

Dynamic fault-tolerant VLIW processor with heterogeneous Function Units

Rafail Psiakis, Angeliki Kritikakou, Olivier Sentieys

▶ **To cite this version:**

Rafail Psiakis, Angeliki Kritikakou, Olivier Sentieys. Dynamic fault-tolerant VLIW processor with heterogeneous Function Units. Microprocessors and Microsystems: Embedded Hardware Design , 2022, 93, pp.104564. 10.1016/j.micpro.2022.104564 . hal-03885490

HAL Id: hal-03885490
https://inria.hal.science/hal-03885490

Submitted on 6 Dec 2022

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.



Distributed under a Creative Commons Attribution 4.0 International License

Dynamic Fault-Tolerant VLIW Processor with Heterogeneous Function Units

Rafail Psiakis, Angeliki Kritikakou, and Olivier Sentieys
Univ Rennes, Inria, IRISA, Rennes, France

Abstract

Instruction Level Parallelism (ILP) of applications is typically limited and variant in time, thus during application execution some processor Function Units (FUs) may not be used all the time. Therefore, these idle FUs can be used to execute replicated instructions, improving reliability. However, existing approaches either schedule the execution of replicated instructions based on compiler schedule or consider processors with identical FUs, able to execute any instruction type. The former approach has a negative impact on performance, whereas the later approach is not applicable on processors with heterogeneous FUs. This work presents a hardware mechanism for processors with heterogeneous FUs that dynamically replicates instructions and schedules both original and replicated instructions considering space and time scheduling. The proposed approach uses a small scheduling window of two cycles, leading to a hardware mechanism with small hardware area. In order to perform such a flexible dynamic instruction scheduling, switches are required, which, however, increase the hardware area. To reduce the area overhead, a cluster-based approach is proposed, enabling scalability for larger hardware designs. The proposed mechanism is implemented on VEX VLIW processor. The obtained results show an average speed-up of 24.99% in performance with an almost 10% area and power overhead, when time scheduling is also considered on top of space scheduling. Compared to the unprotected version, the instruction reliability has increased by 2.2×.

1 Introduction

Modern systems follow technology and voltage scaling trends in order to meet industry's demands in increasing performance, area minimization and energy consumption reduction. However, decreasing transistors size and voltage makes processors more susceptible to reliability violations [1, 2]. Common sources leading to reliability violations, such as radiation-induced Single-Event Upsets (SEUs) and Single-Event Transients (SETs) [3], and electromagnetic interference [4], can cause faults leading to temporary errors (soft errors), permanent errors (hard errors) or semi-permanent (intermittent errors) [5]. To provide protection, systems are extended with error detection/correction capabilities.

The majority of existing works has mostly focused on protecting the storage components of a system, e.g., memories, register file, and pipeline registers, considering the impact on the combinational logic negligible. However, this is not true for technologies of 130nm and beyond [6]. The probability of a component to be affected by faults is highly related to its area. As shown by our experimental results, Function Units (FUs) occupy the largest combinational area (more details in Table 6 - Section 5), making FUs the most vulnerable combinational components of the processor.

To protect these components, either Hardware (HW) or Software (SW) redundancy is used. HW redundancy inserts additional FUs to the original processor, in order to execute in parallel the same instruction and to compare the obtained results [7]. Although small performance overhead is normally ob-

served (due to comparison of the results), the area overhead is significant. SW redundancy modifies the program by inserting replicated instructions to be executed on the original processor. Although the area is not increased, the impact on the execution time is significant [8]. A better area and performance trade-off for fault tolerance can be achieved, when processors with several FUs are used. Very Long Instruction Word (VLIW) processors is an example, where several issues exist that can process instructions in parallel. Depending on the configuration, VLIW issues consist of complex FUs, able to execute all operation types, and simpler FUs, which, however, cannot execute sophisticated operations, such as multiplications and divisions. The set of instructions, executed in parallel, is called instruction bundle. Due to the limited application's Instruction Level Parallelism (ILP) and the different type of FUs, concurrent utilisation of all FUs is not always possible. As shown in the experimental results, the average ILP of ten benchmarks on a VLIW processor having 4 issues is 2.15 (more details in Table 7 - Section 5). Therefore, the idle FUs can be used to execute the replicated instructions, reducing the negative impact of SW redundancy on the execution time.

The exploitation of the idle FUs, to execute redundant instructions, can be achieved statically or dynamically. Static approaches are usually implemented by the compiler, which duplicates or triplicates and schedules the instructions at compile time in the instruction memory and extends the Instruction Set Architecture (ISA) to include special comparison instructions [9]. Although the obtained schedule can theoretically be as dense as possible, code size is significantly increased, increasing the required storage size. To remove this drawback, dynamic approaches have been proposed, implemented by hardware mechanisms. Dynamic hardware approaches reschedule the instructions at run-time despite compilers scheduling, in order to exploit idle slots and achieve fault tolerance.

Existing dynamic approaches, such as [10, 11], couple the VLIW issues, removing the need of a dynamic scheduler. Duplicated instructions are executed on the coupled pipelines, following the schedule of the original instructions, provided by the compiler. However, due to the lack of dynamic scheduling, there is a negative impact on the performance. Approaches with dynamic scheduler are designed for homogeneous VLIW, i.e., each issue includes all FU types, in order to be able to execute any instruction [12, 13]. However, such approaches are not applicable when the VLIW issues have different types of FUs, because the scheduler does not take into account the constraints over the type of FUs in an issue. Extending a heterogeneous VLIW to a homogeneous VLIW by inserting all FU types to each issue significantly increases area, especially when floating-point FUs are used.

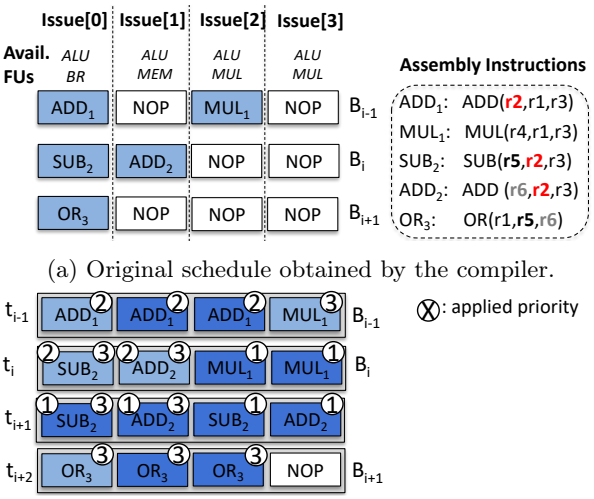
To address these limitations, a hardware mechanism is proposed that enhances VLIW processors with heterogeneous FUs with fault tolerant capabilities. The proposed mechanism dynamically replicates instructions and reschedules original and replicated instructions both in the current bundle (space scheduling - S) and in the next upcoming bundle (time scheduling - T), improving performance, with reduced complexity and area overhead. To achieve this, a scheduling exploration window of two instruction bundles is used. In this way, new opportunities are created for optimizing the schedule during execution, while the scheduling complexity remains low. Within this window, the instruction rescheduling is only restricted by the FUs type and any dependency among the instructions within the scheduling window. This work extends the idea of dynamic instruction rescheduling of [14], where only the concept and some preliminary performance results, based on simulations, were presented. More precisely, this work presents the implementation of proposed mechanism consisting of i) a hardware dependency analyzer, ii) a hardware replication scheduler, and iii) a hardware voting scheduler. The dependency analyzer identifies dependent instructions between two concurrent bundles. The replication scheduler dynamically reschedules original and replicated instructions, avoiding the insertion of additional time slots, by postponing the execution of independent instructions to the next bundle. The voting scheduler synchronises original and replicated instruction in order to compare their outputs, masking any occurring error. Due to dynamic rescheduling, switches have to be inserted in

the VLIW pipeline. To reduce the area and critical delay overhead, we propose a switch hardware design that simplifies the complexity of the scheduler, compared to commonly used switches. Last, but not least, to enable scalability, we propose a novel cluster-based approach; a large VLIW is divided in several clusters, each consisting of a smaller VLIW. The proposed dynamic hardware mechanism is applied within a cluster (intra-cluster), providing significant gains in area, with small performance loss. To support our contributions, we implemented on the VEX VLIW processor and compare the proposed approach that applies instruction triplication, considering space scheduling (TMR-S) and space and time scheduling based on clusters (TMR-S+T+CB), with typical hardware FUs triplication methods and the unprotected baseline VLIW. We performed extensive experimental results, providing performance, area, power and reliability metrics for ten benchmarks.

The remaining of this paper is organized as follows. Section 2 presents the proposed approach through an illustrative example. Section 3 describes the intra-cluster dynamic mechanism. Section 4 describes the proposed cluster-based version. Section 5 presents the experimental results. Section 7 discusses the related work on fault tolerant VLIW processors. Finally, Section 8 concludes this work and summarizes its contributions.

2 Illustrative Example

In this section, we provide an example to illustrate the proposed approach. Our example is depicted in Fig. 1 and it will be used throughout the manuscript as a running example. For simplicity reasons and without loss of generality, we assume an 4-issue VLIW, as the one in Fig. 2. The VLIW configuration has one Arithmetic Logic Unit (ALU) and one Branch unit (BR) in the Issue[0], one ALU and one Memory FU (MEM) in the Issue[1] and one ALU and one Multiplication unit (MUL) in Issue[2] and Issue[3]. Fig. 1a provides (on the left) the schedule given by the compiler and (on the right) the corresponding assembly instructions, with the destination and source registers. For instance, ADD_1 is an



(a) Original schedule obtained by the compiler.

(b) Dynamic schedule obtained by the proposed approach.

Figure 1: Illustration example for an 4-issue VLIW.

instruction that adds the values of the registers $r1$ and $r3$ and stores the result to the register $r2$. The compiler has scheduled these instructions to three instruction bundles, B_{i-1}, B_i and B_{i+1}. B_{i-1} has two instructions (ADD_1, MUL_1) and two idle slots (NOP), B_i has two instructions (SUB_2, ADD_2) and two idle slots, and B_{i+1} has one instruction (OR_3) and three idle slots.

The proposed approach replicates instructions and dynamically schedules original and replicas on the FUs, within a scheduling window of two instruction bundles. Fig. 1b shows the schedule obtained by the proposed dynamic mechanism, when instruction triplication is applied. The light (dark) blue boxes represent original (replicated) instructions.

To obtain this schedule, a *dependency analysis* initially takes place between the instructions of two consecutive bundles (implementation in Sections 3.2.1 and 3.2.2). The goal is to identify dependent instructions, i.e., an instruction in bundle B_{i-1} that uses, as destination register, a destination or source register of an instruction in bundle B_i. Parallel execution of the dependent instructions is forbidden. Independent instructions of B_{i-1} can be postponed and scheduled at

any idle FUs of the next bundle. In Fig. 1, MUL_1 of bundle B_{i-1} is independent, since none instruction of bundle B_i (SUB_1, ADD_2) uses the destination register of MUL_1 as destination or source register. However, SUB_2 and ADD_2 of bundle B_i both depend on ADD_1, since they read register *r2*, updated by ADD_1. In a similar way, the instruction OR_3 of B_{i+1} reads registers *r5* and *r6*, which are used as destination registers by SUB_1 and ADD_2 of B_i.

The results of the dependency analysis are used by the *replication scheduler*, responsible for the dynamic scheduling of original and replicated instructions (implementation details in Section 3.2.3). The scheduler operates according to three priorities, applied in the following order: ① instructions of a previous bundle have a higher priority than instructions of the current bundle, ② the dependent instructions have a higher priority than the independent instructions, and ③ instances of different instructions of the same bundle have higher priority than the replicated instances of the same instruction. These priorities reduce the negative impact on performance, because of the execution of replicated instructions to provide fault tolerance. By scheduling first the dependent instructions, the remaining instructions are independent and can be potentially scheduled in the next bundle. Fig. 1b depicts which of the three priorities the hardware scheduler uses each time in order to obtain the schedule. At the time slot t_{i-1}, the original, the first replica and the second replica of ADD_1 are scheduled, due to priority ②. Then, the original MUL_1 is scheduled, due to priority ③. Since the remaining instructions from bundle B_{i-1} are independent, they are allowed to be scheduled alongside with the instructions of the upcoming bundle B_i. Due to priority ①, the first and second replica of the MUL_1 of bundle B_{i-1} are scheduled at the time slot t_i. Then, each of the original SUB_2 and ADD_2 are scheduled, due to priorities ② and ③. However, the remaining unscheduled instructions of B_i are dependent, and thus, they cannot be executed with the upcoming bundle B_{i+1}. Hence, a new time slot has to be added. At this new time slot t_{i+1}, the remaining dependent and redundant instructions of B_i are scheduled based on priority ① and ③. At t_{i+2}, no instruction remains from the previous bundle, thus original and replicated OR_3 are scheduled, according to priority ③.

Last, but not least, after the execution of the scheduled instruction, the *voting scheduler* is responsible for synchronizing and grouping the original and the replicated instructions in order to compare their results and mask any occurred error (implementation details in Section 3.2.4).

3 Intra-cluster Dynamic Mechanism

The proposed approach is applied over a VLIW processor, which has a fixed issue-width during execution, defined at design-time, running a single application. Dynamically reconfigurable and fault tolerant VLIWs, such as [15], where several applications are executed in parallel, is left as future work. Fig. 2 depicts the original VLIW datapath consisting of a 3-stage pipeline with Fetch (F), Decode (DC) and Execute/Memory-WriteBack (EX/M-WB) in blue color. The yellow color highlights the hardware components that implement the proposed intra-cluster dynamic approach. These hardware components are categorized as follows: i) processing type (the *replication switch*, the *voting switch*, and the *voters*), ii) control type (the *information extraction unit*, the *dependency analyzer*, the *replication scheduler*, and the *voting scheduler*) and iii) storage type (the *ReplicRes* register and the *VotingRes* register).

The *information extraction unit* performs an early decoding in the F stage, providing the necessary information to the *dependency analyzer* and the *replication scheduler*. The *dependency analyzer* is the component responsible for analyzing two subsequent bundles in order to identify potential dependencies. The *replication switch* propagates previously decoded instructions (stored at the *ReplicRes* register) and the currently decoded instructions to the pipeline *DC-toEX* register, following the dynamic schedule provided by the *replication scheduler*. The *voting switch* propagates the results of the previously executed instructions (stored at the *VotingRes* register), as well as the results of currently executed instructions to the

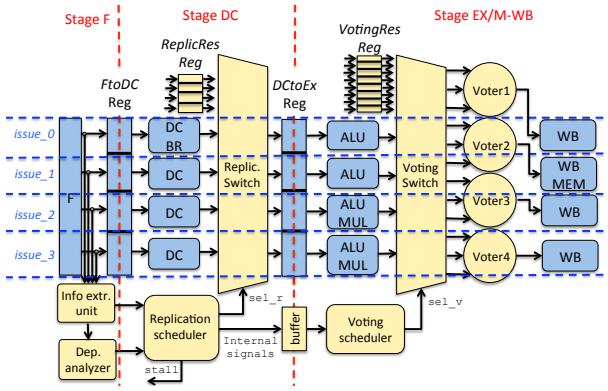


Figure 2: Original VLIW data-path (blue) enhanced with the proposed fault tolerant mechanism (yellow).

voters for correction, following the dynamic schedule provided by the *voting scheduler*. The objective of our approach is to reduce the negative impact on performance, due to the execution of replicated instructions to provide fault tolerance. Therefore, the hardware components, that do not necessarily require to be placed inside the VLIW data-path, are designed to run in parallel and to have a smaller critical path than the VLIW pipeline stages, in order to not affect the clock frequency. The remaining components, that must obligatory added in the VLIW data-path to support the functionality of the proposed approach, are designed with reduced critical path and are placed in different pipeline stages, so as to reduce the impact on the overall clock frequency. For instance, the replication switch is added in the DC stage, which has a smaller critical path than the EX/M-WB stage. Through the information provided by the voting scheduler, the voting switch is significantly simplified, as described in the following paragraphs. We leave as future work the extension of the proposed approach to VLIW processors that support long immediate and multi-cycle instructions. For the rest of the manuscript, let n represent the number of VLIW issues.

3.1 Processing and Storage Components

3.1.1 Replication Switch and ReplicRes Register

The replication switch selects which, among the $2\times n$ possible inputs (i.e., n outputs of the decoders, one per issue, and n positions from the *ReplicRes* register) will be propagated and become the n inputs of the pipeline register *DCtoEX*. Each input/output of the switch is a decoded instruction. For instance, Table 1 depicts the structure of the decoded instruction of the VEX VLIW processor [16], used in the experimental section. The switch input signal has a width of $2\times n\times Size_{Dec}$ bits. To avoid a full switch $2n$-to-$2n$, which decreases the clock frequency, the replication switch is placed in the DC stage. Due to this positing, an exact copy of output of the decoders of the previous cycle is stored to the *ReplicRes* register. In this way, the replication switch implementation is simplified, reducing its complexity, area and impact on clock frequency. The switch output has a width of $n\times Size_{Dec}$ bits. To implement the replication switch, $n\times(2n$-to-1) multiplexers are required with a selection signal of $n\times log_2(2n)$ bits. Fig. 3 illustrates the replication switch for the 4-issue VEX VLIW processor; the input signal has a width of $2\times4\times109=872$ bits, the circuit has 4×8-to-1 multiplexers and a selection signal of 4×3 bits.

3.1.2 Voting Switch and VotingRes Register

The voting switch is placed after the FUs. It is responsible for grouping correctly the results of the original and replicated instructions: a) into the *VotingRes* register, and b) into the voters. For instance, Table 2 shows the structure of the result, after the execution of an instruction, for the VEX VLIW processor. When replicas of one instruction remain to

Table 1: Decoded instruction ($Size_{Dec}$ = 109 bits).

Data from RF	Data from RF/Immediate	Data for Store instruction	Dest. addr.	Opcode
32 bits	32 bits	32 bits	6 bits	7 bits

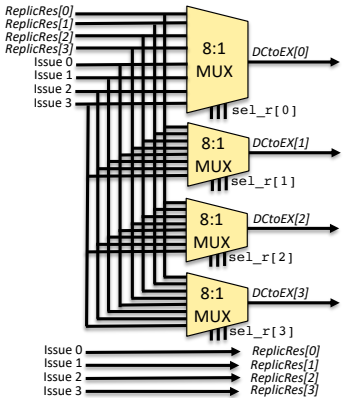


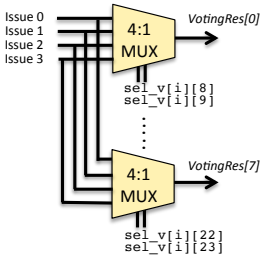
Figure 3: *Replication switch* implementation.

be executed, the results of the already executed instances of this instruction need to be temporarily stored. This storing takes place at the *VotingRes* register. Therefore, the *VotingRes* register has $2 \times n$ entries to store the results of up to two instances of one instruction per issue.

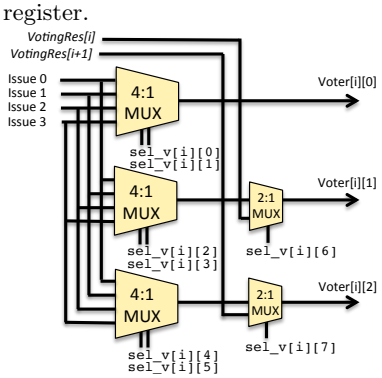
Table 2: Instruction's result ($Size_{res}$ = 78).

FU result	Data to store in memory	Dest. addr.	Opcode	WB Enable
32 bits	32 bits	6 bits	7 bits	1 bit

Typical symmetrical designs for switches have high complexity, which will lead to increased overheads if used by the proposed approach. To reduce complexity, we propose an asymmetrical design for the voting switch reducing the number of required 2-to-1 muxes. The improved design uses 44 2-to-1 muxes, comparing to 132 2-to-1 muxes used by typical designs, reading to 66% gain. The *voting scheduler* groups upfront the results of the same instruction instances to the *VotingRes* register. Then, each such group is attached to a specific voter. To perform this grouping, the results of the two instances of the same instruction are stored in adjacent entries of *VotingRes* register, i.e., in the even entry for the first instance and in the odd entry for the second instance. More precisely, the voting switch has n inputs (coming from the FU output at each issue), which will be stored at the register *VotingRes*, following the grouping provided by the voting scheduler through a select signal of $\log_2(n) \times 2n$ bits. Fig. 4 presents the implementation of the voting switch considering the 4-issue VLIW. Fig. 4a depicts the part related to the grouping of the results. It has 4 inputs to be stored at the register *VotingRes* with 8 entries (*VotingRes[0]* to *VotingRes[7]*). The select signal is 16 bits. Due to



(a) Part for grouping to *VotingRes* register.



(b) Part for connecting a voter to an issue.

Figure 4: *Voting switch* implementation.

this grouping of results, the switch is simplified, i.e., it is not required to connect all the inputs of the voters with all the FUs and all the entries of *VotingRes* register. More precisely, the three inputs of a voter, belonging to an issue i, are connected as follows: 1) the first input 0, *Voter[i][0]*, requires to be connected only to the outputs of the FUs, 2) the second input 1, *Voter[i][1]*, requires to be connected either to the FUs or to the entry *VotingRes[i]*, and 3) the third input 2, *Voter[i][2]*, requires to be connected to either

the FUs or the entry *VotingRes[i+1]*. Fig. 4b depicts the part of the switch related to the connection of one voter. It has $n + 2 = 6$ inputs (n from the FU outputs and 2 from the *VotingRes* register outputs), which have to be connected to the 3 inputs of a voter. Finally, the select signal is $\log_2(n) \times 3 + 2$=8 bits.

3.1.3 Voters

The *Voting switch* provides to the voters the results, already grouped, after the execution of the three instances of one instruction. The voters compare and select which is the correct value to be provided at the output. Note that, the proposed approach replicates the complete decoded instruction, i.e., all the fields shown in Table 1. Since the address for the memory operations is also calculated in the execution stage by the FUs and it is available in the FU result field, we partially protect also the memory operations.

3.2 Control Logic Components

3.2.1 Information Extraction Unit

The *information extraction unit* is depicted in Fig. 5. It reads each instruction from the F stage and extracts the opcode, the destination, and the source registers. To reduce the cost of the *Voting scheduler* and *Replication scheduler*, each instruction, currently in the pipeline, is associated with an Instruction Identifier (ID) and the number of instruction instances that need to be executed (Rem). This early decoded information is stored in an intermediate internal register (*Info* register). The register outputs are read by the *dependency analyzer* and the *replication scheduler*.

An ID value has a size of $Size_{IDvalue} = n + 1$ bits. The first $\log_2(n)$ Least Significant Bits (LSBs) show the type of the instruction. The next $\log_2(n)$ bits show the original position of the instruction inside a bundle, which corresponds to the issue where the instruction has been scheduled by the compiler. The Most Significant Bit (MSB) indicates whether the instruction belongs to the previous ($MSB = 0$) or to the current bundle ($MSB = 1$). The coding of the ID for the 4-issue VLIW is depicted in Table 3. Bit b_4 indicates the bundle where the instruction belongs to

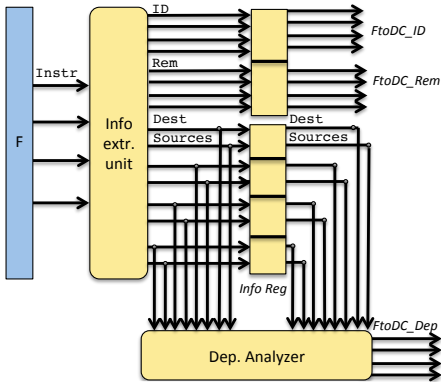


Figure 5: *Information extraction unit* and *Dependency analyzer*.

($b_4 = 0$ for previous bundle, $b_4 = 1$ for current bundle). Bits b_3 and b_2 show the original position of the instruction inside a bundle. Bits b_1 and b_0 carry the information about the instruction type.

Table 3: ID encoding in the *information extraction unit*.

B_{i-1}					B_i					
Bundle	No.Issue		Inst/ons		Bundle	No.Issue		Inst/ons		Instr.
b_4	b_3	b_2	b_1	b_0	b_4	b_3	b_2	b_1	b_0	Type
0	0	0	0	0	1	0	0	0	0	NOP
0	0	1	0	1	1	0	1	0	1	MEM
0	1	0	1	0	1	1	0	1	0	MUL
0	1	1	1	1	1	1	1	1	1	ALU

Each of the four registers, that is considered as an input and an output of the two switches (*FtoDC*, *ReplicRes*, *DCtoEX* and *VotingRes*), is associated with an array of IDs, as depicted in Fig. 6. The size of each array is $Size_{ID} = n \times Size_{IDvalue}$. Two additional arrays (*Rem*) are associated with *FtoDC* and *ReplicRes* registers, i.e., *FtoDC_Rem* and *ReplicRes_Rem*, used for the replication of an instruction. These arrays indicate the number of instances of the instruction that remain to be scheduled. When a bundle is in the DC stage, the values of the *FtoDC_Rem* are initialized to 3, since instruction triplication is applied as fault tolerance method. At the next time slot, some of these instructions are potentially executed on the FUs. Therefore, the values of the *ReplicRes_rem* are updated to depict the num-

ber of unscheduled instruction instances.

To illustrate the above approach, Fig. 6 depicts the *ID* and the *Rem* arrays of the running example of Fig. 1b, at time t_i, when bundle B_i is in the DC stage. The *FtoDC_ID* array is related to bundle B_i: the first element (10011) and the second element (10111) indicate that the instructions in the first and second issue are of *ALU* type, the third element (11000) and the fourth element (11100) indicate that there are not instructions in the third and fourth issue, i.e., the type of these instructions is *NOP*. The array *FtoDC_rem* is initialized to 3, meaning that each of the corresponding instructions require to be executed three times. The *ReplicRes_ID* array is related to the remaining instructions from the previous bundle B_{i-1}: the first element (00011) indicates that the first issue has an *ALU* instruction, the second element (00100) is a *NOP*, the third element (01010) stands for a *MUL* instruction and the last one (01100) stands for a *NOP*. At the time t_{i-1}, three ADD_1 and one MUL_1 instructions have been scheduled (Fig. 1b), so the *ReplicRes_Rem* array has been updated accordingly.

3.2.2 Dependency Analyzer

The *dependency analyzer* (Fig. 5) decides if any dependencies exist among the instructions of two subsequent bundles. To do so, it takes as input the opcode, the destination and the sources of each instruction of the two bundles. Three possible dependencies may exist: 1) Read After Write (RAW), 2) Write After Read (WAR) and 3) Write After Write (WAW). Only RAW and WAW are verified by the dependency analyzer. The dependency analyzer reads the destination of each instruction from the *Info* register and compares it with the sources (RAW) and destination (WAW) of each instruction, currently extracted from the *information extraction unit*. If these values are equal, then there is a dependency. A *Dep* array is associated with *FtoDC* register to store this information. This information is also used by the *replication scheduler* to take dependencies into account. Fig. 6 depicts the *Dep* array of the running example (Fig. 1b) at time t_i, where the two instructions of bundle B_i write registers that serve as sources to the instruction of bundle B_{i+1}. Hence, the corresponding *FtoDC_Dep* entries are set to 1. Notice that the case

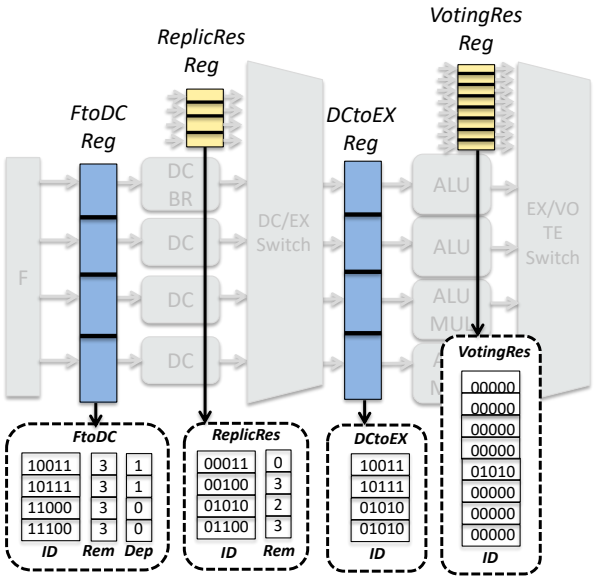


Figure 6: *ID*, *Rem* and *Dep* arrays of example Fig. 1b at time t_i.

of WAR dependency is prevented by the hardware design itself. The proposed mechanism may move the instruction that uses the register as source (performs the "read") to the next bundle, and thus, it may be executed in parallel with the instruction that uses the register as a destination (performs the "write"). However, the mechanism reschedules the instructions after the DC stage, thus the instruction has already obtained the correct value of the register at the DC stage.

3.2.3 Replication Scheduler

The replication scheduler schedules the inputs of the *replication switch* (represented by *ReplicRes_ID* and *FtoDC_ID*) to its output (represented by *DCtoEX_ID*). This hardware scheduler applies a set of priorities in the following order: ① unscheduled instructions from previous bundle (instructions in *ReplicRes* register) have the highest priority, ② the dependent instructions are scheduled next, and then

③ among the remaining instructions, instances of different instructions have higher priority than the replicas of a single instruction. Three groups are created to depict these scheduling priorities: 1) *ReplicRes* for the previous instructions, 2) *FtoDC_Dep* for the dependent current instructions, and 3) *FtoDC* for the independent current instructions. To design a hardware scheduler with low complexity, the scheduler should use simple logic. To achieve that, the IDs are transformed to occupation vectors, enabling the scheduler implementation through simple bitwise logic.

Occupation vectors Each aforementioned group has three entries, since triplication is used as fault tolerant method. Each group entry includes two vectors: 1) *instr*, where each bit corresponds to an issue and indicates if an instruction exists (occupies the issue), and 2) *mul*, where each bit indicates if the instruction of the issue is a multiplication (e.g., in the 3rd and 4th issue of the VLIW configuration of our running example). Table 4 shows how to obtain the *instr* vector. More precisely, the bit i is set if at least one of the LSBs of *ID[i]* is not zero and instruction instances remain to be scheduled (*Rem[i]*). i represents the issue number and $j \in [0,2]$ represents the original instruction, the first replica and the second replica, respectively. Especially, the instructions must be dependent (*Dep[i]*) for the *FtoDC_Dep* group and independent (*!Dep[i]*) for the *FtoDC* group. The bits in the *mul* vector are set, when the IDs in the issues with a multiplication unit have a multiplication instruction, e.g., $i \in \{2,3\}$ in the 4-issue VLIW of Fig. 2 with an $ID[i][1:0] = 10$.

Fig. 7 depicts the occupation vectors of Fig. 1b at time t_i. The instruction in issue 2 (third entry in *ReplicRes*) needs to be executed two more times (*ReplicRes_Rem[2]*=2). Therefore, the 3^{rd} bit of the *ReplicRes_instr[1]* and *ReplicRes_instr[2]* is set. The instruction is a multiplication, thus the vector *mul* is set to 10 to all the entries of *ReplicRes_mul*. The *FtoDC_Dep_instr*, *FtoDC_Dep_mul*, *FtoDC_instr* and *FtoDC_mul* are constructed in a similar way.

Table 4: Creation of *instr* vector.

Group	*instr[i]*
ReplicRes	$(ID[i][0] \parallel ID[i][1])$ & $(Rem[i] \geq 3-j)$
FtoDC_Dep	$(ID[i][0] \parallel ID[i][1])$ & $(Rem[i] \geq 3-j)$ & $Dep[i]$
FtoDC	$(ID[i][0] \parallel ID[i][1])$ & $(Rem[i] \geq 3-j)$ & $!Dep[i]$

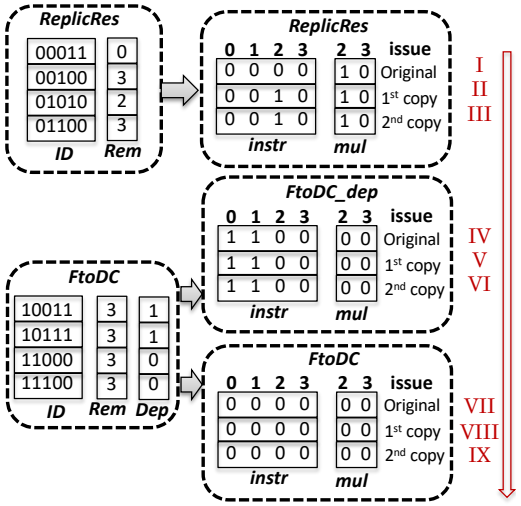


Figure 7: Occupation vectors of the running example.

Bitwise Scheduler Following the scheduling priorities, the groups and the occupation vectors are explored in the order depicted by Fig. 7 (latin numbers). For an occupation vector, the scheduler applies three phases: i) scheduling through *direct assignment*, ii) scheduling through *circular exploration*, and iii) decision for time-slot insertion.

i) Direct assignment: During this phase, only the issue, where the instruction is originally scheduled by the compiler, is checked for scheduling. Hence, no verification of the issue's FU type is required. The scheduler has two inputs: 1) the occupation vector for the instructions to be scheduled (*instr*), and 2) the current occupation of the issues (*issues*) (with bit 3 corresponding to issue 0 and bit 0 to issue 4). Three outputs are created: 1) the updated occupation vector of the issues (*issues_up*), 2) the vector with the scheduled instructions (*fit*) and 3) the vector with

the instructions to be scheduled (*rest*). As shown in Eq. 1, the scheduling is performed through a bitwise OR operation between the *instr* and *issues* vectors. The *issues_up* is compared (bitwise XOR) with the initial output vector *issues*, to decide which of the instructions can be mapped directly to the output (*fit*), and thus, to be scheduled for execution. To reflect the scheduling decision, the *ID* and *Rem* arrays are modified as follows: a) if *fit[i] = 1*, then *Rem[i] = Rem[i] - 1*, and b) *DCtoEX_ID[i] = ID[i]*. The *sel_r* vector signal is updated to reflect the scheduling decision to the *replication switch*: *sel_r[i] = i + n*, when a vector from the *ReplicRes* group is scheduled, and *sel_r[i] = i* for the vectors of the other groups. To obtain the instructions, that could not be scheduled directly (*rest*), the vector *fit* is compared with the initial input vector *instr*. If the result is zero, this phase has scheduled all instructions, and the next input vector is explored. Otherwise, the circular exploration phase is applied to schedule the *rest* instructions to the empty FUs.

$$\begin{aligned} issues_up &= instr \parallel issues, \\ fit &= issues_up \oplus issues, \\ rest &= instr \oplus fit. \end{aligned} \tag{1}$$

Before describing the next scheduling phase, we illustrate how the direct assignment is applied to the running example. Initially, the *issues* vector equals to zero. Following the priorities, the first group to be scheduled is *ReplicRes*. The first vector of the group is zero, thus no instruction exists for scheduling. The second vector is *instr* = 0010. The output *fit* is the same as the input *instr*, thus all instructions can be scheduled (*rest* = 0000). To depict the scheduling decisions, the *ID* and *Rem* arrays are modified as follows: a) $ReplicRes_Rem[2] = Rem[2] - 1$, b) $DCtoEX_ID[2] = ReplicRes_ID[2]$, and $sel_r[2] = 2$. The next vector is *instr* = 0010, while the *issues* is initialised with the previously calculated *issues_up*. Now, the output *fit* is different than the input *instr*, implying that some instructions could not be scheduled through direct assignment (*rest* = 0010).

ii) Circular exploration: During this phase, the type of FU per issue is verified. This mechanism takes three inputs: 1) the occupation vector for the instructions to be scheduled (*instr*), which is equal to the vector *rest* provided by the direct assignment, 2) the *mul* occupation vector for these instructions, and 3) the current occupation of the issues (*issues*), which is equal to the *issues_up* provided by the direct assignment. The output is the assignment signal *assign*, which holds the configuration of the *replication switch* to adequately propagate the instructions that have been scheduled for execution.

Algorithm 1: Representative cases of the circular exploration phase.

```
   Input: instr, issues_up, mul
   Output: assign array
1  switch instr do
2      ...
3      case 1000 do
4          if (issues_up[2] = 0) then assign[1] ← 0;
5          else if (issues_up[1] = 0) then
             assign[2] ← 0 ;
6          else if (issues_up[0] = 0) then
             assign[3] ← 0;
7      case 0010 do
8          if (issues_up[0] = 0) then assign[3] ← 2;
9          else if (mul[1] = 0) then
10             if (issues_up[2] = 0) then
                 assign[1] ← 2;
11             else if (issues_up[3] = 0) then
12                 assign[0] ← 2;
13     case 0011 do
14         if (mul = 10) then
15             if (issues_up[2] = 0) then
                 assign[1] ← 3;
16             else if (issues_up[3] = 0) then
17                 assign[0] ← 3;
18         else if (mul = 01) then
19             if (issues_up[2] = 0) then
                 assign[1] ← 2;
20             else if (issues_up[3] = 0) then
21                 assign[0] ← 2;
22         else if (mul = 00) then
23             if (issues_up[3] = 0) then
                 assign[0] ← 2;
24             if (issues_up[2] = 0) then
                 assign[1] ← 3;
25     ...
```

Algorithm 1 presents some representative cases of the circular exploration phase. In the first two cases, only one instruction needs to be scheduled. When *instr* = 1000, the instruction cannot be a multiplication. When *instr* = 0010, the instruction can be a multiplication. This is because the issue, where the instruction is original scheduled, has a multiplication FU. In the former case (line 3), the instruction can be potentially scheduled in any available issue. In the latter case (line 7), two possibilities exist: a) whatever the value of *mul[1]*, the instruction can be potentially executed in *issues_up[0]* (corresponding to the 4^{th} issue), and b) otherwise, if *mul[1]* = 0, the instruction can be potentially executed in any possible issue. The third case (line 13) has two instructions for scheduling (case 0011). Since these instructions are originally scheduled in issues with the multiplication FU, we have to check if the instructions are multiplications. Three possibilities exist: a) *mul[1]* = 1 and *mul[0]* = 0, b) *mul[1]* = 0 and *mul[0]* = 1, and c) *mul[1]* = *mul[0]* = 0. No available scheduling exists when both instructions are multiplications, due to FU restrictions. For a) and b) cases, the multiplication instruction cannot be scheduled, since the remaining idle issues do not have a multiplication FU. Only the non-multiplication instruction can be scheduled. For c) case, no multiplication instruction exists, thus both instructions can be scheduled in any available issue. When three instructions need to be scheduled (case 1101, 1110, etc.), this has the least complexity. All three instructions have been tested for direct scheduling and failed. The only legitimate action is to potentially schedule the non-multiplication instructions (*instr[2]*, *instr[3]*) (e.g. 1101 → 1011). The signal *assign[i]* is updated with the new issue, where the instruction is now dynamically scheduled. For instance, if *instr* is 1000 and is scheduled to 0100, then *assign[2]* equal to 0, to show that instruction of issue 0 is now scheduled to issue 1. The *assign[i]* signal is used to update the *issues_up* vector, the *ID* arrays and the signal *sel_r*.

In our running example, after the direct assignment of *ReplicRes* group, *rest* = 0010. Since *issues_up* = 0010, the condition of line 8 in Algorithm 1 is met. Therefore, *assign[3]* ← 2. Consequently we have: a) *issues_up* = 0011, $Rem[2] = Rem[2] - 1$ and b) *DCtoEX_ID[3]* = *FtoDC_ID[2]*. The *sel_r* vector signal is updated to *sel_r[3]* = 2.

iii) Time-slot insertion decision: If no appropriate FU is available for scheduling and there are still instructions from the previous bundle (*ReplicRes_rem* ≠ 0) or dependent instructions (*FtoDC_rem* ≠ 0), an additional time slot must be inserted. Instructions from the previous bundle, that have not managed to be scheduled in the current bundle, will need in any case an additional time slot, since there are not enough FUs to schedule them in the current bundle, while the instructions of the current bundle have to be also replicated and scheduled. Pending dependent instructions require also an additional time slot, as they cannot be scheduled in parallel with instructions from the upcoming bundle. The additional time slot is achieved by stalling the fetch and decode stages for one cycle.

3.2.4 Voting Scheduler

This scheduler is responsible for the grouping of: i) the results of currently executed instructions at the *VotingRes* register, and ii) previous and current results to the *voters* and the commit/memory phase. To perform the grouping, we use the array *unGrouped_ID*, created by concatenation of the two inputs of the voting switch, i.e., *DCtoEX_ID* and *VotingRes_ID*. A comparison-based sorting algorithm groups the instructions with the same *IDs* in order to form triplets. The output is a sorted array (*Grouped_ID*). Each triplet in the *Grouped_ID* is linked with the voter of one issue. Based on this sorted array, the values of the signal *sel_v* are defined and used to configure the *voting switch*. For instance, the triplet formed in the three first places of the *Grouped_ID* array, i.e., *Grouped_ID[0]*, *Grouped_ID[1]* and *Grouped_ID[2]*, sends the corresponding instructions to the first voter. In case the triplet is not formed yet, e.g., IDs exist only in *Grouped_ID[0]* and/or in *Grouped_ID[1]*, the results of the corresponding instructions are stored to the *VotingRes[0]* and *VotingRes[1]* registers. In this phase, the constraints on memory operations must be respected. Therefore, memory instructions are allowed to be grouped only in positions linked

with issues that have a MEM FU, e.g., the entries *Grouped_ID[3]*, *Grouped_ID[4]* and *Grouped_ID[5]*, for the 4-issue VLIW of Fig.2, since they are linked to the voter of the second issue, where a memory FU exists.

4 Cluster-Based Approach

By analysing the area overhead of the components of the intra-cluster dynamic mechanism (as shown in Table 8 of the experimental section), we observe that the costliest components, in terms of area, are the switches. Switches also decrease maximum clock frequency. It should be stressed that this holds true for any approach that performs dynamic re-scheduling of the instructions, since this type of techniques requires to dynamically dispatch the instructions to different issues. We have carefully designed the switches of the proposed approach in order to achieve simpler complexity (as explained in Section 2). However, with the increase of the number of issues, the complexity of the switches is also increased, affecting the area overhead and clock frequency. Table 5 presents the number of inputs, outputs and the complexity of the proposed replication and voting switches, in numbers of required 2-to-1 muxes, considering in a single cluster, with respect to the number of VLIW issues. To address this limitation, we propose a cluster-based approach that reduces the area and time overhead, when the number of issues increases.

Table 5: Comparison of number of inputs, number of outputs and complexity for replication switch and original and improved voting switch (Section 3.1.2) in numbers of required 2-to-1 muxes.

	Replication switch				Voting switch			
Issue	In	Out	Single Cluster	Cluster based	In	Out	Single Cluster (Orig. Switch)	Single Cluste (Impr. Switc
4	8	4	28	28	12	12	132	44
8	16	8	120	56	24	24	552	100
16	32	16	496	112	48	48	2256	212
32	64	32	1952	224	96	96	9120	436

352

Instead of having a single large VLIW applying directly the intra-cluster dynamic mechanism of Section 2, we combine several clusters, i.e., instances of a VLIW with a smaller number of issues, and apply the dynamic mechanism of Section 2 inside each cluster. For instance, Fig. 8 shows how the cluster-based approach is applied; to obtain an 8-issue VLIW, two parallel 4-issue VLIWs are used. Table 5 shows the benefits of using the proposed cluster-based approach in terms of number of inputs, outputs and complexity. The complexity of the switches with the cluster-based approach increases in a linear way. For a VLIW with $m \times n$ issues, the switch overhead in area equals to the number of clusters m, times the switch overhead, used internally in the n-issue VLIW. Since the clusters work in parallel, no further delay is inserted, no matter how much we scale the design. In this way, the complexity of the schedulers remains the same, i.e. equal to the schedulers complexity for n-issue VLIW. However, with the proposed cluster-based approach, the dynamic scheduling can be applied only within a cluster.

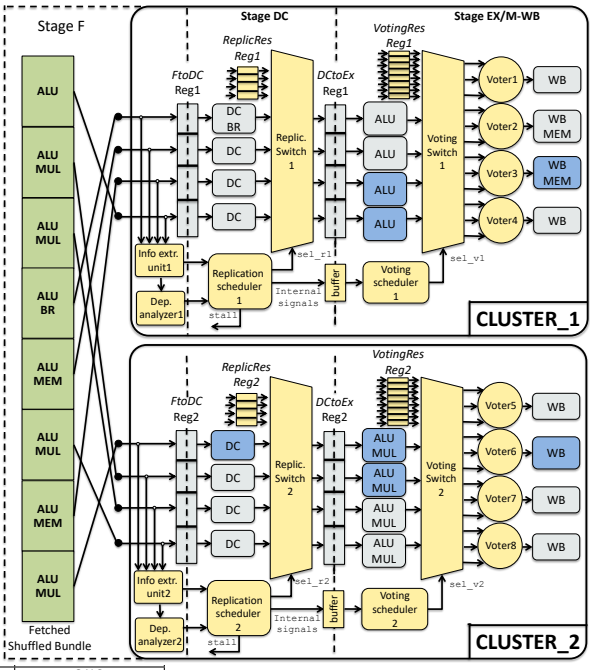


Figure 8: 8-issue VLIW with cluster-based 2×4 approach.

Considering that the compiler usually schedules

the instructions as dense as possible to occupy less area in the memory, it first tries to schedule the instructions in the first issues. Therefore, when the configuration of the large VLIW is provided to the compiler, it will first schedule the instructions to the first cluster, while the rest of the clusters may remain empty. This original schedule can lead to a negative impact on the performance of the proposed cluster-based approach. Instructions are gathered in the first cluster and the potential idle issues of the remaining clusters cannot be exploited, resulting in unnecessary stalls of the VLIW processor. To address this problem, we combine the cluster-based approach with a virtual VLIW configuration. The configuration of the FUs in the overall VLIW is done offline by shuffling FUs randomly. Then, instead of asking the compiler to generate code with the real VLIW configuration, we provide this virtual configuration. The goal is that the compiler generates the corresponding code, achieving a more balanced distribution of the instructions and NOPs into the clusters. In order to keep the area overhead low, a static de-shuffling of the instructions is applied during execution (during the fetch stage), providing the correct instructions to the clusters, based on the real FU configuration. Fig. 8 depicts such an offline shuffling of FUs (green box) and the static de-shuffling occurring during a global fetch, that distributes the 8 instructions from the virtual scheduling to the corresponding issues of the two 4-issue clusters. This static de-shuffling choice might have a negative impact to the benefits of the proposed approach. In order to not negativelly affect the proposed approach, the minimum required number of same FUs in a cluster should be aligned with the fault tolerance method. For instance, when the instructions are triplicated, at least three FUs of the same type should belong to the same cluster, if possible. In this way, an instruction can be triplicated and executed in parallel within a bundle. It is our future work to explore methods to identify a near-optimal configuration of the clusters and the shuffling/de-shuffling, based on the available FUs, the fault tolerance approach and the application instructions. The stall signal (*stall*) of each cluster has to be global for the whole architecture; the fetch and decode stages of all clusters are stalled, if at least one needs an extra time slot.

5 Experimental Results

5.1 Experimental set-up

As a case study, we use as baseline processor an in-house VLIW, which is compatible with the VEX compiler of HP [16] and is described by a high-layer C++ model. This C++ model can be synthesized to the gate level using Mentor Graphics Catapult High Level Synthesis (HLS) and Synopsys Design Compiler. The single cluster VLIW configuration used in the experimental section is the 1×4-issue, consisting of 4 ALU, 2 MUL, 1 MEM, 1 BR. Larger cluster VLIW configurations are multiples of the basic configuration, e.g. 2×4-issue consists $2\times$ the 1×4-issue configuration, except for the BR unit (i.e., 8 ALU, 4 MUL, 2 MEM, 1 BR). Table 6 depicts the combinational logic area of each FU type. As the execute stage has the largest combinational logic, it is more exposed and susceptible to faults, which motivates the focus of our approach to protect FUs.

Table 6: Combinational logic area of FU types (μm^2).

DC	DC with BR	ALU	MUL	MEM/WB
250	2,530	1,533	3,843	358

To evaluate our approach, we have used ten basic media benchmark kernels with different characteristics from the MediaBench suite [17]. We selected Mediabench as a workload to evaluate the behavior of the proposed approach as it is the one of the popular multimedia benchmark suite. The behavior of the proposed approach depends on the number of idle issues left by the application and the compiler. We present the results for ten applications with different characteristics, e.g., as Table 7 shows we have low ILP and several multiple dependencies (adp_dec, adp_denc), high ILP and different dependencies (bcnt, dct, fft32x22s, matrix_mmul) and low ILP and less multiple dependencies (huff_ac_dec, motion,fir, crc). We expect that applications from other benchmarks suites will have a similar behavior

if their benchmark analysis is similar to the analysed ones. We compare the following VLIW processor architectures:

1. Original version, i.e., the unprotected processor.

2. TMR-3FUs version, i.e., a fault tolerant processor where the FUs of the execution stage are triplicated and concurrently execute the same instruction.

3. TMR-S version, i.e., a fault tolerant processor that triplicates instructions and dynamically schedules original and redundant instructions only in the current bundle, called Space scheduling. Extra bundles are added if instructions cannot fit in the current bundle. Such an approach is inspired by [10, 11], considering instruction triplication, instead of duplication, and leveraged in order to be applicable for heterogeneous VLIW processors.

4. TMR-S+T+CB version, i.e., a fault tolerant processor with the proposed approach, that triplicates instructions and dynamically schedules original and redundant instructions within a scheduling window of two instruction bundles, called Space & Time Cluster-Based scheduling.

The aforementioned architectures have been developed in C++ and synthesized using the Catapult High-Level Synthesis (HLS) tool (University Version 10.0b/273613). Following this approach, we are capable of both simulating the processor and synthesizing a functional RTL design. Note that the presented results provide an upper bound in the area and power overhead, since HLS tools are not yet capable to produce the same high quality results as hand written RTL code [18]. However, since all the aforementioned vliw processor variations have been implemented using HLS, the comparative results should not be significantly different from hand written RTL code results. The generation of the gate-level netlist was handled by Design Compiler (Version J-2014.09-SP5-7) from Synopsys using a 28-nm ASIC library. The power analysis is given by Design Compiler, based on a statistical activity factor estimation assuming that every net toggles 10% of the time.

Section 5.2 analyses the benchmarks in order to motivate the proposed approach. Section 5.3 presents the evaluation results with respect to performance, area, and power consumption for the proposed intra-cluster dynamic mechanism, i.e., for the single cluster VLIW configuration, consisting of 4-issues. Section 5.4 evaluates the cluster-based approach for several clusters. Section 5.5 computes the reliability for the proposed approach.

5.2 Benchmark analysis

We analyze the binaries of the benchmarks to measure the i) possibility to execute instructions in parallel, by computing the average number of instructions per bundle (Instruction Level Parallelism - ILP), and ii) number of dependent instructions between two concurrent bundles.

Table 7 illustrates the obtained results for the single cluster configuration. We observe that $1.51 \leq$ ILP ≤ 2.85 (4-issue configuration), $1.75 \leq$ ILP ≤ 4.46 ($2 \times$ 4-issue configuration), $1.89 \leq$ ILP ≤ 5.58 ($4 \times$ 4-issue configuration) and $1.89 \leq$ ILP ≤ 7.4 ($8 \times$ 4-issue configuration). This implies that, in several bundles, there is a sufficient number of idle issues. We also observe that by doubling the number of issues, the ILP is not doubled. This behavior is due to the limited parallelization capability of the applications and to the heterogeneity of the FUs. Through the ILP metric, we can conclude that there are idle issues inside one bundle that can be explored for fault tolerance. Such an exploration is achieved by space scheduling (TMR-S).

To see if we can explore the next bundle (through time scheduling), we compute the percentage of the dependency occurrences, i.e., zero, one, two and three or more instructions are dependent between two concurrent bundles. For the majority of the benchmarks, in more than 50% of the total instruction bundles, two consecutive bundles have zero dependencies. It is quite frequent to have only one dependent instruction. Last, but not least, the case of multiple concurrent dependencies is rare. We can conclude that the proposed approach will affect less negatively the performance, since it can schedule the independent instructions to the next upcoming bundle, avoiding

Table 7: Benchmark analysis

Benchmark	1×4-issue					2×4-issue					4×4-issue					8×4-issue				
	ILP	Dep. Occurrence%				ILP	Dep. Occurrence%				ILP	Dep. Occurrence%				ILP	Dep. Occurrence%			
		0	1	2	3+		0	1	2	3+		0	1	2	3+		0	1	2	3+
adpcm_dec	1.77	60.6	35.2	4.1	0	2.28	50.3	37.7	9.6	2.3	2.93	64	33	3	0	3.31	76.075	22.5	1.425	0
adpcm_enc	1.82	58.7	35.9	4.9	0.5	2.41	48.6	39	9	3.3	3.04	63.05	32	4.05	0.9	3.44	68.6	26.675	4.725	0
bcnt	2.49	36	54	10	0	3.62	27	43.5	5.1	24.3	4.91	55.1	37.6	3.9	3.35	7.4	56.9	23.45	19.65	0
dct	2.22	53.9	30.1	7.7	8.3	3.31	45.4	28.1	7.1	19.4	4.62	50.55	20.7	18.95	9.85	4.91	60.525	30.9	7.975	0.625
fft32x32s	2.85	62.6	17.2	20.2	0	4.19	60.8	6.8	6.5	26	5.3	51.7	32	5.85	10.45	6.37	58.175	31.35	10.475	0
huff_ac_dec	1.51	59.9	36.1	3.6	0.5	1.75	40.7	50.7	7.9	0.7	2.14	70	29.75	0.2	0	2.17	84.075	15.925	0	0
motion	1.94	57	29.1	11.6	2.3	2.39	47.1	30	18.6	4.3	2.84	57.95	39.2	2.85	0	2.86	78.575	21.425	0	0
fir	2.09	62.3	29.8	7.9	0	2.5	51.6	36.8	11.6	0	2.55	77.75	22.25	0	0	2.56	83.15	16.85	0	0
crc	1.76	29.8	65.6	4.5	0.1	1.8	28.2	64.8	6.9	0.1	1.89	70.85	29.15	0	0	1.89	86.55	13.45	0	0
matrix_mul	2.61	51.5	32.2	16.4	0	4.46	21.1	62.7	16.2	0	5.58	41.1	58.9	0	0	5.6	63.675	36.325	0	0
Average	2.15	53.1	36.1	9.75	1.06	2.85	42.9	39.3	10.4	7.31	3.57	60.205	33.455	3.88	2.455	4.041	71.63	23.885	4.425	0.1

stalling the pipeline.

5.3 Intra-cluster Dynamic Mechanism

5.3.1 Analysis of mechanism components

Table 8 presents the area and the delay of each component of the proposed dynamic mechanism. The switches dominate the additional area inserted by the proposed mechanism (76.2% of the total area overhead). It should be clarified that such switches are required by any approach that performs dynamic instruction scheduling. However, it should be stressed that the proposed implementation of the voting switches results in significantly less area overhead than the original switch implementations, as presented in Section 3.1.2 and depicted in Table 5. The voters contribute to 9.7% of the area overhead. Voters and detectors are required by any fault tolerant approach to compare redundant instructions. Then, the specific control components, inserted by the proposed approach, contribute to the remaining 14.2%. Concerning the delay, the processor frequency is affected only by the components inserted inside the VLIW pipeline (*replication switch* in the DC stage with 0.16 ns) and *voting switch* and *voters* in the EX/M-WB stage with 0.33 ns and 0.27 ns, respectively). For the control parts, the largest observed delay is 2.19 ns for the *replication scheduler*, which, however, does not affect the clock frequency, since the critical path is given by the EX/M-WB stage and is equal to 2.62 ns.

Table 8: Area and delay of intra-cluster components.

Component	Area			Delay (ns)
	No. cells	%	%of total CPU	
Replication Switch	3,917.78	18.9	6.24	0.16
Voting Switch	11,891.57	57.3	18.93	0.33
Voters	2,007.68	9.7	3.20	0.27
Info extr. unit	88.13	0.4	0.14	0.12
Dep. Analyzer	303.55	1.5	0.48	0.13
Replic. scheduler	1,720.78	8.3	2.74	2.19
Voting scheduler	822.85	4.0	1.31	1.36

5.3.2 Performance

Execution cycles Table 9 depicts the execution cycles required to execute the benchmarks for the 4-issue configuration for all approaches. These values provide the impact on execution time, when all approaches are using the same frequency. We compute the Overhead (O) of TMR-S and TMR-S+T+CB, compared to the original/TMR-3FUs versions, and the Speed-Up (SU) of TMR-S+T+CB compared to TMR-S. The SU metric shows the impact on performance, when the scheduling window is extended from the current to the next instruction bundle. The overhead of TMR-S is similar among most of the benchmarks (except *crc* benchmark), with an average of 152.95%. The overhead of TMR-S+T+CB depends on the benchmark. The minimum overhead is 21.34% (*huff_ac_dec* benchmark) up to 139.16% (*matrix_mul* benchmark), with an average of 77.52%. Last, but not least. the speed-up of TMR-S+T+CB compared to TMR-

S is from 13.47% (*matrix_mul* benchmark) up to 43.68% (*huff_ac_dec* benchmark), with an average of 29.93%.

Note that the low number of execution cycles in some of the benchmarks is due to the following factors: a) the compiler schedules instructions in parallel to exploit as many idle slots as possible and, b) the architectures use a scratchpad memory with memory access of 1 cycle.

Table 9: Execution cycles (1 × 4-issue).

	Orig./TMR-3FUs	TMR-S		TMR-S+T+CB		
	Cycles	Cycles	O (%)	Cycles	O (%)	Gain (%)
adpcm_dec	386	998	158.55	613	58.81	38.58
adpcm_enc	409	1,066	160.64	652	59.41	38.84
bcnt	478	1,264	164.44	976	104.18	22.78
dct	1,288	3,779	193.40	2,404	86.65	36.39
fft32x32s	569	1,654	190.69	1,282	125.31	22.49
huff_ac_dec	1,101	2,372	115.44	1,336	21.34	43.68
motion	344	821	138.66	546	63.95	31.30
fir	6,852	16,890	146.50	11,714	70.96	30.65
crc	12,228	22,601	84.83	17,823	45.76	21.14
matrix_mul	11,142	30,795	176.39	26,647	139.16	13.47
Average			152.95		77.55	29.93

Maximum frequency The insertion of components to the VLIW datapath, in order to provide fault tolerance capabilities, increases the critical path. The upper part of Table 10 provides the critical path and the overhead (O), compared to the unprotected original processor. The critical path in all approaches is given by the EX/M-WB stage. The delay impact of the TMR-3FUs approach is 0.27 ns due to the need to insert voters for comparison in this stage. The delay impact of TMR-S and TMR-S+T+CB approaches is increased to 0.33 ns due to the voting switch required to correctly group the results for comparison. The replication switch is placed in the DC stage and does not increase the critical path. Then, the lower part of Table 10 depicts the worst case impact in execution time (in ns) for the different benchmarks, when each approach runs with its maximum possible frequency for the 4-issue configuration. TMR-3FUs has the least impact in execution time, i.e., on average 9.92%, but it has high area overhead, as shown in the next section. TMR-S increases the execution time on average 210.18% and TMR-S+t+CB on average 118.21%. Last, TMR-S+t+CB provides a speed-up compared to TMR-S on average 29.93%.

Table 10: Execution time at maximum frequency (1× 4-issue).

	Orig.	TMR-3FUs		TMR-S		TMR-S+T+CB		
	[ns]	[ns]	O (%)	[ns]	O (%)	[ns]	O (%)	Gain (%)
Critical path	2.62	2.88	9.92	3.22	22.09	3.22	22.09	0.
adpcm_dec	1,011.32	1,111.68	9.92	3,213.56	217.76	1,973.86	95.18	38.58
adpcm_enc	1,071.58	1,177.92	9.92	3,432.52	220.32	2,099.44	95.92	38.84
bcnt	1,252.36	1,376.64	9.92	4,070.08	224.99	3,142.72	150.94	22.78
dct	3,377.56	3,709.44	9.92	12,168.38	260.59	7,740.88	129.39	36.39
fft32x32s	1,490.78	1,638.72	9.92	5,325.88	257.25	4,128.04	176.90	22.49
huff_ac_dec	2,884.62	3,170.88	9.92	7,637.84	164.78	4,301.02	49.11	43.67
motion	901.28	990.72	9.92	2,643.62	193.32	1,816.08	101.56	31.30
fir	17,952.24	19,733.36	9.92	54,385.8	202.95	37,719.08	110.11	30.65
crc	32,037.36	35,216.64	9.92	72,775.22	127.16	57,390.06	79.13	21.14
matrix_mul	29,192.04	32,088.96	9.92	99,159.9	239.68	85,803.34	193.93	13.47
Average			9.92		210.18		118.21	29.93

5.3.3 Area and Power

Table 11 depicts the inserted area (number of required cells) and the power (in mW) results, compared to the unprotected original version. The TMR-3FUs approach has the maximum area and power, i.e., 44.60% and 35.96%, respectively. The TMR-S has the lower overhead, with an area increase of 15.56% and a power increase of 23.61%. The TMR-S+T+CB, compared to the unprotected processor, has an area increase of 23.54% and a power overhead of 32.87%.

5.3.4 Overall hardware architecture comparison

Table 11 also depicts the Power×Delay×Area (PDA) and the Power×Delay^2 products of the different hardware architectures. The power is based on statistical activity and the delay on the maximum frequency each architecture can achieve. The PDA product shows that the TMR-3FUs has the highest overhead, mainly due to the area overhead. TMR-S+T+CB has higher PDA than TMR-S, because it includes a small power and area increase, in order to significantly reduce the execution cycles of the benchmarks. The PD^2 product shows that TMR-3FUs

has the lowest overhead, making TMR-3FUs the best candidate when performance is the most important metric, sacrificing area and power. Although TMR-S+T+CB has the largest PD^2 overhead as architecture, this approach significantly reduces the number of cycles, compared to TMR-S. Taking into account also the time required to execute the benchmarks, the estimated average PD^2 for all benchmarks is 50% less compared with TMR-S.

Table 11: Area, Power, PDA and PD^2 product results (1×4-issue).

Approach	Area		Power		PDA		PD^2	
	Cells	O(%)	mW	O(%)	mW*ns*cells	O(%)	mW*ns^2	O(%)
Original	50,844	-	6.48	-	863209	-	44.48	-
TMR-3FUs	73,523	44.60	8.81	35.96	1865484	116	73.07	64.
TMR-S	58,819	15.68	8.01	23.61	1517071	76	83.05	87
TMR-S+T+CB	62,812	23.54	8.61	32.87	1741412	102	89.27	101

5.4 Cluster-based approach

5.4.1 Impact of clustering

First of all, we quantify how much the clustering impacts the proposed approach. To achieve that, we compare the obtained results of the proposed approach between 1×8-issue and 2×4-issue configurations. Regarding area, we estimate that the voting switch for 1×8-issue configuration requires 49,730 cells, leading to, at least, a total area 129,391 cells, increasing the critical path by 0.11 ns. This area overhead (62.43%) between the voting switch for 1×8-issue and 1×4-issue motivated the proposal of the cluster-based approach. Table 12 compares the number of execution cycles required to execute the benchmarks. The proposed cluster-based approach trade-offs performance improvement and area overhead, providing fault tolerance capabilities.

Table 12: Execution cycles of TMR-S+T+CB (1×8-issue and 2×4-issue).

Benchmark	2×4 VLIW	1×8 VLIW	Overhead (%)
adpcm_dec	443	388	14.18
adpcm_enc	493	426	15.73
bcnt	700	555	26.13
dct	1,623	1,519	6.85
fft32x32s	856	722	18.56
huff_ac_dec	1,090	1,029	5.93
motion	383	367	4.36
fir	8,291	6,969	18.97
crc	14,745	11,986	23.02
matrix_mul	14,053	12,196	15.23
Average			14.89

5.4.2 Performance

Table 13 depicts the execution cycles for two, four and eight clusters, i.e., 2×4-issue, 4×4-issue and 8×4-issue. For the 2×4-issue configuration, TMR-S has an average overhead of 102.01% and TMR-S+T+CB of 62.31%, compared to the original version. These overheads are reduced to 51.27% and 32.36%, respectively, for the 4×4-issue configuration, and to 26.17% and 13.60%, respectively, for the 8×4-issue configuration. Regarding the speed-up of TMR-S+T+CB compared to TMR-S, the average speed-up is 19.84% for the 2×4-issue configuration, 11.65% for the 4×4-issue configuration and 9.26% for the 8×4-issue configuration. Overall, as the number of issues is increased, the obtained speed-up, compared to the TMR-S, and the overhead compared, to the original VLIW, are decreased. This behaviour is explained due to the low application ILP and the large number of idle issues. In several cases, the instruction triplication is done within the current bundle, explored by both TMR-S and TMR-S+T+CB.

We observe that, in general, the speed-up of the single cluster is slightly higher compared to the speed-up of the configuration with two or more clusters. This behavior is explained due to:

1. The average *ILP*, compiled for the two-or-more cluster configuration, is low and, thus, there are more idle issues in the current bundle than the idle issues of the single cluster configuration. These are exploited by both TMR-S+T+CB and TMR-S. When the instructions are triplicated, the threshold (after which we need to add an extra time slot) is one instruction (for the single-cluster configuration), two instructions (for the two-cluster configuration), etc. For the single cluster, the ILP is almost 2, and thus, additional time slots are

Table 13: Execution cycles (2×4-issue, 4×4-issue and 8×4-issue).

	2 x 4-issue						4 x 4-issue						8 x 4-issue			
	Orig.	TMR-S		TMR-S+T+CB			Orig.	TMR-S		TMR-S+T+CB			Orig.	TMR-S		TMR
	Cycles	Cycles	O (%)	Cycles	O (%)	SU (%)	Cycles	Cycles	O (%)	Cycles	O (%)	SU (%)	Cycles	Cycles	O (%)	Cycles
adpcm_dec	302	612	102.65	443	46.69	27.61	258	385	49.22	309	19.77	19.74	232	292	25.86	248
adpcm_enc	323	662	104.95	493	52.63	25.53	280	413	47.50	344	22.86	16.71	252	317	25.79	273
bcnt	333	764	129.43	700	110.21	8.38	252	483	91.67	421	67.06	12.84	171	288	68.42	231
dct	872	2,144	145.87	1,623	86.12	24.30	670	1,118	66.87	999	49.10	10.64	636	811	27.52	718
fft32x32s	400	1,006	151.50	856	114.00	14.91	332	653	96.69	523	57.53	19.91	280	377	34.64	335
huff_ac_dec	951	1,525	60.36	1,090	14.62	28.52	925	1,025	10.81	966	4.43	5.76	912	938	2.85	917
motion	280	500	78.57	383	36.79	23.40	253	326	28.85	284	12.25	12.88	251	277	10.36	263
fir	6,709	10,877	62.13	8,291	23.58	23.77	6,007	7,031	17.05	6,728	12.00	4.31	6,007	6,549	9.02	6,247
crc	11,955	15,962	33.52	14,745	23.34	7.62	11,699	12,215	4,41	11,971	2,32	2.00	11,699	11,955	2.19	11,699
matrix_mul	6,533	16,408	151.16	14,053	115.11	14.35	5,236	10,453	99.64	9,227	76.22	11.73	5,236	8,116	55.00	7,532
Average			102.01		62.31	19.84			51.27		32.36	11.65			13.60	

Table 14: Area, Power, PDA and PD^2 product results (measurement method: (+) complete design, (†) all additional components, (*) estimation).

Approach	2 x 4-issue								4 x 4-issue								8 x 4-issue							
	Area		Power		PDA		PD^2		Area		Power		PDA		PD^2		Area		Power		PDA		PD^2	
	Cells	O(%)	mW	O(%)	mW*ns*cells	O(%)	mW*ns^2	O(%)	Cells	O(%)	mW	O(%)	mW*ns*cells	O(%)	mW*ns^2	O(%)	Cells	O(%)	mW	O(%)	mW*ns*cells	O(%)	mW*ns^2	O(%)
Original (+)	79,661	-	7.36	-	1,536,119	-	50.52	-	136,610	-	9.04	-	3235581	-	62.05	-	250,776	-	12.84	-	8,436,305	-	88.14	-
TMR-3FUs (+)	124,137	55.83	11.88	61.41	4,247,273	176	98.54	95	229,889	68.28	18.38	103.22	12,169,036	276	152.45	149	435,232	73.56	31.02	141.60	38,882,582	361	257.29	192
TMR-S (*)	95,258	19.58	9.03	22.69	2,769,779	80	93.63	85	168,380	23.26	13.13	45.26	7,118,871	120	136.14	119	317,975	26.80	20.73	61.45	21,225,022	152	214.94	144
TMR-S+T+CB (†)	103,598	30.05	9.77	32.74	3,259,131	112	101.30	101	184,484	35.05	13.73	51.85	8,156,148	152	142.36	129	346,523	38.18	21.33	66.09	23,800,101	182	221.16	151

needed in most cases. However, for two clusters, the ILP is almost 3, decreasing the probability of needing an extra time slot. For instance, the *crc* benchmark has the lowest speed-up in the two-cluster configuration, because the *crc* ILP changes slightly from the single-cluster to the two-cluster configuration and that is due to its high amount of one dependency occurrences. Hence, the required idle slots are available in the current bundle (both approaches explore them).

2. The application's code structure. For instance, an application with consecutive bundles without NOPs has similar behavior in both TMR-S and TMR-S+T+CB, since no idle slots exist to exploit. Such an example is *matrix_mul*, which achieves the lowest speed-up for the single-cluster case.

3. A high number of dependencies. If idle slots exist in next bundle, but instructions are dependent, they cannot be exploited. However, there are usually less than two dependent instructions (see Table 7). Thus, the remaining independent instructions can use the idle slots. That is why the *crc* benchmark (for single cluster) has high performance speed-up, even though 70% of the bundles have one or two dependent instructions.

5.4.3 Area and Power

Table 14 depicts the area and power for two, four and eight clusters (i.e., 2×4-issue, 4×4-issue and 8×4-issue), compared to the unprotected original version. For the original and the TMR-3FUs, the complete design has been implemented. For the TMR-S and TMR-S+T+CB, the additional components have been synthesized, all together, and added to the original design area/power. Estimated area and power results are provided for the TMR-S, based on the fact that TMR-S needs replication and voting switches, voters, simpler scheduling logic and no shadow registers. As the number of issues is increased, the area and power of the approaches are also increased. For the 8×4-issue configuration, the TMR-3FUs approach has the maximum area overhead of 73.56% and a power overhead of 141.60%. On the contrary, the TMR-S+T+CB, compared to the unprotected pro-

cessor, has an area overhead of 38.18% and a power overhead of to 66.09%. This overhead is ~ 50% less than the TMR-3FUs and ~ 10% more than the TMR-S. The TMR-S has the lower overhead, with an area increase of 26.80% and a power increase of 61.45%.

5.4.4 Overall architecture comparison

Table 14 depicts Power×Delay×Area (PDA) and the Power×Delay^2 products for two, four and eight clusters (i.e., 2 × 4-issue, 4 × 4-issue and 8 × 4-issue), compared to the unprotected original version. Similar to the 1x4-issue, the PDA product shows that the TMR-3FUs has the highest overhead, while the TMR-S+T+CB has higher PDA than TMR-S. The PD^2 product shows that, as the number of issues is increased, the PD^2 overhead is also increased for TMR-3FUs, strengthening the fact that a significant power penalty has to be paid in order to get the best possible performance. TMR-S+T+CB has bigger PD^2 overhead than TMR-S as an architecture. However, TMR-S+T+CB significantly reduces the number of execution cycles of the benchmarks. The estimated average PD^2 for all the benchmarks is reduced 33%, 26% and 23% in comparison with TMR-S for the 2×4-issue, 4×4-issue and 8×4-issue configurations, respectively.

5.5 Instruction Reliability Factor

In this work, we check the complete ISA of the VLIW by performing reliability measurements at the instruction level. We compute the Instruction Reliability Factor (IRF) for the original and the TMR-S+T+CB VLIW, defined as the probability of a fault in a processor's register to not lead to an error in the instruction output. IRF depicts the impact of a fault occurring in FtoDC, DCtoEx, Register File and PC. The IRFs of TMR-3FUs, TMR-S, TMR-S+T+CB are the same, they have the same protection sphere and apply instruction triplication.

To compute the IRF, the VLIW processors are extended with fault injection capabilities. We perform an exhaustive fault injection campaign at the micro-architecture level, for all valid opcodes of the Instruction Set Architecture (ISA) of VLIW processor. For each opcode, we flip all the bits of a pipeline register and all pipeline registers. For the input data of each instruction, we repeated the fault injection experiments 1,000 with random data. Hence, 14,100 total fault injections are performed per instruction (1,000 × 141 bits), which corresponds to confidence interval of 98.25%, considering an error margin of 0.01% [19]. For each experiment, a fault-free execution is performed to obtain the golden references. Then, a fault is injected by flipping one bit of a pipeline register. The process is repeated for all bits within a register and for all pipeline registers. After comparing the output with the golden reference, the result is categorized as: (i) Crash, the execution finishes unexpectedly, an exception is raised and the processor crashes; (ii) Correct, the instruction output and the processor state (registers, stack memory, PC, etc.) match the golden references; (iii) Output mismatch, the instruction output does not match the golden reference (but the processor state does); (iv) Processor state failure, the processor state does not match the golden reference (but instruction output does). Depending on this output, the bit where the fault is injected is categorized as Correct Instruction Output (CIO) (Cases ii and iv) and Faulty Instruction Output (FIO) (Cases i and iii). Then, the IRF_i in pipeline stage i is:

$$IRF_i = \frac{CIO\ bits\ of\ pipeline\ register\ i}{Total\ bits\ of\ processor\ pipeline\ registers} \quad (2)$$

Table 15 depicts the average IRF among similar instructions. The most vulnerable instructions of the original unprotected processor are the arithmetic and logic operations. With TMR-S+T+CB, they become up to 2.2x less vulnerable, compared to the original processor. The IRFs depend on the fault tolerance approach used (triplication) and the comparison that the voter performs (data result). As an exhaustive fault injection takes place, faults are inserted in other fields of the instruction than the data. Thus, IRFs are increased, but they are not equal to 1. This is due to the fact that the voters compare only the arithmetic result of the logical/arithmetic operations (FU result field in Table 2). If the voters are extended to check errors in the opcode and destination registers, instead

Table 15: Average IRF per instruction group.

Opcode	Orig.	Prop.	Opcode	Orig.	Prop.
Logic Operations			**Arithmetic Operations**		
CMPx	0.832	0.974	MPYx	0.707	0.974
CMPxi	0.899	0.974	ADD/SUB	0.447	0.974
AND/OR/NORx	0.667	0.974	ADD/SUBi	0.445	0.974
ANDC/ORCx	0.734	0.978	SHx	0.762	0.974
NOTx	0.679	0.974	SHxi	0.765	0.974
XORx	0.447	0.974	SHADD	0.456	0.974
Memory Operations			SHADDi	0.445	0.974
LDx	0.442	0.973	EXT	0.699	0.974
STx	0.346	0.975	EXTi	0.778	0.980
Control Operations			MOVI	0.715	0.974
BR/CALL/RT	1.000	1.000	NOP	0.950	1.000
CALLR/GOTOR	1.000	1.000			
BRF	1.000	1.000			
STOP	1.000	1.000			

of only the output results, the IRF would be close to 1. This extension is estimated to increase the size of the voters by approximately 22%. The delay is expected to be almost unaffected since the comparisons will occur in parallel. Note that, NOPs are never replicated in our design. Their IRF is improved due to the fact that TMR-S+T+CB uses a flag (sel_v), which is set by the Voting scheduler only when three instances of the same instruction are ready for commit. Hence, when a NOP is "executed", no instruction is committed. For this reason, NOPs never passes to the commit stage, and, in the IRF calculation, these faults are always masked.

5.6 Reliability of introduced hardware mechanisms

The probability of radiation hitting the hardware components, and thus introducing transients, depends on the component area. Thus, the larger the area the more possible it is for an exposed component to be affected by transient faults. In the proposed approach, the voting and replication switches have the biggest area among all added components, i.e. 76,2% of the proposed approach (and 25.17% of the total processor) as shown in Table 8. However, faults occurring in these hardware components are mitigated. Faults in the voting and replication switches will be detected by the voters, since faults will affect either the values passed to the voters or mix the values that will be voted. Therefore, the critical part of the proposed approach is the voters (3.20% area of the processor) which is the case for all fault tolerant approaches. Another part that is critical for the proposed approach is the scheduling (i.e. the replication and voting schedulers, the info extraction unit and the dependency analyser). If a fault occurs in this part, the pipeline may be corrupted and wrong instructions may be issued. However, all of these components occupy around 14% of the mechanisms inserted by the proposed approach (4.67% of the processor). In this work, we do not present any countermeasures for these critical components, since their area is small compared to the overall design, and thus, the probability of radiation affecting this part is low. A typical countermeasure that can be applied is to triplicate the critical parts, since they have low area overhead. Last, error correction codes is required as well for the register file and the memory, and this is an assumption for all the analysed approaches of this study.

6 Further discussions and limitations

Regarding the target VLIW processor, the proposed approach has been implemented considering a baseline VLIW processor of a pipeline depth equal to 3. The baseline VLIW processor with heterogeneous FUs is inspired by commercial VLIWs having large pipelines, e.g., Intel Itanium [20], but it is a simplified version with small pipeline depth. Such an implementation provides the worst case delay that the proposed approach can insert, since no pipelined version of the schedulers can occur with this 3-stage baseline processor without modifying its pipeline. Note that, increasing the VLIW pipeline depth will not impact the functionality of the proposed approach, since it will still exploit two instruction bundles, regardless the pipeline depth of the processor. Exploiting larger instruction windows will insert high overhead. The proposed approach has as goal to combine the benefits of VLIW and dynamic scheduling keeping the overhead low. To achieve that, the size of the instruction window has to remain as low as possi-

ble, which, however, provides some important gains. From the benchmark analysis, we have seen that already a high percentage of instructions between the current bundle and the next bundle are independent. Thus, the replication of instructions of the current and next bundle creates enough instructions to fill the VLIW issues.

Regarding the VLIW processor frequency, the worst delay of the proposed approach is given by the replication and voting schedulers, which are placed in parallel to the decode and execution stage of the baseline VLIW processor. The current implementation can be applied to a processor up to 456 MHz, without affecting the processor pipeline. Note that, for processors achieving higher frequencies, larger pipeline depths are required. In this case, more stages exist, and thus, more opportunities are provided to optimise the proposed implementation, e.g., taking advantage of the processor pipeline stages in order to divide and also pipeline the schedulers.

Regarding the target applications, the proposed approach focuses on applications that allow the existence of idle slots during their execution. Hence, applications with high ILP limit the advantages of the proposed approach, since most of the slots are occupied by original instructions, and thus, the proposed approach degrades to a typical triple execution of instruction bundles. Note that, this is the case for any approach that exploits idle slots. However, as other benchmark analysis have shown, high ILPs are not common and the number of NOPs is usually significantly large [21].

7 Related Work

Static and dynamic approaches have been proposed to replicate instructions and execute them on idle FUs of VLIW processors, in order to provide fault tolerance.

Static approaches are usually implemented by software. They replicate and schedule the instructions at design-time and additional instructions are inserted for comparison of the results. For instance, the approach presented in [22, 23] applies full duplication and full comparison in the compiled code, whereas the approach of [24] reduces the number of compared instructions. CASTED [25] proposes a compiler-based technique to distribute error detection overhead across core/clusters, which is applicable to tightly coupled cores and clustered VLIWs. Although static approaches do not require additional hardware, the code size, storage and energy consumption are increased. To reduce the additional instructions inserted in the code, approaches do not duplicate all instructions or implement the comparisons in hardware. For instance, a packing-oriented duplication is proposed, compared to the common sequential order, to maximize the number of duplicated instructions within the same performance overhead bound [26]. Instruction duplication and scheduling is performed by the compiler, while the comparison is performed by the hardware [27, 9]. In [28], the compiler encodes information in the instructions and a hardware mechanism decodes the information to runtime duplicate the instructions. In [13], instruction duplication and scheduling is done by the compiler and the comparison of results is done by the hardware. If an error is detected, the hardware adds a time slot and re-executes the instruction to another FU.

Dynamic approaches eliminate the need for high storage requirements and additional instructions. They are usually implemented through hardware that replicates and schedules the instructions during execution. A recent approach proposes dynamic recompilation, but this work addresses aging degradation [29]. Some dynamic approaches based on hardware avoid the need of dynamic scheduling. In [30, 10, 11], one-to-one coupling of the VLIW issues is applied, and thus, the duplicated instructions can use the schedule of the original instructions, given by the compiler. Partial instruction duplication is applied [30]. When no idle issue exists, the corresponding instruction is not duplicated. When an error is detected, instruction re-execution is applied. Full instruction duplication is applied in [10, 11]. When an instruction bundle has more instructions than the half of its issue-width, the bundle is divided into two, and a time slot is added. However, due to the lack of dynamic scheduling, there is a more-than-necessary negative impact on performance.

The remaining hardware dynamic approaches apply dynamic scheduling, assuming that VLIW issues include all FUs, thus can execute any instruction. However, when the VLIW consists of issues with different type of FUs, the existing schedulers are not applicable, as they ignore the type of FUs. To remove this limitation, existing approaches transform a heterogeneous VLIW to a homogeneous one, by inserting the missing FUs at each issue, e.g., [12]. With increasing number of VLIW issues and considering floating point FUs, these approaches lead to VLIW processors with significant area overhead. In [31, 32], the instructions are partitioned in groups in order their results to be directly compared after execution. However, one or two idle cycles are inserted for each instruction bundle. To reduce the negative impact, the use of spare FUs is explored.

Compared to the aforementioned approaches, this work proposes a hardware mechanism that replicates and schedules the instructions on heterogeneous VLIWs during execution, taking into account the number and the type of the FUs, and the algorithmic dependencies of the instructions, improving performance, while keeping area overhead low.

8 Conclusion

Due to application's ILP, the available resources of a VLIW processor are not always used. A cluster-based dynamic fault tolerant approach is proposed for heterogeneous VLIW data-paths. The intra-cluster hardware mechanism explores the idle slots in current bundle (space scheduling) and in the next bundle (time scheduling). To keep the hardware cost low, a hardware instruction scheduler is proposed. To enable scalability, a cluster-based approach is proposed, avoiding area and power overhead at the cost of a small decrease in performance. The processor behavior is explored with ten media benchmarks, with an average speed-up of 24.99% in performance and area and power overheads to be almost 10% with respect to existing approaches. Fault injection experiments show up to 2.2x instruction reliability improvement.

References

[1] J. W. McPherson, Reliability challenges for 45nm and beyond, in: IEEE/ACM Design Automation Conference, 2006, pp. 176–181.

[2] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, L. Alvisi, Modeling the effect of technology trends on the soft error rate of combinational logic, in: IEEE/IFIP Int'l Conf. Dependable Systems and Networks, 2002, pp. 389–398.

[3] R. C. Baumann, Radiation-induced soft errors in advanced semiconductor technologies, IEEE Transactions on Device and Materials Reliability 5 (3) (2005) 305–316. doi:10.1109/TDMR.2005.853449.

[4] T. Sudo, H. Sasaki, N. Masuda, J. L. Drewniak, Electromagnetic Interference (EMI) of System-on-Package (SOP), IEEE Transactions on Advanced Packaging 27 (2) (2004) 304–314. doi:10.1109/TADVP.2004.828817.

[5] V. Castano, I. Schagaev, Resilient Computer System Design, Springer Publishing Company, 2015.

[6] T. Heijmen, Radiation-induced soft errors in digital circuits – a literature survey (2002).

[7] J. Klecka, W. Bruckert, R. Jardine, Error self-checking and recovery using lock-step processor pair architecture, U.S. Patent 6,393,582 B1 (2002).

[8] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, Swift: Software implemented fault tolerance, in: Int'l Symp. Code Generation and Optimization, 2005, pp. 243–254.

[9] J. S. Hu, et al., Compiler-directed instruction duplication for soft error detection, in: IEEE/ACM Conference on Design, Automation & Test in Europe, 2005.

[10] A. L. Sartor, et al., Adaptive ilp control to increase fault tolerance for vliw processors, in: IEEE Int'l Conf. Application-specific Systems, Architectures and Processors, 2016, pp. 9–16.

[11] A. L. Sartor, et al., Exploiting idle hardware to provide low overhead fault tolerance for vliw processors, ACM Transactions on Journal on Emerging Technologies in Computing 13 (2) (2017) 13:1–13:21.

[12] A. L. Sartor, A. F. Lorenzon, S. Kundu, I. Koren, A. C. Beck, Adaptive and polymorphic vliw processor to optimize fault tolerance, energy consumption, and performance, in: ACM Int'l Conf. Computing Frontiers, 2018.

[13] M. Schözel, Hw/sw co-detection of transient and permanent faults with fast recovery in statically scheduled data paths, in: IEEE/ACM Design, Automation and Test in Europe Conference, 2010, pp. 723–728.

[14] R. Psiakis, A. Kritikakou, O. Sentieys, NEDA: NOP Exploitation with Dependency Awareness for Reliable VLIW Processors, in: IEEE Computer Society Annual Symposium on VLSI, 2017, pp. 391–396. doi:10.1109/ISVLSI.2017.75.

[15] F. Anjam, S. Wong, Configurable fault-tolerance for a configurable vliw processor, in: Int'l Conf. Reconfigurable Computing: Architectures, Tools, and Applications, Springer-Verlag, 2013, p. 167–178.

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

[17] C. Lee, et al., MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, in: IEEE Int'l Symp. Microarchitecture, 1997, pp. 330–335.

[18] S. Rokicki, E. Rohou, S. Derrien, Hardware-accelerated dynamic binary translation, in: IEEE/ACM Conference on Design, Automation & Test in Europe, 2017, pp. 1062–1067.

[19] I. Tuzov, D. de Andrés, J. Ruiz, Accurate robustness assessment of hdl models through iterative statistical fault injection, in: European Dependable Computing Conference, 2018, pp. 1–8.

[20] H. Sharangpani, H. Arora, Itanium processor microarchitecture, IEEE MICRO 20 (5) (2000) 24–43.

[21] G. Li, Y. Hou, J. Zhu, An efficient and fast vliw compression scheme for stream processor, IEEE Access 8 (2020) 224817–224824.

[22] C. Bolchini, A software methodology for detecting hardware faults in vliw data paths, IEEE Transactions on Reliability 52 (4) (2003) 458–468. doi:10.1109/TR.2003.821935.

[23] D. M. Blough, A. Nicolau, Fault tolerance in super-scalar and vliw processors, in: IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, 1992, pp. 193–200.

[24] J. G. Holm, P. Banerjee, Low cost concurrent error detection in a vliw architecture using replicated instructions, in: Int'l Conf. Parallel Processing, 1992.

[25] K. Mitropoulou, V. Porpodas, M. Cintra, CASTED: Core-Adaptive Software Transient Error Detection for Tightly Coupled Cores, in: IEEE Int'l Parallel and Distributed Processing Symp., 2013.

[26] Y. Ko, S. Kim, H. Kim, K. Lee, Selective code duplication for soft error protection on vliw architectures, Electronics 10 (15) (2021).

[27] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, Compiler-assisted soft error detection under performance and energy constraints in embedded systems, ACM Transactions on Embedded Computing Systems 8 (4) (2009) 27:1–27:30. doi:10.1145/1550987.1550990.

[28] J. Lee, Y. Ko, K. Lee, J. M. Youn, Y. Paek, Dynamic code duplication with vulnerability awareness for soft error detection on vliw architectures, ACM Transactions on Architecture and Code Optimization 9 (4) (2013) 48:1–48:24.

[29] A. Rahimi, R. K. Gupta, Hardware/Software Codesign for Energy Efficiency and Robustness:

From Error-Tolerant Computing to Approximate Computing, Springer International Publishing, 2021, pp. 527–543.

[30] A. L. Sartor, et al., A novel phase-based low overhead fault tolerance approach for vliw processors, in: IEEE Computer Society Annual Symposium on VLSI, 2015, pp. 485–490.

[31] Y.-Y. Chen, et al., An integrated fault-tolerant design framework for vliw processors, in: IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems, 2003, pp. 555–562.

[32] Y.-Y. Chen, K.-L. Leu, Reliable data path design of VLIW processor cores with comprehensive error-coverage assessment, Microprocessors and Microsystems: Embedded Hardware Design 34 (1) (2010) 49 – 61.