Fine-Grained Hardware Mitigation for Multiple Long-Duration Transients on VLIW Function Units

Rafail Psiakis, Angeliki Kritikakou, Olivier Sentieys

▶ **To cite this version:**

Rafail Psiakis, Angeliki Kritikakou, Olivier Sentieys. Fine-Grained Hardware Mitigation for Multiple Long-Duration Transients on VLIW Function Units. DATE 2019 - 22nd IEEE/ACM Design, Automation and Test in Europe, Mar 2019, Florence, Italy. pp.976-979, 10.23919/DATE.2019.8714899 . hal-01941860

HAL Id: hal-01941860
https://inria.hal.science/hal-01941860

Submitted on 2 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fine-Grained Hardware Mitigation for Multiple Long-Duration Transients on VLIW Function Units

Rafail Psiakis* Angeliki Kritikakou* and Olivier Sentieys*
*University of Rennes 1 - IRISA/INRIA, Rennes, France
Email: {rafail.psiakis, angeliki.kritikakou, olivier.sentieys}@inria.fr

***Abstract*—Technology scaling makes hardware more susceptible to radiation, which can cause multiple transient faults with long duration. In these cases, the affected function unit is usually considered as faulty and is not further used. To reduce this performance degradation, the proposed hardware mechanism detects the faults that are still active during execution and reschedules the instructions to use the fault-free components of the affected function units. The results show multiple long-duration fault mitigation with low performance, area, and power overhead.**

I. Introduction

As technology size decreases, the hardware becomes more and more susceptible to radiation. Although multiple simultaneous faults have been neglected for a long time, [7], they can no longer be negligible for technologies of 130nm and beyond [3]. The system clock period and the duration of the transient faults do not scale uniformly [1]. The fault duration is proportionally increasing compared to clock period, and, thus, the occurring faults can last for several clock cycles [2]. These Long-Duration Transient (LDT) faults have a much higher probability of creating system failures. Approaches to deal with LDTs are mask-based, spatial redundancy and sensors. Mask-based approaches have massive performance degradation [4], as the execution is stopped until the two mask inputs become equal. Spatial redundancy inserts spare resources [5] having high area overhead.Built-In Current Sensors (BICS) monitor the induced transient currents to detect single and multiple LDTs with a significantly small cost [6]. However, the execution is stalled as long as the faults are active [7] inserting high performance overhead.

For better trade-off between area and performance overhead for fault tolerance, systems include several Function Units (FUs). Very Long Instruction Word (VLIW) processors usually consist of complex FUs able to execute all types of operations and simpler ones that cannot execute sophisticated operations, such multiplications and divisions. As the Instruction Level Parallelism (ILP) of the applications does not usually lead to full utilization of the FUs, the idle FUs can be used to mitigate faults occurring on the system. When not enough idle FUs exist, new time slots have to be added. However, the state-of-the-art approaches for VLIW processors focus on: 1) short duration transient faults and 2) permanent faults. As the first category assumes a fault duration smaller than the system clock cycle, these approaches do not apply the required restrictions during instruction execution to support LDTs [8]. The second category permanently excludes the faulty FUs, when a fault is detected. Approaches that detect the faulty units before execution cannot be applied for LDTs [10].

Few approaches detect the faulty FUs during execution, and, thus, they could be used for LDTs. However, they perform coarse-grained exploration, i.e. the faulty complex FUs can still be used as simple FUs, whereas the faulty part is permanently excluded for the rest of the execution [9].

To deal with LDTs, we propose a hardware mechanism to detect the active faults during execution and temporally exclude only the faulty components of the affected FUs for as long as it is necessary. Our main contributions are: i) the fine-grained micro-architectural solution that partitions an FU into components, where each component is enhanced with a BICS to identify occurring faults, ii) the online fine-grained instruction scheduling mechanism that reschedules the instructions onto the healthy FU components, and iii) an evaluation analysis of the proposed hardware mechanism.

The organization of this paper is as follows. Section II presents an overview of the proposed mitigation mechanism, Section III and Section IV present the architecture details of the proposed mechanism, Section V presents the experimental results and Section VI concludes this work.

II. Overview and Motivation Example

The target domain is VLIW processors. In the remaining sections, we use the 4-issue heterogeneous VLIW data-path of Fig. 2 to schematically illustrate our approach. The components in blue color correspond to the basic architecture, whereas we highlight the hardware components added or modified by our approach with yellow color. The VLIW consists of a 3-stage pipeline with Fetch (F), Decode (DC) and Execute/Memory-WriteBack (EX/MEM/WB). A number of instructions, named as *instruction bundle* is issued and executed in parallel by the FUs of the processor. The proposed approach focuses on LDT faults occurring in the arithmetic FUs, as they have the largest area footprint of the system combinatorial components based on our experiments in Table III. The faults in the storage components, e.g., register file, memory and pipeline registers, are assumed to be protected (e.g. using Error Correction Codes (ECC)).

We illustrate through an example the main idea of this work. Fig. 1-i depicts the original schedule of two consecutive instruction bundles, B_{k-1} and B_k, obtained by the compiler. Based on the instruction type, the instructions are assigned to different FU components, as depicted in Fig. 1-ii. Assume that,

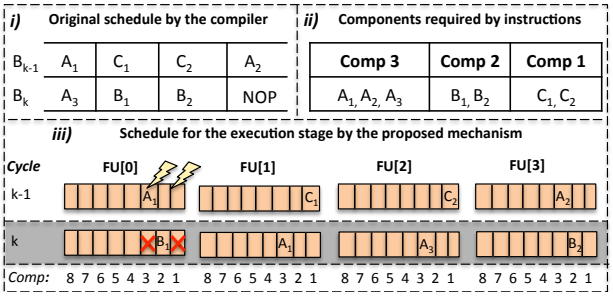

Fig. 1: Illustration example of the proposed mechanism

at cycle *k-1*, one fault that lasts at least two cycles affects the first and the third FU components of the first issue, FU[0], as depicted in Fig. 1-iii. The proposed mechanism decides the instruction rescheduling at cycle $k-1$ so as the instructions to be executed at cycle k. In the example of Fig. 1-iii, the scheduling of the B_{k-1} instructions is decided at cycle $k-2$. As no fault is detected during execution at cycle $k-2$, the VLIW executes the compiler's original schedule. During execution at cycle *k-1*, the mechanism detects two – just occurred – faults. To ensure a correct execution, the faulty B_{k-1} instructions must be re-executed at the next cycle avoiding the currently faulty FU components. Therefore, the A_1 instruction must be stored in order to be re-executed at cycle k. At cycle k, although the LDTs persist and the corresponding components have not yet recovered, the mechanism succeeds in executing the remaining instruction A_1 thanks to the re-scheduling of the B_k instructions. This action is allowed if the instruction A_1 is independent from the B_k instructions.

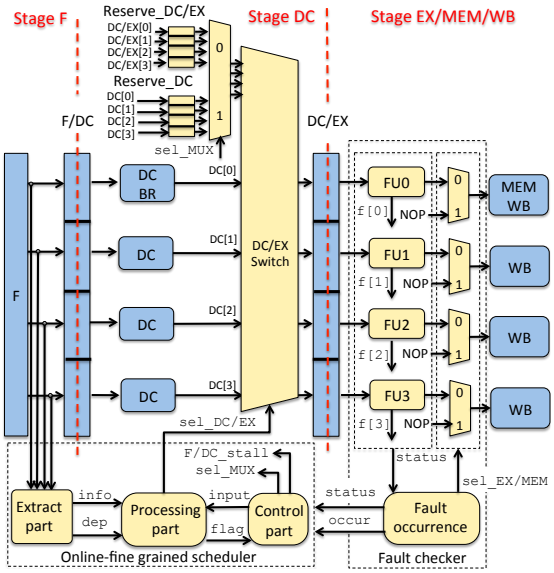

Fig. 2: VLIW enhanced with the proposed mechanism.

III. Fault Checker

The fault checker keeps the faulty status of the FU components, identifies new fault occurrences and takes care of miscalculated results. To achieve a fine-grained use of the components of an FU, each FU is internally enhanced with BICS. Both complex and simple FU types are analyzed in gate-level to identify the individual circuits. In our architecture, we considered a complex FU as a simple FU enhanced with a multiplication operator. The complex (simple) FU has 15 (14) different FU operations and 8 (7) individual circuits. The circuits are grouped based on the instruction opcode. For instance, the circuit that performs the addition of two registers (ADD operation) is partially shared with the circuit that calculates the address of a memory operation (MEM operation) and the circuit that performs ADDSHIFT operations. As they partially share the same execution path, they are grouped to the same individual component. Table I depicts the final obtained individual components for the complex FU. The individual components for the simple FU are the same without component 5. Each individual component and the final multiplexer (comp 0), which selects the result of the executed operation according to the opcode, is an FU component that is enhanced with a BICS sensor [7].

TABLE I: Component groups of a complex FU.

Comp 0	**Comp 1**	**Comp 2**
SELECT	MEM/ADD/ADDSHIFT	AND/NAND/ZEROEXT
Comp 3	**Comp 4**	**Comp 5**
OR/NOR/XOR	CMP/SUB	MUL
Comp 6	**Comp 7**	**Comp 8**
SRL	SRA	SLL

The output of each BICS sensor is combined into a fault status signal, signal `f`, with a size of 9 (8) bits for complex FU (simple FU). The `f` signals of each FU are combined to a global signal, `status`, with a size equal to the number of the VLIW issues. The signal `status` represents which components of the FUs are currently affected by a fault, if the corresponding bit is set. In case of one or more active faults at cycle $k-1$, the results of corresponding instructions – currently residing in the execution stage – are miscalculated, and, thus, they must not be committed. For this purpose, each VLIW issue is enhanced with a multiplexer controlled by the signal `sel_EX/Mem` (size equal to the number of VLIW issues) computed by the fault checker. When a bit in `sel_EX/Mem` is set, the corresponding multiplexer passes a NOP result (instead of the miscalculated result) and the WB and MEM enable of the corresponding issue is disabled. The fault checker stores the `status` signal at cycle $k-1$ to be compared with the `status` signal at cycle k. The comparison identifies the just occurred faults (one bit signal `occur`). Both `status` and `occur` signals pass to the online fine-grained scheduler to be used for the instruction re-scheduling decisions.

IV. Online Fine-Grained Scheduler

The instructions in the *F/DC* register are decoded at cycle $k-1$ and the scheduler at cycle $k-1$ decides the instructions to be executed at cycle k based on the `status` of the faulty FU components. The decoded instructions that couldn't be scheduled at cycle k, due to insufficient FUs or instruction dependencies, are stored to the *Reserve_DC* shadow register. At the same time instance, the EX/MEM/WB stage executes the instructions scheduled for execution at cycle $k-1$ (scheduling decision occurred at cycle $k-2$). The *Reserve_DC/EX* shadow register keeps the instructions executed at cycle $k-1$

in case a fault occurs during their execution. The instructions to be scheduled at cycle $k-1$ can potentially come from three sources: 1) the decoded instructions at cycle $k-1$ (*DC*) 2) the remaining instructions not scheduled at cycle $k-2$ (*Reserve_DC* register) and 3) the executed instructions at cycle $k-1$ (*Reserve_DC/EX* register).

In order to allow the scheduling of the instructions in different issues than the ones defined by the compiler's original schedule, a switch has to be inserted to the VLIW datapath. However, if the switch implemented all combinations between the three instruction inputs (*DC*, *Reserve_DC*, and *Reserve_DC/EX*) to the VLIW issues, the switch complexity would increase significantly. In contrast, the design of our online hardware mitigation mechanism reduces this overhead. A $2n$ to n switch, *DC/EX switch*, passes the instructions from one of the shadow registers and the decoded instructions *DC* to the main pipeline DC/EX register. A $2n$ to n multiplexer is used to decide which shadow register to be used as an input to the switch (signal `sel_MUX`).

The online fine-grained hardware scheduler is implemented by three components. The first one extracts the required information (`info` signal) from the F stage and calculates the dependencies between consecutive bundles (`dep` signal), similar to the resource and dependency analyzers of [8].

The second component is the scheduler processing part that schedules the input (one of the three potential instruction inputs) to the output of the *DC/EX switch* using bit masks, called IDentifiers (IDs) and taking into account the `status` of the faulty components. Each potential instruction input to the *DC/EX switch* is represented by a table (*Res_DC/EX_ID*, *Res_DC_ID* and *DC_ID*) that has a size equal to the number of VLIW issues. Each table element is an ID that corresponds to the instruction scheduled at position i either by the compiler (*DC_ID[i]* and *Res_DC_ID[i]*) or by the proposed mechanism (*Res_DC/EX_ID[i]*). The bit coding shown in Fig. 3 is as

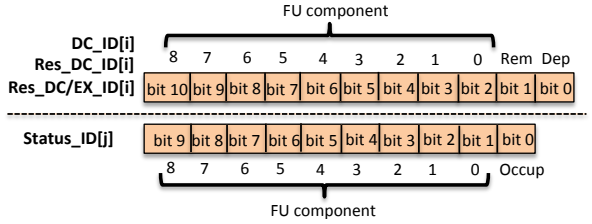

Fig. 3: ID coding

follows: a) *bits 10 to 2:* when a bit is set, its position shows the FU component (Table I) required for the instruction execution, b) *bit 1:* when it is set, it is a remaining instruction, i.e. it has not been scheduled yet, and c) *bit 0:* when it is set, the instruction has a dependency with at least one of the instructions of the next bundle. For instance, the ID="00000001110" is decoded as: the operation requires the FU component 1, i.e. it can be a MEM/ADD/ADDSHIFT operation (bit 3=1), the final multiplexer (comp 0) is required (bit 2=1) and the instruction has not been scheduled yet (bit 1=1). The status of all FUs components is represented by the table *Status_ID*, where each bit is the `f` signal of a FU enhanced by with an additional bit that is set when the FU is occupied.

The scheduling procedure is as follows: For all the instructions i of an input ID, i.e. the *Res_DC/EX_ID*, the *Res_DC_ID* and the *DC_ID* and for each issue j described by *Status_ID*, if the instruction i has not been scheduled and the FU in the j issue is unoccupied, we check if the required component is available. If it is available, the occupied bit of the corresponding *Status_ID* is set, the remaining bit of the corresponding input ID is cleared, since the instruction is scheduled, and the signal *sel_DC/EX* instructs the switch to pass the instruction currently at issue i to issue j. After that, a new instruction is explored.

The third component is the scheduler control part that controls the inputs and the execution of the scheduler processing part depending on the fault occurrence (`occur` signal) and the type of the scheduled instructions (signal `flag`). The state machine diagram of Fig. 4 describes its functionality, where $i \in [0, n]$ and n is the number of issues.

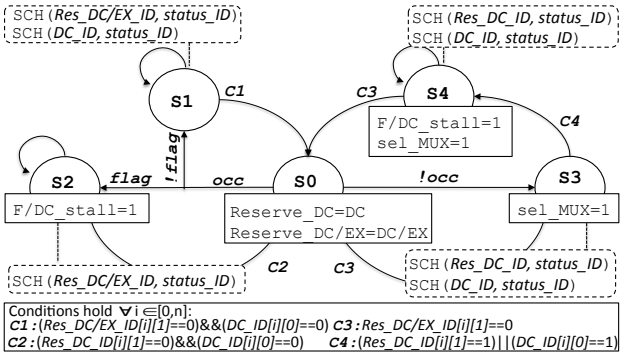

Fig. 4: State machine diagram of the control part.

(S1-S2) One (more) faults occurred at cycle $k-1$ (`occ` = 1): The executed instructions on the faulty FU components at cycle $k-1$ (which reside in *Reserve_DC/EX* register) are not committed and they must be scheduled again for execution at cycle k. Whether or not the Fetch and Decode stages must be stalled (`F/DC_stall`) depends on whether the faulty instructions are decoded instructions at cycle $k-2$ (`flag`=0) or at cycle $k-3$ (`flag`=1). In the first case (*S1*), no stall is required and these instructions are the first to be executed at cycle k (*Input_ID=Res_DC/EX_ID*). Then, the decoded instructions at cycle $k-1$ are explored (*Input_ID=DC_ID*). An example of this case is Fig. 1, where two faults occur during the execution of the instructions of bundle B_{k-1} at cycle $k-1$. During the scheduler decision for execution at cycle k, the remaining instruction A_1 is scheduled at issue 1 and, then, the decoded instructions of B_k are scheduled. In the second case (*S2*), the stall signal is activated (`F/DC_stall=1`) and a new cycle is inserted for the re-execution of the *Reserve_DC/EX* instructions. The process is repeated until no instructions are left in the *Reserve_DC/EX* (guaranteed by condition C2).

S3-S4) No fault occurred at cycle $k-1$ (`occ`=0): If no FU component is affected by a new fault at cycle $k-1$, the mechanism schedules first the remaining decoded instructions from cycle $k-2$ (that now reside in *Reserve_DC*) for execution

TABLE II: Performance comparison (execution cycles) under several multiple faults and average performance overhead (%).

Benchmarks	4-issue									8-issue										
	Original	Fine-grained mechanism				Coarse-grained mechanism				Original	Fine-grained mechanism					Coarse-grained mechanism				
Num. faults	0	1	2	3	4	1	2	3	4	0	2	4	6	8	10	2	4	6	8	10
adpcm_dec	386	388	390	391	397	412	436	464	571	302	304	311	335	393	401	314	366	442	471	566
adpcm_enc	409	413	413	425	426	462	469	504	614	323	324	326	412	415	420	339	437	483	494	633
bcnt	478	479	480	480	482	479	784	785	1159	333	335	336	338	339	397	335	527	545	580	1,085
fft32x32	569	580	587	591	669	667	775	930	1371	400	402	403	439	447	483	424	520	728	835	1,428
motion	344	350	350	358	368	391	392	407	598	280	281	282	285	287	302	282	284	363	377	619
huff	1,101	1,111	1,117	1,119	1,158	1,136	1,137	1,176	1,488	951	952	954	957	958	990	959	961	1,072	1,074	1,466
dct	1,288	1,314	1,315	1,456	1,458	1,353	1,627	1,846	2,606	872	877	912	953	954	992	951	1,322	1,369	1,544	2,578
fir	6,852	6,853	6,854	6,901	7,333	7,693	8,593	8,714	11,599	5,709	5,710	5,712	6,012	6,235	6,740	6,070	7,092	7,213	8,105	11,820
crc	12,228	12,229	12,229	12,231	12,232	12,274	12,275	14,851	20,969	11,955	11,956	11,959	11,960	11,989	12,245	11,956	11,958	12,217	15,107	20,964
mat_mul	11,142	11,143	12,423	13,010	15,011	15,015	16,039	16,358	21,593	6,533	6,535	6,538	6,539	7,102	8,112	6,534	10,951	11719	20,333	20,373
Average overhead %	0.8	2.2	4.6	9.3	10.6	24.1	33.5	82.7		0.3	1.1	6.7	10.3	17.3	3.3	29.1	44.9	69.7	110.6	

at cycle k, and, then, the current decoded instructions. If instructions still reside in the *Reserve_DC* and/or if there is any dependent instruction in the current decoded instructions *DC* that cannot be scheduled (condition C4), the `F/DC_stall` signal is set to stall the F and the DC stage for one cycle in order the mechanism to schedule these instructions.

V. Evaluation results

For the experimental results two heterogeneous VLIW configurations are used: i) 4-issue configured with 2 complex FUs, 2 simple FUs, 1 memory FU (MEM) and 1 branch unit (BR), and ii) 8-issue configured with 4 complex FUs, 4 simple FUs, 2 MEM and 1 BR. The processor has been enhanced with the proposed approach. Both the original unprotected VLIW processor and the VLIW with the proposed online fine-grained mitigation mechanism have been developed in C++ and synthesized using the Catapult High Level Synthesis (HLS) tool to obtain the RTL design. The gate-level netlist was generated by the Design Compiler of Synopsys using 28 nm ASIC technology. To evaluate our approach, we use ten benchmarks from the MediaBench suite. The benchmarks are compiled with VEX compiler for each configuration.

We compare the performance with existing online coarse-grained approaches, such as [9]. In contrast to the proposed fine-grained approach, the coarse-grained approach neither 1) explores the FUs in fine-grained way nor 2) applies temporal exclusion. For fare comparison, we randomly injected multiple faults (i.e. as many as the coarse grained approach can sustain) during the benchmarks' execution and we consider them as persistent, i.e. they last for the rest of the execution. Table II shows the cycles required to execute the ten benchmarks considering: i) 0 faults (Original), ii) 1 up to 4 multiple faults for the 4-issue configuration and iii) 2 up to 10 multiple faults for the 8-issue configuration. When no faults occur, both approaches have the same performance, i.e. the original execution cycles. We observe that: 1) the proposed approach inserts significantly lower overhead than the coarse-grained approach, and 2) in several benchmarks our performance is very close to the original one, i.e. without faults, even for several multiple faults. In contrast to the coarse-grained approach, our gain is achieved because whenever a persistent fault is detected, the proposed approach exploits the healthy FU components in the current and the next bundle execution.

We also present the logic area of each FU of a pipeline stage in Table III, which motivates the focus of the proposed approach on the execute stage, since it covers more area, and, thus, it is more exposed to faults.

TABLE III: Logic Area of pipeline stages (μm^2).

DC	DC_Br	Simple FU	Complex FU	MEM/WB
250	2,530	1,533	3,843	358

Table IV shows the area/power implementation results of the proposed mechanism with a target frequency of 200MHz. System's clock frequency is not affected because the critical path still resides in the EX/MEM/WB stage, while the only unit inserted in the pipeline (*DC/EX switch*) is strategically placed in the DC stage. Compared to the unprotected version, the proposed approach implies an area and a power overhead of up to 34% and 33%, respectively. The overhead of existing approaches is expected to be comparable, since both techniques require a switching mechanism and a re-scheduling logic, which are the costliest components of the design.

TABLE IV: Area footprint and power estimation.

Approach	4-issue		8-issue	
	area(μm^2)	power(mW)	area(μm^2)	power(mW)
Unprotected	50,844	6.48	79,661	7.36
Proposed	62,314	7.92	107,258	9.89

VI. Conclusion

Radiation can cause LDT failures. A hardware mechanism is proposed that, during execution, characterizes the FUs in a fine-grained way and reschedules the faulty instructions to the healthy FU components. Results show that multiple LDT mitigation is achieved with significant performance reduction.

References

[1] A. Simionovski and G. I. Wirth, "A bulk built-in current sensor for set detection with dynamic memory cell," *LASCAS*, pp. 1–4, Feb 2012.

[2] J. Velamala et al., "Design sensitivity of single event transients in scaled logic circuits," *DAC*, pp. 694–699, June 2011.

[3] T. Heijmen, "Radiation-induced soft errors in digital circuits – a literature survey," 2002.

[4] E. Koser and W. Stechele, "A long duration transient resilient pipeline scheme," *TDMR*, vol. 17, no. 1, pp. 12–19, March 2017.

[5] J. Klecka et al., "Error self-checking and recovery using lock-step processor pair architecture," 2002.

[6] R. Viera et al., "Validation of Single BBICS Architecture in Detecting Multiple Faults," *ATS*, Nov 2015.

[7] C. A. Lisboa et al., "Using built-in sensors to cope with long duration transient faults in future technologies," *ITC*, pp. 1–10, Oct 2007.

[8] R. Psiakis et al., "Neda: Nop exploitation with dependency awareness for re- liable vliw processors," *ISVLSI*, pp. 391–396, July 2017.

[9] R. Psiakis et al., "Run-time instruction replication for permanent and soft error mitigation in vliw processors," *NEWCAS*, pp. 321–324, June 2017.

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