Mécanismes d'optimisation des performances des processeurs VLIW à tolérance de fautes

Rafail Psiakis

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THESE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Rafail PSIAKIS

Performance Optimization Mechanisms for Fault-Resilient VLIW Processors

Thèse présentée et soutenue à Rennes, le 21/12/2018
Unité de recherche : INRIA Rennes – Bretagne Atlantique et IRISA UMR 6074
Thèse N° :

Rapporteurs avant soutenance :

Alberto Bosio	Professeur à l'Ecole Centrale de Lyon - INL
Arnaud Virazel	Maître de Conférences HDR à l'Université de Montpellier - LIRMM

Composition du Jury :

Examinateurs :

Alberto Bosio	Professeur à l'Ecole Centrale de Lyon - INL
Arnaud Virazel	Maître de Conférences HDR à l'Université de Montpellier - LIRMM
Sébastien Pillement	Professeur à l'Université de Nantes - IETR
Georgios Keramidas	Chercheur à l'Université de Patras, Grèce
Angeliki Kritikakou	Maître de Conférences à l'Université de Rennes 1- IRISA

Directeur de thèse :

Olivier Sentieys	Professeur à l'Université de Rennes 1 – IRISA/INRIA

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Mécanismes d'optimisation des performances des processeurs VLIW à tolérance de fautes.

Mots clés : tolérance aux fautes, VLIW processeurs, exploitation des ressources inactives, optimisation de performance, injection de fautes, analyse de la vulnérabilité.

Résumé : Les processeurs intégrés dans des domaines critiques exigent une combinaison de fiabilité, de performances et de faible consommation d'énergie. Very Large Instruction Word (VLIW) processeurs améliorent les performances grâce à l'exploitation ILP (Instruction Level Parallelism), tout en maintenant les coûts et la puissance à un niveau bas. L'ILP étant fortement dépendant de l'application, les processeurs n'utilisent pas toutes leurs ressources en permanence et ces ressources peuvent donc être utilisées pour l'exécution d'instructions redondantes. Cette thèse présente une méthodologie d'injection fautes pour les processeurs VLIW et trois mécanismes matériels pour traiter les pannes légères, permanentes et à long terme menant à quatre contributions. La première contribution présente un schéma d'analyse du facteur de vulnérabilité architecturale et du facteur de vulnérabilité d'instruction pour les processeurs VLIW. La deuxième contribution explore les ressources inactives hétérogènes au moment de l'exécution, à l'intérieur et à travers des ensembles d'instructions consécutifs. La technique se concentre sur les erreurs légères. La troisième contribution traite des défauts persistants. Un mécanisme matériel est proposé, qui réplique au moment de l'exécution les instructions et les planifie aux emplacements inactifs en tenant compte des contraintes de ressources. Afin de réduire davantage le surcoût lié aux performances et de prendre en charge l'atténuation des erreurs uniques et multiples sur les transitoires de longue durée (LDT), une quatrième contribution est présentée. Nous proposons un mécanisme matériel qui détecte les défauts toujours actifs pendant l'exécution et réorganise les instructions pour utiliser non seulement les unités fonctionnelles saines, mais également les composants sans défaillance des unités fonctionnelles concernées.

Title: Performance Optimization Mechanisms for Fault-Resilient VLIW Processors.

Keywords: fault tolerance, VLIW processors, idle resource exploitation, performance optimization, fault injection, vulnerability analysis

Abstract: Embedded processors in critical domains require a combination of reliability, performance and low energy consumption. Very Long Instruction Word (VLIW) processors provide performance improvements through Instruction Level Parallelism (ILP) exploitation, while keeping cost and power in low levels. Since the ILP is highly application dependent, the processors do not use all their resources constantly and, thus, these resources can be utilized for redundant instruction execution. This dissertation presents a fault injection methodology for VLIW processors and three hardware mechanisms to deal with soft, permanent and long-term faults leading to four contributions. The first contribution presents an Architectural Vulnerability Factor (AVF) and Instruction Vulnerability Factor (IVF) analysis schema for VLIW processors. The second contribution explores heterogeneous idle resources at run-time both inside and across consecutive instruction bundles. The technique focuses on soft errors. The third contribution deals with persistent faults. A hardware mechanism is proposed which replicates at run-time the instructions and schedules them at the idle slots considering the resource constraints. In order to further decrease the performance overhead and to support single and multiple Long-Duration Transient (LDT) error mitigation a fourth contribution is presented. We propose a hardware mechanism, which detects the faults that are still active during execution and re-schedules the instructions to use not only the healthy function units, but also the fault-free components of the affected function units.

To my mother and father...

Résumé étendu en français

La conception des systèmes embarqués modernes est très complexe car ces systèmes doivent simultanément satisfaire à un certain nombre de critères qui se contredisent généralement. Les applications embarquées doivent généralement s'exécuter et fournir leur résultat dans un délai déterminé, le temps total d'exécution est donc essentiel. Une défaillance dun système embarqué peut avoir des conséquences fatales, la fiabilité est donc devenue un facteur très important. D'autre part, l'industrie des circuits intégrés s'efforce de réduire les coûts unitaires. La minimisation de la surface et de la puissance revêt donc une grande importance.

L'augmentation de la fréquence d'horloge est une pratique courante pour améliorer les performances du système et, par conséquent, le temps d'exécution de l'application. Cependant, en augmentant la fréquence d'horloge, la consommation d'énergie augmente également. Une solution à ce problème consiste à faire évoluer les architectures mono-cur classiques vers des architectures prenant en charge une sorte de parallélisme, telles que les processeurs VLIW *(Very Long Instruction Word)* utilisés dans cette thèse. La figure 1 présente un chemin de données VLIW simplifié pouvant exécuter jusqu'à quatre instructions simultanément, prenant ainsi en charge l'exécution en parallèle via lILP *(Instruction Level Parallelism)* du processeur. La phase dexécution a été amplifiée pour montrer le parallélisme inhérent à de tels systèmes.

Pour réduire la consommation électrique, une autre pratique courante consiste à réduire la tension et la fréquence de fonctionnement du système. Cependant, la réduction de la tension de fonctionnement, associée à la taille décroissante des transistors, rend les systèmes intégrés plus vulnérables aux erreurs et, par conséquent, moins fiables. Pour répondre à la demande croissante de fiabilité, les systèmes embarqués sont généralement conçus avec des capacités de détection et/ou de correction, atténuation et masquage des erreurs.

Cependant, augmenter la fiabilité implique généralement l'utilisation d'une forme de

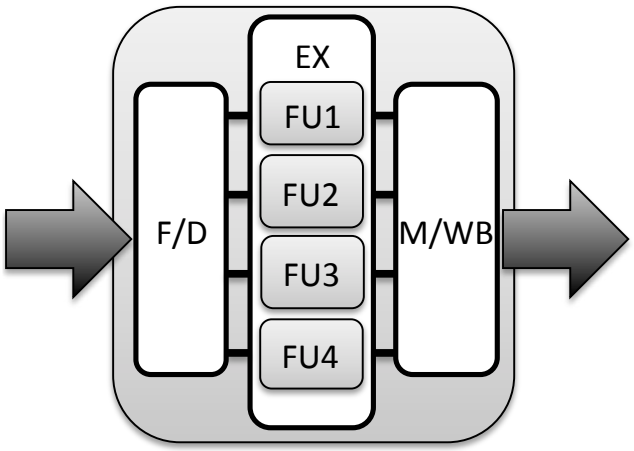


Figure 1: Architecture VLIW à quatre voies. Détail de la phase d'exécution avec ses quatre unités fonctionnelles parallèles.

redondance, qu'elle soit spatiale ou temporelle, ayant un impact négatif sur la surface (et donc le coût du système) et le temps d'exécution. Des techniques ont été développées pour assurer la fiabilité du niveau du transistor jusqu'au niveau de l'application, en utilisant une redondance matérielle (HW) ou logicielle (SW). La figure 2 présente schématiquement les différences entre la redondance HW (Fig. 2.b) et SW (Fig. 2.c) dans un scénario utilisant un processeur VLIW et un ordonnancement des calculs résultant dune compilation donnée (Fig. 2.a). Les approches utilisant la redondance matérielle étendent le matériel du système non protégé à lorigine en ajoutant des ressources supplémentaires pour exécuter les instructions redondantes. D'autre part, les mécanismes de redondance logicielle réutilisent les ressources disponibles pour ordonnancer des instances redondantes des instructions sur le système d'origine.

Dans cette thèse, nous explorons des moyens efficaces pour fournir des systèmes intégrés fiables grâce à la redondance logicielle, tout en limitant les surcoûts en surface et temps d'exécution. Nous montrons que les méthodes de redondance logicielle permettent daméliorer la fiabilité à moindre coût lorsqu'elles sont combinées à des architectures à redondance inhérente, telles que les processeurs VLIW. Nous proposons trois mécanismes matériels qui explorent les ressources inactives des processeurs VLIW. Nous effectuons tout dabord une analyse sur plusieurs *benchmarks* pour obtenir leur ILP moyen. Cette analyse montre que $1{,}51 \leq \text{ILP} \leq 2.85$ pour la configuration à quatre voies. Pour la configuration à 8 voies, nous observons que $1.75 \leq \text{ILP} \leq 4.46$. Cela implique que chaque cycle durant lexé-

cution comporte un nombre suffisant de *slots* inactifs à exploiter. Une analyse temporelle de dépendance entre les instructions a également été effectuée pour chaque application afin de détecter les ressources inactives potentielles pouvant être exploitées pour la tolérance aux fautes. Les résultats montrent que les cas avec zéro dépendance entre deux ensembles *(bundles)* consécutifs du VLIW est supérieur à 50% et que le cas avec exactement une dépendance est également assez fréquent (40%). Cela implique que les créneaux inactifs peuvent également être exploités de façon temporelle entre séquences dinstructions.

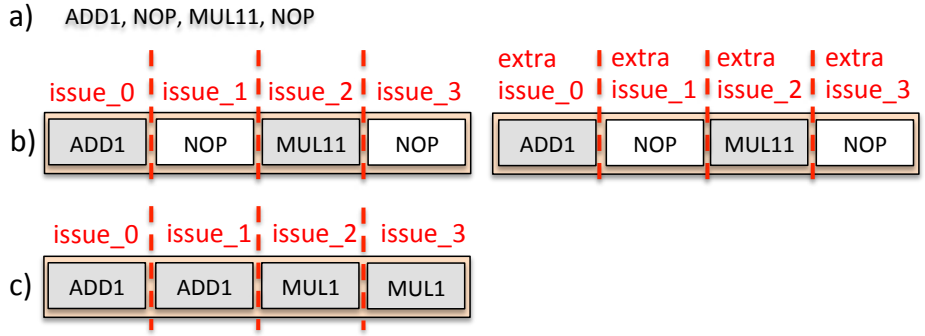


Figure 2: Redondance SW/HW dans un scénario VLIW.

Cette thèse présente ensuite une méthodologie d'injection de fautes pour vérifier et analyser la vulnérabilité des processeurs VLIW non protégés ainsi que les trois mécanismes matériels exploitant les ressources inactives pour traiter les fautes transitoires, permanentes et à long terme, ce qui mène aux trois contributions principales de la thèse.

La première contribution présente un environnement danalyse de l*Architectural Vulnerability Factor* (AVF) et de l*Instruction Vulnerability Factor* (IVF) pour les processeurs VLIW. Ces métriques définissent le facteur de vulnérabilité architecturale et micro-architecturale d'un processeur, quelle que soit la fréquence des occurrences de fautes. Lobjet de létude AVF et IVF dans cette thèse est de mettre en évidence les capacités de masquage de fautes des processeurs VLIW de type RISC et de trouver les parties les plus critiques de la conception qui doivent être protégées contre ces fautes. Pour cela, une méthodologie dinjection de fautes au niveau de différentes structures de mémoire est proposée pour extraire les capacités de masquage aux niveaux architecture et instruction du processeur. Un schéma de classification des défaillances de haut niveau est présenté pour catégoriser la sortie du processeur.

La deuxième contribution explore les ressources inactives hétérogènes au moment de l'exécution, à l'intérieur dun ensemble et entre plusieurs ensembles d'instructions consé-

cutifs. Pour ce faire, une technique dordonnancement des instructions optimisée pour le matériel est appliquée en parallèle avec le pipeline afin de contrôler efficacement la réplication et lordonnancement des instructions. Suivant les tendances à la parallélisation croissante, une conception ciblant les architectures VLIW clustérisées est également proposée pour traiter les problèmes de passage à léchelle, tout en maintenant un surcoût en surface et puissance raisonnable. La technique proposée accélère la performance de 43,68% avec un surcoût en surface et en puissance de ∼10% par rapport aux approches existantes. Les analyses AVF et IVF évaluent la vulnérabilité du processeur avec le mécanisme proposé. Les résultats montrent quen raison du mécanisme proposé et de la technique de réplication appliquée, les instructions les plus vulnérables de larchitecture protégée, cest-à-dire les opérations arithmétiques et mémoire en entier, sont jusqu'à 2,2x moins vulnérables que celles de larchitecture non protégée.

La troisième contribution traite des défauts persistants. Un mécanisme matériel, qui réplique au moment de l'exécution les instructions et les planifie aux emplacements inactifs en tenant compte des contraintes de ressources, est proposé. Si une ressource devient défaillante, l'approche proposée permet de réaffecter efficacement les instructions d'origine et les instructions répliquées pendant l'exécution. Les premiers résultats dévaluation de performance montrent un gain de performance jusquà 49% par rapport aux techniques existantes.

Afin de réduire davantage le surcoût en performance et de prendre en charge latténuation des erreurs simples et multiples de type transitoires longues *(Long-Duration Transients – LDT)*, une troisième contribution est présentée. Nous proposons un mécanisme matériel qui détecte les défauts toujours actifs pendant l'exécution et réorganise les instructions pour utiliser non seulement les unités fonctionnelles saines, mais également les composants sans défaillance des unités fonctionnelles concernées. Lorsque le défaut disparaît, les composants de l'unité fonctionnelle concernée peuvent être réutilisés. La fenêtre dordonnancement du mécanisme proposé comprend deux ensembles d'instructions pouvant explorer des solutions d'atténuation des fautes lors de l'exécution de l'ensemble dinstructions en cours et de l'ensemble d'instructions suivant. Les résultats obtenus sur l'injection de fautes montrent que l'approche proposée peut atténuer un grand nombre de fautes avec des surcoûts faibles en performance, surface et puissance.

Contents

Chapter 1

Introduction and Motivations

The design of modern embedded systems is very challenging, since these systems have to simultaneously meet a number of criteria that usually contradict one another. Embedded applications usually have to react and provide their result within a fixed delay, so the total execution time is essential. A failure of an embedded system can have fatal consequences, so reliability has become a very important factor. On the other hand, the Integrated Circuit (IC) industry has been striving towards unit cost reduction, so area and power minimization is of great importance.

Increasing the clock frequency is a common practise to increase system performance, and, thus, the application execution time [86]. However, by increasing the clock frequency, the power consumption is increased as well. A solution to this issue was the evolution of the classic single-core architectures to architectures supporting a kind of parallelism. The shrinking of the transistor size enables hundreds of millions transistors to be placed on a single chip reducing manufacturing cost and increasing integration and computer processing abilities. Three levels of parallel execution exist: a) Thread Level Parallelism (TLP), where different tasks are executed in parallel on an architecture with several processing elements (e.g. Multi-cores, Simultaneous multithreading (SMT) processors), b) Data Level Parallelism (DLP), where parallelism arises from executing essentially the same code on a large number of data (e.g. Single Instruction Multiple Data (SIMD) processors, vector processors and Graphics Processing Units (GPUs)), and c) Instruction Level Parallelism (ILP), where parallelism is explored by executing in parallel independent instructions (e.g. Very Large Instruction Word (VLIW) processors and superscalars). Architectures exploring TLP (such as General Purpose Processors (GPP)) are used in multi-application domains

where task parallelization is of critical importance and they introduce increased complexity and cost [8]. Architectures exploring DLP introduce large area and power overhead and they are useful only in case of applications with extensive data level parallelism (e.g. rendering applications [35], etc.). Architectures exploring ILP can offer parallelism with minimum complexity [68] being a very promising candidate for the domain of embedded systems striving for performance, without introducing high area overhead (e.g. Hexagon DSP [17], etc.).

To reduce power consumption, another common practise is to reduce the operating voltage and frequency of the embedded system. However, reducing the operating voltage in combination with the decreasing size of the transistors, makes the embedded systems more susceptible to reliability violations and, thus, less reliable [42], [76]. Reliability violations occur due to Process, Voltage, and Temperature (PVT) variations [74], circuit aging-wearout induced by failure mechanisms, such as Negative-Bias Temperature Instability (NBTI), Hot Carrier Injection (HCI) [49], radiation-induced Single-Event Effects (SEEs), such as Single-Event Upsets (SEUs) and Single Event Transients (SETs) [7], clock skews [58], thermal stress [79], electromagnetic interference, such as cross-talk and ground bounce [81], etc. These phenomena can cause errors that may affect the embedded system temporarily (soft errors), permanently (hard errors) or semi-permanently (intermittent errors) [13]. To satisfy the increasing demand for reliability, embedded systems are usually designed with error detection and/or error correction/mitigation capabilities.

However, to increase the reliability, it usually implies a form of redundancy, either spatial or temporal, impacting in a negative way on the area and/or the execution time. Several techniques have been developed to provide reliability from the transistor level up to the application level, using either Hardware (HW) or Software (SW) redundancy [56]. Approaches using HW redundancy extend the hardware of the original unprotected system by adding spare resources. Then, the same instructions are executed several times on the extra resources and their results are compared to provide error detection and/or correction [32]. As the instructions are executed in parallel, normally no execution time overhead is observed. However, the area overhead is significantly increased. On the other hand, SW redundancy mechanisms introduce redundancy through software modifications on the application running on the original unprotected system. They reuse the available resources to execute the fault tolerant instructions, and, thus, increasing the execution time [60].

In this dissertation, we explore efficient ways to provide reliability to the

embedded systems, while keeping the area and execution time overhead low. We expect that SW redundancy methods are able to achieve reliability with less cost when they are combined with architectures that have inherent redundancy, such as VLIW processors.

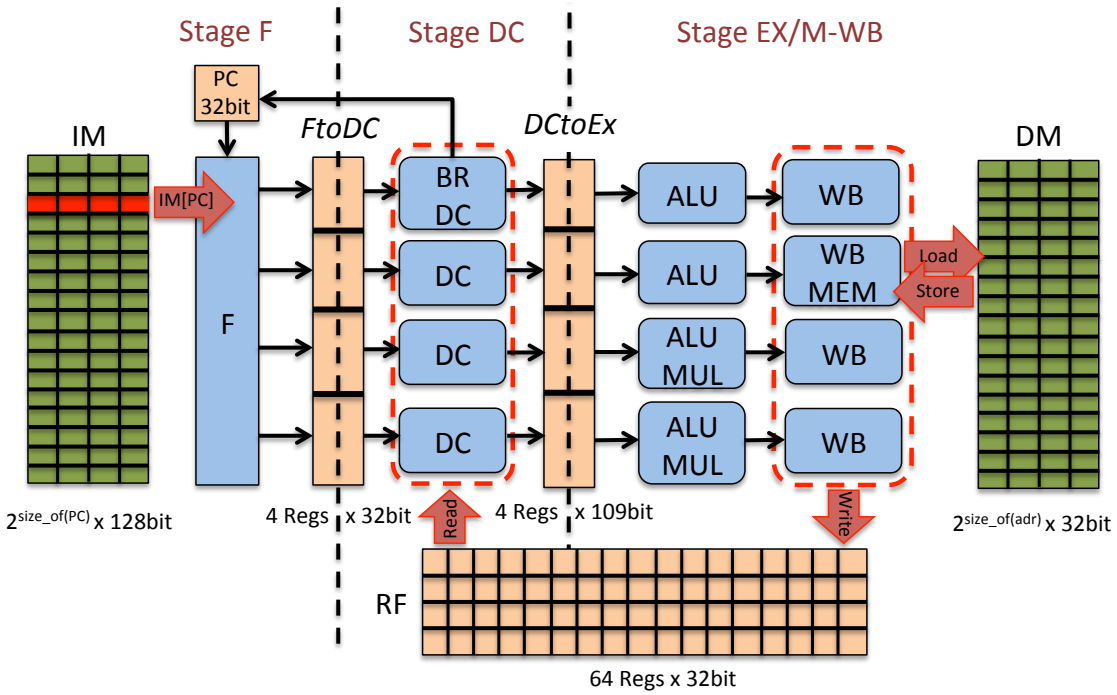


Figure 1.1: VLIW architecture with 4 issues.

1.1 VLIW Processors

An example of a VLIW architecture (similar to VEX [20]) is presented in Fig. 1.1. The figure depicts a processor's data path with four issues. The data-path consists of a 3-stage pipeline with Fetch (F), Decode (DC) and Execute/Memory-WriteBack (EX/M-WB) and, thus, it does not require a bypass logic. The processor has one Instruction Memory (IM) port, that fetches an $n \times 32$-bit word each cycle according to the current value of the Program Counter (PC). Thus, it can issue up to n instructions per cycle, where n is the number of issues of the architecture. These instructions are executed on:

- Memory units (MEM) that perform load and store instructions. There are as many memory units as data cache memory ports connected to the processor. More precisely, there should be at least $\lceil n/4 \rceil$ data cache ports to interact with the Data Memory (DM). In the presented example there is one MEM in the second issue.

- Integer and logic units that execute the common set of integer, compare, shift and logical instructions on registers or immediate operands. More precisely, the VLIW FUs are either complex FUs, which are able to execute all types of operations and simpler FUs, which cannot execute sophisticated operations, such as multiplications and divisions. In this configuration, there are $\lceil n/2 \rceil$ simple FUs, including only ALUs and $\lceil n/2 \rceil$ complex FUs, including ALUs and multiplication units. These FUs commit their result in the registers of a Register File (RF) comprising of 64 general-purpose 32-bit registers.

- Single branch unit that executes control instructions based on conditional results stored in registers. The control instructions can be conditional branches, unconditional jumps, direct and indirect calls, and returns. In this configuration the BRanch (BR) unit is only available in the first issue.

The Instruction-Set Architecture (ISA) considered in this dissertation follows the RISC ISA implementation and encoding (similar to the ISA of RISC-V processor [84]). The number of instructions executed in parallel per cycle depends on VLIW's FU parallelization capability (i.e. issue-width), the configuration of the VLIW (type of FUs used) and the intrinsic ILP available in each application. The instructions, which are issued and executed in parallel, form a bundle named *instruction bundle*. The VLIWs execute instructions in parallel, based on a fixed schedule determined when the programs are compiled.

For instance, suppose that the following matrix multiplication operation needs to be calculated:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E \\ G \end{pmatrix} = \begin{pmatrix} A \times E + B \times G \\ C \times E + D \times G \end{pmatrix} \tag{1.1}$$

As shown, for the calculation of this equation we need to perform four independent multiplications and two independent additions afterwards. Compilers can determine this parallelization opportunity and exploit processor's resources to obtain better performance, while respecting dependencies (e.g. the addition must be executed after the calculation of the multiplications, thus it is dependent from them). Fig. 1.2 schematically presents three consecutive instruction bundles B_i, B_{i+1} and B_{i+2} scheduled by the compiler targeting the architecture of Fig. 1.1 for the computation of the above equation. The compiler has scheduled two multiplication instructions at *issue 2* and *issue 3* for the bundle B_i in order to calculate the two multiplications needed, i.e. $A \times E$ and $B \times G$, for the first element of the output matrix. Two multiplication instructions are scheduled also at *issue 2* and *issue*

3 for the bundle B_{i+1} in order to calculate the two multiplications needed, i.e. $C \times E$ and $D \times G$, for the second element of the output matrix. In addition, the compiler also schedules an addition instruction at *issue 0* of B_{i+1} in order to calculate the $AE + BG$. At B_{i+2}, when the results of the two multiplications of B_{i+1} are ready, an addition instruction is scheduled to compute the last element $(CE + DG)$ of the output matrix. In order to perform this calculation, a normal processor without ILP exploitation capabilities would require one cycle for each of these instructions, i.e. six cycles in total, while in our case the cycles needed are only three.

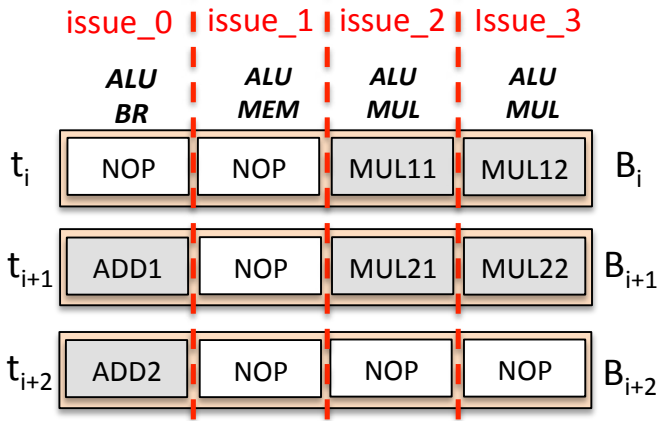


Figure 1.2: Three instruction bundles scheduled by the compiler for the computation of Eq. 1.1.

Since the order of execution of instruction and the decision of which operations can be executed simultaneously is handled by the compiler, any scheduling hardware (such as the instruction queue, reorder buffer, dependency-checking) that is needed when using out-of-order processors is avoided. Thus, VLIWs offer good computing power, high parallelization and performance gain with reduced hardware complexity and power consumption. For this reason VLIWs have been commercialized and used in several implementations during the past years (e.g., Intel Itanium [75], Trimedia CPU64 [83], Hexagon DSP [17], etc.).

1.2 VLIWs and Fault Tolerance

In the context of VLIW processors, the compiler is not always able to fill the entire bundle with instructions [1], because either there is no parallelism in the application or the processor configuration provides limited resources. Hence, idle slots are introduced in the form of No OPeration instructions (NOP). These idle slots can be used to execute redundant

instructions. In this way, processor's reliability increases since SW redundancy is applied, while the execution time overhead, introduced by the instruction replication, decreases. The SW redundancy approach applied on VLIWs can be implemented either in software or in hardware.

Approaches following a software implementation insert redundant instructions during design-time (i.e. applying a kind of redundancy in the original code itself) and/or during compile-time (i.e. the compiler is programmed to apply a kind of redundancy during compilation). They can efficiently explore the idle slots to schedule the redundant instructions without additional hardware control. However, the code size, the storage requirements and the power consumption are increased, whereas they cannot deal with dynamically changing faulty environments. For instance, the compiler duplicates the operations and schedules them in different FUs of a VLIW processor [11] or it exploits the idle FUs for soft error mitigation by adding a new time slot, whenever the idle FU exploitation in the current time slot is not possible [29]. For the comparison of the results of the replicated instructions extra comparison instructions should be executed [10, 9]. To reduce the number of executed instructions, software-implemented approaches are combined with hardware implementations. The instruction duplication and scheduling is performed in the software by the compiler, but the comparison is performed by the hardware. In case of an error, a simple HW instruction rebinding takes place so as to re-execute the instruction at the next time slot [72].

Hardware mechanisms replicate the instructions at run-time. Existing approaches maintain the compiler's result and explore the use of idle FUs in space, i.e., only inside the current instruction bundle. For instance, in [63], the idle issue slots inside the current instruction bundle are used for the execution of the duplicated instructions. If no idle slots exist, the instructions are not duplicated, which reduces the reliability of the processor. In [64, 65] the technique is extended by adding an extra time slot, so as to duplicate the instruction bundles that have more that half of its issue-width filled with instructions. However, the execution overhead is increased.

In this dissertation, we explore hardware mechanisms for SW redundancy, as they can efficiently deal with dynamically changing faulty environments compared to software implementations. We expect that by exploring the idle FUs of VLIWs, not only in space, but also in time through rescheduling of independent instructions of subsequent bundles, the reliability will be increased, while the execution time overhead will decrease.

1.3 Motivations

To motivate the benefits of the design of such mechanisms, we need to reply to two main questions:

1. **How many of these resources are actually available to be used for fault tolerance each cycle?**

2. **How many of each application's instructions can be potentially rescheduled to enable further idle FU exploitation?**

In order to answer to these fundamental questions, we analyzed ten basic media benchmarks from the MediaBench suite [36] with respect to their intrinsic parallelization capability. Similar media applications have been used by others in the literature to evaluate fault tolerant approaches [65], [10], [43]. In order to perform our analysis we utilized the VEX C compiler[1] provided by HP to compile our benchmarks. Initially, we perform an analysis of the binaries obtained by the VEX compiler in order to export the benchmarks' characteristics. Table 1.1 illustrates the average number of instructions per bundle, ILP, for the 4-issue and 8-issue VLIW configurations. We observe that $1.51 \leq \text{ILP} \leq 2.85$ for the 4-issue configuration. For the 8-issue configuration, we observe that $1.75 \leq \text{ILP} \leq 4.46$. This implies that in several bundles, there is a sufficient number of idle slots to be exploited. By duplicating the issue width, it does not imply a duplication of the ILP, because of the limited parallelization capability of the applications. Therefore, more idle slots exist in the 8-issue configuration. ILP metric provides us relevant information for the idle FUs exploitation in space, meaning that only the idle FUs within one bundle are explored.

When idle FUs exploitation occurs also in time, meaning that the FUs exploitation is performed among several bundles, we require to explore another metric, i.e. the percentage of the dependency occurrences per application. In this way, we implicitly know how many instructions are independent and, thus, they can be postponed in time so as to explore idle FUs in upcoming bundles. We analyzed the benchmarks to identify the number of

[1]The VEX compiler is derived from the Lx/ST200 C compiler, which is a descendant of the Multiflow C compiler. VEX compiler allows complex program compilation, custom instruction experimentation, and scalability. It targets C language and concentrates on acyclic scheduling (no software pipelining is supported). It supports partial predication and its only region shape is a trace (no superblocks nor treegions). It uses trace scheduling as its global scheduling engine. A programmable machine model determines the configuration of the target architecture. Some of the tuneable parameters that allow architecture exploration without having to recompile the compiler are for instance the number of clusters, the number of execution units, the issue width, and the latency of specific instructions.

the dependent instructions between two bundles. In Table 1.1 we present the dependency occurrence (%) for each application for zero, one, two and three or more simultaneous dependencies between consecutive bundles. As we observe, for most of the applications, the

Table 1.1: VEX Compiled Applications' Profiling

Benchmark	**4-issue**					**8-issue**				
	ILP	**Dep. Occurrence%**				**ILP**	**Dep. Occurrence%**			
		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
adpcm_enc	1,82	58,7	35,9	4,9	0,5	2,41	48,6	39	9	3,3
bcnt	2,49	36	54	10	0	3,62	27	43,5	5,1	24,3
dct	2,22	53,9	30,1	7,7	8,3	3,31	45,4	28,1	7,1	19,4
fft32x32s	2,85	62,6	17,2	20,2	0	4,19	60,8	6,8	6,5	26
huff_ac_dec	1,51	59,9	36,1	3,6	0,5	1,75	40,7	50,7	7,9	0,7
motion	1,94	57	29,1	11,6	2,3	2,39	47,1	30	18,6	4,3
fir	2,09	62,3	29,8	7,9	0	2,5	51,6	36,8	11,6	0
crc	1,76	29,8	65,6	4,5	0,1	1,8	28,2	64,8	6,9	0,1
matrix_mul	2,61	51,5	32,2	16,4	0	4,46	21,1	62,7	16,2	0

case of having zero dependencies between two consecutive bundles is more than 50%. The case with exactly one dependency is also quite frequent (∼40%), whereas the case of multiple dependencies is rather rare (∼15%). Considering the limited ILP of the applications and the limited number of dependent instructions between consecutive bundles, **we assume that the idle FU exploitation in time would be beneficial in terms of execution time overhead improvement.**

The need for increased reliability in combination with the above-mentioned findings regarding applications' characteristics motivate the development of efficient hardware mechanisms that exploit idle resources both in space (limited ILP of the applications) and time (independent instructions that can be postponed later). The next subsection presents an overview of the contributions of this work which are all driven by the observation of several idle resources in modern VLIW processors.

1.4 Dissertation Contributions

The main contributions of this dissertation with respect to the research topic of reliable VLIW processors are the following.

- **Vulnerability analysis through fault injection:** A fault injection software methodology is developed to test the vulnerability of the presented VLIW architecture. Both Architectural Vulnerability Factor (AVF) and Instruction Vulnerability Factor (IVF) analyses are performed to motivate the need for increased reliability against faults occurring in VLIW data paths. A high-level failure classification scheme is presented to categorize the output of the processor.

- **Time and space instruction rescheduling:** We propose a hardware mechanism capable of i) replicating at run-time the original instructions to provide VLIW processors with fault tolerance and ii) dynamically scheduling these original and replicated instructions to efficiently explore the idle FUs of current and upcoming bundles to improve execution time overhead. To achieve that, a hardware-optimized scheduling technique is proposed based on bit-wise logic. A cluster-based architecture is proposed, to support larger VLIW configurations and scalability. The detailed hardware implementation of the proposed mechanisms in the VLIW data path is presented and evaluated through extensive results on performance, area and power consumption. The AVF and IVF analysis is performed to evaluate the vulnerability factor of the VLIW architecture enhanced with the proposed mechanisms. Early results of this work have been published in [Psi17a], whereas an extended version has been submitted for publication in [Psi19b].

- **Instruction rescheduling for persistent errors**: When errors become persistent, the number of available FUs is reduced, thus affecting the execution time of the applications. A coarse-grained mitigation mechanism is proposed, that replicates and binds the instructions at run-time in order to provide error detection and mitigation. When a permanent error is detected, the instruction execution is modified to avoid the faulty FU. Both instruction replication and binding explore the healthy FUs taking the limitations on the type and the number of resources into account. A set of evaluation performance experiments with respect to the execution time overhead are performed and show up to 49% performance gain over existing techniques. This work has been published in [Psi17b].

  In order to further decrease the performance overhead, while supporting dynamic faulty environments including single and multiple Long-Duration Transient (LDT) faults, we propose a fine-grained mitigation mechanism that detects the active faults during execution and excludes only the faulty components of the affected FUs and for as long as it is necessary. To achieve that, a fine-grained micro-architectural solution is proposed

that partitions an FU in components, where each component is an individual circuit that executes a group of instructions. Each FU component is enhanced with a Built-In Current Sensor (BICS) mechanism, so as to identify the exact location of the fault and the duration that the fault is active. An online fine-grained instruction re-scheduling mechanism is proposed that explores idle healthy FU components in the current and the next instruction execution. Finally, we perform exhaustive fault injection simulations (214K) varying the number of total faults, the number of concurrently occurring faults and the fault duration. The obtained results show a minor performance degradation ($\sim$ 9% for four concurrent faults) even for several occurring multiple long duration faults. This work has been accepted for publication in [Psi19a].

1.5 Thesis Organization

The dissertation is organized into six chapters including this one. Chapter 2 discusses the state of the art research works w.r.t. fault resilient VLIW processors. Chapter 3 presents the details of the VLIW architecture used in this dissertation, w.r.t. the architectural and instruction vulnerability factor. In Chapter 4, we present the second contribution of this dissertation, i.e. the time and space instruction rescheduling, whereas in Chapter 5 we present our third contribution, i.e. the fine-grained instruction rescheduling. Finally, the thesis concludes with an overview of the presented work and a discussion on future research directions in Chapter 6.

Chapter 2

Background and state of the art

In the following sections we present the state of the art concerning fault injection and fault mitigation methods on modern embedded architectures and more specifically VLIW processors. We present the work related to the aforementioned research area split in four sections. In the first section, we discuss the latest published works on fault injection and vulnerability analysis on modern embedded processors. In the following sections, we discuss about the state of the art techniques concerning transient, permanent, and long duration fault detection and mitigation.

2.1 Fault Injection and Vulnerability Analysis

According to [87], the fault injection techniques can be classified as: 1) Hardware-based, 2) Software-based, 3) Emulation-based, 4) Simulation-based, and 5) Hybrid methods.

Hardware-based fault injection techniques test processors in realistic experimental conditions (e.g., radiation chambers, thermal stress experiments). In [23] an approach for SET detection and measurement is proposed. Real fault injection experiments are performed targeting a custom test chip, which is irradiated with heavy ions and pulsed lasers, while it is triggered by random inputs. FIST5 [25], developed at the Chalmers University of Technology in Sweden, uses heavy ion radiation to create transient faults at random locations inside a chip when the chip is exposed to the radiation and can thus cause single- or multiple- bit-flips. Messaline [2], developed at LAAS-CNRS, in Toulouse, France, uses both active probes and sockets to conduct pin-level fault injection. Messaline can inject stuck-at, open, bridging, and complex logical faults. Although such approaches are very accurate, a drawback is that they require the physical implementation of the device under test and

that the tested chip is heavily stressed, being defected after the experiment.

Software-based fault injection techniques are applied at the application level and mainly focus on applications' masking capabilities. Due to application-level fault masking (operation-level fault masking, fault masking due to fault propagation, and algorithm-level fault masking), the manifested errors may have zero impact on the application outputs. In [26], an application-level fault modelling is proposed. The proposed method models the behaviour of the faults which have already been manifested in the application-level. In [67], software-implemented fault injection is studied and a pitfalls interpretation is presented. The fault probability is approximated using the Poisson distribution, which is used to inject single errors in the memory. Ways to reduce the experiment effort are also studied using fault sampling and defuse, and pruning analysis. A high-level fault injector for Intel Xeon Phi, built upon GDB (the GNU debugger), is presented in [50]. Results show that 75% of the injected faults do not generate an observable error. Software-based techniques cannot inject faults into locations that are inaccessible to software and, thus, they have limited accuracy. On the other hand, since they are applied at high level, they require minimum simulation time and they can be easily implemented.

Emulation-based fault injection has been presented as an alternative solution for reducing the time spent during simulation-based fault injection campaigns. It is based on using Field Programmable Gate Arrays (FPGAs) for speeding-up fault simulation. This technique allows the designers to study the actual behavior of the circuit in an application environment, taking real-time interactions (e.g. I/Os) into account. However, when an emulator is used, the initial architecture or algorithmic description must be synthesizable to an FPGA design.

In simulation-based fault injection, there is a simulator that simulates the hardware as well as the injected faults. For instance, in [57], the authors propose analytical equations to model the propagation of a voltage pulse to flip flops. Random errors are injected at all possible nodes of a circuit gate-level netlist in order to calculate the Soft Error Rate (SER). Verification afterwards is performed using HSPICE. A tool for automated integration of fault injection modules is presented in [78]. The gate-level netlist is enhanced with injection modules after each gate and flip-flop. These modules are parametrized by the user for various fault specifications. In [80] a gate-level fault injection methodology for logical and electrical masking effects in case of reconvergent fanouts is presented. Delayed fault glitches are generated in case of reconvergent fanouts. In [21] the authors propose a simulator

implemented fault injection tool, where faults are induced by altering the logical values of the model elements during the simulation. They enhance MGSim [53] with a fault injection capability, thus the fault injection is performed at component-level. In [44] the relationship between glitches in the gates and latched errors in the flip flops is studied using mathematical models. The results of the proposed method are compared with HSPICE's results. The work in [61] introduces faulty behavior signatures that are computed after several gate-level injections. These signatures give the error occurrence probability of each output vector and are used by a saboteur that injects faults on a high level representation model of the tested design. This technique benefits from the high accuracy of the low-level injection and the simulation speed of a high-level injection. In [18], the authors measure the SER of a processor starting from a technology response model up to application masking. Only the injected errors from lower levels, which were latched by a memory element, are considered in the higher level and, thus, simulation time is reduced due to masking. In [52], a Monte-Carlo-based fault injection technique is proposed, taking into account multiple faults. In order to obtain accurate results, the injected nodes are selected according to their proximity to the error source in the place and route diagram.

AVF (Architecture Vulnerability Factor) was proposed by [47]. It concerns the probability of a soft error to result in an error of the program visible output. AVF estimation of modern microprocessors, using Statistical Fault Injection (SFI) for MBUs, is proposed in [41]. The presented method partitions the design into various hierarchical levels and systematically performs incremental fault injections to generate vulnerability estimates. Fault injection times are accelerated by 15x on average. A simulator-level fault injection framework is proposed in [82] that targets the Multi2Sim simulator. It measures the AVF of each application for a specific architecture, when single or multiple bit-flips into memory structures occur. In [4], the authors propose a new reliability metric named as Instruction Vulnerability Factor (IVF). Each instruction is tested with different operands in all the stages of a processor's pipeline under soft errors to measure the IVF. Results show that the execution stage is the most vulnerable part of the tested processor according to the average IVF of each stage. The study concludes with a technique for a fast and accurate AVF estimation using the IVFs of running instructions. Fault injection experiments at the Register Transfer Level (RTL) and Instruction Set Simulator (ISS) level are performed in [19]. The results are compared in order to find a correlation between the two approaches. Each application's instruction diversity is measured (i.e., the number of different opcodes)

in order to detect which areas in the RTL are actually affected. The authors in [46] propose a multi-level simulation that switches between ISS-level and RTL at run-time. The fault injection is performed when the simulation has passed to the RT level, evaluating the impact of soft errors in the pipeline of a RISC processor. The multi-level simulation benefits from the accuracy of the RTL-level simulation and the fast simulation time of the ISS-level simulation. Finally, simulation-based techniques provide accurate enough results, because the simulation models have most of the low-level hardware information while the run-time injection concept provides an accurate enough simulation of real scenarios. Additionally, when using simulation-based fault injection techniques, there is no risk to damage the system in use. A drawback of such approaches is the significant overhead in simulation and development time.

Few hybrid techniques also exist. A method combining software-based and simulation-based fault injection is presented in [27]. It uses pin-level forcing or generates interrupts to activate software fault injection. Hybrid methods benefit from the the advantages of both categories.

In this dissertation, we developed a simulation-based fault injection methodology and we used the AVF and IVF metrics to evaluate the VLIW architecture and the proposed hardware mechanism. To the best of our knowledge, this is the first AVF, IVF study on VLIW processors with heterogeneous issues. Valuable insights are obtained from this analysis, concerning the masking capabilities and the vulnerability characteristics of VLIW processors.

2.2 VLIW Processors Under Soft Errors

Software-based and hardware-based techniques have been proposed to take advantage of the additional resources in statically scheduled processors with inherent resource redundancy and to provide error detection and/or error correction. Software-based approaches replicate and schedule the instructions at design-time and additional instructions are inserted for comparison of the results. Software error detection approaches apply the duplication of the instructions after the compilation of the code and, thus, they can control where the original and replicated instructions are executed, e.g., in different function units. For instance, the approaches presented in [10, 9] apply full duplication and full compari-

son at the compiled code, whereas the approach of [28] reduces the number of compared instructions. CASTED [45] proposes a compiler-based technique to distribute error detection overhead across core/clusters applicable to tightly-coupled cores and clustered VLIWs. To reduce the number of additional instructions, software-based approaches are combined with hardware-based ones. The instructions duplication and scheduling is performed by the compiler whereas the comparison is performed by the hardware [30, 29]. In [37], a hardware/software-based technique is proposed, where the compiler encodes information in the instructions and a hardware mechanism decodes the information to run-time duplicate the instructions.

However, the aforementioned approaches do not provide any correction means. In software-based techniques no additional hardware control is required, but code size, storage and power consumption are increased. The code size increase has also a negative impact on system reliability, as more bits are present in the system, leading to a higher soft error rate [59]. Additionally, software-based techniques cannot deal with dynamic fault situations, meaning that they are not able to change the schedule according the current affected unit. To avoid these limitations, hardware approaches replicate the instructions at run-time using specific hardware mechanisms.

Hardware-based approaches eliminate the need of high storage requirements and additional instructions and are either applied to homogeneous or heterogeneous VLIWs. In [66], a dynamically adaptive homogeneous processor design is proposed, which is capable of reconfiguring the processor in order to achieve the best trade-off among fault tolerance, performance, and energy consumption. The applied fault tolerance technique exploits the spatial and temporal identical idle resources of a VLIW. In [15, 16], the authors propose a common approach for short transient and permanent faults. The instructions are partitioned in groups in order to be able to be directly compared, inserting one or two idle cycles for each instruction bundle. Due to the increased performance overhead, the use of spare function units is explored. Results are provided for one spare unit and homogeneous issues with ALU FUs.

Combination of a software and a hardware approach is presented in [73]. The instruction duplication and scheduling is performed by the compiler and the comparison of the instructions is performed by the hardware. In case of an error, re-execution takes place through a simple HW operation rebinding that adds an additional slot and re-binds the operation

to another FU. However, the VLIW also consists of homogeneous issue slots with FUs that can execute any type of operations.

Although hardware-based approaches that exploit the idle resources inside homogeneous VLIWs introduce a less complex control logic, they cannot be applied for heterogeneous VLIW data paths. In addition, their scalability is argued because an N-issue implementation would require N identical FUs, with N multipliers, N memory units, etc. On top of that, homogeneity in FUs does not usually reflect to realistic VLIW processor configurations with limited resources (e.g., [75]). Especially for architectures that employ floating point units, the area and power overhead can be very high. Existing approaches for heterogeneous VLIWs do not explore dependencies and idle slots among instructions bundles, adding unnecessary performance overhead.

In [63, 64, 65], one-to-one coupling of heterogeneous VLIW pipelines is applied and, thus, the duplicated instructions can use the schedule of the original instructions given by the compiler. In [63], error detection is applied through instruction duplication. If no idle slots exist, the instructions are not duplicated. When an error is detected, instruction re-execution is applied. In [64, 65], the technique is extended with ILP reduction. When a VLIW bundle has more than half of its issue-width filled with instructions, the bundle is divided into two and an additional time slot is added.

Compared with existing approaches, **in this dissertation we propose a hardware-based approach for heterogeneous VLIW data paths**, which explores at run-time the idle slots in space and time, i.e. both inside and among instruction bundles, to decrease performance overhead. Restrictions due to both the number and the type of resources and the dependencies are taken into account. The technique proposed in this thesis is applied to a VLIW with a combination of simple and complex FUs, but it can be easily extended to VLIWs supporting floating point arithmetic operations. Supporting floating point operations would require FUs with significant area/power overhead, where their replication to achieve fault tolerance is forbidden when resources are limited. Thus, the proposed technique would have a more significant impact since it provides fault tolerance without adding extra FUs.

2.3 VLIW Processors Under Permanent Faults

The approaches designed for permanent errors have to modify the execution of the program to avoid the use of faulty units. As these methods focus on permanent faults, the detection of the faulty unit is usually assumed to be done upfront. These approaches can be implemented either off-line in software or on-line in hardware.

Several software approaches exist. For instance, the compiler duplicates the operations and schedules them in different FUs of VLIW processors [11] or exploits the idle slots for soft errors [29]. The authors in [45] propose a compiler based technique to distribute error detection overhead to the available resources of architectures with abundant ILP, like VLIWs. The software approach of [31] stores several versions of the scheduling, where each scheduling is an alternative implementation for a given error. The permanent faults are detected offline and the program memory is modified adequately, in case of an off-line detected permanent fault. The work in [85] focuses on permanent faults in the registers of the VLIW and proposes a recompilation technique with a register pressure control to re-assign variables to fault-free registers. Concerning the software approaches, usually no additional hardware control is required, but code size, storage and power consumption are increased.

Several hardware approaches also exist. Few of them are capable of online identifying and handling the permanent errors. The following techniques add spare hardware resources for the error handling, thus the area and control overhead are increased. In [15], a spare function unit is added for error detection in VLIW, whereas single errors of one type are considered. In [16], spare function units are inserted to support Triple Modular Redundancy (TMR) and when not enough resources exist and the recovery is performed by re-execution of the faulty instruction. In [14, 51], a coarse-grained reconfiguration is proposed for a single permanent fault for each hardware class of ASICs based on the partitioning of the time and instruction bundle space. The technique has been extended for multiple faults by assuming one fault at each band and each reconfiguration of the scheduling can isolate one faulty unit. The fault detection is assumed to have been done in advance. In [77], the fault is detected by adding smaller ALUs and a reconfiguration logic is inserted in the execution stage to avoid the use of the faulty unit. Concerning the hardware approaches, usually no software modifications are required, but area and power consumption are increased due to extra hardware.

Some approaches combine software and hardware implementations. In [71], a software

repair routine modifies the instructions permanently in the memory. During start-up, a self-test takes place to identify the faulty slots. This information is used to change the schedule stored in the memory. If the repair routine fails, a simple hardware binding mechanisms adds time slots and sequentially maps the instructions that cannot be assigned to other slots. In [69], the approach is extended to cover pipeline registers, the register ports and the bypass logic. In [70], the approach is combined with adaptive software-based self-test, assuming though that the permanent errors have been already detected.

When permanent errors occur, either spare units have to be used or the executed program has to be modified through self-repair routine or through the use of several stored versions. However, these solutions introduce high area overhead for the additional resources, time overhead for the execution of the repair algorithm and storage overhead of the multi-versioning. **To address these limitations, a hardware mechanism is proposed (Chapter 5, Section 1) which at run-time replicates the instructions and schedules them at the idle FUs considering the resource constraints.** If a resource becomes faulty, the proposed approach efficiently rebinds both the original and replicated instructions during execution.

2.4 VLIW Processors Under Long-Term Faults

To the best of our knowledge, there is no technique targeting VLIW processors that deals with Long-Duration Transient (LDT) faults. Few approaches exist that focus on LDTs in general and they mostly focus on the error detection part at the transistor level using Built-In Current Sensors (BICS). In [5], a comparison of different BISCs can be found. Existing approaches usually stall the computation as long as the LDTs are valid or apply re-execution of the faulty instruction to single instruction processors. In [39], a BICS is proposed as a SET sensor connected directly to the bulk of transistors. In [6], a new lower area BICS scheme is propose using a single circuit to monitor at the same time both CMOS networks. When the SET is vanished, the computation restarts. In [38], a recomputing instruction mechanism is combined with BICS for transient errors on a micro-controller. In [33, 34], DMR is applied to detect any corruption of the application logic in a pipeline processor and a new micro-rollback scheme is applied to correct long duration transients, single event upsets and timing violation. Other approaches insert spare resources increasing

the area overhead. For instance, a fault tolerant technique with a double-mirror BISC is proposed in [24], allowing the detection of abnormal current consumption. If a defect occurs, redundant circuits are used.

Concerning VLIW processors, as most of the existing approaches are designed for transient and/or permanent errors, they are either not applicable or too pessimistic for LDTs. As mentioned in the previous sections, several software-based and hardware-based techniques have been proposed to take advantage of the abundant resources inside the VLIW data paths in order to provide error detection and/or error correction.

Software-based approaches for transient errors apply duplication of the instructions at the compilation time and. Thus, they can control where the original and replicated instructions are executed, i.e., in different function units, to support the detection of both transient and permanent faults [10, 9, 28, 30, 29, 73]. Although these techniques could be applied for LDTs, the performance overhead introduced by the continuous re-execution of the faulty instructions because of the long duration faults is rather significant. Hardware-based approaches for transient errors replicate the instructions at run-time using specific hardware [64, 65, 54]. Since no restrictions are applied to the on-line schedule, these techniques cannot detect persistent errors and, thus, they are not applicable to LDTs.

The approaches designed for permanent errors have to modify the execution of the program to avoid the use of faulty units. Approaches based on the modification of the execution of the program can be implemented either off-line in software or on-line in hardware. Software approaches such as [31, 71, 69, 70] assume that the detection of the faulty unit is done upfront, which is not possible in the case of LDTs. Few hardware approaches are capable of online identifying the permanent errors [55, 77, 15, 16], and, thus, could be applied for LDTs. However, the exclusion of the faulty unit is permanent leading to pessimistic results for LDTs.

To eliminate the performance overhead due to the re-execution or due to the pessimistic FU exclusion for LDTs on VLIW processors, we propose a fine-grained mitigation hardware mechanism combined with BISC FUs (Chapter 5, Section 2). During execution, this mechanism characterizes the components of each FU, identifies LDTs, reschedules the faulty instructions to the healthy FU components, and temporarily excludes the faulty ones. When LDTs vanish, the faulty FU components can be reused once again.

Chapter 3

Architecture's Vulnerability Analysis

Today's increased demand for reliable systems rises questions such as:

1. Given a processor's architecture with a given fault masking capability, what is the frequency of the errors a system experiences from its environment?

2. Is it necessary to incorporate a mitigation technique inside a given design, or the probability of a fault propagated to the user level is rather negligible (fault masking)?

3. Which part(s) of an architecture should be protected the most?

4. Which is the most suitable mitigation technique to be adopted?

Although the first question is not always easy to be answered since it depends on external to the system factors (e.g., radiation, PVT), the research community found a way to respond to the second and third questions by introducing two new metrics. These metrics define the architectural and micro-architectural vulnerability factor of a processor, regardless of the frequency of the fault occurrences. In the following sections, we explore the Architectural Vulnerability Factor (AVF) and the micro-architectural / Instruction Vulnerability Factor (IVF) of the VLIW architecture used in this dissertation, in order to point out its fault masking capabilities and find the most critical parts of the design that should be protected against faults.

3.1 Architectural Vulnerability Factor Analysis

A structure's architectural vulnerability factor (AVF) is the probability that a fault in a processor will result in a visible error in the final output of a program [4, 47]. For instance, an error in the offset part of a NOP instruction has zero impact to the executed application, it is considered as masked and, thus, it reduces the overall AVF of the processor. The AVF is measured only for the storage cells (bits) of an architecture. Adopting the classification of [4, 47] the important bits are the Architecturally Correct Execution (ACE) bits, while the remaining bits are un-ACE bits. The AVF is defined as the portion of the important bits, which are required for the correct calculation of the final output of a program, to the total number of bits and it is given by

$$AVF = \frac{ACE\ storage\ bits}{Total\ storage\ bits\ of\ the\ processor}. \tag{3.1}$$

Since the processor changes its state at each cycle, the ACE bits also change. Therefore, the AVF is calculated per cycle.

We define the following six classes that refer to the application's output and the processor's state after the complete execution of a program with one fault injected, as compared to the golden values obtained from an execution without faults:

1. **Correct**: The program is executed correctly. The program's output and processor's internal states (registers, stack memory, PC, etc.) match with the golden values.

2. **Execution Time Violation (ETV)**: The output and processor's internal states are as expected, but the program finishes later than expected.

3. **Crash**: Execution finishes unexpectedly. An exception is raised and the processor crashes.

4. **Hang**: Execution enters in an infinite loop.

5. **Application Output Mismatch (AOM)**: The program exits correctly, but its output does not match with the golden one.

6. **Processor's State Failure (PSF)**: The output of the program is correct, but there is(are) mismatch(es) in processor's internal state (registers, stack memory, PC, etc.).

When a fault occurs in the ACE bits, the result of the execution is either Crash, Hang or AOM. On the other hand, faults in the un-ACE bits result in Correct, ETV or PSF outputs.

A processor's simulator that simulates the behaviour of the VLIW architecture was developed during this thesis. The processor's model is developed in C++ and it is capable of executing vex binaries. The simulator is enhanced with: a) a non-synthesizable fault injection function, which injects faults at user specified injection points and random time-stamps, and b) a function which checks the output of the application and the state of the processor, compares them with the golden values, and categorizes the result in one of the six presented classes. The injection points can be in all the storage structures of the processor. The memory units (IM and DM) are excluded for better simulation performance under the assumption that ECC codes or similar techniques have been used for their protection.

The fault injection experiment is presented by Alg. 1. We execute once the binary file with our simulator without injecting faults to obtain the golden output values of the application and the golden state of the processor (line 1). Having the information about the cycles needed for one complete execution, we decide the number of cycles we will inject faults on (*inj_cycles*). Then, at each iteration, we choose randomly one unique cycle inside the valid range where the fault will be injected (line 3). For each of these cycles we iterate over all issues, all memory structures and all bits of each structure (lines 4,5,6).

Algorithm 1 AVF Fault Injection Experiment Algorithm

```
1:  ./vliw "binary_file" > gold_log
2: for (i = 0, i < inj_cycles, i++) do
3:     c[i] = unique_random(max_cycles, c, size_of(c))
4:    for each issue ($is) do
5:       for each struct ($s) do
6:          for each bit ($b) do
7:              ./vliw "bin" "gold_log" $is $s $b $c > inj_log
8:          end for
9:       end for
10:   end for
11: end for
```

For the VLIW architecture used here we have four issues; the structures and their bits are shown in Table 3.1 (refer to Fig. 1.1). Thus, in every iteration, we inject one fault (modelled as bit-flip) to one bit of one of the FtoDC, DCtoEx, PC and RF registers. The results are compared with the golden ones from the *gold_log* file and a report is generated and stored into the *inj_log* file, categorizing each injection to one of the six presented categories.

Table 3.1: Bit composition for the used VLIW architecture.

	FtoDC	**DCtoEx**					**PC**	**RF**
	instr	dataa	datab	datac	dest	opcode		
issues x struct_bits	4 x 32	4 x 32	4 x 32	4 x 32	4 x 6	4 x 7	32	64 x 32

Fig. 3.1 presents the per cycle AVF of the processor when it executes a matrix multiplication application, calculated from Equation 3.1. The whole execution of the application takes 30228 cycles, thus an exhaustive fault injection simulation injecting faults at each cycle would require several days. Small intervals, such as one cycle difference, leads to a more accurate estimation with the AVF being more sensitive to the instantaneous behaviour of the application. On the other hand, when the interval length is too large, a lot of AVF variations may be lost. To tackle this issue and decrease the simulation time, we randomly choose 1000 unique cycles to inject faults at. The periodic behaviour of the AVF observed in Fig. 3.1 with instantaneous changes from max to low values shows the existence of loops in the executed code. The average AVF of the matrix multiplication application is 0.0534.

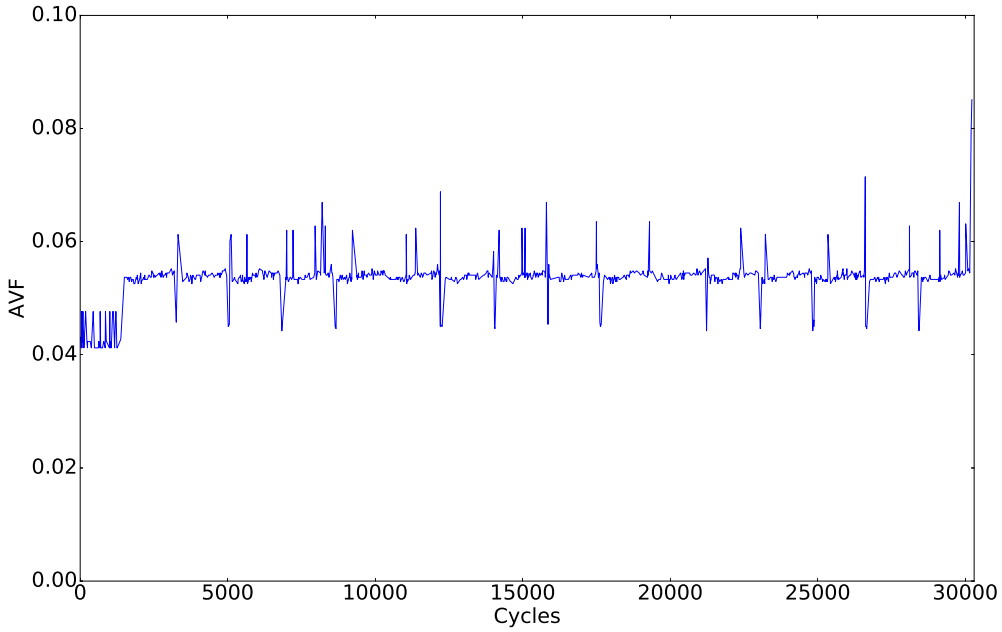


Figure 3.1: Per cycle AVF for VEX processor when executing a matrix multiplication

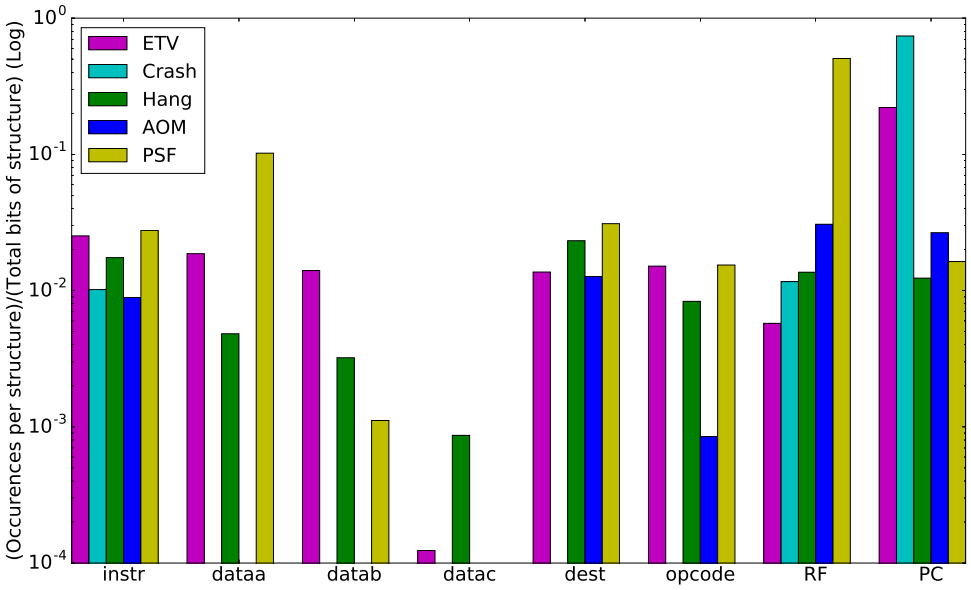
Fig. 3.2 presents the categorization of processor's AVF according to the aforementioned six classes when executing the matrix multiplication application. The results are normalized

in order to be independent from the ILP and are presented in logarithmic scale. We observe that for most of the structures, the output of the application is rarely affected. This is because of:

a) application masking (e.g. fault injection to a register that is not used by the application),

b) architectural masking (e.g. the MSBits of the output of a multiplication instruction are affected, but the executed instruction passes only the LSBits to the output registers),

c) logical masking (e.g. an AND operation between '0' and '0' has the same result as an AND operation between '1' and '0').



Class	instr	dataa	datab	datac	dest	opcode	RF	PC
ETV	0.025	0.019	0.014	0.0	0.014	0.015	0.006	0.221
Crash	0.01	0.0	0.0	0.0	0.0	0.0	0.012	0.74
Hang	0.017	0.005	0.003	0.001	0.023	0.008	0.014	0.012
AOM	0.009	0.0	0.0	0.0	0.013	0.001	0.031	0.027
PSF	0.028	0.102	0.001	0.0	0.031	0.015	0.506	0.016

Figure 3.2: Error occurrences per storage structure for the matrix multiplication (Normalized)

We observe that *ETV* is more prominent in PC error injections. That is because *ETV* refers to these cases where the execution is correct, but there is a cycle violation. An

example of *ETV* is a bit flip in a counter causing a drawback to a previous value or a bit flip in PC which drives the program to a previous stage of the execution which happens to be idempotent with the current one.

In our architecture, *Crash* errors are mostly frequent because of segmentation faults, and, thus they are more prominent in PC injections as well. Infinite loop errors or *Hang* errors usually happen when there is a violation in a checking condition of a loop. We observe that it is mostly fault injections in structures such as the PC, the instr, the RF and the dest that cause these types of errors. *AOM* and *PSF* errors refer to mismatches in the output of the application and the state of the processor respectively, and, they are more prominent in case of faults injected in the RF.

The goal of this analysis is to identify the most vulnerable parts of the VLIW architecture. We observe that the PC is the most vulnerable part of the architecture, but since it is only a 32-bit register, it can be easily protected with negligible hardware cost (e.g., ECC codes, radiation hardening). The next candidate for protection is the RF, which has the biggest number of *AOM* and *PSF* errors among all the other structures. *PSFs* are not considered as errors, since they do not affect the executed program, but they potentially corrupt other program's data, thus they should not be ignored. Errors in the RF might be either because of a direct fault injected in this storage structure or because of a transient error occurring in the execution stage and committed to the RF. Although error correction methods for memory elements of embedded processors have been studied thoroughly the past years [48, 22, 12], there is not much work regarding the faults occurring in the execution stage of (VLIW) processors.

To identify the most vulnerable parts of a processor, except of the vulnerability factor of each tested structure, one should also consider the probability of an error occurring in this particular structure. According to the empirical model provided in [76], this probability in case of soft errors is proportional to:

a) the neutron flux with energy > 1 MeV,

b) the area of the circuit sensitive to particle strikes,

c) the critical charge, and

d) the charge collection efficiency of the device.

Consequently, in order to measure this probability, we compare the logic area of each FU

of each pipeline stage, as depicted in Table 3.2. We observe that the area coverage of the execution stage components, including both simple and complex FUs, in comparison with the area coverage of all the stages of this architecture is greater than 70%. This observation in combination with the aforementioned findings, motivate the focus of the proposed approaches of this dissertation on the protection of the execute stage. Consequently, in the remaining chapters, we present novel techniques to mitigate errors mainly in the execution stage of VLIW processors.

Table 3.2: Area of pipeline stages (μm^2).

	DC	**DC_BR**	**ALU**	**ALU_MUL**	**WB**	**MEM_WB**	**Issue_Total**
issue_0	×	2,530	1,533	×	11	×	4074
issue_1	250	×	1,533	×	×	358	2141
issue_2	250	×	×	3,843	11	×	4104
issue_3	250	×	×	3,843	11	×	4104
FU_Total	750	2530	3066	7686	33	358	14423

3.2 Instruction Vulnerability Factor Analysis

A processor's Instruction Vulnerability Factor (IVF) is the probability that a fault in a processor's pipeline register will result in a visible error in the final output of the instruction under study [4]. For instance, an error in the offset part of a LOAD instruction will result in a wrong memory address calculation and, thus, in a wrong memory read. The IVF of this particular LOAD instruction is consequently decreased. A counter paradigm is a fault occurring in a value which is used by a logical operation (e.g., AND, OR, XOR). In Table 3.3 presented in [3], P_M shows the masking probability of a soft error occurring in one of the inputs of an logic gate. For example, when both inputs are 0, errors in each of the inputs of an OR or XOR gates are never masked. The IVF of these particular instructions is consequently high. P_F shows the error propagation probability to the outputs of a gate and is calculated by

$$P_F = 1 - \frac{\sum_{i=1}^{2^n} P_{M_i}}{2^n}, \tag{3.2}$$

where n is the number of inputs of the each gate.

Adopting the classification of [4] for IVF, the bits which are involved in the execution of an instruction are divided into two groups: Correct Execution of Instruction (CEI) and Un-CEI. If a fault in a pipeline register bit causes incorrect instruction output or abnormal

Table 3.3: Logical masking for three logic gates [3].

Inputs		**AND**		**OR**		**XOR**	
in_1	in_2	out	P_M	out	P_M	out	P_M
0	0	0	1	0	0	0	0
0	1	0	0.5	1	0.5	1	0
1	0	0	0.5	1	0.5	1	0
1	1	1	0	1	1	0	0
P_F			0.5		0.5		1

behaviour (e.g., segmentation fault), then the bit is classified as CEI. Otherwise the bit is an Un-CEI one. The IVF is defined as the portion of the CEI bits of the each pipeline stage to the total number of bits of the processor and it is given by

$$IVF = \frac{CEI\ storage\ bits\ of\ stage}{Total\ storage\ bits\ of\ the\ processor}. \tag{3.3}$$

We define the following four classes that refer to instruction's output and processor's state after the execution of an instruction with one fault injected, as compared to the golden values obtained from an execution without faults.

1. **Correct:** The instruction is executed correctly. The output and processor's internal states (registers, stack memory, PC, etc.) match with the expected golden values.

2. **Crash:** Execution finishes unexpectedly. An exception is raised and the processor crashes.

3. **Application Output Mismatch (AOM):** The instruction is executed, but its output does not match with the golden one.

4. **Processor's State Failure (PSF):** The output of the instruction is correct but there is(are) mismatch(es) in processor's internal state (registers, stack memory, PC, etc.).

The faults in the CEI bits are responsible for Crash and AOM errors. Faults in un-ACE bits result in Correct or PSF outputs.

Similarly to the AVF experimental setup, we developed a processor's simulator to simulate the behaviour of the VLIW architecture when it executes vex binaries of individual instructions. Since the IVF experiment is performed for one instruction at the time,

the VLIW pipeline is reduced to one issue. The simulator is enhanced with: a) a non-synthesizable fault injection function which injects faults at user specified injection points into the pipeline registers, and b) a function which checks the output of the instruction and the state of the processor, compares them with the golden values and reports in which class the output belongs to.

The fault injection experiment is presented by Alg. 2. For each valid opcode of the Instruction Set Architecture (ISA) we generate *rest_n0* random numbers from *0* to *max_rest* for the rest part of the instruction (line 2, 3). The DM and RF are also initialized with random values. The experiment is executed several times for the same opcode with a randomized instruction and processor configuration in order to test different input scenarios and, thus, avoid IVF miscalculation because of using specific inputs. Next step is to run our simulator for this particular generated instruction and this particular processor configuration and register the golden output values into a log file (line 5). For each of these generated instructions and configurations we iterate over all the pipeline storage structures (FtoDC, DCtoEx) and all the bits of each structure (lines 6, 7) and we inject one fault in the corresponding stage during the instruction execution (Fetch in cycle 0, Decode in cycle 1). The results are compared with the golden values from the *gold_log* file and a report is generated and stored into a log file (*inj_log*) categorizing each injected fault to one of the four above-mentioned categories.

Algorithm 2 IVF Fault Injection Experiment

```
1:  for each opcode ($op) do
2:      for (i = 0, i < rest_n0, i++) do
3:          r = random(max_rest)
4:          randomize(DM,RF)
5:          ./vliw $op $r > gold_log
6:          for each struct ($s) do
7:              for each bit ($b) do
8:                  ./vliw "gold_log" $op $r $s $b > inj_log
9:              end for
10:         end for
11:     end for
12: end for
```

For each instruction and for each bit of each structure of the VLIW architecture we generate 1000 test cases in order to create a uniform distribution of the input masking probability. For each injected stage the IVF is calculated from Equation 3.3.

Table 3.4 presents the IVF of each instruction of each stage of the processor. The table

depicts the IVF of all the logical, multiplication, memory, control and integer arithmetic operations of the ISA of a RISC processor (similar to the ISA of RISC-V processor [84]). For all the instructions, the IVF of the fetch stage is greater than the IVF of the decode stage, meaning that the decode stage is more vulnerable than the fetch stage. This is because the registers of the decode stage are more than the registers of the fetch stage and, thus, there are more bits and more fault injection points, which affect the correct execution of the instruction. The logical masking effect, which was discussed in this section and analyzed for specific instructions in Table 3.3, is prominent enough for the XOR instructions ($P_M = 0,\ \forall\ (in_1, in_2)$ and $P_F = 1$) which have the lowest IVF among all the other logical operations. We observe that integer arithmetic and memory operations have a lower IVF in the decode stage than other instructions (e.g. from the logical operations). This is explained because: a) these instructions use most of the pipeline registers in order to perform computations between registers and/or immediate values (integer arithmetic operations) or for the computation of the memory addresses (memory operations), and b) there is no logical masking in integer arithmetic operations, thus errors are usually propagated to the output. The multiplication operations are implemented in the same way, but since only a part of the multiplication result (Hi or Low) passes as a result of the operation, there is some masking introduced which increases the IVF when errors are injected into the decode stage registers. The control operations change the PC and redirect the execution to the desired position inside the program. They operate in the decode stage, thus any fault injected into the decode stage registers has zero effect to the output of these operations, thus the IVF is always one.

3.3 Conclusion

In order to identify the most vulnerable parts of the adopted processor architecture, a fault injection methodology has been developed for VLIW processors. For that, we performed experiments to measure the Architectural Vulnerability Factor (AVF) and the Instruction Vulnerability Factor (IVF) of the processor. Vulnerability factor analysis results in combination with measurements for the logic area of each FU of each pipeline motivated the focus of the proposed approaches of this dissertation (Chapters 4 and 5) on the protection of the execute stage.

Table 3.4: Per stage IVF for all operations of the ISA

Logical Operations		
OPCODE	**Fetch**	**Decode**
CMPEQ	0.882569	0.800358
CMPGE	0.880146	0.837737
CMPGEU	0.885730	0.842606
CMPGT	0.878686	0.840891
CMPGTU	0.880526	0.843241
CMPLE	0.880438	0.841562
CMPLEU	0.879891	0.842343
CMPLT	0.875088	0.833307
CMPLTU	0.875496	0.834365
CMPNE	0.884139	0.801453
CMPEQi	0.905533	0.912861
CMPGEi	0.912431	0.897628
CMPGEUi	0.912453	0.897759
CMPGTi	0.912445	0.897635
CMPGTUi	0.912438	0.897664
CMPLEi	0.912416	0.897460
CMPLTi	0.905153	0.890460
CMPLTUi	0.905153	0.890314
CMPNEi	0.905204	0.912475
AND	0.819788	0.690934
ANDi	0.815328	0.751708
ANDC	0.885117	0.704453
ANDCi	0.906555	0.776095
OR	0.819810	0.674073
ORi	0.815102	0.609774
ORC	0.870752	0.690058
ORCi	0.897701	0.765102
NOR	0.817555	0.671314
NORi	0.814088	0.606350
NOT	0.861365	0.678832
NOTi	0.861365	0.678832
XOR	0.819577	0.448993
XORi	0.766606	0.445401

Multiplication Operations		
OPCODE	**Fetch**	**Decode**
MPYLL	0.885080	0.704453
MPYLLU	0.884964	0.704453
MPYLH	0.894161	0.713409
MPYLHU	0.884964	0.704453
MPYHH	0.885080	0.704453
MPYHHU	0.892292	0.711672
MPYL	0.892146	0.711650
MPYLU	0.877898	0.697255
MPYH	0.886978	0.699015
MPYHU	0.899445	0.718847
MPYHS	0.892263	0.711650

Memory Operations		
LDW	0.766423	0.437956
LDHU	0.770204	0.441584
LDH	0.770036	0.441672
LDBU	0.770460	0.443547
LDB	0.770701	0.443606
STW	0.766431	0.248175
STH	0.766423	0.364971
STB	0.766635	0.423453

Control Operations		
CALLR	0.905117	1.000000
BR	0.817518	1.000000
BRF	0.978102	1.000000
RETURN	0.810219	1.000000
GOTO	0.824818	1.000000
GOTOR	0.905124	1.000000
CALL	0.810219	1.000000
STOP	0.948905	1.000000

Integer Arithmetic Operations		
OPCODE	**Fetch**	**Decode**
ADD	0.817518	0.445255
ADDi	0.766533	0.445387
SUB	0.819555	0.448956
SUBi	0.766628	0.445365
SRL	0.829642	0.764204
SRLi	0.838234	0.767715
SRA	0.829876	0.763569
SRAi	0.838387	0.767927
SLL	0.820591	0.757124
SLLi	0.829723	0.760131
SH1ADD	0.817518	0.445255
SH2ADD	0.817518	0.452555
SH3ADD	0.817518	0.459854
SH4ADD	0.817518	0.467153
SH1ADDi	0.766577	0.438022
SH2ADDi	0.781029	0.452555
SH3ADDi	0.773810	0.445336
SH4ADDi	0.773737	0.445255
ZXTH	0.877964	0.697277
ZXTB	0.878131	0.697496
SXTH	0.885343	0.704679
SXTB	0.877934	0.697255
ZXTHi	0.908504	0.776095
ZXTBi	0.908708	0.776234
SXTHi	0.914095	0.781730
SXTBi	0.908504	0.776095
MOVI	0.788321	0.715328
NOP	0.951402	0.949620

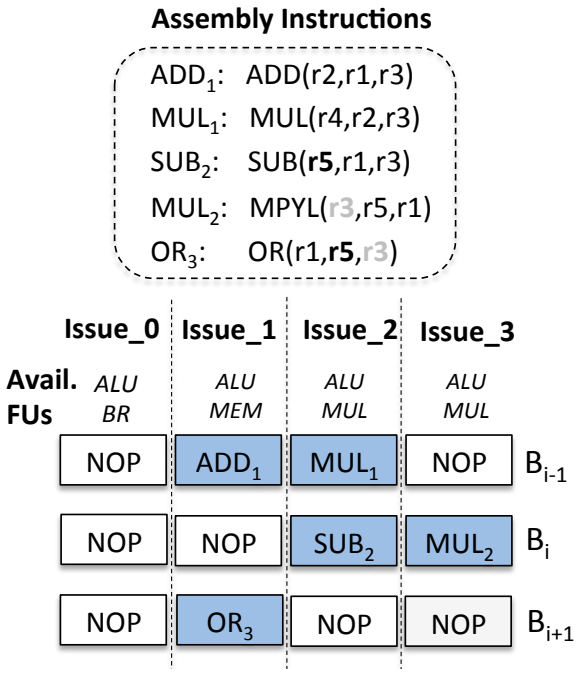
Chapter 4

Time and Space Instruction Rescheduling

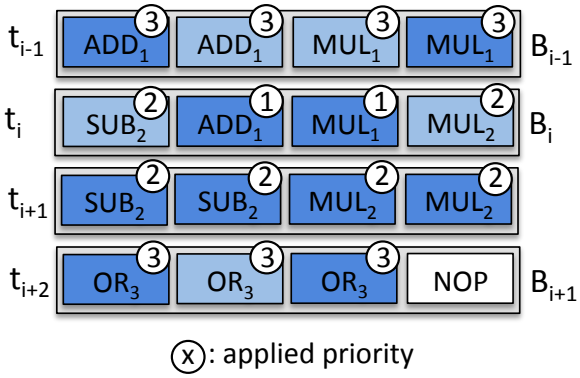
We propose a hardware mechanism that combines the benefits of the software redundancy and the hardware mechanisms for heterogeneous VLIW data-paths. The proposed mechanism is capable of both replicating at run-time the original instructions to achieve fault tolerance and dynamically scheduling them to efficiently explore the idle slots in time and space to improve performance, while preserving reliability. The scheduling exploration window is two instruction bundles and their potential additional time slots. To support the scheduling decisions, a hardware dependency analysis takes place between the two original instruction bundles. The independent instructions can use the idle slots of the next bundle and the potential time slots added for the execution of its replicated instructions. As the instruction scheduling is flexible, a more efficient idle slot exploitation takes place.

4.1 Running example

In this section we present the proposed approach through a running example. Fig. 4.1 depicts the proposed approach on the 4-issue VLIW of Fig. 1.1 with one Arithmetic Logic Unit (ALU) and one Branch unit (BR) in the Issue_0, one ALU and one Memory function unit (MEM) in the Issue_1, and one ALU and one Multiplication unit (MUL) in Issue_2 and Issue_3. Fig. 4.1a is the compiler's original schedule to the VLIW issues. Three consecutive instruction bundles of the original code are depicted, 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 also two instructions



(a) Compiler's original schedule.



(b) Schedule of the proposed approach.

Figure 4.1: Scheduling running example on an 4-issue VLIW.

(SUB_2, MUL_2) and two idle slots, while B_{i+1} has one instruction (OR_3) and three idle slots. The upper box of the Fig. 4.1a presents the corresponding assembly instructions with their implied registers. For instance, ADD_1 is an instruction that adds the values of the registers *r1* and *r3* and stores the result to the register *r2*.

Fig. 4.1b describes our approach for idle slot exploitation in consecutive bundles. We assume that the applied fault tolerance technique is the triplication of the instructions and, thus, every instruction has to be executed three times. The light blue boxes represent original instructions and the dark blue ones represent the replicated instructions. Initially, an instruction *dependency analysis* takes place (implementation details in Sections 4.4.1 and 4.4.2) between the instructions of two consecutive original bundles. If no dependency exists, the instructions that do not fit in the current bundle are allowed to be scheduled to the potential idle slots of the next bundle. In our example, when our approach is applied to the bundle B_{i-1}, none of the instructions (ADD_1, MUL_1) uses as destination register one of the source or destination registers of the instructions of the bundle B_i (*SUB2*, MUL_2). Hence, no dependency exists. On the other hand, concerning the bundles B_i and B_{i+1}, the instruction OR_3 reads the registers *r5* and *r3*, which are also used as destination registers by SUB_2 and MUL_2 of B_i. Hence, there is a dependency between the instructions of B_i and the instruction of B_{i+1}. Therefore, a potential parallel execution of these instructions is forbidden.

The dependency analysis results are used by the *replication scheduler* (implementation details in Section 4.4.3), which is responsible for the dynamic scheduling of the original and duplicated instructions. The scheduler operates according to three priorities 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 ones, and ③ the first copy of an instruction has a higher priority than the second copy of an instruction. These three priorities applied by the hardware scheduler to the considered instructions are illustrated in Fig. 4.1b by their respective encircled number.

In Fig. 4.1b, for t_{i-1}, one copy of each of the ADD_1 and MUL_1 is placed along with the original instructions occupying the two available idle slots, due to priority ③. At t_i, the scheduler is applied on the bundle B_i and the remaining instructions of the previous bundle are scheduled first (due to priority ①). Then, the original dependent instructions of the bundle B_i are scheduled (priority ②). Due to the dependency explained above, no exploration of the idle slots of B_{i+1} can take place and a new time slot has to be added.

At this new time slot t_{i+1}, the same scheduling policy is applied, filling the complete slot with the remaining dependent redundant instructions of B_i (priority ②). At t_{i+2}, there are not any remaining instructions from the previous bundle, thus OR_3 and its replicas are scheduled according to priority ③.

4.2 Overview of the proposed architecture

We use the 4-issue VLIW data-path of Fig. 1.1 and the triplication as fault tolerant method to illustrate the proposed approach. Fig. 4.2 depicts the original VLIW data path in blue color. The yellow color highlights the hardware components added to implement our approach. It is important to mention that the relative size of the blue and yellow boxes in the figure is not representative of the relative size of the actual hardware. The control components of the proposed fault tolerant mechanism are: 1) the *information extraction unit*, 2) the *dependency analyzer*, 3) the *replication scheduler*, and 4) the *voting scheduler*. The processing components are: 1) the *replication switch*, 2) the *voting switch*, and 3) the *voters*. 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* allocates previously decoded instructions (from the *ReplicRes* register) and the currently decoded instructions (from the decoders) to the pipeline *DCtoEX* register following the schedule provided by the *replication scheduler*. The *voting switch* allocates and groups the results of the previously executed instructions (from the *VotingRes* register) as well as the currently executed instructions (from the FUs) to the *voters* for correction following the schedule provided by the *voting scheduler*. As our main goal is to reduce the performance overhead, the processing components required to be added in the VLIW data-path are strategically placed, whereas all the control components of our architecture are designed to run in parallel with the main data path of the processor, so as to not affect the clock frequency.

4.3 Processing Components

To enable the dynamic scheduling of the instructions, our approach requires two switches and a voter to be added in line with the VLIW data path, as shown in Fig. 4.2. In the rest of the chapter, n will represent the number of the issues.

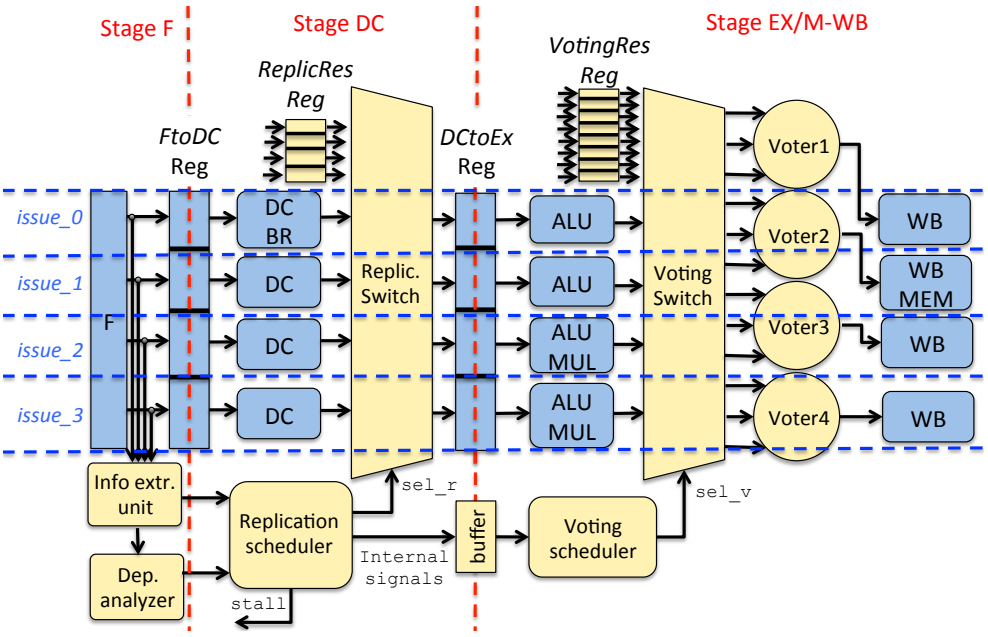


Figure 4.2: Original VLIW datapath (blue) enhanced with the proposed fault tolerant mechanism (yellow).

4.3.1 Replication Switch

The replication switch selects some of the $2 \times n$ possible inputs – i.e., the output of the decoders and the *ReplicRes* register – and places them to the n inputs of the pipeline register *DCtoEX*. The *ReplicRes* register stores an exact copy of the decoders' output in the previous cycle. Each input/output of the switch consists of 109 bits, i.e., the size of the decoded instruction. The decoded instruction is depicted in Fig. 4.3. Therefore, the input signal has a width of $109 \times 2n$ bits, while the output has a width of $109 \times n$ bits. For the switch implementation, $n \times (2n\text{-to-}1)$ multiplexers are required and the selection signal has $n \times log_2(2n)$ bits. Fig. 4.4 presents the implementation of the switch for a 4-issue VLIW.

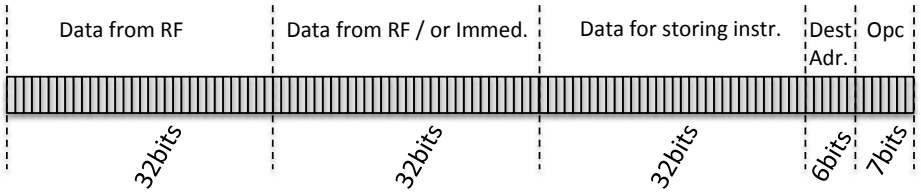


Figure 4.3: Decoded instruction

The input signal has a width of $109 \times 8 = 872$ bits. The circuit has 4×8-to-1 multiplexers and a 4×3-bit selection signal is required. The output is connected to the 4×109 bits

pipeline register *DCtoEX* shown in Fig. 4.2.

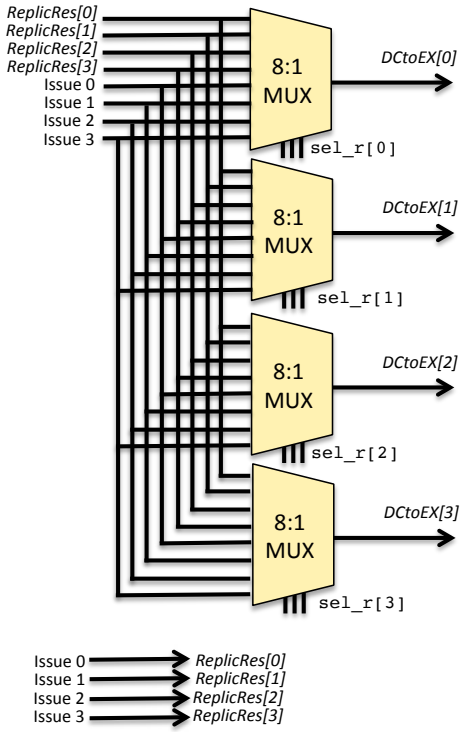


Figure 4.4: *Replication switch* implementation details

4.3.2 Voting Switch

This switch is placed after the FUs and it is responsible for the grouping of the results of the replicated instructions: a) into the voters and b) into the *VotingRes* register. The results of the instructions, whose replicated instructions have not been executed yet, are stored to the *VotingRes* register (Fig. 4.6b). The voting scheduler is responsible for the correct storing of the results in the *VotingRes* register. The results of the two instances of the same instruction are stored in adjacent positions in the *VotingRes* register, i.e., in the even position for the first instance and in the odd position for the second instance. Using this configuration, the first input of a voter, *Voter[i][0]*, is required to be connected only to the FUs. The second input, *Voter[i][1]*, is connected either to the FUs or to the *VotingRes[2i]* and the third input, *Voter[i][2]*, to either the FUs or the *VotingRes[2i+1]* register, $\forall i \in \{0, .., n-1\}$ (Fig. 4.6a). In this way the switch complexity and its corresponding area are reduced.

In case of the 4-issue VLIW, the switch is presented in Fig. 4.6. Each individual input and output has a size of 78 bits and it is structured as in Fig. 4.5. Fig. 4.6b depicts the

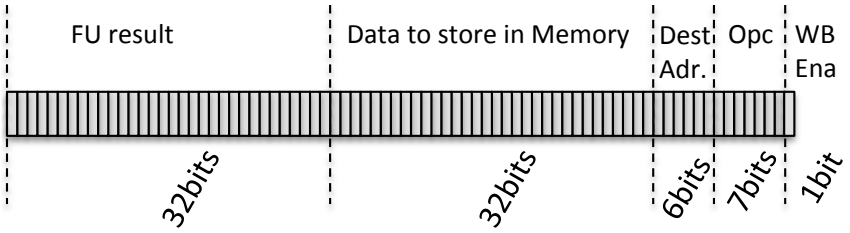


Figure 4.5: *Voting switch* I/O instruction.

part of the switch that stores back the results at the *VotingRes* register. It has 4 inputs (from the FUs outputs) that have to be connected to a register of 8 positions (*VotingRes[0]* to *VotingRes[7]*). Fig. 4.6a depicts the part of the switch required for one voter. It has 12 inputs (4 coming from the FUs outputs and 8 from the *VotingRes* register outputs), which have to be connected to 12 voters inputs. Finally, the select signal is 96 bits.

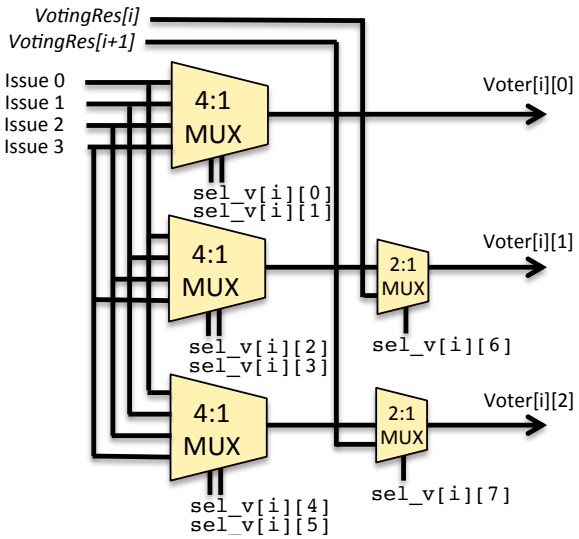
4.3.3 Voters

The voters are similar to the majority voters like the ones used in [40]. These voters detect and correct errors between the values that were grouped by the *Voting switch*. The voters compare the three 32-bit results of the EX stage, either coming from the FUs or the *VotingRes* register. They can be also easily extended to compare the rest of the values in the registers in order to provide fault tolerance for the control flow (opcode comparison) and the register address (destination register comparison).

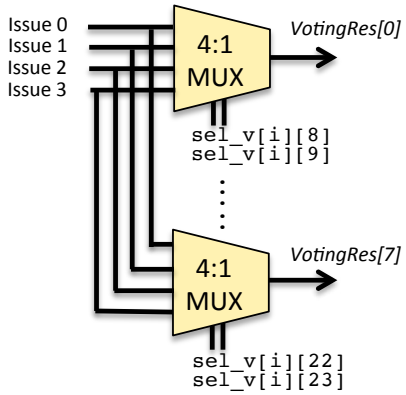
4.4 Control Logic Components

4.4.1 Information Extraction Unit

The *information extraction unit* is depicted in Fig. 4.8. It reads each instruction from the F stage and extracts the opcode, the destination, and the source registers. This early decoded information is stored in an intermediate internal register (*Info* register). The outputs are read from the *dependency analyzer* and the *replication scheduler*. To implement both the scheduling of the instructions to the *replication switch* and the grouping of the instruction to be voted in the *voting switch*, we need to introduce a logic that associates an instruction with an identifier. An Instruction Identifier (ID) corresponds to an instruction currently in the data path. These IDs provide the required information about the inputs of the *replication switch* in order to decide its output, through the selection signal vector `sel_r` of



(a) Part of the *voting switch* required for one voter.



(b) Part of the *voting switch* for storing back to *VotingRes* register.

Figure 4.6: Implementation of the *voting switch*.

the *replication switch*. They also provide the information to the *voting scheduler* in order to decide the grouping of instruction in the *voting switch* through the signal vector `sel_v`. Each of the four registers, *FtoDC*, *ReplicRes*, *DCtoEX* and *VotingRes*, is associated with an array of IDs, as depicted in Fig. 4.7.

The coding of the ID is depicted in Table 4.1. An ID value uses 5 bits. The 3 Most Significant Bits (MSBs) indicate the position of the instruction: the first bit indicates if the instruction belongs to the previous or to the current bundle and the next two bits show the original position of the instruction inside a bundle. Hence, the values from $0-3$ $(000-011)$ are assigned for the previous bundle and $4-7$ $(100-111)$ for the current bundle. The 2 Least Significant Bits (LSB) carry the information about the instruction type: NOP,

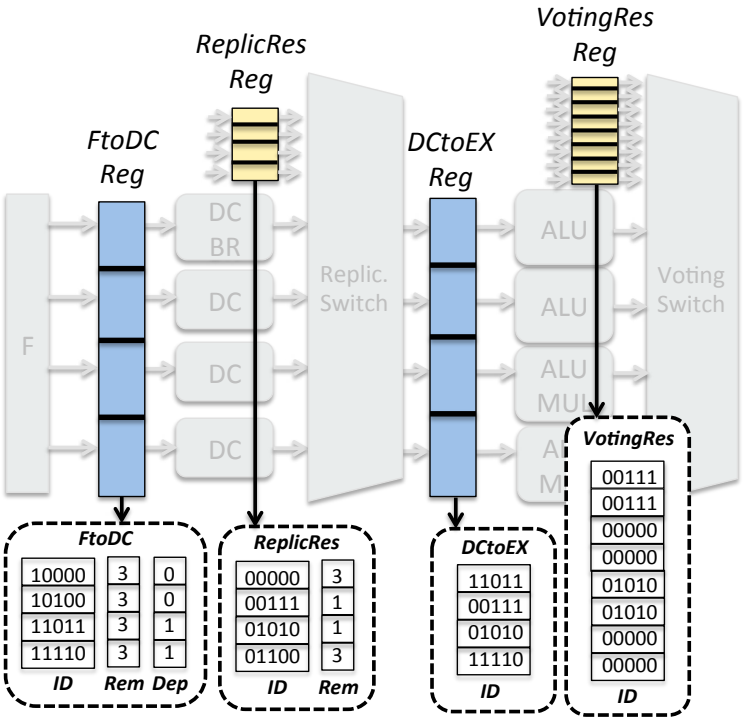


Figure 4.7: *ID*, *Rem* and *Dep* arrays of illustration example from Fig. 4.1b at time t_i.

memory, multiplication and ALU operations.

Table 4.1: ID encoding in the *information extraction unit*.

Bundle	Issue Number			Instructions		
	b_4	b_3	b_2	b_1	b_0	Type
B_{i-1}	0	0	0	0	0	NOP
	0	0	1	0	1	MEM
	0	1	0	1	0	MUL
	0	1	1	1	1	ALU
B_i	1	0	0	0	0	NOP
	1	0	1	0	1	MEM
	1	1	0	1	0	MUL
	1	1	1	1	1	ALU

Two additional register arrays are used to store the remaining instructions that need to be scheduled, i.e., the *FtoDC_rem* and *ReplicRes_rem*. For the bundle currently in the DC stage, the values of the *FtoDC_rem* are initialized to 3, since instruction triplication is applied as fault tolerance method. During scheduling, some of the instructions are executed on the FUs of the VLIW datapath. Therefore, the values of the *ReplicRes_rem* are updated to depict the new number of the unscheduled instructions.

In Fig. 4.7 we illustrate the *ID* and the *Rem* arrays of our running example presented in Fig. 4.1b at time t_i. The *FtoDC_ID* array is related to the current bundle B_i: the first element (10000) and the second one (10100) indicate that the instructions in the first and second slot of the current bundle are *NOPs*, the third element (11011) stands for an ALU operation and the last one (11110) stands for a *MUL* operation. The array *FtoDC_rem* is initialized to 3, meaning that each of the corresponding instructions have to get triplicated. The *ReplicRes_ID* array is related to the remaining instructions from the previous bundle B_{i-1}: the first element (00000) indicates that the first slot is a *NOP*, the second one (00111) stands for an ALU instruction, the third one (01010) stands for a *MUL* operation and the last one (01100) stands for a *NOP*. At the time t_{i-1} two ADD_1 and two MUL_1 instructions have been scheduled (Fig. 4.1b), so the *ReplicRes_rem* array has been updated accordingly by setting the corresponding values to 1.

4.4.2 Dependency Analyzer

The *dependency analyzer* in Fig. 4.8 decides about the dependencies between two subsequent bundles. To do so, it reads the opcode, the destination and the sources of each instruction of each bundle. Three possible dependency cases may exist: 1) Read After Write (RAW), 2) Write After Read (WAR) and 3) Write After Write (WAW). RAW and WAW are taken care by the dependency analyzer. However, WAR never occurs as it is prevented by architecture design: The proposed mechanism may move the 'read' instruction in the next bundle and, thus, it will be executed in parallel with the 'write' instruction. However, as the rescheduling occurs after the decode stage, the read instruction has already obtained the correct value from the register file. For the RAW and WAW cases, the *dependency analyzer* reads the destination of each instruction from the *Info* register and compares it with the sources and destination values of the instructions currently extracted from the information extraction unit. If they are the same, the corresponding outputs of the vector signal `FtoDC_Dep` are set to one to inform the replication scheduler for the detected dependency.

In Fig. 4.7 we illustrate the *Dep* array of our running example at time t_i (refer to Fig. 4.1b), where both instructions of the current bundle (B_i) are dependent on the memory instruction of the next bundle (B_{i+1}), thus the corresponding *FtoDC_Dep* positions are set to 1.

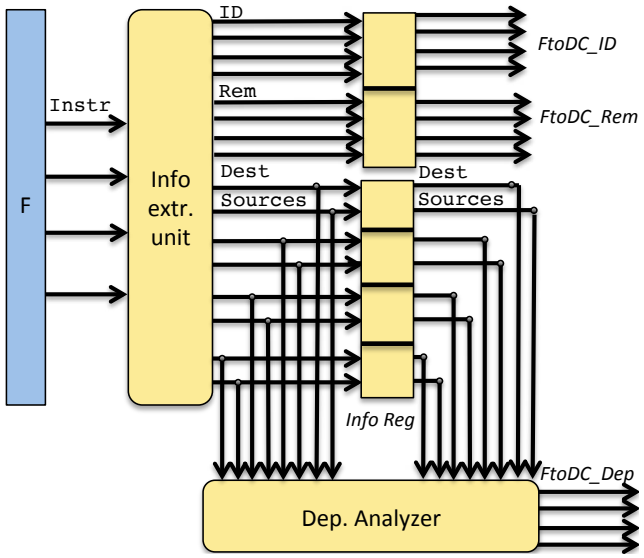


Figure 4.8: *Information extraction unit* and *Dependency analyzer*.

4.4.3 Replication Scheduler

The replication scheduler is responsible for the scheduling of the inputs of the *replication switch*, represented by *ReplicRes_ID* and *FtoDC_ID*, to the output of the switch, represented by *DCtoEX_ID*. The scheduling priorities are given in the following order: 1) whichever instruction remains in the *ReplicRes* register has the highest priority, 2) only the dependent instructions in the *FtoDC* register are scheduled and 3) the independent instructions from *FtoDC* register are scheduled.

Pre-processing In order to reduce the overhead of the replication scheduler, our approach works on instruction occupation vectors instead of the ID arrays. Following the above priority rules we introduce the three following vector groups: 1) *ReplicRes* for the previous instructions, 2) *FtoDC_dep* for the dependent current instructions and 3) *FtoDC* for the independent current instructions. Each group includes two vectors: 1) *instr*, which indicates the existence of instructions in the slots and, 2) *mul*, which indicates if the instructions are multiplications based on the current VLIW configuration. Thus, each element of the vector *instr* has a size of n, while each element in *mul* has a size equal to the number of issues with *MUL* FUs. Concerning the memory operations, as their execution is performed after the *EX/VOTE switch*, the *voting scheduler* is responsible for their correct scheduling.

Table 4.2 presents the transformation required to obtain the *instr* vector: Each bit of the vector is set, if at least one of the two LSBs of each *ID[i]* is not zero and there

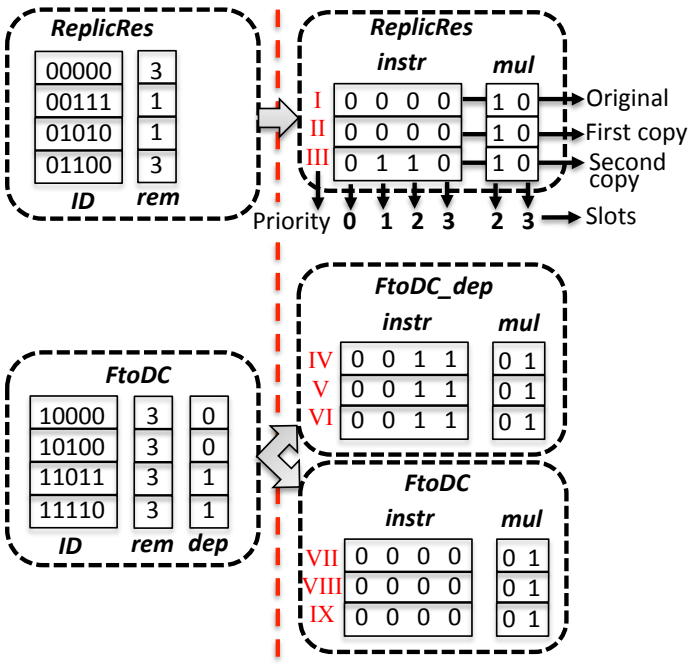


Figure 4.9: Pre-processing of *IDs* to occupation arrays.

are enough remaining instructions (*rem[i]*), where i is the issue number and j a variable corresponding to the execution of the original, the first or the second copy of the instructions. For the *FtoDC_dep* and *FtoDC* groups, the instructions must be also dependent (*dep[i]*) and independent (*!dep[i]*), respectively.

Table 4.2: *ID* to *instr* vector transformation.

Group	Transformation
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	$(ID[i][0] \parallel ID[i][1])$ & $(rem[i] \geq 3-j)$ & $!dep[i]$

The *mul* vector for each group is obtained by checking whether the IDs in the positions with a multiplication unit (i.e., $i \in \{2,3\}$) belong to a multiplication instruction or not $(ID[i][0]$ & $!ID[i][1])$.

Fig. 4.9 illustrates the input occupation vectors for the running example of Fig. 4.1b at time t_i. The instructions in slots 1 and 2 of the *ReplicRes* need to be executed one more time (*ReplicRes_Rem[1]*=1, *ReplicRes_Rem[2]*=1). Therefore, the second and third bits of the *ReplicRes_instr[2]* are set. The instruction in slot 2 is a multiplication, while the instruction in slot 3 is a NOP. Thus, the vector *mul* is set to 10 for all its entries. The

FtoDC_dep_instr, *FtoDC_dep_mul*, *FtoDC_instr* and *FtoDC_mul* are constructed in a similar way.

Bitwise Logic Based on the scheduling priorities, the occupation arrays are explored in the order depicted by Fig. 4.9 (Latin numbers in red). For each occupation array, two scheduling mechanisms are applied: 1) direct assignment, i.e., the issue where the instructions are originally scheduled is not modified and, thus, no verification of the type of FUs is required, and 2) circular exploration, which is applied after the direct assignment in case there are still instructions to be scheduled. These instructions are scheduled to the remaining idle slots taking into account the type of the FUs.

Direct assignment: This scheduling mechanism has two inputs: 1) the occupation vector for the instructions to be scheduled (*instr*) and 2) the current occupation of the issues (*issues*), and has three outputs: 1) the updated occupation of the issues (*issues_up*), 2) the scheduled instructions at this step (*fit*) and 3) the remaining ones to be scheduled (*rest*). The mathematical representation of the direct assignment algorithm is defined as:

$$\begin{aligned} &issues\_up[j] = instr[j] \parallel issues, \\ &fit[j] = issues\_up[j] \oplus issues, \\ &rest[j] = instr[j] \oplus fit[j],\ j \in \{0, .., 2\}. \end{aligned} \tag{4.1}$$

The scheduling is performed through a bitwise OR operation between the *instr[j]* and the *issues* vectors. The *issues_up[j]* result is compared (bitwise XOR) with the initial output vector *issues* to decide which of the instructions can be mapped directly to the output (*fit[j]*). According to the *fit[j]*, the *ID* and *rem* arrays are modified as follows: a) if *fit[j][i] = 1*, then *Rem[i] = Rem[i] - 1*, b) *DCtoEX_ID[i] = ID[i]*. The `sel_r` vector signal is also updated: *sel_r[i] = i + n* for the *ReplicRes* group and *sel_r[i] = i* for the others. To obtain the instructions that could not be scheduled directly, *rest[j]*, the vector *fit[j]* is compared with the initial input vector *instr[j]*. If the vector *rest[j]* is zero, it means that all the instructions were scheduled and the next input vector can be explored for scheduling.

The fig. 4.10 shows how the direct assignment technique is applied to the running example. Initially, the *issues* vector is initialized to zero. We start by testing the *ReplicRes* vectors. Since *instr[0] and instr[1]* are both zero, no instruction exists to be scheduled. Then, *instr[2]* is assigned to the binary value 0110. The output *fit[2]* is the same as the input *instr[2]*, meaning that all instructions can be scheduled (*rest[2]* = 0000). Accord-

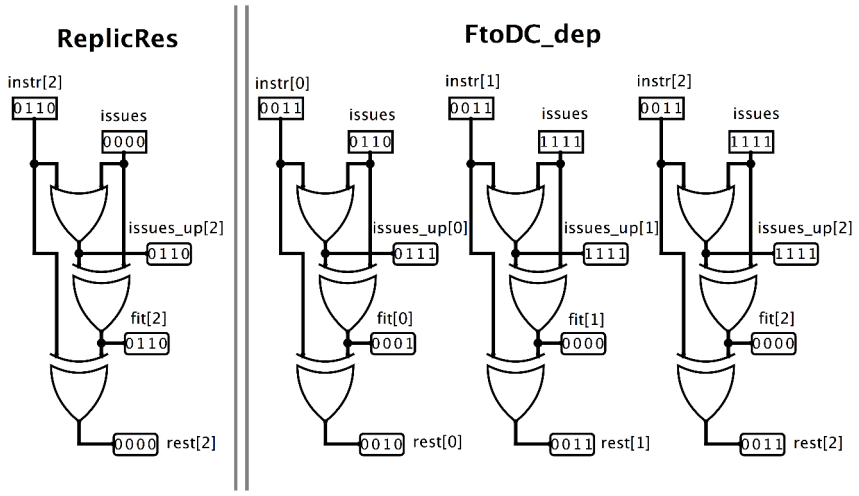


Figure 4.10: Direct assignment algorithm on the running example.

ing to the *fit[2]*, the *ID* and *rem* arrays are modified as follows: a) *ReplicRes_Rem[1] = 0*, b) *ReplicRes_Rem[2] = 0*, c) *DCtoEX_ID[1] = ReplicRes_ID[1]*, d) *DCtoEX_ID[2] = ReplicRes_ID[2]*). We also update the `sel_r` vector signal as *sel_r[1] = 5* and *sel_r[2] = 6*.

The next vectors to be checked are the *FtoDC_dep* vectors. The new *issues* is the previous *issues_up[2]* and the input vector is *instr[0]*. The output *fit[0]* is not the same as the input *instr[0]*, meaning that some instructions cannot be scheduled (*rest[0]* = 0010). Finally *issues_up[0]* is the updated *issues* occupation vector. According to *fit[0]* we update the IDs and *sel_r* as we did above. As the *rest* signal is not zero and the bits in *issues_up* are not all set, there are still empty FUs that could be potentially used to schedule the *rest* instructions by applying the following circular exploration mechanism.

Circular exploration: This mechanism takes three inputs: 1) the occupation vector for the instructions to be scheduled (*instr[j]*) initialized with the vector *rest[j]* from the direct assignment, 2) the current occupation of the issues (*issues*) initialized with the *issues_up[j]* result coming from the direct assignment, and 3) the *mul[j]*. The output is the assignment signal *assign[i]*. This signal holds the configuration values for each decided assignment.

Algorithm 3 presents the most representative cases of the circular exploration mechanism. The first two cases represent the situations when only one instruction needs to be scheduled. When *instr[j]* = 1000 the instruction cannot be a multiplication, whereas in case *instr[j]* = 0010 the instruction can be a multiplication. In the first case, the instruction can be tested in any available issue. In the second case, two possibilities exist: a) whatever the

Algorithm 3 Representative cases of the circular exploration mechanism.

```
1: Inputs:instr[j], issues_up[j], mul[j]
2: Outputs:assign[ ]
3: procedure SWITCH
4:     switch instr[j] do
5:         case ...
6:         case 1000 :
7:             if (issues_up[j][2] = 0) then
8:                 assign[1] ← 0;
9:             else if (issues_up[j][1] = 0) then
10:                assign[2] ← 0;
11:            else if (issues_up[j][0] = 0) then
12:                assign[3] ← 0
13:            end if
14:        case 0010 :
15:            if (issues_up[j][0] = 0) then
16:                assign[3] ← 2;
17:            else if (mul[j][1] = 0) then
18:                if (issues_up[j][2] = 0) then
19:                    assign[1] ← 2;
20:                else if (issues_up[j][3] = 0) then
21:                    assign[0] ← 2;
22:                end if
23:            end if
24:        case 0011 :
25:            if (mul[j] = 10) then
26:                if (issues_up[j][2] = 0) then
27:                    assign[1] ← 3;
28:                else if (issues_up[j][3] = 0) then
29:                    assign[0] ← 3;
30:                end if
31:            else if (mul[j] = 01) then
32:                if (issues_up[j][2] = 0) then
33:                    assign[1] ← 2;
34:                else if (issues_up[j][3] = 0) then
35:                    assign[0] ← 2;
36:                end if
37:            else if (mul[j] = 00) then
38:                if (issues_up[j][3] = 0) then
39:                    assign[0] ← 2;
40:                else if (issues_up[j][2] = 0) then
41:                    assign[1] ← 3;
42:                end if
43:            end if
44:        case ...
45: end procedure
```

value of *mul[j][1]*, the instruction can be potentially executed in *issues_up[j][0]* and b) otherwise, the instruction can be potentially executed in any possible position, if and only if this instruction is not a multiplication (*mul[j][1]* = 0). In the third case, two instructions need to be scheduled (case 0011). Because of their position we have to check if the instructions are multiplications. Three possibilities exist: a) *mul[j][1]* = 1 and *mul[j][0]* = 0, b) *mul[j][1]* = 0 and *mul[j][0]* = 1 and c) *mul[j][1]* = *mul[j][0]* = 0. No available scheduling exists for both instructions being multiplications due to resource restrictions. For a) and b) cases, the multiplication instruction cannot move, since the remaining untested issues do not have a multiplication FU. Only non-multiplication instructions can be scheduled. For c) case, no multiplication instruction exists, thus the instructions can be scheduled in any available slot. The case in which there are three instructions to be scheduled (case 1101, 1110, etc.) has the least complexity. Since all three instructions have been tested and failed to be scheduled in their current position by the direct assignment technique, the only legitimate action is to potentially schedule the non-multiplication instructions (*instr[j][2]*, *instr[j][3]*) (e.g. 1101 $\rightarrow$ 1011). In any of the aforementioned cases, if an instruction is scheduled, the signal *assign[i]* is updated with the original scheduled position of the instruction, where i is the new issue where the instructions is currently scheduled. According to the *assign[i]* signal, we update the *issues_up[j]* vector, the *ID* arrays and the signal *sel_r*.

For instance, in Fig. 4.10 after the direct assignment of the *instr[0]* of the *FtoDC_dep*, the *rest[0]* = 0010. Since *mul[0][1]* = 0, *issues_up[0][0]* = 1, *issues_up[0][2]* = 1 and *issues_up[0][3]* = 0, the condition of line 11 in Algorithm 3 is met and, thus, *assign[0]* $\leftarrow$ 2. Consequently we have: a) *issues_up[0][3] = 1*, *rem[2] = rem[2] - 1* and b) *DCtoEX_ID[0] = FtoDC_ID[2]*. The `sel_r` vector signal is updated to *sel_r[0] = 2*. Finally, the *issues* is updated to 1111 for the next direct assignment, as depicted in Fig. 4.10, and, thus, no further scheduling exploration can be performed.

Time-slot insertion decision: After the instruction scheduling phase, the *replication scheduler* decides whether there is a need to stall the fetch and decode stages. The stalling decision occurs when: a) there are still unscheduled instructions in the *ReplicRes* (*ReplicRes_rem* $\neq$ 0), or b) there are still unscheduled dependent instructions in the *FtoDC* (*FtoDC_rem* $\neq$ 0).

4.4.4 Voting Scheduler

This scheduler decides the scheduling of the currently executed instructions either to the *voters* and the commit/memory phase or to the *VotingRes* register to be stored. It has two inputs: 1) *DCtoEX_ID* and 2) *VotingRes_ID*, which are concatenated to form an array named *unGrouped_ID*. A comparison-based sorting algorithm is applied to group the instructions with the same *IDs* in order to form triplets respecting the memory resource constraint. The output is a sorted array named *Grouped_ID*. The way the instructions have been sorted in triplets determines the values of the signal *sel_v*, which configures the *voting switch*. For instance, if a triplet is formed in the three first places of the *Grouped_ID* array, i.e., there are instruction IDs in *Grouped_ID[0]*, *Grouped_ID[1]* and *Grouped_ID[2]*, the corresponding instructions are sent to the first voter. In case of an incomplete triplet in the first places of the *Grouped_ID*, i.e., IDs only in *Grouped_ID[0]* and/or in *Grouped_ID[1]*, the corresponding instructions are stored to the *VotingRes[0]* and *VotingRes[1]* registers. Memory instructions are grouped only in *Grouped_ID[3]*, *Grouped_ID[4]* and *Grouped_ID[5]* in order to be scheduled to the second voter that is connected to a memory unit.

4.5 Cluster-based approach

The proposed architecture follows an n-issue configuration. Based on the experimental results with respect to the area overhead (Fig. 4.14 in the Exmperimental Results Section 4.6 page 57), the most dominant components are the switches. Therefore, with the increase of the number of issues n, we expect a non-linear increase of the area and an increase in the delay, since the switch complexity has a non-linear increase, as shown in Table 4.3. To avoid this limitation, we propose a cluster-based approach based on multiple instances of a VLIW with smaller issue number. For instance, for a $n \times 4$ cluster approach we employ n parallel 4-issue components, that ideally each handles the $1/n$ of the instructions, as shown in Fig. 4.11 for $n = 2$. In this way, the area grows in a linear way since the hardware needed for a $n \times 4$ approach is n times the hardware needed for the 4-issue approach. In addition, the delay of the data path is preserved no matter how much we scale the design since there is zero scaling of the switches.

However, following the cluster-based approach, the exploitation of the idle slots is restricted only within a cluster. The compiler usually schedules the instructions as dense as possible in order to occupy less area in the memory and, thus, it tries to first fit the instruc-

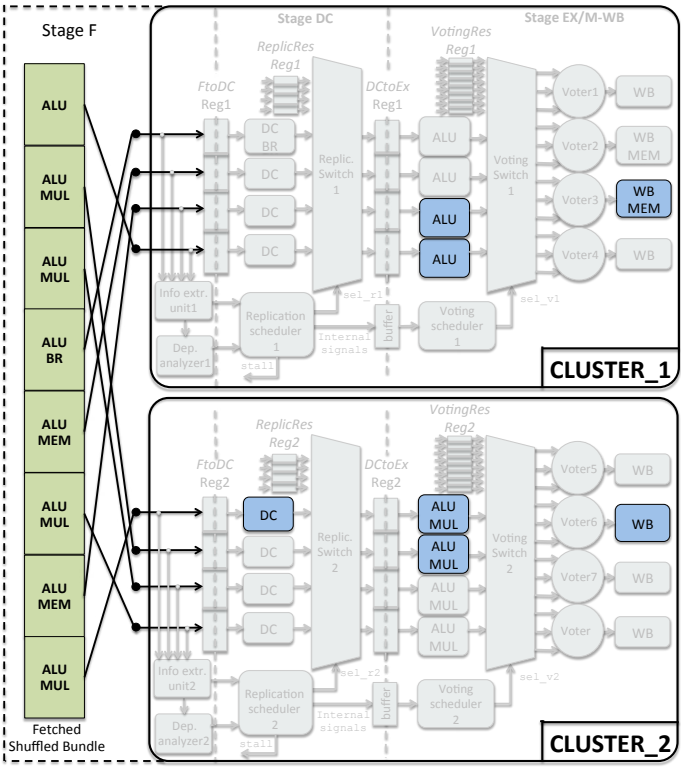


Figure 4.11: Example of a cluster-based 2×4 VLIW configuration.

Table 4.3: Implementation complexity of the *Replication switch* (Fig. 4.4) for different n-issue configurations.

Configuration	Size		Complexity
	In	Out	(No. of 2:1 mux)
4-issue	8	4	28
8-issue	16	8	120
16-issue	32	16	496
32-issue	64	32	1952

tions to the first cluster. Because of this behaviour, the instructions are not assigned to the other clusters and the potential idle slots cannot be exploited resulting in unnecessary stalling of the processor. To deal with this limitation, we propose a shuffling approach, where we provide the compiler with a shuffled FU VLIW configuration so as to generate instruction bundles, where the instructions are more uniformly distributed. To avoid inserting additional area overhead, a static de-shuffling according to the real VLIW cluster configuration is performed during the fetch stage. An example of such case is depicted in Fig. 4.11, where the fetched bundle of this 2x4 configuration is shuffled in a random way and a static de-shuffling is performed in order to drive the instructions to their corresponding VLIW issues.

In addition, the real VLIW cluster configuration is defined in such a way to allow a better FU exploitation: the multiplication (or memory) function units are gathered in the same cluster. With this configuration the FUs of the same type are grouped, allowing the exploitation of the FUs of a specific type. Fig. 4.11 illustrates a 2×4-issue VLIW with 1 branch unit, 8 ALUs, 4 multiplication units and 2 memory units. A global fetch occurs that distributes the 8 shuffled fetched instructions to the two 4-issue clusters. The stall signal (*stall*) generated by the hardware scheduler 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 of them needs an extra time slot.

4.6 Experimental Results

In this section we discuss about performance, area and power results of our approach. We employ the VEX VLIW processor [20] under two different VLIW configurations (based on realistic commercial VLIWs, e.g., Intel Itanium [75]):

1. 4-issue width (4 ALUs, 2 Mult, 1 Mem, 1 Br)

2. 8-issue width (8 ALUs, 4 Mult, 2 Mem, 1 Br)

The VEX processor has been modified and enhanced with the proposed approach. Both the original unprotected processor and the components of our approach have been developed in C++ and synthesized using the Catapult High Level Synthesis (HLS) tool. Following this approach, we are capable of both simulating the processor and synthesizing a functional RTL design. However, it should be highlighted that modern HLS tools still fail to produce the

same high quality results as hand written RTL code [62]. This especially holds for complex irregular designs with algorithms that have a highly data dependent behavior, such as our schedulers. Therefore, the obtained results provide an upper bound in the area and power overhead.

4.6.1 Performance

We compare the behaviour of the proposed approach (*TMRi*) with: a) the unprotected architecture (*Unprotected*), b) the architecture that triplicates the FUs and votes for the result (*FU triplication*), and c) the architecture which exploits the idle slots in space, i.e. only inside the current bundle (*TMR*), without the dependency exploitation. For the performance metric, both the unprotected architecture and the FU triplication architecture have the same performance results.

In Table 1.1 in the first chapter, we have presented the dependency occurrence (%) for each application for 0, 1, 2 or 3+ number of simultaneous dependencies between consecutive bundles. As we observe, for most of the applications the case of having zero dependencies between two consecutive bundles is more than 50%, pointing out a high potential benefit for our approach. The case with exactly one dependency is also quite frequent (∼30%). Our approach exploits these cases as dependent instructions are prioritized before the independent ones. Thanks to this prioritization policy our technique can benefit even in case there are more dependent instructions simultaneously in one bundle.

Fig. 4.12 depicts the execution cycles of the 10 benchmarks executed on a processor with the 4-issue configuration. The proposed *TMRi* architecture has a performance speed-up from 13.47% for the *matrix_mul* benchmark up to 43.68% for the *huff_ac_dec* benchmark compared with the *TMR* architecture. The average speed-up that our architecture achieves for the 4-issue configuration is 30.15%.

Fig. 4.13 depicts the execution cycles of the benchmarks with the 2×4-issue configuration. Compared with the *TMR* architecture, we observe a speed-up from 7.62% for the *crc* benchmark up to 27.61% for the *adpcm_dec* benchmark. The average speed-up that our architecture achieves for the 2×4-issue configuration is 19.84%. We observe that in general the speed-up for the 4-issue is slightly higher than for the 2×4-issue. This occurs because of the following reasons:

1. The average *ILP* of the applications compiled for an 2×4-issue configuration is low and, thus, there are more idle slots in the current bundles than the idle slots in a 4-issue

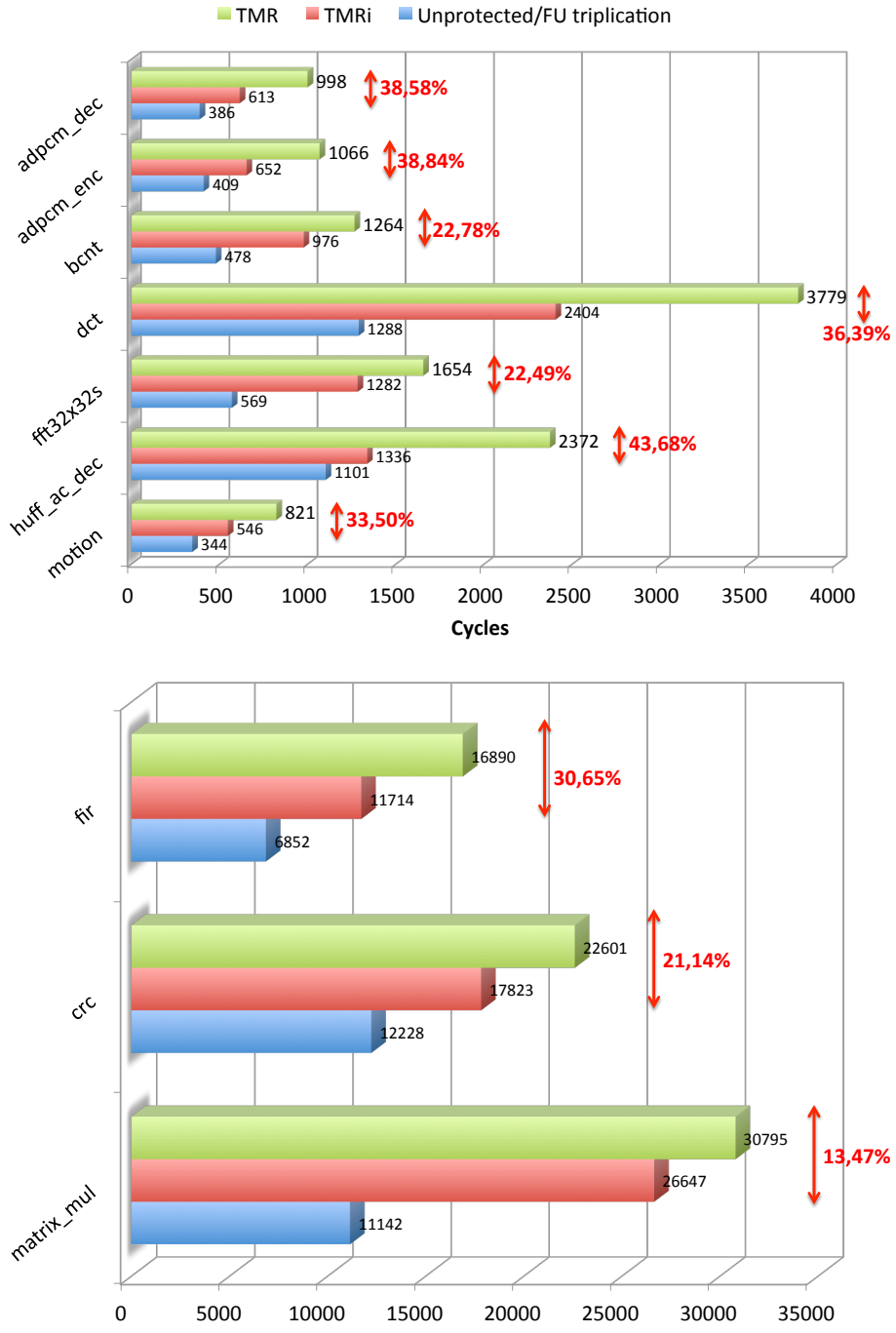


Figure 4.12: 4-issue performance results.

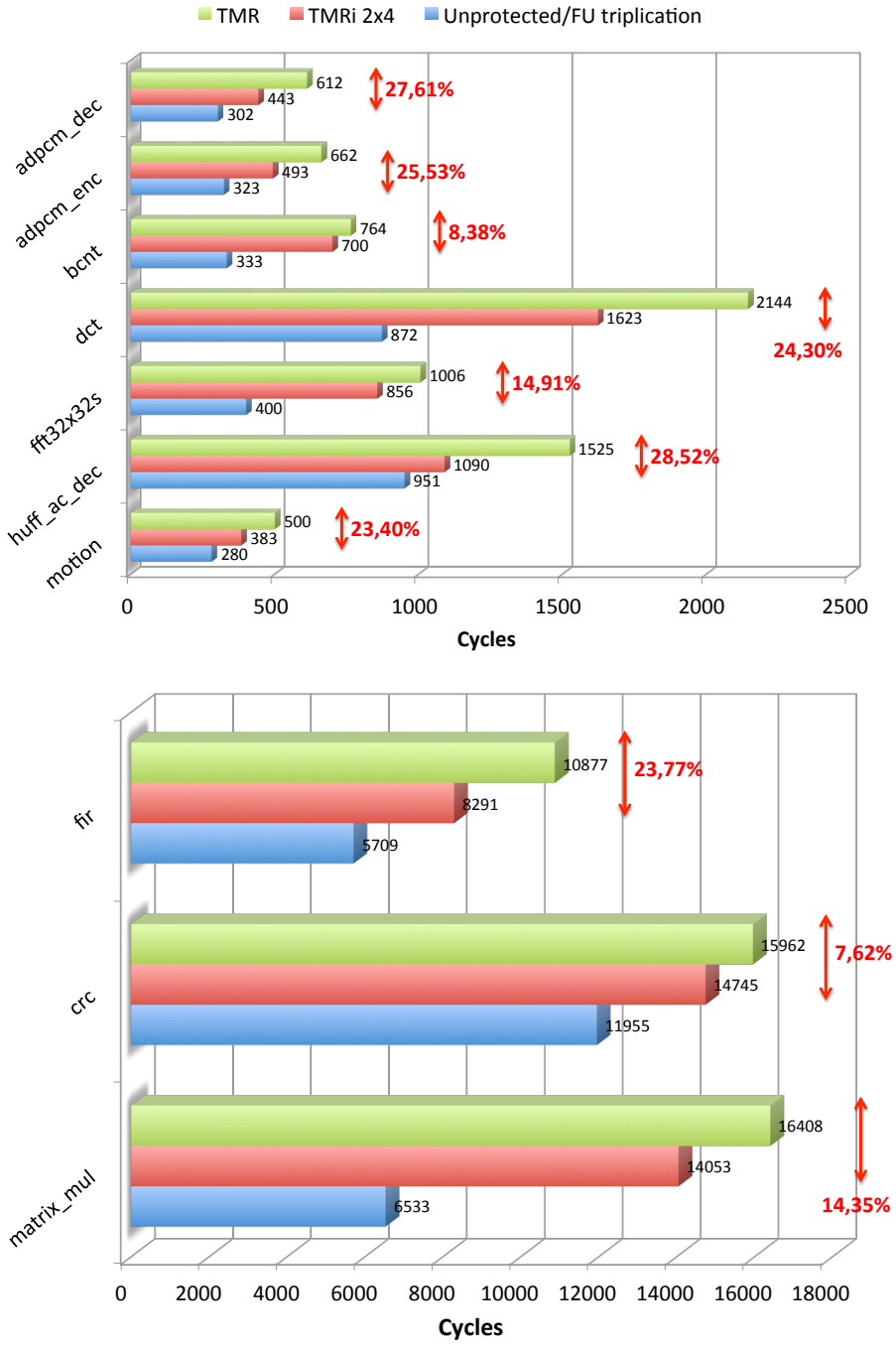


Figure 4.13: 8-issue performance results.

configuration for both techniques to exploit. For the *TMR*, the threshold – after which we need to add an extra time slot – is one instruction for the 4-issue configuration and two instructions for the 2×4-issue configuration. Concerning the 4-issue, where the ILP is ~ 2, in most cases additional time slots are needed, whereas in the case of 2×4-issue, where the ILP is ~ 2.5, the probability of needing an extra time slot is decreased. The *crc* benchmark for the 2×4-issue configuration is a paradigm of this case. It has the lowest observed speed-up because the ILP of the *crc* changes slightly from the 4-issue configuration to the 2×4-issue configuration, and, thus, both *TMRi* and *TMR* approaches have the required idle slots available in the current bundle.

2. The speed-up also depends on the code structure of each application. For instance, an application with consecutive bundles full of instructions has a similar behavior in both *TMR* and *TMRi*. Both techniques have no idle slots to exploit neither in the current nor in the next bundle. A paradigm of such an application is the *matrix_mul*, which achieves the lowest performance speed-up for the 4-issue configuration.

3. A third reason is the number of dependencies. If there are idle slots to be exploited in the next bundles, but the instructions to be scheduled are dependent, then no idle slot exploitation is allowed and new time slots have to be added. However, in most of the cases, there are less than two dependent instructions (see Table 1.1) and the remaining independent instructions can use the idle slots of the next bundle. This is the reason why the *crc* benchmark for the 4-issue configuration has high performance speed-up even though 70% of the bundles have one or two dependent instructions.

4. The last reason is the scheduling of the memory instructions. In our approach, these instructions can be scheduled in any issue, since the *voting switch* leads them correctly to the appropriate functional unit that implements the memory operations. In *TMR*, the memory instructions are scheduled from the beginning only in the issues supplied with a memory unit. A paradigm of this case is the *huff_ac_dec* benchmark, which has a low ILP and a lot of bundles with just one memory instruction. In this case, *TMRi* rarely adds time slots, whereas *TMR* requires each time two additional time slots in order to triplicate a memory instruction, given the VLIW configuration with only one memory unit.

4.6.2 Area and Power

All the techniques have been synthesized using Catapult HLS tool (University Version 10.0b/273613) to obtain the RTL design. The last step of generating the gate-level netlist was handled by Design Compiler (Version J-2014.09-SP5-7) from Synopsys using a 28-nm ASIC library. Area and power results obtained with a target frequency of 200MHz are shown in Table 4.4. Note that the power analysis of each approach, provided by Design Compiler, is based on a statistical activity factor estimation with the assumption that every net toggles 10% of the time. In Table 4.5 we present the overhead of each of the techniques when they are compared with the unprotected architecture.

Table 4.4: Area footprint and power estimation results.

Approach	**4-issue**		**8-issue**	
	area(μm^2)	**power(mW)**	**area(μm^2)**	**power(mW)**
Unprotected	50843,82	6,48	79661,02	7,36
TMR	58818,67	8,01	95257,57	9,03
TMRi 2x4	62812,26	8,61	103597,9	9,77
FU triplication	73522,58	8,81	124136,94	11,88

Table 4.5: Area and power overhead to the unprotected approach.

Approach	**4-issue**		**8-issue**	
	area(%)	**power(%)**	**area(%)**	**power(%)**
TMR	15,68	23,61	19,58	22,69
TMRi 2x4	23,54	32,87	30,05	32,74
FU triplication	44,60	35,96	55,83	61,41

We observe that the FU triplication architecture implies an area overhead up to 55,83% and a power overhead up to 61,41%. The *TMRi* has less overhead, i.e. up to 30,05% for area and up to 32,74% for power. The *TMRi* has the lower overhead, i.e. up to 19,58% and up to 22,69% for the power overhead. However, *TMRi* can provide a performance speed-up up to 43,68% with an area/power overhead of $\sim$ 10% over the *TMR* approach.

Finally, we explore the participation of each component of the proposed *TMRi* architecture in the area overhead of our approach. Fig. 4.14 depicts graphically the area coverage of each of the basic components of our design. The area of the switches dominates the area of our design with the 76,2% of the total area overhead. The other components (the schedulers, the voters and the rest of the logic) contribute to the remaining 23,8%. Concerning

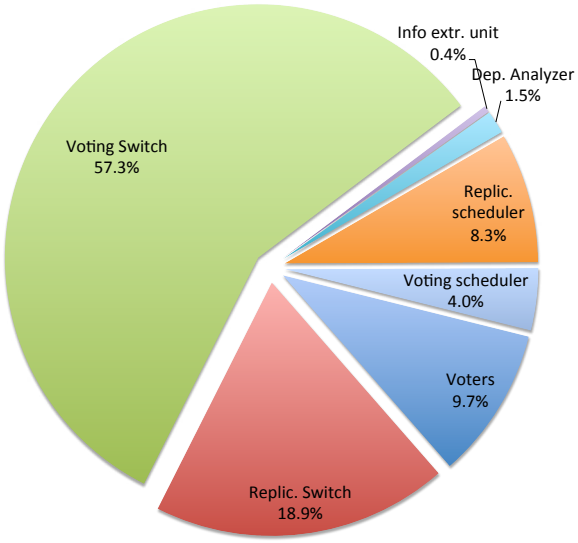
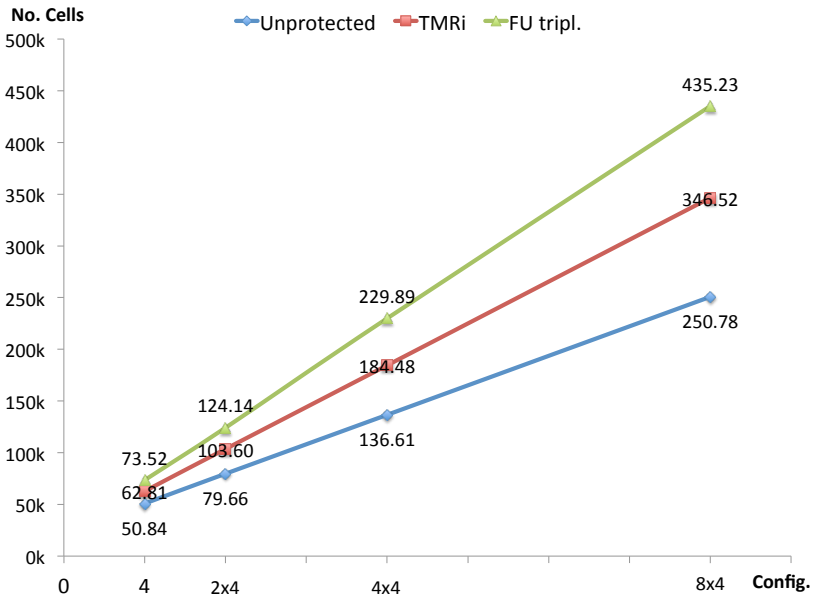


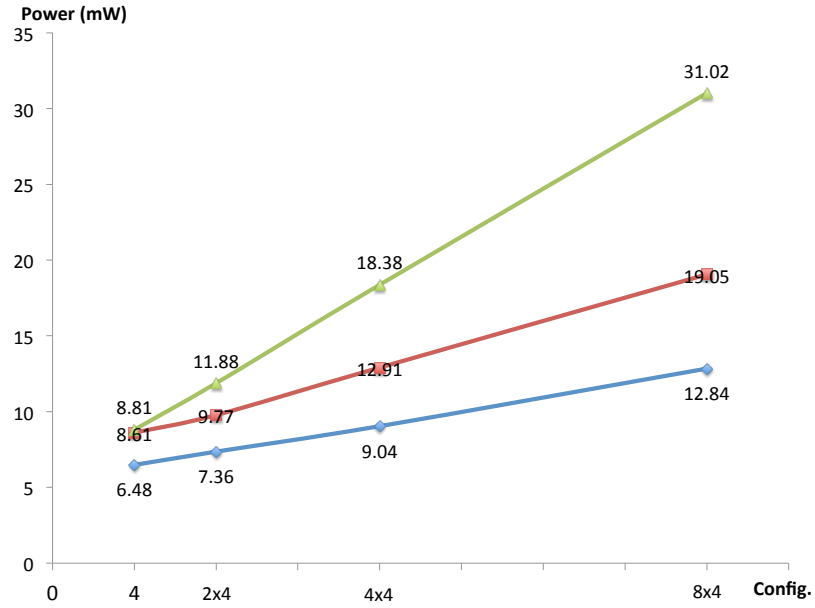
Figure 4.14: Area coverage of each of our technique's components.

the delay of the components inserted inside the VLIW data path (*replic. switch*, *voting switch* and *voters*), they add a rather low amount of delay (0,16ns, 0,33ns and 0,27ns, respectively). The largest delay is given by the *replication scheduler* (2,19ns), which, however, does not affect the clock speed of the processor, since it is strategically placed in parallel with the data path.

According to the obtained results for the 4-issue and the 8-issue configurations, we give an extrapolation on how the proposed approach scales. Fig 4.15a illustrates the extrapolated area estimations, while Fig. 4.15b presents extrapolated estimations for the power consumption. As we observe TMRi is growing slower than the FU triplication, making it a better candidate if area is a design constraint. Same observations are performed for the power, where we observe that the slope of the FU triplication is steeper , making our approach a good candidate if power is also a constraint.



(a) Area scaling (μm^2).



(b) Power scaling (mW).

Figure 4.15: Scaling of the proposed approach

4.6.3 AVF and IVF analysis

We perform an AVF and IVF analysis similar to the analysis presented in Chapter 3, so as to explore the behaviour of the proposed architecture.

AVF analysis: Fig. 4.16 presents the per cycle AVF of the proposed 4-issue *TMRi* architecture (i.e. the original processor enhanced with our technique's components) when it executes a matrix multiplication application. The whole execution of the application takes 31529[1] cycles. An exhaustive fault injection simulation injecting faults at each cycle would require several days of simulation. Additionally, although for a precise calculation of the AVF we should inject faults to all the storage structures of our architecture, in this study we inject faults only in the following storage elements: a) FtoDC, b) DCtoEx, c) Register File and d) PC, same as for the original architecture. The storage elements added by our mechanism are assumed to be protected (e.g. with ECC codes). This assumption was made in order to keep the simulation time cost low, since an exhaustive fault injection scheme would require significantly long simulation time. We inject faults at the exact same unique

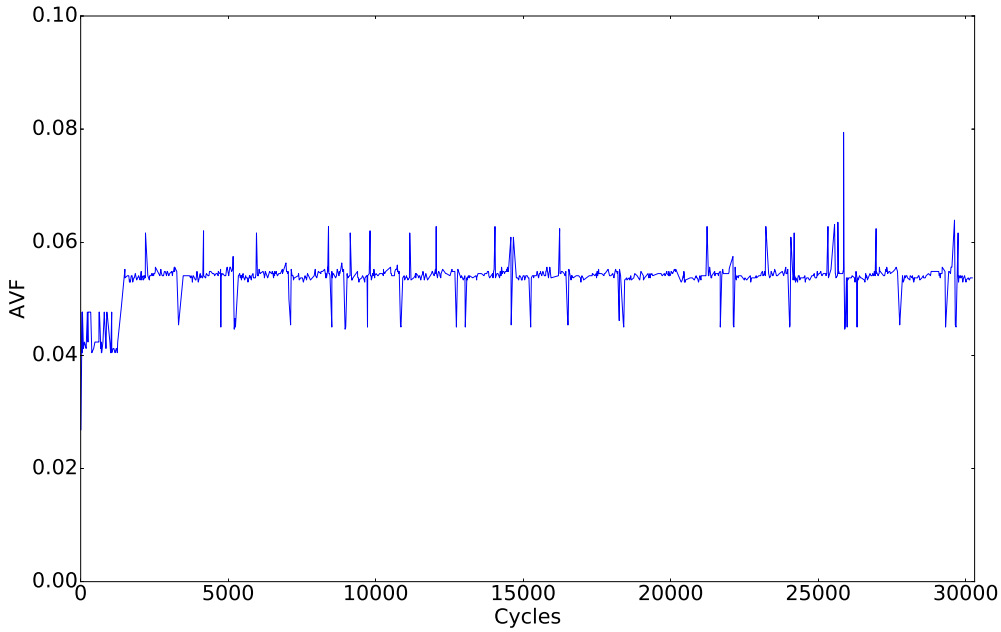
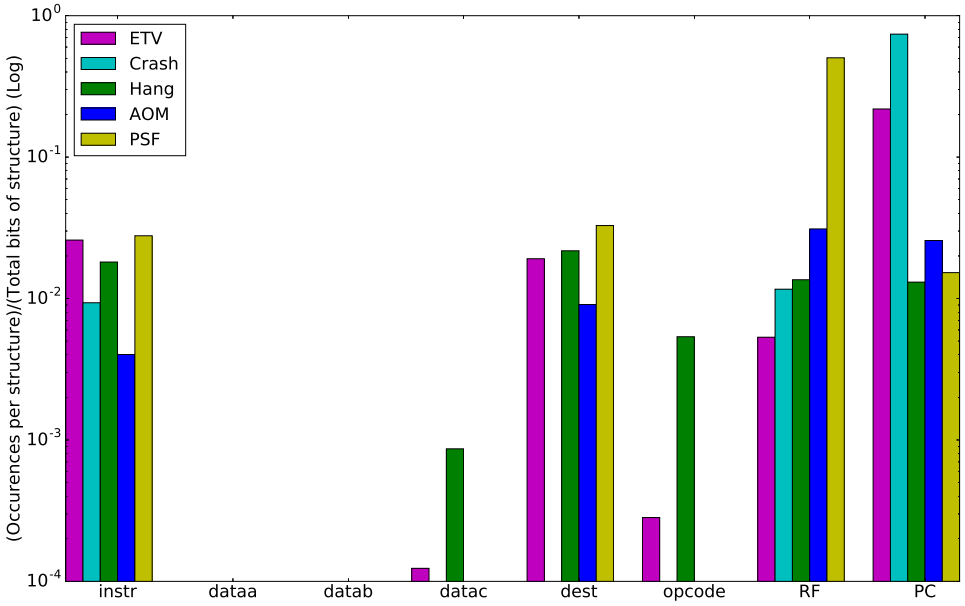


Figure 4.16: Per cycle AVF for VEX processor

[1]The execution cycles needed are different from the cycles in section 4.6.1 because a different matrix multiplication implementation has been used.

cycles randomly chosen in Chapter 3 for the AVF analysis. The same simulation scenario is applied here (i.e. exactly the same registers as injection points and same injection cycles) in order to have exactly the same faults injected, so as to be able to compare the proposed *TMRi* approach with the *Unprotected* one. We observe that the AVF of the *TMRi* (Fig. 4.16) is similar to the AVF of the unprotected original architecture (Fig. 3.1) and the average AVF is 0.0533. This can be explained as follows: The AVF calculation as well as the fault injection experiment itself are mainly dominated by injections in the registers of the RF (32x64 = 2048 bits over a total of 2644). Since our technique does not protected the RF directly from errors happening in the registers, it does not improve to the AVF value contributed by the RF. On the other hand our technique protects the RF from indirect errors happening inside the data path and being propagated to the registers of the RF. Another reason why the AVF of the *TMRi* is similar to the unprotected one is because according to Fig. 3.2 the masking capabilities of the unprotected architecture, when an error is injected in the DCtoEX register are quit high ($\sim 99\%$).

Fig. 4.17 presents the categorisation of processor's output running a matrix multiplication for the fault injection experiment according to the classes presented in Chapter 3. The results are presented in logarithmic scale. Since *TMRi* applies an instruction triplication scheme before the DCtoEx register, we are able to detect and correct errors occurring in these registers. In the current implementation the voters compare only the obtained results of the operations after the EX stage and, thus, only the errors in DCtoEX_dataa and DCtoEX_datab are always guaranteed to be corrected. Errors in DCtoEX_dest are corrected only in case the result chosen by the voters is the correct one. On the other hand most of the times errors in DCtoEX_opcode are corrected, since an execution of another instruction instead of the original one will result in a different calculated result, which can be detected and corrected in the voters. Hang errors in case of injections in the DCtoEX_opcode occur rarely and in cases such as: A triplicated LDB instruction, which loads an 8-bit value, resides in the DCtoEX register and one bit of the DCtoEX_opcode register changes, referring to instruction LDW, which loads an 32 bit value. After the execution stage the result will be the same for all three copies of the instruction, so the voters will not detect the error. Therefore, they will commit the faulty instruction, which will load a value with a different size into a register. In case this register is used to evaluate a loop condition, this error may lead to a hang error.



Class	instr	dataa	datab	datac	dest	opcode	RF	PC
ETV	0.026	0.0	0.0	0.0	0.019	0.0	0.005	0.219
Crash	0.009	0.0	0.0	0.0	0.0	0.0	0.012	0.741
Hang	0.018	0.0	0.0	0.001	0.022	0.005	0.014	0.013
AOM	0.004	0.0	0.0	0.0	0.009	0.0	0.031	0.026
PSF	0.028	0.0	0.0	0.0	0.033	0.0	0.504	0.015

Figure 4.17: Error occurrences per storage structure for the matrix multiplication (Normalized)

IVF analysis: For each instruction opcode and for each bit of each of the structures (FtoDC, DCtoEx, Register File and PC) of *TMRi*, we generate 1000 different test cases with random inputs in order to create a uniform distribution of the input masking probability. Table 4.6 presents the IVF of each instruction of each stage of the VLIW processor. For all the instructions, the IVF of the decode stage is greater than the IVF of the fetch stage, because of the applied fault tolerant technique. More precisely, the Table 4.6 presents the IVF of all the logical, multiplication, memory, control and integer arithmetic operations of the Instruction Set Architecture (ISA). Because of the proposed triplication mechanism, the most vulnerable instructions of the unprotected architecture, i.e. the integer arithmetic and memory operations (see Table 3.4) are now up to 2.2x less vulnerable compared to the integer arithmetic and memory operations of Table 4.6. In case of an implementation with voters checking for errors in the both the opcode and destination registers, the IVF of all

the instructions in the decode stage would be one.

4.6.4 Conclusion

In this chapter, a hardware-initiated approach for heterogeneous VLIW data-paths was proposed to reuse the idle slots to provide fault tolerance. The proposed approach explores the idle slots in the current and next bundles and prioritizes dependent instructions. The result is a more compact schedule for both original and replicated instructions. In order to keep the hardware cost low while still providing with full rescheduling flexibility, a hardware-friendly instruction scheduler is proposed. In addition, for scalability purposes, a cluster-based approach is presented and evaluated to avoid area and power overhead with a small decrease in performance. The processor is tested with 10 different media benchmarks and the obtained results show a 43.68% maximum speed-up with both area and power overheads to be $\sim$10% with respect to existing approaches. Finally, we performed an AVF and IVF analysis of the proposed architecture, in order to evaluate its vulnerability and compare it with the unprotected original architecture.

Table 4.6: Per stage IVF for all operations of the ISA

Logical Operations		
OPCODE	**Fetch**	**Decode**
CMPEQ	0.883569	0.9741606
CMPGE	0.880248	0.9740000
CMPGEU	0.884861	0.9743504
CMPGT	0.878577	0.9734307
CMPGTU	0.880839	0.9741387
CMPLE	0.880664	0.9738759
CMPLEU	0.880212	0.9742774
CMPLT	0.875409	0.9736569
CMPLTU	0.875854	0.9742555
CMPNE	0.884956	0.9739343
CMPEQi	0.905679	0.9745182
CMPGEi	0.912467	0.9744015
CMPGEUi	0.912482	0.9735839
CMPGTi	0.912467	0.9737153
CMPGTUi	0.912460	0.9736496
CMPLEi	0.912453	0.9737737
CMPLTi	0.905175	0.9738832
CMPLTUi	0.905175	0.9744891
CMPNEi	0.905263	0.9743212
AND	0.819912	0.9742847
ANDi	0.814737	0.9738832
ANDC	0.884934	0.9751752
ANDCi	0.905898	0.9810292
OR	0.819927	0.9739562
ORi	0.815715	0.9738613
ORC	0.870540	0.9744891
ORCi	0.896905	0.9804891
NOR	0.817547	0.9740584
NORi	0.813270	0.9745985
NOT	0.861423	0.9741898
NOTi	0.861423	0.9740657
XOR	0.819533	0.9745109
XORi	0.766584	0.9732190

Multiplication Operations		
OPCODE	**Fetch**	**Decode**
MPYLL	0.884847	0.9733431
MPYLLU	0.884745	0.9732774
MPYLH	0.893891	0.9742628
MPYLHU	0.884745	0.9738029
MPYHH	0.884847	0.9750584
MPYHHU	0.892088	0.9750073
MPYL	0.891942	0.9748759
MPYLU	0.877650	0.9740219
MPYH	0.886701	0.9752847
MPYHU	0.899241	0.9752774
MPYHS	0.892044	0.9744161

Memory Operations		
LDW	0.927015	0.9733796
LDHU	0.919730	0.9731387
LDH	0.919730	0.9732628
LDBU	0.919708	0.9722190
LDB	0.919708	0.9722774
STW	0.766423	0.9754088
STH	0.766423	0.9752263
STB	0.766672	0.9748686

Control Operations		
CALLR	0.905110	1.0000000
BR	0.817518	1.0000000
BRF	0.978102	1.0000000
RETURN	0.810219	1.0000000
GOTO	0.824818	1.0000000
GOTOR	0.905124	1.0000000
CALL	0.810219	1.0000000
STOP	1.000000	1.0000000

Integer Arithmetic Operations		
OPCODE	**Fetch**	**Decode**
ADD	0.817518	0.9740584
ADDi	0.766547	0.9739489
SUB	0.819540	0.9738248
SUBi	0.766606	0.9739854
SRL	0.830073	0.9735766
SRLi	0.838212	0.9743431
SRA	0.830015	0.9735985
SRAi	0.838380	0.9744526
SLL	0.820752	0.9737591
SLLi	0.828942	0.9744015
SH1ADD	0.817518	0.9746934
SH2ADD	0.817518	0.9745036
SH3ADD	0.817518	0.9744818
SH4ADD	0.817518	0.9742409
SH1ADDi	0.766569	0.9740949
SH2ADDi	0.781022	0.9738394
SH3ADDi	0.773818	0.9735839
SH4ADDi	0.773723	0.9741825
ZXTH	0.877781	0.9746861
ZXTB	0.877964	0.9742555
SXTH	0.885204	0.9744964
SXTB	0.877737	0.9740073
ZXTHi	0.907854	0.9801971
ZXTBi	0.908051	0.9803942
SXTHi	0.913482	0.9802190
SXTBi	0.907854	0.9804161
MOVI	0.788321	0.9743796
NOP	1.000000	1.0000000

Chapter 5

Instruction Rescheduling for persistent errors

We propose a **coarse-grained mitigation mechanism** that takes advantage of the idle issue slots of VLIW instruction bundles to: 1) execute original and replicated instructions in order to provide fault tolerance and 2) rebind instructions in case of permanent errors. It can be combined with several fault tolerant techniques, i.e. duplication and triplication of instructions supporting error detection and mitigation, and it is applicable for any VLIW structure, i.e. any issue width and number and type of FUs.

In order to further decrease the performance overhead introduced by the coarse grained mitigation mechanism and also to support single and multiple Long-Duration Transient (LDT) faults a **fine-grained mitigation mechanism** is proposed. This mechanism detects the active faults during execution and temporally excludes only the faulty components of the affected FUs for as long as it is necessary.

5.1 Coarse Grained Mitigation for Permanent Errors

5.1.1 Motivation example and overview

Fig. 5.1 illustrates the output of the proposed approach using duplication of instructions and one permanent error. It shows the scheduled operations for an instruction bundle, the assembly instructions of which are shown inside the box on the top of the figure. The VLIW data path consists of three stages F, DC and EX/MEM/WB and it is depicted in fig. 5.2. The available FUs for an 8-issue VLIW configuration are: 8 Arithmetic and Logic

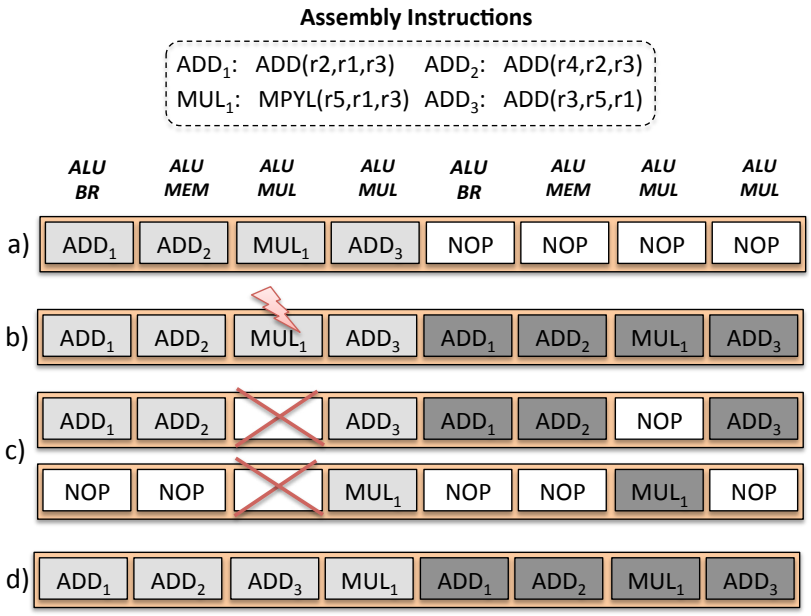


Figure 5.1: Illustration of the proposed approach

Units (ALUs), 4 Multipliers (MULs), 2 Memory operation units (MEMs) and 1 Branch unit (BR). The schedule of the original instruction bundle is depicted in Fig. 5.1(a) and the instruction bundle with the duplicated instructions in Fig. 5.1(b). In case of a permanent error detected in the multiplication unit of the third slot, existing hardware techniques re-execute the instruction scheduled at the third slot to another FU of another time slot, as depicted in Fig. 5.1(c). Consequently, the execution time increases.

To improve performance, the proposed approach explores at run-time the rebinding of original and replicated instructions to explore the existing FUs, as depicted in Fig. 5.1(d). In this example, the instruction ADD_3 is moved to the third slot, whereas the instruction MUL_1 is moved to the fourth slot without adding a new time slot. In case there is a need for an extra time slot, the instructions that fit in the first time slot are scheduled, while the remaining ones are scheduled in the next time slot.

Fig. 5.2 depicts the two hardware components added to the VLIW data path: the Instruction Replication and Binding (IRB) and the fault detector. The IRB takes the *decode stage result*, the *mode*, and the *faulty information* as input, and has the *binding info* and the *fetch stall* as output. The mode is defined by the designer and it defines which fault tolerance technique is implemented: i) duplication of instructions, ii) triplication of instructions, or iii) duplication and re-execution. Depending on the mode, the IRB duplicates or triplicates the

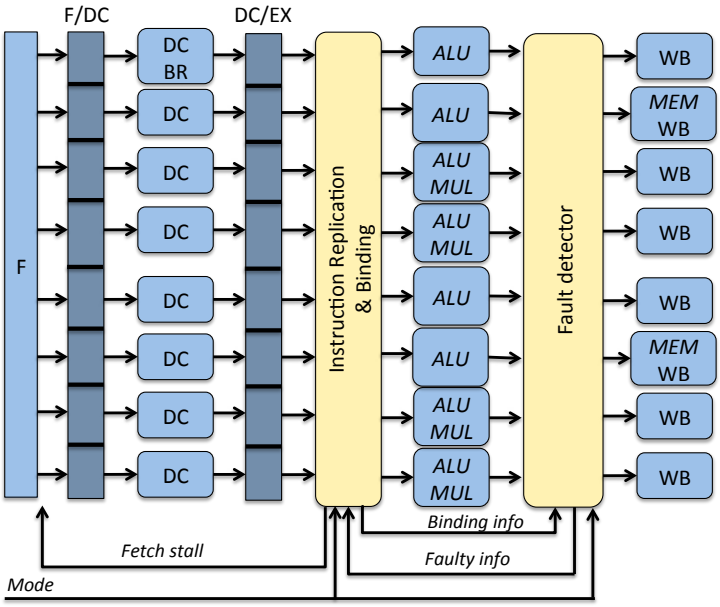


Figure 5.2: Hardware components inserted in the VLIW pipeline.

instructions and binds them in the idle slots of the instruction bundle taking into account the limitations on the number and the type of resources. In case not enough idle slots or FUs exist, a new time slot is added by stalling the fetch of new instruction bundles. The stall of the fetch stage is performed by propagating the fetch stall command to the fetch stage. The faulty information is used by the IRB in case of detected errors. When duplication with re-execution is selected as mode and a temporary error occurs, the IRB stalls the pipeline and re-executes the faulty instruction in a different FU. If a permanent error is detected, the IRB updates the state of the FUs and explores the idle slots and the available FUs to bind efficiently the original and replicated instructions. The binding information is sent to the fault detector to inform how the instruction binding has been performed. The fault detector uses this binding information to decide which results are ready to be compared and committed. When an error is detected, it is initially assumed to be a temporary error. If a number of sequential instructions continue to indicate that the FU is faulty, then the fault detector decides that the error is permanent and sends this information to the IRB to update the status of the FUs. If the selected mode is triplication, the fault detector corrects the error and propagates the corrected value for commit.

5.1.2 Performance Evaluation

We have performed a set of experiments to evaluate the performance gain of the proposed approach. For the experimental part, we have used basic media benchmarks extracted from MediaBench [36] and the VEX VLIW processor [20] with HP VEX compiler. The VLIW is configured based on realistic configuration of resources used by commercial VLIWs, e.g. Intel Itanium [75], as depicted in Fig. 5.2. A simulation tool is developed to calculate the execution cycles of each application compiled with the HP VEX compiler (see fig. 5.3). Intermediate files (.cs.c) from compiled simulator step are instrumented with a code enhancing script, in order to provide us with processor's execution instruction sequence traces when they are linked and compiled with GCC compiler. Our tool parses these traces to calculate the processor's execution cycles and, thus, estimates the performance of each approach. We

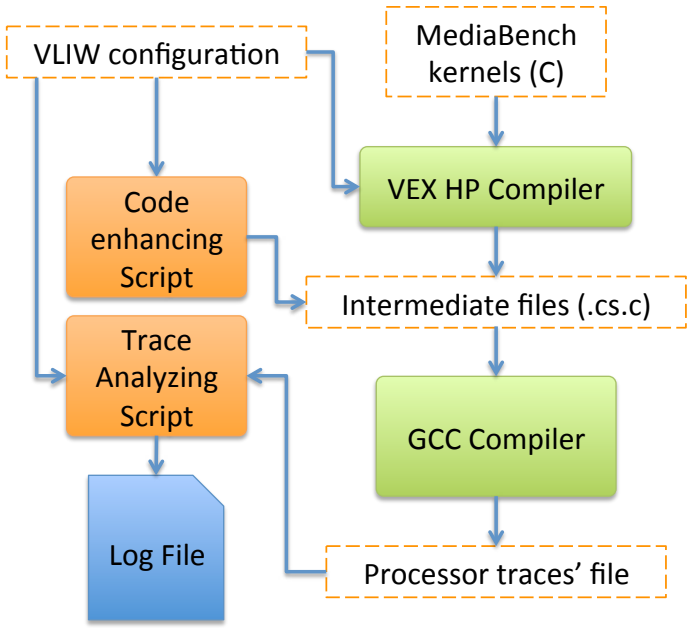


Figure 5.3: Simulation tool flow for performance evaluation results.

perform experiments by applying two fault tolerance techniques with our approach: duplication and triplication of the instructions. We provide performance results for one up to five concurrent permanent errors occurring in any combination of the four different types of FUs of the VLIW. Each time, at least one non-faulty FU exists for each type of required FUs. Otherwise the processor is declared as "out of service", as it is not able to execute every instruction anymore.

Fig. 5.4 depicts the execution cycles estimated for running the applications on the un-

protected original code (N) and on the proposed VLIW approach with duplication (DMR) and triplication of the instructions (TMR) for $p = 0, \ldots, 5$ permanent errors. With our method, the overhead of DMR and TMR with $p = 0$ permanent errors is in most of the cases less than 100%. This occurs as the proposed approach efficiently explores the idle FUs.

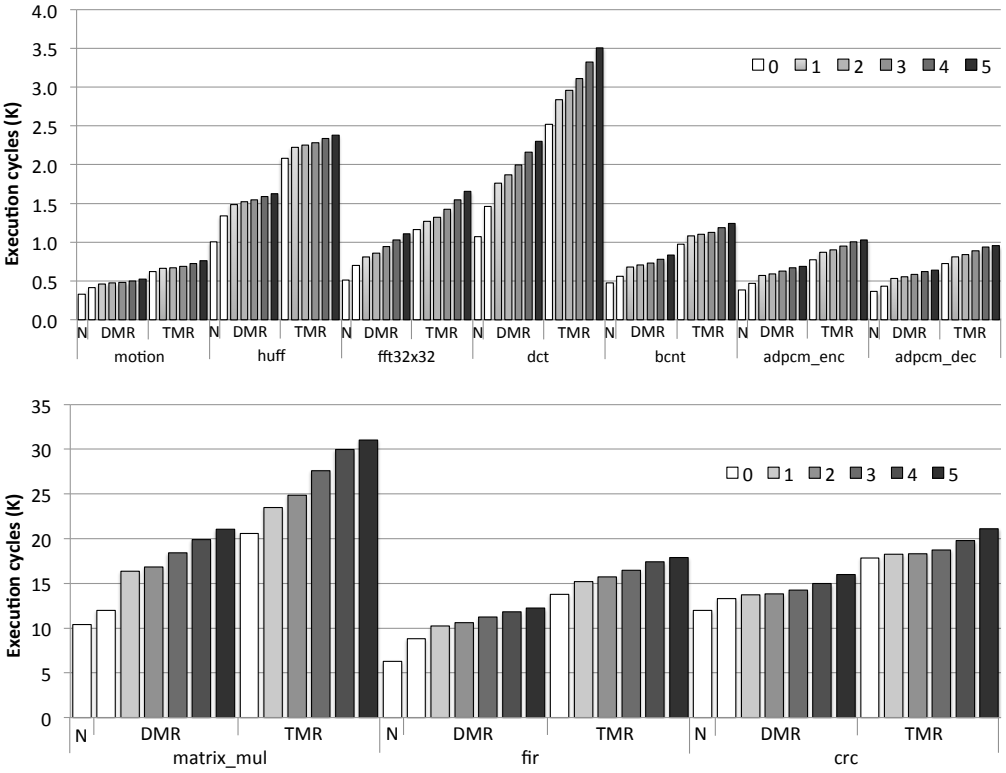


Figure 5.4: Performance comparative results for $p = 0, \ldots, 5$ permanent errors

Table 5.1 depicts the impact of the multiple permanent errors in the performance of our approach implementing DMR and TMR. The performance overhead of DMR (TMR) with $p = 1, \ldots, 5$ permanent errors is calculated relatively to DMR (TMR) with no errors (p = 0). For the DMR, the performance overhead for *motion*, *huff* and *crc* remains quite small, up to 27%, even for 5 concurrent permanent errors. As these applications have a relatively small number of memory and multiplication instructions, the impact in performance is low when errors occur in MEM and MUL units. A similar overhead exists for most of the remaining benchmarks with up to 2 permanent errors. We observe that the concurrent permanent errors affect the performance of *matrix_mul*, with a maximum overhead of 75% in the case of 5 errors. This overhead, in contrast to the aforementioned cases, occurs

because *matrix_mul* intensively uses the multiplication FUs, and, thus, the lack of these resources due to errors leads to high performance overhead. The last row shows the average overhead over all the benchmarks.

Table 5.1: DMR (TMR) performance overhead (%) for the proposed approach with respect to DMR (TMR) without faults.

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

Table 5.2: Performance gain (%) estimation of the proposed approach over existing approaches for multiple permanent errors.

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

Existing hardware approaches are applicable for single permanent errors [73]. Extending them for multiple permanent errors means that all the instructions scheduled by the compiler on a permanently faulty unit have to be re-executed to another unit by adding an extra time slot, like in Fig. 5.1(c). Similarly, hybrid approaches re-execute the instruction that can not be assigned to a slot of the current bundle [71]. The performance overhead of these approaches can be estimated as the execution cycles of the fault tolerance technique, taking

into account the effect of the re-execution each time the instructions have been scheduled by the compiler to permanently faulty units.

Table 5.2 depicts the performance gain of the proposed approach over the existing approaches that re-execute the faulty instructions adding extra cycles. For our approach using the DMR, we observe that for all the benchmarks we achieve a high performance gain even for 5 multiple concurrent permanent errors, as depicted in the left part of Table 5.2. The highest gains have been observed for the *crc* benchmark, from 49% for one permanent error up to 40% for 5 permanent errors and the smallest gains for *matrix_mul* from 32% for 1 permanent error up to 12% for 5 permanent errors. In the case where our approach uses TMR, it has also achieved a high performance gain for all the benchmarks, as depicted in the right part of Table 5.2. We observe that for up to 2 permanent errors we have high gains for all the benchmarks, whereas for the *matrix_mul*, which has high ILP, the gains are slightly reduced for 3, 4 and 5 permanent errors. The gains of the TMR compared to the DMR are higher, even for the instructions with high ILP. This occurs because due to the triplication of the instructions, more time slots are required to be added, which also increases the number of idle issues. As the proposed approach efficiently explores these idle issues and the available FUs, it provides higher gains.

5.2 Fine-Grained Mitigation for Multiple Long-Duration Transients

In order to further decrease the performance overhead introduced by the previous technique and also to support single and multiple Long-Duration Transient (LDT) faults we extend our approach with a new fine-grained hardware mechanism. This mechanism detects the active faults during execution and temporally excludes only the faulty components of the affected FUs for as long as it is necessary. A fine-grained micro-architectural solution is proposed that partitions an FU into components, i.e. an individual circuit that executes a group of instructions. Each component is enhanced with a Built-In Current Sensor (BICS) mechanism to identify the exact location of the fault and the duration that the fault is active. The online fine-grained instruction scheduling mechanism excludes only the faulty FU components for as long as they are affected and reschedules the instructions onto the remaining healthy FU components exploring mitigation solutions in the current and the next instruction execution.

5.2.1 Overview and Motivating Example

We use the 4-issue heterogeneous VLIW data-path of Fig. 5.5 to schematically illustrate our approach. The components in blue color correspond to the basic architecture, whereas we highlight the hardware components added or modified by our approach with yellow color. The proposed approach focuses on LDT faults occurring in the arithmetic FUs, as they have the largest area footprint of the system combinatorial components based on our experiments in Table 3.2. The faults in the storage components, e.g., register file, memory and pipeline registers, are assumed to be protected with other methods, such as Error Correction Codes (ECC).

Before describing in details the two main components of the proposed mechanism, i.e. the fault checker and the online fine-grained scheduler, we illustrate through an example the main idea of this work. Fig. 5.6-a depicts the original schedule of two consecutive instruction bundles, B_{k-1} and B_k, obtained by the compiler. Based on the instruction type, the instructions are assigned to different FU components, as depicted in Fig. 5.6-b. Assume that, at cycle *k-1*, one fault that lasts at least two cycles affects the first and the third FU components of the first issue, FU[0], as depicted in Fig. 5.6-c. The proposed mechanism decides the instruction rescheduling at cycle $k-1$ for the instructions to be executed at

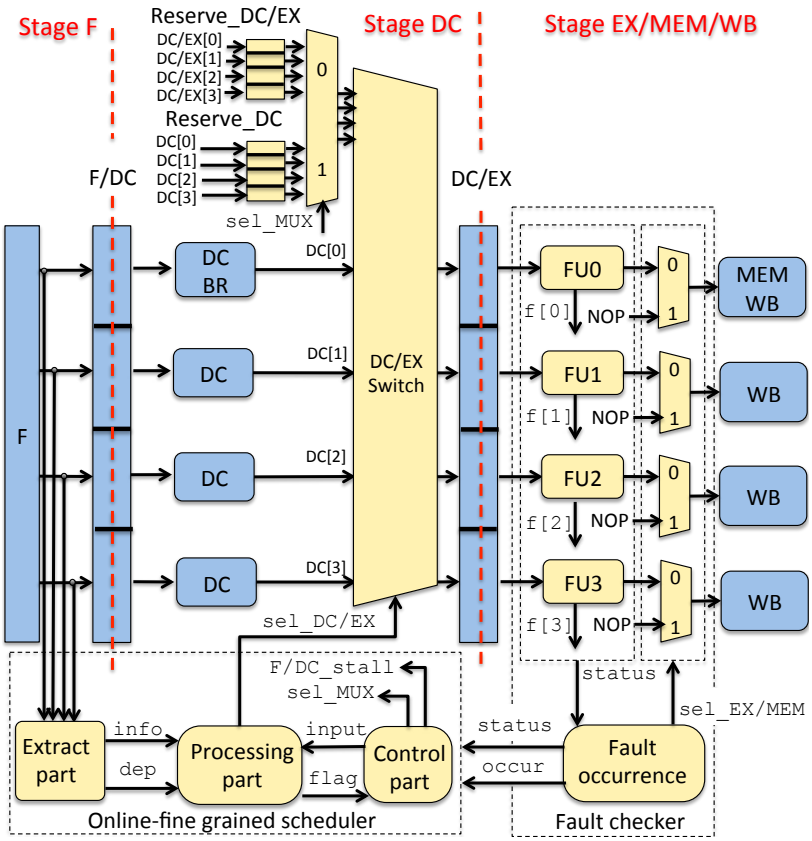


Figure 5.5: VLIW enhanced with the proposed mechanism.

cycle k. In the example of Fig. 5.6-iii, the scheduling of the B_{k-1} instructions is decided at cycle $k-2$. As no fault is detected during execution at cycle $k-2$, the VLIW executes the compiler's original schedule. During execution at cycle *k-1*, the mechanism detects two – just occurred – faults. To have a correct execution, the faulty B_{k-1} instructions must be re-executed at the next cycle avoiding the currently faulty FU components. Therefore, the A_1 instruction must be stored in order to be re-executed at cycle k. At cycle k, although the LDTs persist and the corresponding components have not yet recovered, the mechanism succeeds in executing the remaining instruction A_1 thanks to the re-scheduling of the B_k instructions. This action is allowed, if the instruction A_1 is independent from the B_k instructions.

5.2.2 Fault Checker

The fault checker keeps the faulty status of the FU components, identifies new fault occurrences and takes care of miscalculated results. To achieve a fine-grained use of the

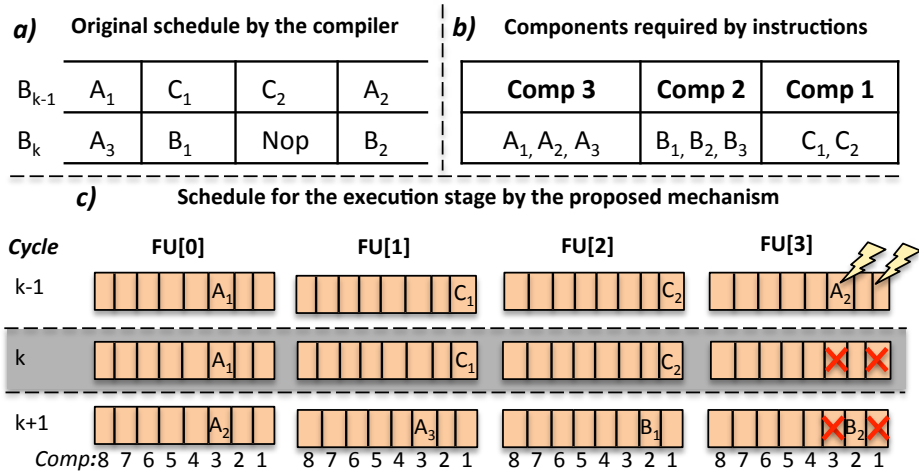


Figure 5.6: Illustration example of the proposed mechanism

components of an FU, each FU is internally enhanced with BICS. Both complex and simple FU types are analyzed to identify the individual circuits. In our architecture, we considered a complex FU as a simple FU enhanced with a multiplication operator. The complex (simple) FU has 15 (14) different FU operations and 8 (7) individual circuits. The circuits are grouped based on the instruction opcode. For instance, the circuit that performs the addition of two registers (ADD operation) is partially shared with the circuit that calculates the address of a memory operation (MEM operation) and the circuit that performs ADDSHIFT operations. As they partially share the same execution path, they are grouped to the same individual circuit. Fig. 5.7 depicts the final obtained individual circuits for the complex FU. The individual circuits for the simple FU are the same without component 5. Each individual circuit and the final multiplexer, which selects the result of the executed operation according to the opcode, is an FU component that is enhanced with a BICS sensor [5]. A BICS is attached to a group of transistors. During normal operation, the current in the bulk of these transistors is approximately zero. Only the leakage current flows through the biased junction, which is still very low compared to the current generated by energetic particles. When an energetic particle generates a current in the bulk, the bulk-BICS captures that a transient fault occurs. The bulk-BICS has a reset mechanism that allows the fault detection to be active only as long as it takes to dissipate the transitory energy pulse. When the fault is vanished, the affected transistors can be used once again.

The output of each BICS sensor is combined into a fault status signal, signal `f`, with a size of 9 (8) bits for complex FU (simple FU). Then, the `f` signals of each FU are combined

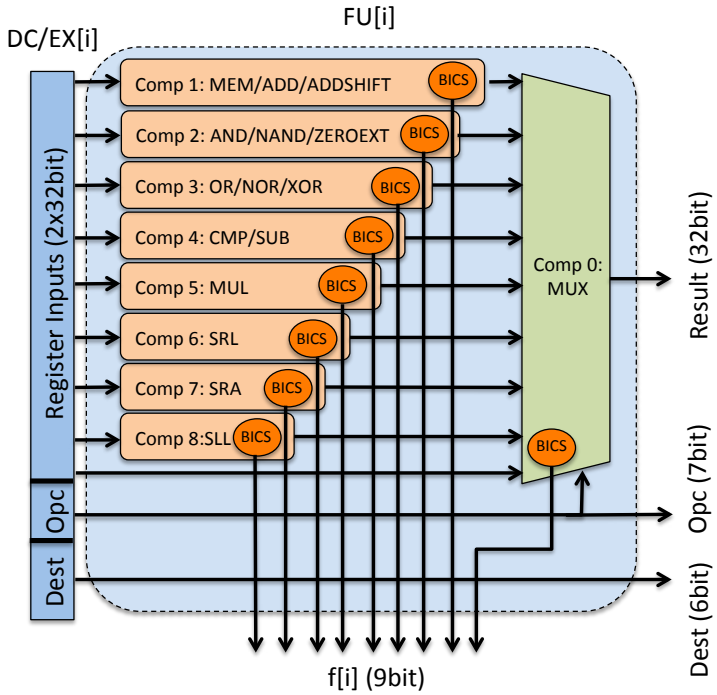


Figure 5.7: Components of complex FU enhanced with BICS.

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

5.2.3 Online fine-grained scheduler

The instructions in the F/DC register are decoded at cycle $k-1$ and the scheduler at cycle $k-1$ decides the instructions to be executed at cycle k based on the `status` of the faulty

FU components. The decoded instructions that couldn't be scheduled at cycle k, due to insufficient FUs or instruction dependencies, are stored to the *Reserve_DC* shadow register. At the same time instance, the EX/MEM/WB stage executes the instructions scheduled for execution at cycle $k-1$ (scheduling decision occurred at cycle $k-2$). The *Reserve_DC/EX* shadow register keeps the instructions executed at cycle $k-1$ in case a fault occurs during their execution. The instructions to be scheduled at cycle $k-1$ can potentially come from three inputs: 1) the decoded instructions at cycle $k-1$ (*DC*) 2) the remaining instructions not scheduled at cycle $k-2$ (*Reserve_DC* register) and 3) the executed instructions at cycle $k-1$ (*Reserve_DC/EX* register).

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

The online fine-grained hardware scheduler is implemented by three components. The first component extracts the required information (`info` signal) from the F stage and the instruction dependencies (`dep` signal). The second component is the scheduler processing part that schedules the input (one of the three potential instruction inputs) to the output of the *DC/EX switch* using bit masks and taking into account the `status` of the faulty components. The third component is the scheduler control part that decides which of the three input signals (*DC*, *Reserve_DC*, and *Reserve_DC/EX*) has to be used as input to the processing part (`input` signal) based on occurrence of new faults (signal `occur`) and the type of the scheduled instructions (signal `flag`). Finally, the output of the processing part is the signal `sel_DC/EX` that controls the *DC/EX switch*. The rest of this subsection describes in details the three scheduler components.

5.2.3.1 Extract part

It performs an early decoding of the instructions at the Fetch stage to create the `info` signal, which consists of the opcode, the destination registers and the source registers of

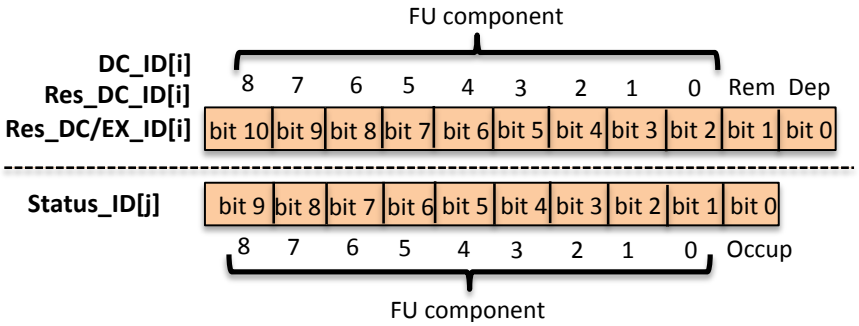


Figure 5.8: ID coding

each instruction (similarly to the information extraction unit of chapter 4). The `info` signal at cycle $k-1$ is locally stored in order to be compared with the `info` signal of the next fetched instruction at cycle k, so as to identify instruction dependencies. The destination of each instruction of the `info` signal at cycle $k-1$ is compared with the destination and source registers of each instruction of the `info` signal at cycle k. If they are the same, the corresponding outputs of the vector signal `dep` are set.

5.2.3.2 Processing part

This part is implemented based on bit masks, called IDentifiers (IDs). Each potential instruction input to the *DC/EX switch* is represented by a table (*Res_DC/EX_ID*, *Res_DC_ID* and *DC_ID*) that has a size equal to the number of VLIW issues. Each table element is an ID that corresponds to the instruction scheduled at position i either by the compiler (*Res_DC_ID[i]* and *Res_DC_ID[i]*) or by the proposed mechanism (*Res_DC/EX_ID[i]*). Fig. 5.8 shows the table element and the 11 bits are coded as follows: a) *bits 10 to 2:* when a bit is set, its position shows the FU component (Fig. 5.7) required for the instruction execution, b) *bit 1:* when it is set, it is a remaining instruction, i.e. it has not been scheduled yet, and c) *bit 0:* when it is set, the instruction has a dependency with at least one of the instructions of the next bundle. For instance, the ID="00000001110" is decoded as: the operation requires the 1 FU component, i.e. it can be a MEM/ADD/ADDSHIFT operation (bit 3=1), the final multiplexer (0 FU component) is required (bit 2=1) and the instruction has not been scheduled yet (bit 1=1). The status of all FUs components is represented by the table *status_ID*, where each element is the `f` signal of a FU enhanced by with an additional bit that is set when the FU is occupied.

The scheduling procedure is given by Alg. 4. The inputs are: 1) the ID of the input signal

to be scheduled, *Input_ID* (each time equal to one of the *Res_DC/EX_ID*, *Res_DC_ID* and *DC_ID*), and 2) the *status_ID*. The outputs are: 1) the control signal *sel_DC/EX* of the DC/EX switch, 2) the updated ID of the input signal and 3) the updated *status_ID*. The procedure is as follows: For all the instructions i of the *Input_ID* (line 3) and for each issue j described by *Status_ID* (line 6), if the instruction i has not been scheduled (line 5) and the FU in the j issue is unoccupied (line 6), check if the required component is available (line 7). If this is true, the occupied bit of the corresponding *Status_ID* is set (line 8), the remaining bit of the *Input_ID* is cleared, since the instruction is scheduled (line 9), and the signal *sel_DC/EX* instructs the switch to pass the instruction currently at issue i to issue j (line 10) and the next instruction is explored (line 11).

Algorithm 4 SCH procedure

```
1: Inputs: Input_ID, Status_ID
2: Outputs: Input_ID, Status_ID, sel_DC/EX
3: for i ∈ {0, n} do                                   ▷ for each instruction in Input_ID
4:     for j ∈ {0, n} do                               ▷ for each issue in Status_ID
5:         if (Input_ID[i][1]) then                    ▷ if the instruction is not scheduled
6:             if Status_ID[j][0] then                 ▷ if the issue is not occupied
7:                 if (Input_ID[i][2 : 10] & Status_ID[j][1 : 9]) then
8:                     Status_ID[j][0] = 1;            ▷ The Status_ID is set to occupied
9:                     Input_ID[i][1] = 0;             ▷ The Input_ID is set to scheduled
10:                    sel_DC/EX[i] = j;               ▷ Instruction at position i pass to issue j
11:                    break;
12:                end if
13:            end if
14:        end if
15:    end for
16: end for
```

5.2.3.3 Control part

The last part controls the inputs and the execution of the scheduler processing part depending on the fault occurrence (`occur` signal) and the type of the scheduled instructions (signal `flag`). The state machine diagram of Fig. 5.9 describes its functionality, where $i \in [0, n]$ and n is the number of issues.

(S1-S2) One (more) faults occurred at cycle $k-1$ (`occ` = 1): The executed instructions on the faulty FU components at cycle $k-1$ (which reside in *Reserve_DC/EX* register) are not committed and they must be scheduled again for execution at cycle k. Whether or not the Fetch and Decode stages must be stalled (`F/DC_stall`) depends on whether the faulty instructions are decoded instructions at cycle $k-2$ (`flag`=0) or at cycle $k-3$ (`flag`=1).

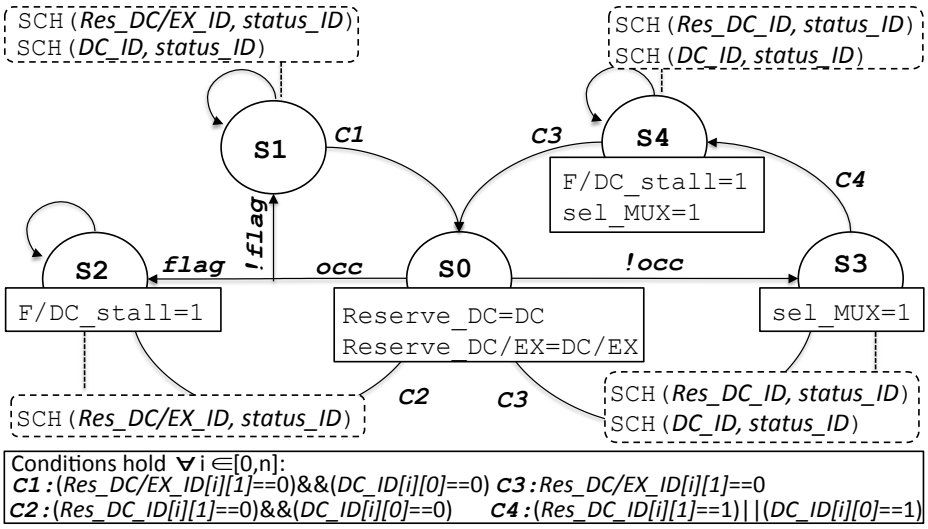


Figure 5.9: Control part.

In the first case (*S1*), no stall is required and these instructions are the first to be executed at cycle k (*Input_ID=Res_DC/EX_ID*). Then, the decoded instructions at cycle $k-1$ are explored (*Input_ID=DC_ID*). An example of this case is Fig. 5.6, where two faults occur during the execution of the instructions of bundle B_{k-1} at cycle $k-1$. During the scheduler decision for execution at cycle k, the remaining instruction A_1 is scheduled at issue 1 and, then, the decoded instructions of B_k are scheduled. In the second case (*S2*), the stall signal is activated (`F/DC_stall=1`) and a new cycle is inserted for the re-execution of the *Reserve_DC/EX* instructions. The process is repeated until no instructions are left in the *Reserve_DC/EX* (guaranteed by condition C2).

S3-S4) No fault occurred at cycle $k-1$ (`occ=0`): If no FU component is affected by a new fault at cycle $k-1$, the mechanism schedules first the remaining decoded instructions from cycle $k-2$ (that now reside in *Reserve_DC*) for execution at cycle k, and, then, the current decoded instructions. If there are still instructions inside the *Reserve_DC* and/or if there is any dependent instruction in current decoded instructions *DC* that cannot be scheduled (condition C4), the `F/DC_stall` signal is activated to stall the Fetch and the Decode stage for one cycle. During this inserted cycle, the mechanism schedules these instructions.

5.2.4 Evaluation results

For the experimental results we used the VEX VLIW processor [20] with two heterogeneous configurations: i) 4-issue configured with 2 complex FUs, 2 simple FUs, 1 memory FU

Table 5.3: Performance comparison (execution cycles) under several multiple faults and average performance overhead (%).

Benchmarks	**4-issue**								
	Original	Fine-grained mechanism				Coarse-grained mechanism			
Num. faults	0	1	2	3	4	1	2	3	4
adpcm_dec	386	388	390	391	397	412	436	464	571
adpcm_enc	409	413	413	425	426	462	469	504	614
bcnt	478	479	480	480	482	479	784	785	1159
fft32x32	569	580	587	591	669	667	775	930	1371
motion	344	350	350	358	368	391	392	407	598
huff	1,101	1,111	1,117	1,119	1,158	1,136	1,137	1,176	1,488
dct	1,288	1,314	1,315	1,456	1,458	1,353	1,627	1,846	2,606
fir	6,852	6,853	6,854	6,901	7,333	7,693	8,593	8,714	11,599
crc	12,228	12,229	12,229	12,231	12,232	12,274	12,275	14,851	20,969
mat_mul	11,142	11,143	12,423	13,010	15,011	15,015	16,039	16,358	21,593
Average overhead %		0.8	2.2	4.6	9.3	10.6	24.1	33.5	82.7

Benchmarks	**8-issue**										
	Original	Fine-grained mechanism					Coarse-grained mechanism				
Num. faults	0	2	4	6	8	10	2	4	6	8	10
adpcm_dec	302	304	311	335	393	401	314	366	442	471	566
adpcm_enc	323	324	326	412	415	420	339	437	483	494	633
bcnt	333	335	336	338	339	397	335	527	545	580	1,085
fft32x32	400	402	403	439	447	483	424	520	728	835	1,428
motion	280	281	282	285	287	302	282	284	363	377	619
huff	951	952	954	957	958	990	959	961	1,072	1,074	1,466
dct	872	877	912	953	954	992	951	1,322	1,369	1,544	2,578
fir	5,709	5,710	5,712	6,012	6,235	6,740	6,070	7,092	7,213	8,105	11,820
crc	11,955	11,956	11,959	11,960	11,989	12,245	11,956	11,958	12,217	15,107	20,964
mat_mul	6,533	6,535	6,538	6,539	7,102	8,112	6,534	10,951	11719	20,333	20,373
Average overhead %		0.3	1.1	6.7	10.3	17.3	3.3	29.1	44.9	69.7	110.6

(MEM) and 1 branch unit (BR), and ii) 8-issue configured with 4 complex FUs, 4 simple FUs, 2 MEM and 1 BR. The processor has been enhanced with the proposed approach. Both the original unprotected VLIW processor and the VLIW with the proposed online fine-grained mitigation mechanism have been developed in C++ and synthesized using the Catapult High Level Synthesis (HLS) tool to obtain the RTL design. The gate-level netlist was generated by the Design Compiler of Synopsys using 28 nm ASIC library. Following this approach, we can both simulate the processor and synthesize a functional RTL design. To evaluate our approach, we use the same ten benchmarks from the MediaBench suite [36], which were used to evaluate the architectures proposed in the previous chapters. The benchmarks are compiled with VEX compiler for each configuration.

5.2.4.1 Performance

We compare the performance of the fine-grained mitigation mechanism with the coarse-grained mitigation approach. In the coarse-grained approach, when a fault occurs in: i) a complex FU, the part that is still healthy – either the simple FU part or the additional multiplication operator – is used and the faulty part is permanently excluded, or ii) a simple FU, the FU is permanently excluded. In contrast, the fine-grained approach explores the FUs in a fine-grained way and applys temporal exclusion.

1)*Fine-grained FU exploration:* In the first experimental part, we evaluate the benefit of our approach only due to the proposed fine-grained FU exploration. To do so, we randomly injected multiple faults during the benchmarks' execution and we consider them as permanent, i.e. they last for the rest of the execution. Table 5.3 shows the cycles required to execute the ten benchmarks considering: i) 0 faults (Original), ii) 1 up to 4 multiple faults for the 4-issue configuration and iii) 2 up to 10 multiple faults for the 8-issue configuration. In this experimental part, we have to limit the number of injected faults to 4 and 10 for the 4-issue and 8-issue configuration, respectively, since this is the maximum number of concurrent faults that the coarse-grained approach can sustain. The performance results are obtained by taking the mean value of 20 simulations running the same benchmark, but the faults are injected at random cycles for each simulation. When no faults occur, both approaches have the same performance, i.e. the original execution cycles. From the Table 5.3, we observe that: 1) the proposed approach inserts significantly lower overhead than the coarse-grained approach, and 2) in several benchmarks our performance is very close to the original one, i.e. without faults, even for several multiple faults. In contrast

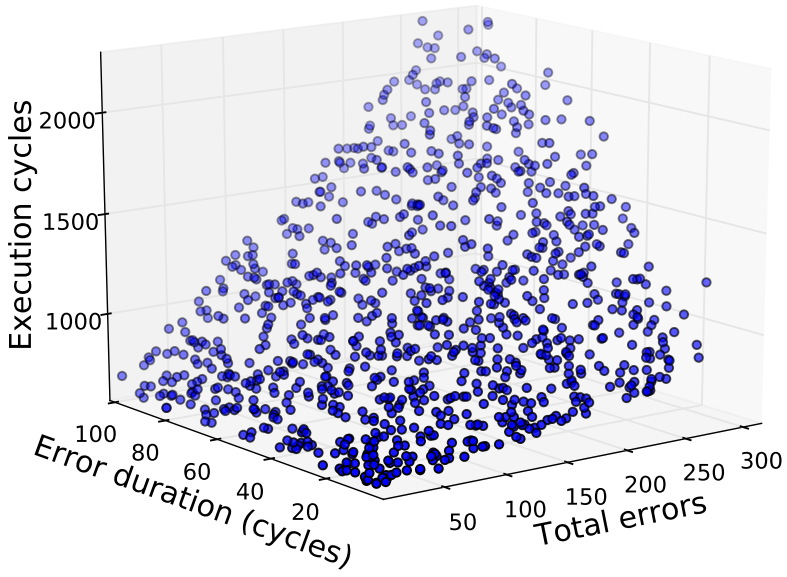
to the coarse-grained approach, the gain of the fine-grained mechanism is achieved because whenever a persistent fault is detected, the proposed approach is capable of still utilizing the healthy FU components in the current and the next instruction bundle.

2)*Error duration and fault stressing:* In the second experimental part, a fault injection simulation framework has been developed that explores the behaviour of the proposed approach under a wide range of faulty scenarios. A script which simulates scenarios with several faults in the FUs is applied to the processor simulator in order to evaluate the performance of the architecture using the proposed mechanism. Depending on the total number of original cycles needed per benchmark, we group them and we stress them under different faulty scenarios. The tuned parameters are depicted in Table 5.4, where Group 1 has *bcnt, adpcm_enc, adpcm_dec, fft* benchmarks and Group 2 has *huff* and *dct* benchmarks. Note that up to 5 new faults can simultaneously occur in a cycle, but the architecture may already suffer from previously occurred, but still active, faults.

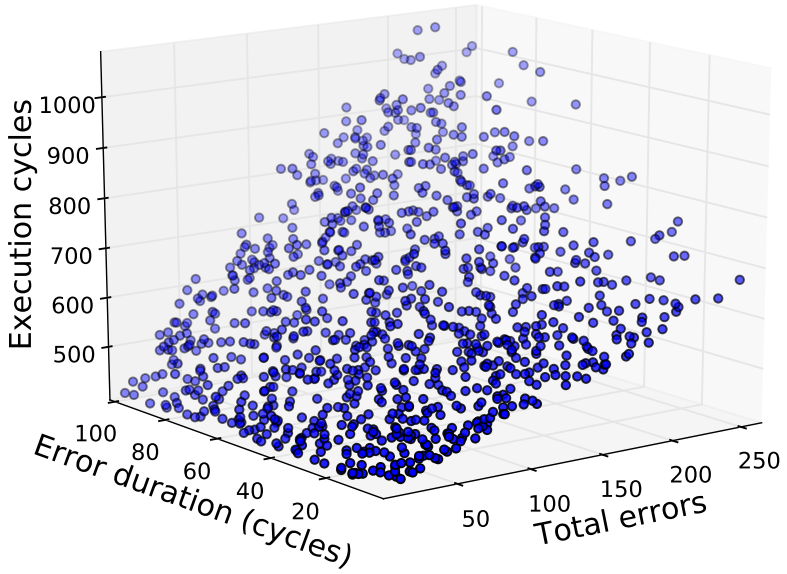
Table 5.4: Tuned parameters (Min, Max, Step) for each group.

Bench. Group	**Total injected faults**	**Fault duration**	**Parallel occ. faults**	**Total simulations**
1 (4-issue)	0, 310, 10	0, 100, 1	0, 5, 1	15,500
1 (8-issue)	0, 260, 10	0, 100, 1	0, 5, 1	13,000
2 (4-issue)	0, 1000, 10	0, 200, 1	0, 5, 1	100,000
2 (8-issue)	0, 850, 10	0, 200, 1	0, 5, 1	85,000

Out of 214,000 simulations performed, we show the obtained results for the *fft* benchmark. This benchmark has been selected as it has a steeper slope compared to the rest. Since it has a higher average ILP (4-issue: 2.85, 8-issue: 4.19) than the rest benchmarks, it introduces less idle slots, and, thus, it further stresses the proposed approach, especially for the 4-issue configuration. The 3-dimensional scatter diagram of Fig. 5.10 is plotted by randomly selecting 1,500 simulations from the total number of experiments. Each dot shows the execution cycles of one complete execution with the proposed approach with respect to the number of injected faults and the fault duration. Generally, we observe that for all the benchmarks with a reasonable amount of injected faults and regardless their duration, the performance degradation is rather negligible. This is because the proposed mechanism: a) is capable of partially utilizing FUs even if they are severely damaged, and b) exploits opportunities in the next instruction bundles, when no sufficient healthy FUs components



(a) 4-issue



(b) 8-issue

Figure 5.10: Proposed mechanism performance for *fft* benchmark under different number of faults and fault duration

exists in the current instruction bundle. From the experimental results, we observe that the proposed approach is capable of dealing with large numbers of injected faults, even for scenarios that the existing approaches are not applicable.

5.2.4.2 Area and power

Table 5.5 shows the area and power results of the implementation of the proposed mechanism with a target frequency of 200MHz. Compared to the unprotected version, the proposed approach implies an area and a power overhead of up to 34% and 33%, respectively. The overhead of the coarse grained approach is expected to be comparable, since both techniques require a switching mechanism and a re-scheduling logic, which are the most costly components of the design.

Table 5.5: Area footprint and power estimation.

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

5.3 Conclusion

In this chapter we proposed a coarse-grained hardware mechanism for permanent errors in order to replicate and schedule the instructions at run-time, thus exploring the idle slots under constraints in the FU number and type. When permanent errors occur, less FUs are available and the proposed approach efficiently rebinds the original and the replicated instructions based on the available resources and idle slots.

In order to protect embedded processors with several FUs against transient faults with a long duration, while also further improving the coarse-grained hardware approach, we proposed an online fine-grained hardware mechanism with low performance, area and power overhead. The FUs of the processor are enhanced with Built-In Current Sensors (BISC) for fine-grained multiple Long Duration Transient (LDT) fault detection and correction. An online rescheduling of the faulty and the current decoded instructions has been proposed that temporarily excludes the faulty FU components. From the obtained results, the proposed

approach can sustain several multiple faults even with very long duration with significant reduction in the performance, area and power overhead.

Chapter 6

Summary and Future Work

6.1 Thesis Summary

Technology scaling and harsh environments have significantly increased the error occurrences in embedded systems making fault tolerance an essential topic for a wide range of domains, especially the safety critical ones. Errors can harm processors temporarily, permanently and semi-permanently. To satisfy the increasing demand for reliability, the systems are designed with error detection and/or error correction capabilities. At the same time, modern systems have to meet the increasing demands in performance, energy, and area efficiency. Very Long Instruction Word (VLIW) processors excel in this category because they offer high performance through Instruction Level Parallelism (ILP) exploitation, while keeping cost and power in low levels.

This dissertation is motivated by the fact that ILP exploitation is not always possible due to applications' fluctuating ILP, thus the available resources of a VLIW processor are not always used. In order to prove this statement benchmarks were analyzed and the results showed that the average ILP is less than 2.85 and 4.46 for a 4-issue and 8-issue configurations, respectively. In order to identify the most vulnerable part of the VLIW architecture used, a fault injection mechanism was also developed that evaluates the Architectural Vulnerability Factor (AVF) and the Instruction Vulnerability Factor (IVF) of the processor. Extensive simulations with fault injections on a simulator software representation of the adopted processor showed that the execution stage takes significant area and it should be prioritized compared to other stages. The findings from the benchmark experiments and the vulnerability analysis in conjunction with the need for fault resilient execution led us

to propose three hardware mechanisms for reliable execution on VLIW processors.

Initially, a hardware rescheduling mechanism for heterogeneous VLIW data-paths was proposed to reuse the idle slots and to provide fault tolerance. The proposed approach explores the idle slots in the current and next bundles and prioritizes dependent instructions. The result is a more compact schedule for both original and replicated instructions. In order to keep the hardware cost low while still providing with full rescheduling flexibility, a hardware-friendly bit-wise instruction scheduler was proposed. In addition, to support scalability, a cluster-based approach is presented and evaluated to avoid area and power overhead with a small decrease in performance. The processor is tested with 10 different media benchmarks and the obtained results show a 43.68% maximum speed-up with an area and power overhead of ∼10% with respect to existing approaches.

A second hardware mechanism was proposed that focuses on permanent error mitigation. The technique replicates and schedules the instructions at run-time, thus exploring the idle slots under constraints in the FU number and type (coarse-grained exploration). When permanent errors occur, less FUs are available and the proposed approach efficiently rebinds the original and the replicated instructions based on the available resources and idle slots. Early evaluation results for an 8-issue VLIW processor prove that the proposed method is able to sustain up to 5 concurrent permanent errors with a significant, though logical performance penalty of up to ∼ 350% compared to other approaches that are not applicable.

In order to further reduce this penalty and enable the FU recovery in case of single/-multiple Long Duration Transient (LDT) errors, a fine-grained hardware mechanism was proposed. During execution, this mechanism characterizes the components of each FU by using Built In Current Sensor (BICS) circuits; it reschedules the faulty instructions to the healthy FU components, and temporarily excludes the faulty ones. From the obtained results, several multiple faults even with very long duration are mitigated with significant reduction in the performance, area and power overhead.

6.2 Directions for Future Work

There are several directions of research that can be followed based on the techniques proposed in this dissertation.

Concerning the AVF analysis performed in Chapter 3, an exhaustive AVF analysis for the proposed technique can be considered in order to obtain a more accurate AVF estima-

tion. An even more accurate fault injection framework can be developed taking the SETs occurring in the combinational logic of the various stages of the pipeline into account. The goal is a modelization for high-level fault injection from gate-level exhaustive fault injection.

Concerning the second contribution, it would also be interesting to explore the implementation of the proposed technique to other systems with multiple function units, such as the Coarse Grained Reconfigurable Arrays (CGRAs).

Concerning the third contribution, a hardware implementation of the rebinding technique applying different fault mitigation techniques (DMR, TMR etc.) could be explored. The improved technique is applied for Long Duration Transient (LDT) errors with the assumption that Built-In Current Sensors (BISC) exist. The BICS enhanced FUs could be physically implemented and the obtained SoC could be tested under realistic radiation stressing conditions.

For all the proposed mechanisms different application domains other than media applications can be tested to see if the performance improvement would be as significant. The emerging domains of approximate computing, cryptographic algorithms, machine learning etc., may be the interesting choices to explore for both performance and energy efficiency.

Acknowledgements

To my life-coach, my mother, Margarita who has always been there for me in the most difficult and yet happy moments of my academic life. Thanks a lot for teaching me that life is all about fighting and never giving up, and that happiness is a state of mind. To my father, Marios, who helped me discover and love science, by stimulating my childish brain. To my sister, Georgiana, my grandma, Anastasia and the rest of my family for their love and support throughout all these years of my studies. Many many thanks and love!! I am also grateful to my friends who have supported me along the way. I also want to thank the God for giving me courage and hope in these moments when everything else has failed!!

I would like to thank my supervisors, Professor Olivier Sentieys and Assistant Professor Angeliki Kritikakou, for the patient guidance, encouragement and advice they provided throughout my time as their student. I consider myself lucky to have such supervisors who cared so much about my work, and who responded to my questions and queries so promptly. In particular, I am extremely thankful and indebted to my co-supervisor Angeliki Kritikakou for sharing expertise, and sincere and valuable guidance and encouragement extended to me. Her help and guidance has been so valuable that I would not be able to reach here without them.

A very special gratitude goes out to the whole Cairn team of the Inria research center, which accommodated and supported me all these 3 years of my PhD research. With a special mention to Simon Rokicki, who provided me with valuable help with his technical expertise. I will miss our brainstorming sessions my friend!! I am also grateful to our beloved team assistant, Nadia Derouault, for her unfailing support and assistance.

Last but by no means least, a special thanks goes to University of Rennes 1 for funding this PhD research and providing me with everything needed for its completion.

Thanks a lot for all your support and encouragement!

Publications

[Psi17a] R. Psiakis, A. Kritikakou, O. Sentieys, "NEDA: NOP Exploitation with Dependency Awareness for Reliable VLIW Processors," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2017.

[Psi17b] R. Psiakis, A. Kritikakou, O. Sentieys, "Run-Time Instruction Replication for Permanent and Soft Error Mitigation in VLIW Processors," *15th IEEE Int. NEW Circuits And Systems Conference (NEWCAS)*, June 2017.

[Psi19a] R. Psiakis, A. Kritikakou, O. Sentieys, "Fine-Grained Hardware Mitigation for Multiple Long-Duration Transients on VLIW Processors," *IEEE/ACM Design Automation and Test in Europe (DATE)*, 2019, **Accepted**.

[Psi19b] R. Psiakis, A. Kritikakou, O. Sentieys, "HW initiated idle resource exploitation for heterogeneous fault tolerant VLIW processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019, **Under Review**.

References

[1] Shail Aditya, Scott A. Mahlke, and B. Ramakrishna Rau. Code size minimization and retargetable assembly for custom epic and vliw instruction formats. *ACM Trans. Des. Autom. Electron. Syst.*, 5(4):752–773, October 2000.

[2] J. Arlat, Y. Crouzet, and J. . Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 348–355, June 1989.

[3] A. Azarpeyvand, M. E. Salehi, and S. M. Fakhraie. Civa: Custom instruction vulnerability analysis framework. In *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 318–323, April 2012.

[4] Ali Azarpeyvand, Mostafa E. Salehi, Seid Mehdi Fakhraie, and Saeed Safari. Fast and accurate architectural vulnerability analysis for embedded processors using instruction vulnerability factor. *Microprocessors and Microsystems*, 42:113 – 126, 2016.

[5] R. P. Bastos, J. M. Dutertre, and F. S. Torres. Comparison of bulk built-in current sensors in terms of transient-fault detection sensitivity. In *2014 5th European Workshop on CMOS Variability (VARI)*, pages 1–6, Sept 2014.

[6] R. P. Bastos, F. S. Torres, J. M. Dutertre, M. L. Flottes, G. Di Natale, and B. Rouzeyre. A single built-in sensor to check pull-up and pull-down cmos networks against transient faults. In *2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 157–163, Sept 2013.

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

[8] G. Blake, R. G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, November 2009.

[9] D. M. Blough and A. Nicolau. Fault tolerance in super-scalar and vliw processors. In *Proceedings of 1992 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 193–200, 1992.

[10] C. Bolchini. A software methodology for detecting hardware faults in vliw data paths. *IEEE Transactions on Reliability*, 52(4):458–468, Dec 2003.

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

[12] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron cmos technology. *IEEE Transactions on Nuclear Science*, 43(6):2874–2878, Dec 1996.

[13] Victor Castano and Igor Schagaev. *Resilient Computer System Design*. Springer Publishing Company, Incorporated, 2015.

[14] Wah Chan and A. Orailoglu. High-level synthesis of gracefully degradable asics. In *EDTC*, pages 50–54, Mar 1996.

[15] Yung-Yuan Chen et al. An integrated fault-tolerant design framework for vliw processors. In *DFT*, pages 555–562, Nov 2003.

[16] Yung-Yuan Chen and Kuen-Long Leu. Reliable data path design of vliw processor cores with comprehensive error-coverage assessment. *MICPRO*, 34(1):49 – 61, 2010.

[17] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule. Hexagon dsp: An architecture optimized for mobile multimedia and communications. *IEEE Micro*, 34(2):34–43, Mar 2014.

[18] M. Ebrahimi, A. Evans, M. B. Tahoori, E. Costenaro, D. Alexandrescu, V. Chandra, and R. Seyyedi. Comprehensive analysis of sequential and combinational soft errors in an embedded processor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1586–1599, Oct 2015.

[19] Jaime Espinosa, Carles Hernandez, Jaume Abella, David de Andres, and Juan Carlos Ruiz. Analysis and rtl correlation of instruction set simulators for automotive microcon-

troller robustness verification. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 40:1–40:6, New York, NY, USA, 2015. ACM.

[20] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[21] Jian Fu. *A fault tolerance framework in a concurrent programming environment*. PhD thesis, University of Amsterdam, 10 2014.

[22] B. Gill, M. Nicolaidis, F. Wolff, C. Papachristou, and S. Garverick. An efficient bics design for seus detection and correction in semiconductor memories. In *Design, Automation and Test in Europe*, pages 592–597 Vol. 1, March 2005.

[23] M. Glorieux, A. Evans, D. Alexandrescu, C. Boatella-Polo, K. Sanchez, and V. Ferlet-Cavrois. Damsel?dynamic and applicative measurement of single events in logic. *IEEE Transactions on Nuclear Science*, 65(1):354–361, Jan 2018.

[24] Badi Guibane, Belgacem Hamdi, Abdellatif Mtibaa, and Brahim Bensalem. Fault tolerant system based on iddq testing. *International Journal of Electronics*, 105(6):1025–1035, 2018.

[25] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 340–347, June 1989.

[26] Luanzheng Guo, Hanlin He, and Dong Li. Application-level resilience modeling for hpc fault tolerance. 04 2017.

[27] J. Guthoff and V. Sieh. Combining software-implemented and simulation-based fault injection into a single fault injection method. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 196–206, June 1995.

[28] John G. Holm and Prithviraj Banerjee. Low cost concurrent error detection in a vliw architecture using replicated instructions. In *ICPP*, 1992.

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

[30] Jie Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, N. Vijaykrishnan, and Mary J. Irwin. Compiler-assisted soft error detection under performance and energy constraints in embedded systems. *ACM Trans. Embed. Comput. Syst.*, 8(4):27:1–27:30, July 2009.

[31] R. Karri et al. Computer aided design of fault-tolerant application specific programmable processors. *TC*, 49(11):1272–1284, Nov 2000.

[32] J.S. Klecka, W.F. Bruckert, and R.L. Jardine. Error self-checking and recovery using lock-step processor pair architecture, 2002.

[33] E. Koser and W. Stechele. Tackling long duration transients in sequential logic. In *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 137–142, July 2016.

[34] E. Koser and W. Stechele. A long duration transient resilient pipeline scheme. *IEEE Transactions on Device and Materials Reliability*, 17(1):12–19, March 2017.

[35] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, July 2003.

[36] Chunho Lee et al. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, pages 330–335, Dec 1997.

[37] Jongwon Lee, Yohan Ko, Kyoungwoo Lee, Jonghee M. Youn, and Yunheung Paek. Dynamic code duplication with vulnerability awareness for soft error detection on vliw architectures. *ACM Trans. Archit. Code Optim.*, 9(4):48:1–48:24, January 2013.

[38] F. Leite, T. Balen, M. Herve, M. Lubaszewski, and G. Wirth. Using bulk built-in current sensors and recomputing techniques to mitigate transient faults in microprocessors. In *2009 10th Latin American Test Workshop*, pages 1–6, March 2009.

[39] C. A. Lisboa, F. L. Kastensmidt, E. Henes Neto, G. Wirht, and L. Carro. Using built-in sensors to cope with long duration transient faults in future technologies. In *2007 IEEE International Test Conference*, pages 1–10, Oct 2007.

[40] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209, April 1962.

[41] M. Maniatakos, M. Michael, C. Tirumurti, and Y. Makris. Revisiting vulnerability analysis in modern microprocessors. *IEEE Transactions on Computers*, 64(9):2664–2674, Sept 2015.

[42] J. W. McPherson. Reliability challenges for 45nm and beyond. In *DAC*, pages 176–181, July 2006.

[43] Mojtaba Mehrara, Mona Attariyan, Smitha Shyam, Kypros Constantinides, Valeria Bertacco, and Todd Austin. Low-cost protection for ser upsets and silicon defects. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1146–1151, San Jose, CA, USA, 2007. EDA Consortium.

[44] Natasa Miskov-Zivanov and Diana Marculescu. Mars-c: Modeling and reduction of soft errors in combinational circuits. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 767–772, New York, NY, USA, 2006. ACM.

[45] Konstantina Mitropoulou, Vasileios Porpodas, and Marcelo Cintra. Casted: Core-adaptive software transient error detection for tightly coupled cores, 2013.

[46] D. Mueller-Gritschneder, M. Dittrich, J. Weinzierl, E. Cheng, S. Mitra, and U. Schlichtmann. Etiss-ml: A multi-level instruction set simulator with rtl-level fault injection support for the evaluation of cross-layer resiliency techniques. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 609–612, March 2018.

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

[48] M. Nicolaidis. Design for soft error mitigation. *IEEE Transactions on Device and Materials Reliability*, 5(3):405–418, Sept 2005.

[49] F. Oboril and M. B. Tahoori. Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, June 2012.

[50] Daniel Oliveira, Vinicius Frattin, Philippe Navaux, Israel Koren, and Paolo Rech. Carol-fi: An efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators. In *Proceedings of the Computing Frontiers Conference*, CF'17, pages 295–298, New York, NY, USA, 2017. ACM.

[51] A. Orailoglu. Microarchitectural synthesis of gracefully degradable, dynamically reconfigurable asics. In *DFT*, pages 112–117, Oct 1996.

[52] G. I. Paliaroutis, P. Tsoumanis, N. Evmorfopoulos, G. Dimitriou, and G. I. Stamoulis. Placement-based ser estimation in the presence of multiple faults in combinational logic. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–6, Sept 2017.

[53] R. Poss, M. Lankamp, Q. Yang, J. Fu, I. Uddin, and C. R. Jesshope. Mgsim - a simulation environment for multi-core research and education. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 80–87, July 2013.

[54] R. Psiakis, A. Kritikakou, and O. Sentieys. Neda: Nop exploitation with dependency awareness for reliable vliw processors. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 391–396, July 2017.

[55] R. Psiakis, A. Kritikakou, and O. Sentieys. Run-time instruction replication for permanent and soft error mitigation in vliw processors. In *2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)*, pages 321–324, June 2017.

[56] Georgia Psychou, Dimitrios Rodopoulos, Mohamed M. Sabry, Tobias Gemmeke, David Atienza, Tobias G. Noll, and Francky Catthoor. Classification of resilience techniques against functional errors at higher abstraction layers of digital systems. *ACM Comput. Surv.*, 50(4):50:1–50:38, October 2017.

[57] R. Rajaraman, J. S. Kim, N. Vijaykrishnan, Y. Xie, and M. J. Irwin. Seat-la: a soft error analysis tool for combinational logic. In *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06)*, pages 4 pp.–, Jan 2006.

[58] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33–42, Oct 1990.

[59] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August. Design and evaluation of hybrid fault-detection systems. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 148–159, June 2005.

[60] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *CGO*, CGO '05, pages 243–254. IEEE Computer Society, 2005.

[61] R. Robache, J. . Boland, C. Thibeault, and Y. Savaria. A methodology for system-level fault injection based on gate-level faulty behavior. In *2013 IEEE 11th International New Circuits and Systems Conference (NEWCAS)*, pages 1–4, June 2013.

[62] Simon Rokicki, Erven Rohou, and Steven Derrien. Hardware-accelerated dynamic binary translation. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, pages 1062–1067, 3001 Leuven, Belgium, Belgium, 2017. European Design and Automation Association.

[63] A. L. Sartor et al. A novel phase-based low overhead fault tolerance approach for vliw processors. In *ISVLSI*, pages 485–490, July 2015.

[64] A. L. Sartor et al. Adaptive ilp control to increase fault tolerance for vliw processors. In *ASAP*, pages 9–16, July 2016.

[65] Anderson L. Sartor et al. Exploiting idle hardware to provide low overhead fault tolerance for vliw processors. *JETC.*, 13(2):13:1–13:21, January 2017.

[66] Anderson L Sartor, Arthur F Lorenzon, Sandip Kundu, Israel Koren, and Antonio CS Beck. Adaptive and polymorphic vliw processor to optimize fault tolerance, energy consumption, and performance. In *ACM International Conference on Computing Frontiers*. sn, 2018.

[67] H. Schirmeier, C. Borchert, and O. Spinczyk. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 319–330, June 2015.

[68] Michael Schlansker, B Ramakrishna Rau, Scott Mahlke, Vinod Kathail, Richard Johnson, Sadun Anik, Santosh G Abraham, and HPL PlayDoh. *Achieving high levels of instruction-level parallelism with reduced hardware complexity*. 1997.

[69] Mario Schölzel. Fine-grained software-based self-repair of VLIW processors. In *DFT*, pages 41–49, 2011.

[70] Mario Schölzel et al. A comprehensive software-based self-test and self-repair method for statically scheduled superscalar processors. In *LATS*, pages 33–38, April 2016.

[71] Mario Schölzel and S. Muller. Combining hardware- and software-based self-repair methods for statically scheduled data paths. In *DFT*, pages 90–98, Oct 2010.

[72] M. Schözel. Hw/sw co-detection of transient and permanent faults with fast recovery in statically scheduled data paths. In *DATE*, pages 723–728, March 2010.

[73] M. Schözel. Hw/sw co-detection of transient and permanent faults with fast recovery in statically scheduled data paths. In *DATE*, pages 723–728, March 2010.

[74] S. Sengupta, K. Saurabh, and P. E. Allen. A process, voltage, and temperature compensated cmos constant current reference. In *Proceedings of the 2004 IEEE International Symposium on Circuits and Systems (IEEE Cat. No.04CH37512)*, volume 1, pages I–325–I–328 Vol.1, May 2004.

[75] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *MICRO*, 20(5):24–43, 2000.

[76] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, pages 389–398, Washington, DC, USA, 2002. IEEE Computer Society.

[77] Smitha Shyam, Sujay Phadke, Benjamin Lui, Hitesh Gupta, Valeria Bertacco, and David Blaauw. Voltaire: Low-cost fault detection solutions for vliw microprocessors. In *Workshop on Introspective Architecture*, 2006.

[78] A. Simevski, R. Kraemer, and M. Krstic. Automated integration of fault injection into the asic design flow. In *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 255–260, Oct 2013.

[79] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions in Architecture and Code Optimisation*, 1(1):94–125, March 2004.

[80] M. Slimani and L. Naviner. A tool for transient fault analysis in combinational circuits. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 125–128, Dec 2015.

[81] T. Sudo, H. Sasaki, N. Masuda, and J. L. Drewniak. Electromagnetic interference (emi) of system-on-package (sop). *IEEE Transactions on Advanced Packaging*, 27(2):304–314, May 2004.

[82] R. Ubal, D. Schaa, P. Mistry, X. Gong, Y. Ukidave, Z. Chen, G. Schirner, and D. Kaeli. Exploring the heterogeneous design space for both performance and reliability. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[83] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. I. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken. Trimedia cpu64 architecture. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*, pages 586–592, 1999.

[84] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. Patterson, and K. Asanovic. The risc-v instruction set. In *2013 IEEE Hot Chips 25 Symposium (HCS)*, pages 1–1, Aug 2013.

[85] W. Xu. *Software-based Permanent Fault Recovery Techniques Using Inherent Hardware Redundancy*. University of Massachusetts Amherst, 2007.

[86] Zhiyi Yu. Towards high-performance and energy-efficient multi-core processors. In K Iniewski, editor, *CMOS Processors and Memories*, pages 29–51. Springer, 01 2010.

[87] H. Ziade, R. Ayoubi, and R. Velazco. A survey on fault injection techniques. *International Arab Journal of Information Technology*, Vol. 1, No. 2, July:171–186, 2004.

List of Figures

List of Tables

List of Abbreviations

ACE Architecturally Correct Execution.

AOM Application Output Mismatch.

AVF Architectural Vulnerability Factor.

BICS Built-In Current Sensor.

BR BRanch.

CEI Correct Execution of Instruction.

DC Decode.

DLP Data Level Parallelism.

DM Data Memory.

DMR Dual Modular Redundancy.

ECC Error Correction Codes.

ETV Execution Time Violation.

EX Execute.

F Fetch.

GPP General Purpose Processors.

HCI Hot Carrier Injection.

HLS High Level Synthesis.

HW Hardware.

IC Integrated Circuit.

ID Instruction Identifier.

ILP Instruction Level Parallelism.

IM Instruction Memory.

IRB Instruction Replication and Binding.

ISA Instruction-Set Architecture.

ISS Instruction Set Simulator.

IVF Instruction Vulnerability Factor.

LDT Long-Duration Transient.

LSB Least Significant Bits.

MEM Memory.

NBTI Negative-Bias Temperature Instability.

NOP No OPeration instructions.

PC Program Counter.

PSF Processors State Failure.

PVT Process, Voltage, and Temperature.

RAW Read After Write.

RF Register File.

SFI Statistical Fault Injection.

SIMD Single Instruction Multiple Data.

SMT Simultaneous Multithreading.

SW Software.

TLP Thread Level Parallelism.

TMR Triple Modular Redundancy.

VLIW Very Long Instruction Word.

WAR Write After Read.

WAW Write After Write.