhead when the DWP is used (Fig. 2b). However, 8 instructions would not be duplicated. Using ILP reduction with threshold equal to 1 (Fig. 2c), we would have 50% of performance degradation with the ability to duplicate all instructions. If a threshold equal to 2 (Fig. 2d) is chosen,

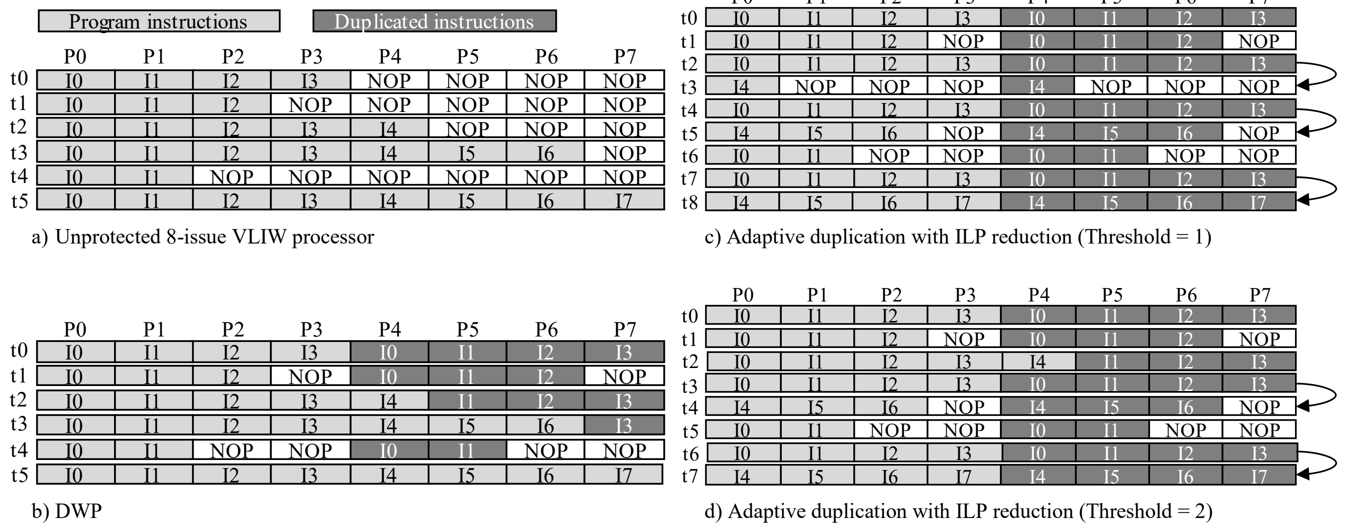

Fig. 2. Code execution example

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. In addition to these two proposed techniques, a phase-configurable duplication mechanism was implemented in [15], using the same processor. Therefore, exploiting idle program phases, instead of a per cycle verification.

III. Results

A. Methodology

The benchmark set is composed of a subset of 10 applications from the WCET benchmark suite [16]: ADPCM, CJPEG, CRC, DFT, Expint, FIR, Matrix Multiplication, NDES, Sums (recursively executes multiple additions on an array), and x264. The HP VEX compiler was chosen because it is more stable and robust than the GCC VEX compiler.

Scripts in TCL (Tool Command Language) were created to inject transient faults in all issue-slots and checkers of the processor. The script chooses a random bit from an arbitrary signal to be flipped at a random cycle during the execution of the application. The fault duration is one clock cycle, and one fault is injected per application's execution. The total number of injected faults was 5.5 million (so there was the same number of application executions).

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 RTL compiler to obtain power dissipation and ASIC (Application Specific Integrated Circuit) area, using a 65nm CMOS cell library from STMicroeletronics. The operation frequency was set to 65MHz for the FPGA and 200MHz for the ASIC. Both data and control flow failures are verified: the former, comparing the memory dump to the golden copy and the latter, with the number of executed cycles.

B. Failure rate and performance

Fig. 3 presents the performance degradation (Y axis) when varying the threshold (X axis) for ILP reduction. The degradation varies from zero to 27.25% with the threshold equal to 1 (i.e., the lowest value for the threshold). 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 (as it will be presented next).

Table I presents the failure rate and performance of the benchmarks for the following versions: *Unprotected* VLIW processor; using DWP; and adaptive duplication with ILP reduction, considering *Threshold = 1.75, 2,* or *2.5* (depending on the benchmark), and *Threshold = 1* (energy results will be discussed in the next subsection). Note that, in some benchmarks, results of the adaptive version are not shown for a Threshold greater than 1, since the failure rate does not decrease significantly. As expected, the DWP approach does not have any performance degradation. When threshold is used and its value decreases, fault tolerance increases together with the number of executed cycles. On average, the unprotected processor has a failure rate of 3.73%. When using DWP, it goes to 0.75%; and with the threshold equal to 1, 0.40% (the average for the other threshold values is not presented because the failure rate does not significantly vary for all the benchmarks).

Fig. 4 depicts the tradeoff between failure rate and performance of the adaptive approach with different thresholds when compared to the unprotected version. The *T* for the threshold. The failure rate reduction varies from 61.68% (DWP on CJPEG) to 97.09% (Matrix multiplication with T=1), while performance degradation reaches up to 27.25% (CJPEG 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 65.16% 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 in fault tolerance, which comes at the high cost of performance. For benchmarks

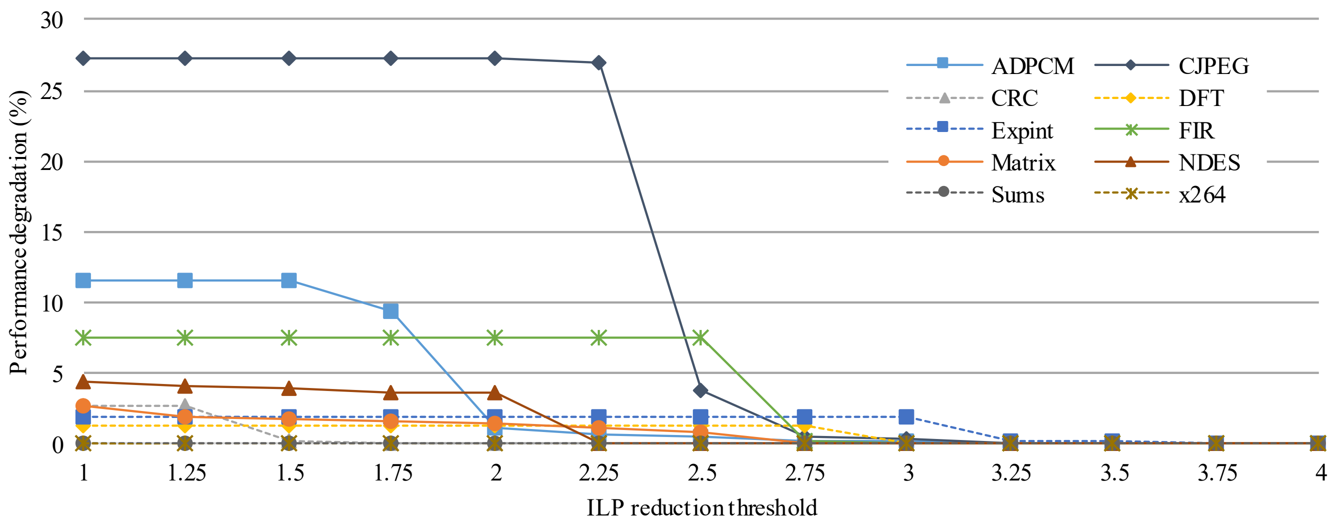

Fig. 3. Performance degradation when varying the threshold for ILP reduction

TABLE I. FAILURE RATE, PERFORMANCE, AND ENERGY CONSUMPTION COMPARISON

		Failure rate (%)	Execution cycles	Energy cons. (J)
ADPCM	Unprotected	3.66	568	6.05E-08
	DWP	0.66	568	8.54E-08
	Threshold=2	0.66	574	9.92E-08
	Threshold=1.75	0.65	621	1.08E-07
	Threshold=1	0.59	633	1.10E-07
CJPEG	Unprotected	6.07	411	5.13E-08
	DWP	2.33	411	6.66E-08
	Threshold=2.5	2.12	426	7.87E-08
	Threshold=1	0.79	523	9.67E-08
CRC	Unprotected	2.95	13,270	1.29E-06
	DWP	0.33	13,270	1.84E-06
	Threshold=1	0.32	13,616	2.20E-06
DFT	Unprotected	2.68	32,575	3.01E-06
	DWP	0.38	32,575	4.23E-06
	Threshold=1	0.15	32,979	5.04E-06
Expint	Unprotected	2.37	9,097	8.44E-07
	DWP	0.13	9,097	1.18E-06
	Threshold=1	0.13	9,257	1.42E-06
FIR	Unprotected	5.93	111,769	1.32E-05
	DWP	1.21	111,769	1.78E-05
	Threshold=1	0.93	120,095	2.19E-05
Matrix Multiplication	Unprotected	5.68	111,025	1.01E-05
	DWP	1.30	111,025	1.39E-05
	Threshold=2	0.53	112,547	1.68E-05
	Threshold=1	0.17	113,929	1.70E-05
NDES	Unprotected	2.09	27,499	2.63E-06
	DWP	0.42	27,499	3.69E-06
	Threshold=1	0.24	28,667	4.52E-06
Sums	Unprotected	2.96	319	2.77E-08
	DWP	0.37	319	3.87E-08
	Threshold=1	0.37	319	4.60E-08
x264	Unprotected	2.94	15,089	1.44E-06
	DWP	0.33	15,089	2.09E-06
	Threshold=1	0.33	15,090	2.43E-06

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% (DWP) to 97.09% (threshold=1).

C. *Area and energy consumption*

Table II presents the processor area for both FPGA and ASIC for all VLIW configurations. As it can be observed, the overhead of the DWP is almost negligible in terms of area when compared to the unprotected version. The area overhead is of 4.6% for the FPGA (given in LUTs) and 4.5% for the ASIC cells, which includes the extra functional units, the rollback mechanism, and the checkers. The overhead for the

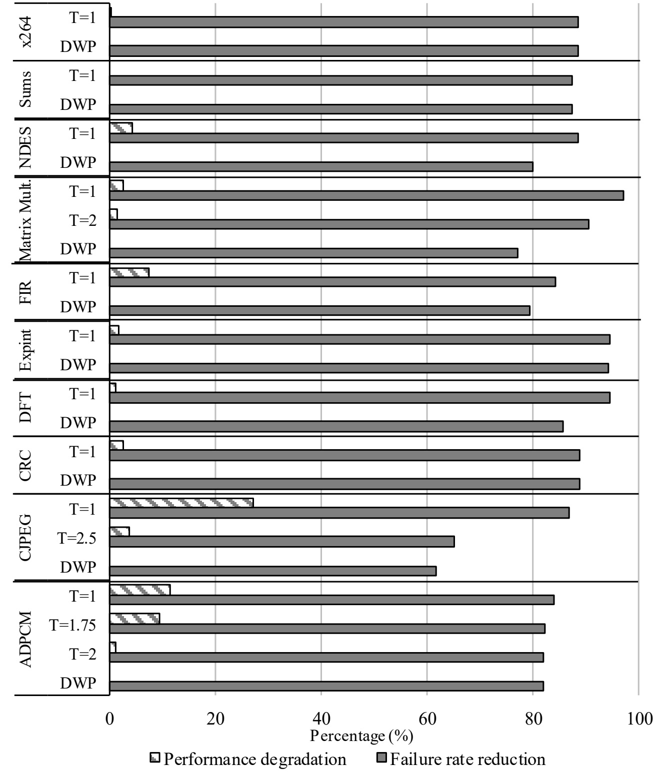

Fig. 4. Adaptive with ILP reduction compared to the unprotected version

TABLE II. AREA COMPARISON

	FPGA		ASIC
	Registers	LUTs	Cells
Unprotected 8-issue	3,974	35,075	65,281
DWP	4,206	36,672	68,241
Adaptive with ILP reduction	4,834	41,485	79,774

adaptive approach with ILP reduction is higher because of extra control circuitry, even though, it is lower than other techniques (as it will be discussed next), being 18.3% for the FPGA and 22.2% for the ASIC.

Assessing Table I again, now with the focus on the energy consumption for the ASIC version of the processor, two different switching activity levels (which influences dynamic power) were considered: activity of 30% (when executing an operation) and 0% (when idle). The switching activity of 30% was chosen because it is the traditionally assumed value for system level analysis of microprocessors [17]. During the VHDL simulations, it is verified whether the functional unit is used or not at each cycle. This simplified model was used due to the complexity of measuring the real switching activity of each part of the circuit, given the very significant simulation times.

On average, the DWP and adaptive versions consume 39% and 72% more energy than the unprotected version, respectively. They consume more energy mainly due to the

execution of duplicated instructions (there will be more active issue-slots). The second factor is the additional circuitry, which implies in 8.3% extra power dissipation for the DWP version and 21.7% for the adaptive. Note that the energy consumption overhead is for the processor only. If the whole system were to be considered (i.e., memory and other components), the resulting overhead would be much lower, as memories consume a significant amount of the energy.

IV. 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 the authors in [18] and [19] 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 of 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 always compares the results and executes again only the instruction word that presented the error. Moreover, 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 [20] and [21]. In these cases, they only triplicate the functional units of a VLIW processor rather than the entire processor; reducing area and power dissipation costs. In [20], the authors proposed the Reduced TMR, in which both hardware and software needed to be changed. If 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. Hence, errors that may occur before or after the execution stage are not detected. The proposed approach occupies less area and dissipates less power than [20] and [21], and does not change the binary code of the application, as it is done in [20].

In [22], the authors propose a similar approach to [20]. 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). In [23] the authors proposes a TMR approach on the synchronous flip-flops. However, the area and energy consumption overheads are higher than the proposed technique.

In [24] and [25], the authors 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. Authors in [26] 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.

Exploiting idle streaming processors on GPGPUs by executing replicated warps is proposed by [27]. 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, power dissipation with little or no performance overhead. The approach proposed in this paper, even though implemented in hardware, have low overhead in area (22.2%), when compared to other techniques.

Adaptive fault tolerance:

Some works exploit the previous techniques in order to provide an adaptive fault tolerance mechanism. In [28], the authors 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: triple modular redundancy, duplication with comparison, algorithm-based fault tolerance (ABFT), internal TMR, and high-performance (no fault tolerance). This approach is exclusive for FPGAs, as the hardware needs to be reconfigured when changing techniques. On the other hand, the proposed approach can be used on both FPGAs and ASICs.

An adaptive checkpoint mechanism was proposed in [29], in which the checkpointing interval is adjusted during the execution based on the occurrence of faults and the available slack. In order to determine the parameters that are provided to the online checkpointing procedure, an offline preprocessing based on linear programming is used. Even though the checkpointing is made adaptively, there is still latency for the error detection. On the other hand, the proposed approach has zero latency error detection and it is completely dynamic, and does not need any preprocessing. In [30], the authors replicate a task and execute the replicated task in a processor with lower processor speed, to save energy. If the main task completes successfully, the duplicated one is terminated; otherwise, the duplicated task takes over to complete the computation, possibly at an increased processor frequency. Although this technique is able to reduce the energy consumption when compared to regular task duplication, the overhead in area and power is still huge, as it needs two processors.

Also, some works that aim to increase the performance of VLIW processors may be used to complement the proposed techniques. For instance, [31] 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.

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, this approach may be applied to any VLIW configuration, with more or less area overhead depending on the available functional units. Moreover, the duplication mechanism could be modified to work with other compilers and VLIW processors, as well as to be extended to superscalar architectures. In all these cases, as the application's code is not modified, only the fetch unit would need modification to dynamically reschedule the instructions to the functional units, which is applicable to both VLIW and superscalar architectures.

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, energy consumption and code size increase. As it can be noticed, the proposed approaches have the lowest area overhead and energy consumption 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.

V. 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. Moreover, the tradeoff between fault tolerance and performance may be exploited by using the ILP reduction mechanism.

As future work, we will consider the implementation of the proposed technique in a superscalar processor and other VLIW configurations (e.g., 2- and 4-issue versions) with different issue-slot organizations. In these cases, the bundle will proportionally be much more used than the one from the 8-issue (the compiler will fill the bundles with more instructions than NOPs), so the impact of the adaptive ILP approach will me even more evident. Also, the support for dynamic adjustment of the threshold will be added.

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.

VI. References

[1] 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," *Proc. Int. Conf. Dependable Syst. Networks*, pp. 389–398, 2002.

[2] A. C. S. Beck, C. A. L. Lisbôa, and L. Carro, *Adaptable embedded systems*. Springer Science & Business Media, 2012.

[3] H. Sharangpani and K. Arora, "Itanium processor microarchitecture," *Micro, IEEE*, vol. 20, no. 5, pp. 24–43, 2000.

[4] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E.-J. D. Pol, M. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken, "TriMedia CPU64 architecture," in *Computer Design, 1999.(ICCD'99) International Conference on*, 1999, pp. 586–592.

[5] C. Villalpando, D. Rennels, R. Some, and M. Cabanas-Holmen, "Reliable multicore processors for NASA space missions," in *Aerospace Conference, 2011 IEEE*, 2011, pp. 1–12.

[6] J. Gaisler, "Evaluation of a 32-bit microprocessor with built-in concurrent error-detection," in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, 1997, pp. 42–46.

[7] C. McNairy and R. Bhatia, "Montecito: A dual-core, dual-thread Itanium

Table III. VLIW fault tolerance techniques comparison

Technique	Error Coverage	Area overhead	Performance degradation	Energy consumption overhead	Code size increase
DWP	~100%	4.5%	~0%	30-45%	0%
Adaptive with ILP reduction	~100%	22.2%	~0-27.25%	66-88%	0%
DMR with rollback [18] *and* [19]	~100%	0%	51%-100%	>0% (not available)	100%
TMR	~100%	200%	~0%	~200%	0%
Partial TMR [21]	95%-99%	100%	0.6%-34.3%	>100%	0%
Reduced TMR [20]	~100%	100%	0%-100%	>100%	> 0%
Reduced TMR - SW [22]	~100%	0%	30%-60%	>0% (not available)	100%
Flip-flops TMR [23]	~100%	200%	~0%	~200%	0%
DWC - SW [24] *and* [25]	~100%	0%	28%-106%	>0% (not available)	109-217%
DWC opt. - SW [26]	~100%	0%	29%	>0% (not available)	100-150%

processor," *IEEE micro*, no. 2, pp. 10–20, 2005.

[8] T. J. Slegel, R. M. Averill III, M. Check, B. C. Giamei, B. W. Krumm, C. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, and others, "IBM's S/390 G5 microprocessor design," *Micro, IEEE*, vol. 19, no. 2, pp. 12–23, 1999.

[9] S. Aditya, S. A. Mahlke, and B. R. Rau, "Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 4, pp. 752–773, 2000.

[10] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, "Instruction fetch mechanisms for VLIW architectures with compressed encodings," in *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, 1996, pp. 201–211.

[11] J.-W. de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.-P. van Itegem, D. Amirtharaj, K. Kalra, and others, "The TM3270 media-processor," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005, pp. 331–342.

[12] B. Hubener, G. Sievers, T. Jungeblut, M. Porrmann, and U. Ruckert, "CoreVA: A Configurable Resource-Efficient VLIW Processor Architecture," in *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, 2014, pp. 9–16.

[13] S. Wong, T. Van As, and G. Brown, "ρ-VEX: A reconfigurable and extensible softcore VLIW processor," in *ICECE Technology, 2008. FPT 2008. International Conference on*, 2008, pp. 369–372.

[14] A. Brandon and S. Wong, "Support for dynamic issue width in VLIW processors using generic binaries," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013, pp. 827–832.

[15] A. L. Sartor, A. F. Lorenzon, L. Carro, F. Kastensmidt, S. Wong, and A. Beck, "A Novel Phase-Based Low Overhead Fault Tolerance Approach for VLIW Processors," in *VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on*, 2015, pp. 485–490.

[16] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The M{ä}lardalen WCET Benchmarks: Past, Present And Future.," *WCET*, vol. 15, pp. 136–146, 2010.

[17] B. Geuskens and K. Rose, *Modeling microprocessor performance*. Springer Science & Business Media, 2012.

[18] R. Xiaoguang, X. Xinhai, W. Qian, C. Juan, W. Miao, and Y. Xuejun, "GS-DMR: Low-overhead soft error detection scheme for stencil-based computation," *Parallel Comput.*, vol. 41, pp. 50–65, 2015.

[19] J.-M. Yang and S. W. Kwak, "A checkpoint scheme with task duplication considering transient and permanent faults," in *Industrial Engineering and Engineering Management (IEEM), 2010 IEEE International Conference on*, 2010, pp. 606–610.

[20] M. Schölzel, "Reduced Triple Modular redundancy for built-in self-repair in VLIW-processors," in *Signal Processing Algorithms, Architectures, Arrangements and Applications, 2007*, 2007, pp. 21–26.

[21] Y.-Y. Chen and K.-L. Leu, "Reliable data path design of VLIW processor cores with comprehensive error-coverage assessment," *Microprocess. Microsyst.*, vol. 34, no. 1, pp. 49–61, 2010.

[22] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-directed instruction duplication for soft error detection," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*, 2005, pp. 1056–1057.

[23] F. Anjam and S. Wong, "Configurable fault-tolerance for a configurable VLIW processor," in *Reconfigurable Computing: Architectures, Tools and Applications*, Springer, 2013, pp. 167–178.

[24] C. Bolchini, "A software methodology for detecting hardware faults in VLIW data paths," *Reliab. IEEE Trans.*, vol. 52, no. 4, pp. 458–468, 2003.

[25] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-assisted soft error detection under performance and energy constraints in embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 4, p. 27, 2009.

[26] K. Mitropoulou, V. Porpodas, and M. Cintra, "DRIFT: Decoupled CompileR-Based Instruction-Level Fault-Tolerance," in *Languages and Compilers for Parallel Computing*, Springer, 2014, pp. 217–233.

[27] J. Tan and X. Fu, "RISE: improving the streaming processors reliability against soft errors in gpgpus," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 191–200.

[28] A. Jacobs, G. Cieslewski, A. D. George, A. Gordon-Ross, and H. Lam, "Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 5, no. 4, p. 21, 2012.

[29] Y. Zhang and K. Chakrabarty, "Dynamic adaptation for fault tolerance and power management in embedded real-time systems," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 2, pp. 336–360, 2004.

[30] B. Mills, T. Znati, and R. Melhem, "Shadow computing: An energy-aware fault tolerant computing model," in *Computing, Networking and Communications (ICNC), 2014 International Conference on*, 2014, pp. 73–77.

[31] A. K. Jones, R. Hoare, D. Kusic, J. Stander, G. Mehta, and J. Fazekas, "A vliw processor with hardware functions: Increasing performance while reducing power," *Circuits Syst. II Express Briefs, IEEE Trans.*, vol. 53, no. 11, pp. 1250–1254, 2006.