Run-Time Instruction Replication for Permanent and Soft Error Mitigation in VLIW Processors

Rafail Psiakis, Angeliki Kritikakou and Olivier Sentieys
University of Rennes 1 - IRISA/INRIA, Rennes, France
rafail.psiakis@inria.fr

Abstract—**Error occurrence in embedded systems has significantly increased. Although inherent resource redundancy exist in processors, such as in Very Long Instruction Word (VLIW) processors, it is not always used due to low application's Instruction Level Parallelism (ILP). Approaches benefit the additional resources to provide fault tolerance. When permanent and soft errors coexist, spare units have to be used or the executed program has to be modified through self-repair or by using several stored versions. However, these solutions introduce high area overhead for the additional resources, time overhead for the execution of the repair algorithm and storage overhead of the multiversioning. To address these limitations, a hardware mechanism is proposed which at run-time replicates the instructions and schedules them at the idle slots considering the resource constraints. If a resource becomes faulty, the proposed approach efficiently rebinds both the original and replicated instructions during execution. In this way, the area overhead is reduced, as no spare resources are used, whereas time and storage overhead are not required. Results show up to 49% performance gain over existing techniques.**

I. Introduction

The reliability requirements of embedded systems are highly increased due to both software and hardware characteristics [1], such as critical applications, harsh operation environments, transistor decreasing size or low supply voltage. To provide correct system functionality, the system has to be enhanced with fault tolerant techniques for both temporary errors, caused by heat, radiation, electromagnetic interference, etc., and permanent errors, caused by circuit manufacturing or ageing. Several fault tolerance techniques use either hardware or software redundancy. Hardware redundancy inserts additional resources to the system in order to execute several times the instructions [2] providing performance almost equal to the unprotected version, but in cost of area overhead. Software redundancy executes the replicated instructions on the same resources increasing the execution time [3].

Several embedded processors are designed with a high number of parallel Function Units (FUs) providing inherent resource redundancy. Due to the restricted ILP of most applications, several of these resources remain idle during the application execution, so they could be exploited to improve software redundancy techniques. Such approaches can be implemented either in software or in hardware. Software-based approaches replicate and schedule the instructions at design-time [3] and thus increase the code size, the storage needs and the power consumption compared with hardware approaches, which perform the duplication at run-time [4]. However, when permanent errors also exist, these approaches cannot be applied as they do not modify the execution of the program to exclude faulty parts. To deal with permanent errors, either the hardware has to be reconfigured using spare units [5] or the executed program has to be modified in order to avoid the use of faulty units. The use of spare units highly increases the area overhead for both the FUs and the control. The modification of the executed program is performed using software by re-writing the program in the memory [6] or by using multiple different versions of the program [7]. In the first case, time overhead is introduced, whereas in the second case the storage requirements are highly increased. A simple hardware approach has been combined with a software repair routine in [6], which sequentially schedules the instructions that cannot be scheduled to other slots, whereas the detection of the permanent errors has been performed up-front.

In this work, we propose a hardware-based approach for both single and multiple, permanent and soft errors for heterogeneous statically scheduled data paths. Our technique removes time and storage overhead and the need for spare units. At run-time, it replicates and binds the instructions to provide error detection and mitigation. If permanent errors are detected, an efficient instruction binding is applied which modifies at run-time the executed program. Both instruction replication and binding explore the idle VLIW slots taking into account the limitations on the type and the number of resources. To support our contribution, we perform a set of experiments for a VLIW processor with heterogeneous slots. Duplication of instructions for error detection and triplication for error mitigation are used by our approach in combination with a number of possible permanent errors. Results show up to 49% performance gain over existing techniques.

The remaining of this paper is organized as follows: Section II describes the proposed technique, Section III presents the experimental results, whereas Section IV discusses the related work. Section V gives a conclusion of this work.

II. Run-Time Instruction Replication and Binding

The target domain of the proposed approach is VLIW processors, where a number of instructions is formatted as one big instruction, named *instruction bundle*, which is issued in parallel to the pipelined FUs of the processor. The proposed approach takes advantage of the idle issue slots of VLIW instruction bundles to: 1) execute original and replicated instructions in order to provide fault tolerance and 2) rebinding instructions in case of permanent errors. It can be combined

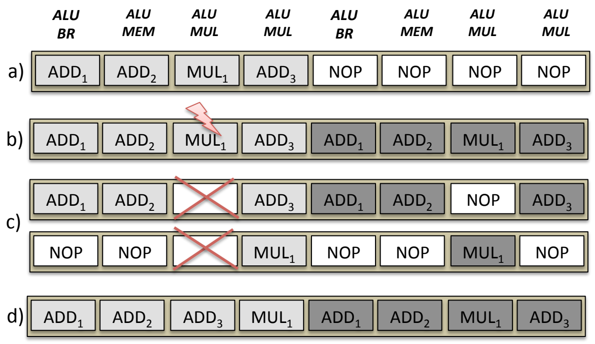
ALU BR
ALU MEM
ALU MUL
ALU MUL
ALU BR
ALU MEM
ALU MUL
ALU MUL
a) ADD1 ADD2 MUL1 ADD3 NOP NOP NOP NOP
b) ADD1 ADD2 MUL1 ADD3 ADD1 ADD2 MUL1 ADD3
c) ADD1 ADD2 ADD3 ADD1 ADD2 NOP ADD3
NOP NOP MUL1 NOP NOP MUL1 NOP
d) ADD1 ADD2 ADD3 MUL1 ADD1 ADD2 MUL1 ADD3

Fig. 1: Illustration of the proposed approach

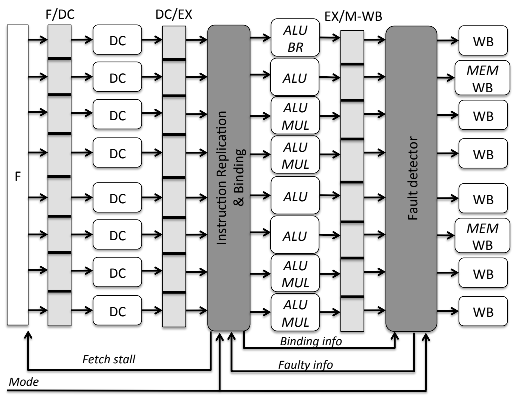

Fig. 2: Hardware components inserted in the VLIW pipeline.

with several fault tolerant techniques, i.e. duplication and triplication of instructions supporting error detection and mitigation, and it is applicable for any VLIW structure, i.e. any issue width and number and type of FUs.

Fig. 1 illustrates the output of the proposed approach for a simple case using duplication of instructions and one permanent error. It shows the scheduled operations for an instruction bundle and the available FUs for an 8-issue VLIW processor: 8 Arithmetic and Logic Units (ALUs), 4 Multipliers (MULs), 2 Memory operation units (MEMs) and 1 Branch unit (BR). The original instruction bundle is depicted in Fig. 1(a) and the instruction bundle with the duplicated instructions in Fig. 1(b). In case of a permanent error detected in the multiplication unit of the third slot, existing hardware techniques re-execute the instruction scheduled at the third slot to another FU of another time slot, as depicted in Fig. 1(c). To improve performance, the proposed approach explores at run-time the rebinding of original and replicated instructions to explore the existing FUs and the idle slots, as depicted Fig. 1(d). In this example, the instruction ADD_3 is moved to the third slot, whereas the instruction MUL_1 is moved to the fourth slot without adding a time slot. In case there is a need for an extra time slot, the instructions that fit in the first time slot are placed there and the remaining ones in the next time slot.

Fig. 2 depicts the two components added to the VLIW pipeline: the Instruction Replication and Binding (IRB) and the fault detector. The VLIW consists of four stages: Fetch (F), Decode (DC), Execute (EX) and Memory/Write Back (M/WB). The IRB takes the *decode stage result*, the *mode*, and the *faulty information* as input, and has the *binding info* and the *fetch stall* as output. The mode is defined by the designer and it defines which fault tolerance technique is implemented: i) duplication of instructions, ii) triplication of instructions, or iii) duplication and re-execution. Depending on the mode, the IRB duplicates or triplicates the instructions and binds them in the idle slots of the instruction bundle taking into account the limitations on the number and the type of resources. In case not enough idle slots or FUs exist, a new time slot is added by stalling the fetch of new instruction bundles. The stall of the fetch stage is performed by propagating the fetch stall command to the fetch stage. The faulty information is used by the IRB in case of detected errors. When duplication with re-execution is selected as mode and a temporary error occurs, the IRB stalls the pipeline and re-executes the faulty instruction in a different FU. If a permanent error is detected, the IRB updates the state of the FUs and explores the idle slots and the available FUs to bind efficiently the original and replicated instructions. The binding information is sent to the fault detector to inform how the instruction binding has been performed. The fault detector uses this binding information to decide which results are ready to be compared and committed. When an error is detected, it is initially assumed to be a temporary error. If a number of sequential instructions continue to indicate that the FU is faulty, then the fault detector decides that the error is permanent and sends this information to the IRB to update the status of the FUs. If the selected mode is triplication, the fault detector corrects the error and propagates the corrected value for commit.

III. Experimental Results

For the experimental part, we have used basic media benchmarks extracted from MediaBench [8] and the VEX VLIW processor [9] with HP VEX compiler. The VLIW is configured based on realistic configuration of resources used by commercial VLIWs, e.g. Intel Itanium [10], as depicted in Fig. 1 and Fig. 2. A simulation tool has been developed using Python to calculate the execution cycles of each application compiled with the HP VEX compiler. Intermediate files (.cs.c) from compiled simulator step are instrumented, linked and compiled with GCC in order to provide processor's execution instruction sequence traces. Our tool scans these traces to calculate the processor's execution cycles and, thus, estimate the performance of each approach. The area and power evaluation of our technique is under development. We perform experiments by applying two fault tolerance techniques with our approach: duplication and triplication of the instructions. We provide performance results for one up to five concurrent permanent errors occurring in any combination of the four different types of FUs of the VLIW. Each time, at least one non-faulty FU exists for each type of required FUs. Otherwise the processor is declared as ”out of service”, as it is not able to execute every instruction anymore.

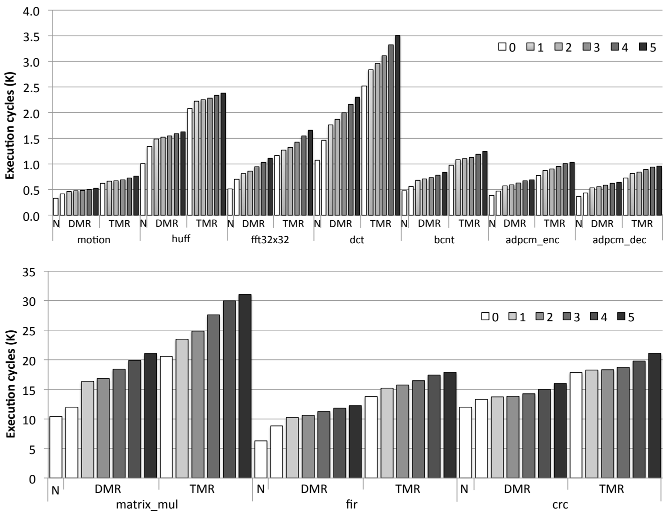

Fig. 3: Performance comparative results

Fig. 3 depicts the execution cycles of the unprotected original code (N) and the proposed approach that applies duplication (DMR) and triplication of the instructions (TMR). With our method, the overhead of DMR and TMR with $p = 0$ permanent errors is in most of the cases less than 100%, which is lower than the cycles expected when we execute twice or three times the original code (N). This occurs as the proposed approach efficiently explores the idle slots.

Table I depicts the impact of the multiple permanent errors in the performance of our approach implementing DMR and TMR. The performance overhead of DMR (resp. TMR) with $p = 1, \ldots, 5$ permanent errors is calculated relatively to DMR (resp. TMR) with no errors (p = 0). For the DMR, the performance overhead for *motion*, *huff* and *crc* remains quite small, up to 27%, even for 5 concurrent permanent errors. As these applications have a relatively small number of memory and multiplication instructions, the impact in performance is low when errors occur in MEM and MUL units. A similar overhead exists for most of the remaining benchmarks with up to 2 permanent errors. We observe that the concurrent permanent errors effect most the performance of *matrix_mul*, with a maximum overhead of 75% in the case of 5 errors. This overhead, in contrast to the first case, occurs because *matrix_mul* intensively uses the multiplication FUs, and, thus, the lack of these resources due to errors leads to high overhead. The last row shows the average overhead over the benchmarks.

Existing hardware approaches are applicable for single permanent errors [11]. Extending them for multiple permanent errors means that the instructions scheduled on a faulty unit have to be re-executed to another unit by adding an extra time slot. Similarly, hybrid approaches re-execute the instruction that can not be assigned to a sequential slot [6]. The behavior of these approaches can be estimated by using twice the execution cycles of the fault tolerance technique without any permanent errors, one time for the performance of the technique and a second time for the effect of the re-execution of the instructions scheduled to faulty units. This provides a lower bound because the execution cycles used are computed without permanent errors and the multiplication by two implies that all the faulty instructions of a bundle can be executed in only one additional time slot. Therefore, these values compared with our technic's results give an estimation of the lowest gain of the proposed approach (see Table II).

For our approach using the DMR, we observe that for all the benchmarks we achieve a high performance gain even for 5 multiple concurrent permanent errors, as depicted in the left part of Table II. The highest gains have been observed for the *crc* benchmark, from 49% for one permanent error up to 40% for 5 permanent errors and the smallest gains for *matrix_mul* from 32% for 1 permanent error up to 12% for 5 permanent errors. In the case where our approach uses TMR, it has also achieved a high performance gain for all the benchmarks, as depicted in the right part of Table II. We observe that for up to 2 permanent errors we have high gains for all the benchmarks, whereas for the *matrix_mul* which has high ILP, the gains are slightly reduced for 3, 4 and 5 permanent errors. The gains of the TMR compared to the DMR are higher, even for the instructions with high ILP. This occurs because due to the triplication of the instructions, more time slots are required to be added, which also increases the number of the idle slots. As the proposed approach efficiently explores these idle slots and the available FUs, it provides higher gains.

By comparing also the TMR approach with the optimal best case, i.e. twice the time of the DMR without any permanent error occurring, we observe some noticeable gain values. For *motion*, *huff*, and *crc* we achieve the gains: 20%, 17% and 31% for 1 permanent error and 8%, 11% and 21% for 5 permanent errors. Due to the low ILP of some applications, several idle slots exist and the proposed approach efficiently explores them even in the case of triplicating instructions. As the ILP increases and more permanent errors occur, it is normal the gains, using TMR, to be reduced compared with DMR. For the benchmarks *fft*, *bcnt*, *adpcm_enc*, *adpcm_dec* and *fir*, the TMR of the proposed approach provides gains of 14% up to 3% for up to 3 permanent errors, whereas for the *dct* and *matrix_mul* the gain is only 2% for 1 permanent error.

IV. Related Work

To provide fault tolerance in processors with inherent resource redundancy, software-based and hardware-based techniques have been proposed to take advantage of the additional

TABLE I: Performance overhead (%)

Benchmark	DMR					TMR				
	1	2	3	4	5	1	2	3	4	5
motion	11	14	17	21	27	7	9	12	17	23
huff	11	14	15	19	21	7	8	10	12	14
fft	16	23	35	47	59	9	13	22	33	42
dct	21	28	37	48	58	13	17	23	32	39
bcnt	21	26	31	39	49	11	13	16	22	28
adpcm_enc	21	26	34	42	47	12	16	23	30	33
adpcm_dec	23	28	35	43	47	12	16	23	30	32
matrix_mul	36	40	54	66	75	14	21	34	46	51
fir	16	20	27	34	38	10	14	20	26	30
crc	3	4	7	12	20	2	3	5	11	18
average	18	22	29	37	44	10	13	19	26	31

TABLE II: Lower bound on performance gain (%)

Benchmark	DMR					TMR				
	1	2	3	4	5	1	2	3	4	5
motion	44	43	42	40	37	46	46	44	41	38
huff	45	43	42	41	39	47	46	45	44	43
fft	42	39	33	27	21	46	43	39	34	29
dct	40	36	32	26	21	44	41	38	34	30
bcnt	39	37	35	30	25	44	43	42	39	36
adpcm_enc	39	37	33	29	27	44	42	39	35	34
adpcm_dec	38	36	32	29	26	44	42	39	35	34
matrix_mul	32	30	23	17	12	43	40	33	27	24
fir	42	40	36	33	31	45	43	40	37	35
crc	49	48	46	44	40	49	49	48	45	41
average	41	39	35	31	28	45	43	41	37	34

resources. Software-based approaches replicate and schedule the instructions at design-time. In this way, no additional hardware control is required, but code size, storage and power consumption are increased. For instance, the compiler duplicates the operations and schedules them in different FUs of VLIW processors [3] or exploits the idle slots for soft errors [12]. Hardware-based approaches duplicate the instructions at run-time using specific hardware using the compiler's result. To do so, coupling of the VLIW pipelines is applied [4], [13]. When the duplicated instructions do not fit in the current bundle, an additional time slot is added. Combinations of software and hardware approaches also exist. In [11] the instruction duplication and scheduling is performed by the compiler, whereas the comparison of the instructions is performed by the hardware. In case of an error, re-execution takes place through a simple HW operation rebinding. However, the VLIW consists of homogeneous issue slots with FUs executing any type of operations, whereas single errors are considered. However, these approaches cannot be applied in the case of permanent errors, as they do not modify program's execution.

In case of permanent errors, either the hardware has to be reconfigured using spare units or the execution of the program has to be changed to avoid the faulty units. By using spare hardware resources, the area and control overhead are increased. The technique in [14] performs redundant re-execution of the instructions by adding a set of spare functional units in general purpose multiprocessors. In [15] a spare function unit is added for error detection in VLIW, whereas single errors of one type are considered. In [5], spare function units are inserted to support TMR and when not enough resources exist, the recovery is performed by re-execution of the faulty instruction. The modification of the execution of the program can be achieved either off-line in software or on-line in hardware. The software approach of [7] modifies the program memory, in case of an off-line detected permanent fault, stores several versions of the scheduling. In [16], [17], a coarse-grained L/U reconfiguration is proposed for a single permanent fault for each hardware class of ASICs based on band partitioning. The technique has been extended for multiple faults by assuming one at each band and each reconfiguration can isolate one faulty unit. The fault detection is assumed to be already done. Some approaches combine software and hardware implementations. In [6] a software repair routine modifies the instructions permanently in the memory. During start-up, a self-test takes place to identify the faulty slots. This information is used to change the schedule stored in the memory. If the repair routine fails, a simple hardware binding sequentially maps the instructions that cannot be assigned to other slots. In [18], the approach is extended to cover pipeline registers, the register ports and the bypass logic, and in [19] is combined with adaptive software-based self-test, assuming though that the permanent errors have been already detected.

V. Conclusion

VLIW processors provide hardware redundancy that is not always exploited by the application. The remaining idle slots can be used to execute replicated instructions for reliability purposes. A hardware technique is proposed to replicate and schedule the instructions at run-time, thus exploring the idle slots under constraints in the FU number and type. When permanent errors occur, less FUs are available and the proposed approach efficiently rebinds the original and the replicated instructions based on the available resources and idle slots.

References

[1] J. W. McPherson, "Reliability challenges for 45nm and beyond," in *DAC*, July 2006, pp. 176–181.

[2] J. Klecka, W. Bruckert, and R. Jardine, "Error self-checking and recovery using lock-step processor pair architecture," 2002.

[3] C. Bolchini and F. Salice, "A software methodology for detecting hardware faults in vliw data paths," in *DFT*, 2001, pp. 170–175.

[4] A. L. Sartor *et al.*, "Adaptive ilp control to increase fault tolerance for vliw processors," in *ASAP*, July 2016, pp. 9–16.

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

[6] M. Schölzel and S. Muller, "Combining hardware- and software-based self-repair methods for statically scheduled data paths," in *DFT*, Oct 2010, pp. 90–98.

[7] R. Karri *et al.*, "Computer aided design of fault-tolerant application specific programmable processors," *TC*, vol. 49, no. 11, pp. 1272–1284, Nov 2000.

[8] C. Lee *et al.*, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *MICRO*, Dec 1997, pp. 330–335.

[9] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[10] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *MICRO*, vol. 20, no. 5, pp. 24–43, 2000.

[11] M. Schölzel *et al.*, "Hw/sw co-detection of transient and permanent faults with fast recovery in statically scheduled data paths," in *DATE*, March 2010.

[12] J. S. Hu *et al.*, "Compiler-directed instruction duplication for soft error detection," in *DATE*, March 2005.

[13] A. L. Sartor *et al.*, "Exploiting idle hardware to provide low overhead fault tolerance for vliw processors," *JETC.*, vol. 13, no. 2, pp. 13:1–13:21, Jan. 2017.

[14] J. B. Nickel and A. K. Somani, "Reese: a method of soft error detection in microprocessors," in *DSN*, July 2001, pp. 401–410.

[15] Y.-Y. Chen *et al.*, "An integrated fault-tolerant design framework for vliw processors," in *DFT*, Nov 2003, pp. 555–562.

[16] W. Chan and A. Orailoglu, "High-level synthesis of gracefully degradable asics," in *EDTC*, Mar 1996, pp. 50–54.

[17] A. Orailoglu, "Microarchitectural synthesis of gracefully degradable, dynamically reconfigurable asics," in *DFT*, Oct 1996, pp. 112–117.

[18] M. Schölzel, "Fine-grained software-based self-repair of VLIW processors," in *DFT*, 2011, pp. 41–49.

[19] M. Schölzel *et al.*, "A comprehensive software-based self-test and self-repair method for statically scheduled superscalar processors," in *LATS*, April 2016, pp. 33–38.