Binary Tree Classification of Rigid Error Detection and Correction Techniques

Angeliki Kritikakou, Rafail Psiakis, Francky Catthoor, Olivier Sentieys

Binary Tree Classification of Rigid Error Detection and Correction Techniques

ANGELIKI KRITIKAKOU, Univ Rennes, Inria, CNRS, IRISA, France
RAFAIL PSIAKIS, Univ Rennes, Inria, CNRS, IRISA, France
FRANCKY CATTHOOR, IMEC, KU Leuven, Belgium
OLIVIER SENTIEYS, Univ Rennes, Inria, CNRS, IRISA, France

Due to technology scaling and harsh environments, a wide range of fault tolerant techniques exists to deal with the error occurrences. Selecting a fault tolerant technique is not trivial, whereas more than the necessary overhead is usually inserted during the system design. To avoid over-designing, it is necessary to in-depth understand the available design options. However, an exhaustive listing is neither possible to create nor efficient to use due to its prohibited size. In this work, we present a top-down binary tree classification for error detection and correction techniques. At each split, the design space is clearly divided into two complementary parts using one single attribute, compared with existing classifications that use splits with multiple attributes. A leaf inherits all the attributes of its ancestors from the root to the leaf. A technique is decomposed into primitive components, each one belonging to a different leaf. The single attribute splits can be used to efficiently compare the techniques and to prune the incompatible parts of the design space during the design of a technique. This essential single attribute division of the design space is required for the improvement of the techniques and for novel contributions to the fault tolerance domain.

CCS Concepts: • **Computer systems organization** → **Reliability**; **Redundancy**; *Processors and memory architectures*; • **Hardware** → **Error detection and error correction**; **Redundancy**; **Self-checking mechanisms**; **System-level fault tolerance**; **Hardware reliability**;

Additional Key Words and Phrases: Classification, Design Space Exploration, Binary tree

ACM Reference Format:
Angeliki Kritikakou, Rafail Psiakis, Francky Catthoor, and Olivier Sentieys. 2020. Binary Tree Classification of Rigid Error Detection and Correction Techniques. *ACM Comput. Surv.* 1, 1, Article 1 (January 2020), 12 pages. https://doi.org/10.1145/3397268

1 INTRODUCTION

The demand for reliability in modern systems has been significantly increased, especially the last years that technology downscales, making electronics more susceptible to errors [23]. More and more the systems suffer from reliability disturbances, such as Process, Voltage, and Temperature

Authors' addresses: Angeliki Kritikakou, Univ Rennes, Inria, CNRS, IRISA, Campus de Beaulieu, 263 Avenue General Leclerc, Rennes, 35042, France, angeliki.kritikakou@inria.fr; Rafail Psiakis, Univ Rennes, Inria, CNRS, IRISA, Campus de Beaulieu, 263 Avenue General Leclerc, Rennes, 35042, France, rafail.psiakis@inria.fr; Francky Catthoor, IMEC, KU Leuven, Kapeldreef 75, Leuven, 30001, Belgium, Francky.Catthoor@imec.be; Olivier Sentieys, Univ Rennes, Inria, CNRS, IRISA, Campus de Beaulieu, 263 Avenue General Leclerc, Rennes, 35042, France, olivier.sentieys@inria.fr.

0360-0300/2020/1-ART1 $15.00
https://doi.org/10.1145/3397268

(PVT) variations [46], circuit aging-wearout induced by failure mechanisms, such as Negative-Bias Temperature Instability (NBTI), Hot Carrier Injection (HCI) [28], radiation-induced Single-Event Effects (SEEs), such as Single-Event Upsets (SEUs) and Single Event Transients (SETs) [2], clock skews [38], thermal stress [48], electromagnetic interference, such as cross-talk and ground bounce [49], etc. These sources can cause errors which can harm systems temporarily (soft errors), permanently (hard errors) and semi-permanently (intermittent errors). To satisfy the increasing demand for reliability, the systems are designed with error detection and/or error correction abilities. Error detection is responsible for identifying the existence of an error during the system execution, whereas error correction is responsible for taking immediately actions to correct the error, so as the system to have an error-free output.

At the same time, the systems have to meet the increasing demands in performance, energy, and area efficiency [21]. Within thirty years, the code size of automotive, space and avionics applications has significantly increased [6]. As the system design complexity significantly grows and performance, power and area requirements have to be preserved, reliable solutions are required to meet the system requirements, without inserting unnecessary overhead [5]. A wide range of error detection and correction techniques have been proposed by engineers and researchers. But in order to decide the most suitable techniques for a given design, an in-depth understanding of the available design options is required. However, an exhaustive list of all the possible design options is not possible due to the prohibited size of the design space, and thus, hierarchical structures (a.k.a. classifications) are required in order to group approaches with similar attributes.

Existing classifications of error detection and correction techniques are usually created bottom-up by studying the published work and grouping the techniques with similar characteristics. As a result, the structure of the proposed classifications is usually a wide tree with multi-way splits. A multi-way split is driven by several attributes at the same time. Such classifications can provide a quick overview of the main existing approaches. However, they suffer from some limitations, when a more detailed and in-depth understanding of the available design options is required. First of all, due to the nature of multiple-way splits, several attributes are repeated in different classes, and clear bounds, that explicitly separate the classes, are missing. As a result, the classification of error detection and correction approaches can be ambiguous. For instance, a very common used split in existing multiple-way classifications is between redundancy and coding, such as in [18]. However, this split has no clear bounds among its classes. Error Correction Code (ECC) approaches, such as [3], belong to the coding class, but at the same time require redundancy in space in order to store the extra bits. REcomputed with Shifted Operands (RESO) [29] approach classified as temporal redundancy. However, the shifting operations perform an encoding and decoding of the operands, it has also a part of encoding. Further illustration is provided in Section 3. Another restriction of existing approaches is related to the fact that multi-way splits are obtained by analyzing a set of specific techniques. An ad-hoc selection does not imply that the complete design space is covered, whereas systematically sampling such a set of techniques is not straightforward. Therefore, a classification with clear bounds among classes, that unambiguously structure the available options of the design space, is essential.

The purpose of this work is to present such a classification for the design space of error detection and correction techniques, i.e., approaches able to either only detect an error or to detect and reconstruct the correct value from incorrect data during execution. The proposed classification is created by applying a top-down recursive partitioning methodology. Each split is i) univariate, i.e. it is driven by a single attribute, and ii) binary, i.e. it partitions the parent class into two non-empty and non-overlapping subclasses that are complementary. With this top-down binary split the design space is divided into two subclasses, each one describing a unique part: one class includes the part of the design space that has the attribute and the other class includes the part of the

design space that does not have the attribute. The attribute defines the most important subclass to be explored first during the exploration of the design space. To continue the partition process, the subclass that does not include the attribute (described by a negative attribute expression) is transformed to an affirmative one and the process is reapplied for each of the subclasses. At the end, a leaf describes a class that inherits all the characteristics progressing from the root node to the leaf node. Therefore, the obtained binary tree classification divides the design space into a set of non-overlapping and complementary classes, removing the limitations of existing multi-way classifications. For instance, the proposed binary classification resolves the aforementioned problem of redundant and encoding classes. It applies a binary split between the type of the additional information to be used for comparison (left branch) and the way this information is created (right branch). Hence, based on the left branch, hardware duplication approaches [10] create identical information as the original system, ECC approaches [3] create compressed information, and RESO approach [29] creates encoding information. Then, based on the right branch, a second identical hardware circuit is inserted in [10] and an extra cache [3] (similar to spatial redundancy), whereas approach [29] re-computes encoded information using the original hardware (similar to temporal redundancy). The proposed classification describes a technique as a combination of primitive components which belong to different classes. We provide examples of techniques that belong at each class in order to provide in-depth understanding of the classes and the attributes. To further support our contribution and to evaluate our classification, we decompose a set of representative and diverse existing error detection and correction techniques, which largely cover the exploration space. From the obtained result, we can extract a number of observations, that indicate which of the design options have been highly explored by existing approaches and which classes remain still unexplored. In this work, we focus on rigid techniques, i.e. static or dynamic techniques without self-adaptation capabilities. This means that the approach output depends on information obtained during execution, but its functionality, i.e., the principle followed to take decisions, cannot be modified during execution.

The rest of this study is organized as follows. Section 2 presents the target domain and formulates the problem under study. Section 3 provides the related work on classifications and surveys on fault tolerance. Section 4 gives the top-down partitioning methodology. Section 5 analyses the binary tree classification, presents the attributes of each split and provides examples of each class using existing techniques. Section 6 presents the decomposition of a set of representative techniques in the proposed classes and provides examples on how the proposed classification highlights the differences and the similarities of the techniques. Finally, Section 7 concludes this study. The annex 1 presents the details of the complete decomposition of all the studied techniques.

2 TARGET DOMAIN AND PROBLEM FORMULATION

Our target domain is systems implemented as Systems-on-Chips (SoC), i.e. an integrated circuit that includes processors and numerous digital peripherals packaged into a single chip. Our target domain includes platforms that i) have hardware components that are programmable using *instructions* and ii) implemented by mapping directly the algorithm operations to hardware. The first case includes typical platforms that are programmable using an Instruction Set Architecture (ISA), such as a RISC processor. This category also covers platforms with flexible hardware components, that can be programmed using a set of control bits, such as a Coarse-Grained Reconfigurable Arrays (CGRAs). The second case refers to fine grained configurable platforms, where the algorithm operations are directly mapped to hardware operators, such as Field-Programmable Gate Arrays (FPGAs), where the Control Logic Blocks (CLB) are configured to implement the hardware operation.

To make a system reliable, fault tolerant mechanisms have to be designed and added to the system. We are focusing on the techniques that are: 1) rigid, i.e. without self-adaptation capabilities

during execution and 2) capable of detecting an error and potentially also correcting it during execution. Rigid techniques include static or dynamic approaches, whose output depends on the information and the system state, obtained during execution. However, the principle followed in order to take decisions, during execution, is defined at design time and cannot be modified during execution. Approaches with self-adaptation capabilities change the principle followed to take decisions during execution, and thus, they have additional characteristics considering how to decide the self-adaptation and how to implement it during execution. Therefore, the classification of approaches with self-adaptation capabilities is an extension of the classification of rigid approaches, and we will cover it in our future work. Our work focuses on error correction approaches that have as goal to reconstruct immediately the correct value during execution. Error mitigation approaches, that have as goal to over-write the impact of the error, are excluded from the proposed classification. Such approaches are usually based on excluding the faulty hardware components from execution (e.g., task re-mapping approaches) or keeping intermediate correct states, that can be used to roll back to a known correct state and continue execution (e.g., check-pointing approaches). The work of [33] presents such a classification for the error mitigation approaches under functional errors, being orthogonal to the proposed classification for error detection and correction approaches. The error detection and correction techniques under study are dedicated to faults occurring on the hardware of the SoC and they may manifest themselves as errors and failures at the software part. It excludes the errors whose source is the inconsistent code of the software part, i.e. software flaws. The classification is not restricted with respect to the type of the fault occurring on the systems, i.e. it includes techniques for both parametric and functional errors [40]. The classification does not focus on separating the techniques to different abstraction levels of the digital system design, as each class can be further instantiated to the appropriate abstraction levels depending on the requirements of the design. With respect to the top-down classification for reliability modeling [40], the proposed classification further projects the lower layers of [40] focusing on error detection and correction techniques.

The goal of this work is to provide a binary classification which divides the design space into classes describing "how to" perform error detection and correction, so as to support the design of reliable systems under our target domain. The classification is described by a set of complementary and non-overlapping classes. Hence, a technique is divided to a set of primitive components and each component is classified into one of the proposed classes.

3 RELATED WORK

The large literature in fault tolerant approaches clearly shows the importance of protecting the system from faults. Several works exist that present a classification of fault tolerant approaches. However, most of these classifications are usually based on tree structures with multi-way splits, i.e. splits driven by several attributes, and independent classes, i.e., classes that cannot be combined with each other. Although a multi-way tree reduces the depth of the classification tree, the use of several attributes to characterize a single class may lead to partially overlapping classes and an ambiguous categorization of the approaches. Other existing works are surveys addressing the new advances and challenges of fault tolerant techniques, without having as a goal to propose a new classification scheme.

One example of existing classifications with multi-way split is the classification of [7], which is a taxonomy of on-line error detection mechanisms focusing on multicore processors. The schematic representation is depicted in Fig. 1. This taxonomy is restricted to a smaller scope than the proposed classification, as it focuses only on techniques applied on the core level for multicore processors. Therefore, several error detection techniques are excluded, such as methods which extend the hardware, like double sampling techniques [27] and error correction approaches.

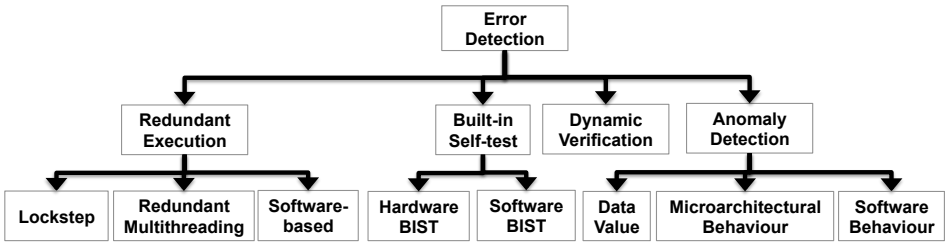


Fig. 1. Error detection taxonomy for multi-core architectures of [7].

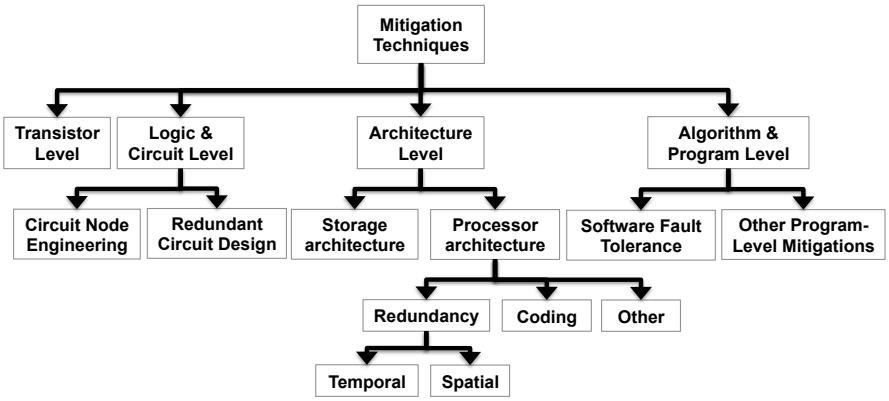


Fig. 2. The classification of soft error detection and mitigation techniques in [18].

Another classification is the work presented in [18] for mitigation techniques against soft errors at processor architectures. The classification is mainly performed based on the abstraction level of the mitigation techniques, i.e. transistor-level up to program-level approaches, as depicted in Fig. 2. A specific distinction is made for the architecture level between error detection and recovery. However, inside each category a survey-like presentation of the state of the art approaches is given, without further systematic classification. The architecture level is further categorized into storage and processor architecture. The storage branch describes mainly error detection and correction codes. The processor branch is divided into redundancy (spatial and temporal), coding and others. However, as the splits are multi-way, driven by several attributes at the same time, ambiguities in the categorization of the techniques can rise. Such an example is the RESO approach [29]. The technique shifts the operands and re-executes the instruction. Although this approach is classified under the temporal redundancy due to the instruction re-execution, the shifting operations are performing encoding and decoding, and, thus, this technique has also coding aspects. On the other hand, the proposed binary tree classification based on univariate splits resolves this ambiguity. As shown in section 6, RESO is categorized by combining the "encoded information" and the "postponed" execution classes.

Another error detection classification is presented in [16]. The schematic representation of the classification is depicted in Fig. 3. The main goal is to classify the techniques based on the platform hierarchy where they are applied to, i.e. circuit level, architecture level, software system, application level and hybrid between different levels. As the provided splits are multi-way, several categories presented in each level can include redundant information. For instance, the approaches using any

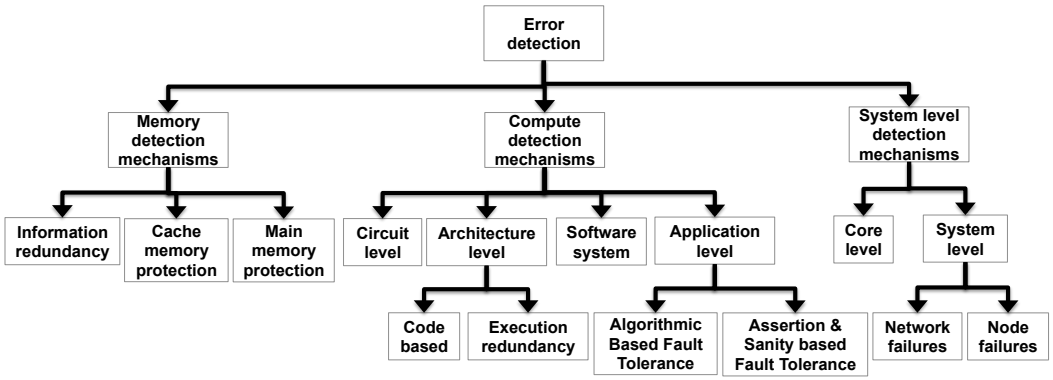


Fig. 3. The classification of error detection and mitigation techniques in [16].

type of coding, such as error detection codes, belong to all the categories presented for the memory detection, i.e. information redundancy, cache memory error protection and main memory error protection. Another example is the code-based approaches and the execution redundancy of the architecture-level. As the code-based redundancy techniques require a parallel execution of part of the instruction to create the redundant information, they describe a part of the design space that overlaps with the execution redundancy class. On the contrary, the binary splits of the proposed classification tree describe these code-based approaches by combining the "compressed value" class and the classes that represent the modifications in the computational functionality of the data path and the storage, as explained in Section 4.

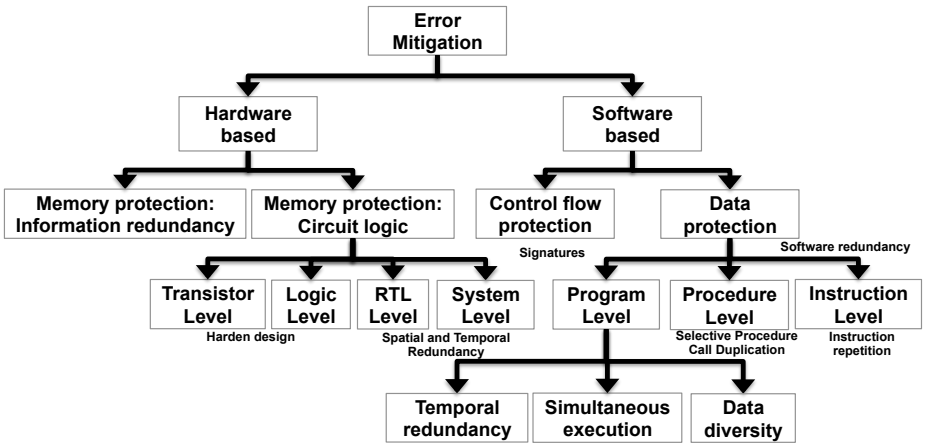


Fig. 4. Error mitigation classification for soft-core processors [22].

The survey of [11] focuses on checker architectures for error detection. It classifies the architectures based on: 1) their abstraction layer (specialized circuits/pipeline functional units, spare cores, spare threads and software modules), 2) the application (single thread or multi-thread), 3) the fault type and coverage and 4) the checker structure. In each of the categories, a set or architectures is listed and categorized based on the fault type and fault coverage. Complementary to this survey, the

proposed classification focuses on further refining the classes with respect to the implementation of the techniques.

The classification of [22] focus on soft-error mitigation approaches in soft-core processors and it is depicted in Fig. 4. Most of the classes focus on some kind of redundancy, either in software or hardware, in space or in time. Partial hybrid solutions are enabled by allowing to combine hardware and software approaches. However, techniques based on monitoring, such as hardware fault screeners [34], are left outside of this classification scheme.

A classification of variability mitigation techniques on timing faults in modern integrated systems is presented in [36]. The classification is based on the Y-chart model, where the techniques are categorized into three axis based on when and how the timing errors are manipulated: 1) error prediction and prevention techniques, 2) error detection and correction techniques, and 3) error acceptance techniques. Each axis is divided into different abstraction levels, i.e. circuit, hardware, software and application/algorithm. The main classified error detection methods are sensing at the circuit level, whereas the architecture level is based on these sensors to provide mitigation. This work allows two hybrids to exist between the axes of Y-chart model: 1) error prediction and prevention with error detection and correction, and 2) error detection and correction with error acceptance. In addition, cross-layer approaches are allowed by combining techniques in different abstraction levels. On the other hand, the proposed classification uses single attributes to refine the implementation of the error detection and correction techniques.

Another classification of fault management techniques for SRAM-based FPGAs is presented in [47]. The techniques are categorized to failure prevention (including design time fault avoidance and operational failure avoidance) and failure tolerance (including failure masking, failure recovery and goal change). However, no error detection classification is presented and classes based on monitoring are excluded, such as hardware fault screeners [34].

The goal of [1] is to provide the main definitions with respect to dependability and to explain general concepts. According to the fault tolerant techniques classification, the error detection techniques are divided into concurrent techniques, which take place during normal service delivery, and preemptive techniques, which are applied when the normal service is suspended. This categorization describes only a part of the techniques, mainly the part relevant to "when" the detection technique is applied.

Survey studies also exist, which, however, present the new advances of the error detection and correction techniques or the future challenges. They do not expand the previous classifications to include the new approaches or provide a new complete classification scheme. A paradigm of this category is the work presented in [39], where various techniques for different electronic systems and their components are discussed and listed. Another example is the work [8]. The mitigation techniques are presented into the following classes: 1) hardware-level, 2) software-level, 3) operating system (thermal management), and 4) application-level. However, no claim is made that this work presents a classification approach, as the main contribution is the presentation of the future perspectives and trends in the reliability domain.

The existing related work on classification of error detection and correction techniques is based on multi-way splits, that take into account several attributes each time, usually leading to categories describing overlapping parts of the design space. Consequently, the existing classifications have limited usage, when researcher and engineers need to select the most suitable error detection and correction techniques for a given design under study. In this work, we provide a binary tree classification that divides the design space into complementary and non-overlapping parts, using a single attribute at each split, supporting a clear distinction of the techniques' characteristics, that can drive the design of error detection and correction techniques.

4 TOP-DOWN PARTITIONING METHODOLOGY

In this work, we present a binary tree classification structure based on a top-down recursive partitioning methodology. The root of our classification tree is the design space describing the error detection and correction techniques for the target domain under study described in Section 2. Initially, the parent class (root) is divided by applying a top-down split.

Definition 4.1 (Top-down split). A top-down split (Fig. 5(a)) is univariate, i.e. is driven by a single attribute Q, and binary, i.e divides the design space of a class P into two primitive subclasses $S1$ and $S2$, that are i) non-empty, ii) complementary, iii) non-overlapping and iv) complete [1].

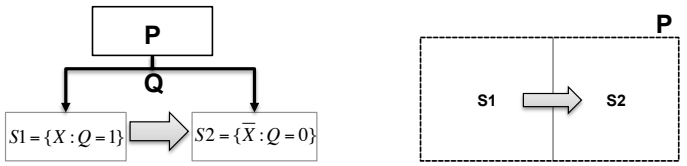


Fig. 5. a) Top-down split and b) design space division. Arrows show the subclasses unidirectional propagation.

Through a top-down split, the parent design space is divided in two sub-spaces, either including the attribute or not. In order to enable the repetition of applying top-down splits to both created subclasses, the subclass that describes the negative part of the attribute has to be reformulated:

Definition 4.2 (Negative attribute reformulation). During reformulation, the negative part of the attribute is rephrased into an equivalent affirmative description in order both subclasses to be positive[2].

A top-down split instantiates the parent class by projecting it to the dimension of the attribute that drives the split. Both subclasses inherit the attributes of the parent class, whereas each subclass adds the characteristic of satisfying or not the attribute that drove the split. Therefore, we define the top-down split type based on the attribute that drives the split:

Definition 4.3 (Top-down split types). A top-down split can be one of the following types:

- *What* type: The "what" top-down split instantiates the parent class into two subclasses that describe the different functionalities of the parent class.
- *How* type: The "how" top-down split instantiates the parent class into two subclasses that describe the different ways to implement the functionality of the parent class.

During the refinement of the classification, the more general splits are applied close to the root of the classification tree, i.e. in the higher levels, whereas in the lower levels the splits become more specific, due to the restriction coming from the attributes inherited from the higher levels. A desired, but not strictly required, feature of the top-down split is balance. A balanced split supports an approximately equal search of the different characteristics of parent space, i.e. the characteristics are approximately equally distributed over the subclasses[3]. The top-down partitioning methodology can stop when the distinctive aspects, that are desired to be present in the classification, have

[1] $S1, S2 \subset P$, where i) $S1, S2 \neq \emptyset$, ii) $S1 = \{X: Q = 1\}$ and $S2 = \{\overline{X}: Q = 0\}$, iii) $S1 \cap S2 = \emptyset$, and iv) $S1 \cup S2 = P$
[2] $Y = \overline{X} \implies S2 = \{Y: Q = 0\}$
[3] $|S1| \approx |S2|$

been expressed. At the end of the top-down partitioning methodology, a leaf has all the attributes progressing from the root node to the leaf node accumulated by conjunction, i.e. using an operation of AND type.

Few binary tree classifications exist based on a similar methodology, but applied in different contexts: design-time scheduling approaches [13], reliability violations [40] and error mitigation for functional errors [33]. In contrast to these earlier works, we apply the top-down methodology for error detection and correction techniques. In addition, we further refine this methodology in order to structure the design space (i.e. through the proposed classification) in such a way that can be used for the design of an error detection and correction technique, i.e. design space exploration. To do so, the top-down split definition is extended to provide an ordering between the two subclasses. This ordering can be used during design space exploration, when both subclasses of a split are eligible to be part of the technique. The ordering implies which part of the design space (the one related to $S1$ or the one related to $S2$) should be explored first. This ordering is important, because the design decisions taken during the exploration of the first subclass are becoming design constraints during the exploration of the second subclass. For instance, the decision on "what type of additional information will be used for detection and correction" becomes a constraint during the decision of "how this additional information will be created". The propagation flows from the decision over the type towards the implementation, since this information is required in order to decide how to create it. The attribute Q, which drives each split, defines the ordering of the subclasses. It should be noted that the ordering presented in this classification is based only on the principles deriving from the structure of the design space itself, as described in Definition 4.5. The approach could be further extended by inserting potential constraints from the application and the platform specifications.

Definition 4.4 (Subclasses unidirectional propagation). During design space exploration, when both subclasses $S1$ and $S2$ of a parent class P are eligible to be part of the technique, the unidirectional propagation shows the order of exploration between two subclasses[4]. This ordering is depicted by the horizontal arrows of Fig. 5(b). The direction depends on the attribute Q, driving the top-down split. When no combination of the $S1$ and $S2$ is possible, no propagation exists.

Definition 4.5 (Principles of structure-based constraints). The principles applied over the attribute Q, to define the propagation direction, are based on constraints from the design space structure:

(1) *Search space restriction:* This principle is mainly related to the "how" top-down split type. As subclasses of a "how" split describe the different possible implementations of the parent class, propagating the design choices of one subclass, e.g. $S2$, may unnecessary restrict the options in the design space of the other subclass, e.g. $S1$. This unnecessary restriction refers to the pruning of potentially promising design options of $S1$ design space, only due to the propagation of the early design choices of $S2$, and not due to the real constraints of the problem under study. In order to avoid that, the constraint propagation has to flow from $S1$ to $S2$.[5] This principle is schematically depicted in Fig. 6(a), where the propagation of the early design choices of $S2$ prunes a large part of the design space of $S1$ (gray part). However, in Fig. 6(b) the decisions of $S1$ are propagated to the $S2$ and the space of $S2$ is lightly restricted. In the remainder of this paper, the term *R_SPACE* is used to annotate the classification splits that follow the search space restriction principle.
(2) *Implied decision:* This principle is related to both the "what" and the "how" type of top-down splits. The design choices of one subclass $S1$ are required in order to select meaningful options in the design space of the second subclass $S2$. To avoid random and ad-hoc decisions for $S2$,

[4]The unidirectional order can be expressed as $S1 \rightarrow S2$
[5]$S2 \rightarrow S1 \implies |S1| \approx 0$, while $S1 \rightarrow S2 \implies |S2| > 0$

the propagation has to flow from $S1$ to $S2$. This principle is depicted in Fig. 6(c), where the decisions for $S2$ cannot be taken, as essential information coming from $S1$ is missing. However, in Fig. 6(d) meaningful decisions in both subclasses can be made. The decisions in $S1$ do not require information of $S2$ and for $S2$ all required information is available, after the propagation of the information from $S1$ to $S2$. In the remainder of this paper, the term *IMPL* is used to annotate the classification splits that follow the implied decision principle.

To illustrate the above principles, we present two examples:

- *Search space restriction:* Based on the previous example relevant to the split of "How to obtain the additional value to be compared with the original one in order to detect/correct a fault" the two subclasses are "a golden reference" ($S1$) and "online produced value" ($S2$). Assume that the ordering between the two subclasses, when they are combined, is to first explore the options in the design space of the online produced value and, then, explore the options in the golden reference. Exploring the online produced value first, propagating the selected design options to the golden reference, removes promising options from the design space. As long as it has already been decided how to online produce the value to compare with, the golden reference decision cannot be of any assistance. However, exploring first the design space of the golden reference, it is possible to combine it with the design options in the online value creation. The choice on the golden reference can provide information to drive the choices on the online produced value.
- *Implied decision:* Assume the split between the "additional information" ($S1$)– what is the type of the additional information to be used so as to perform error detection/correction – and the creation of this additional information ($S2$). We cannot make a meaningful decision on how to create the additional information if we are not aware of which type of information we want to create.

However, some splits are not linked to the aforementioned structure-based constraints. Instead, several objectives exist that can be traded-off in an N-dimensional cost space, e.g., execution time, area, energy etc. For these cases, we group these objectives under a high-level trade-off principle, that has as goal to keep low the overhead inserted by an error detection and correction approach at any of these objectives.

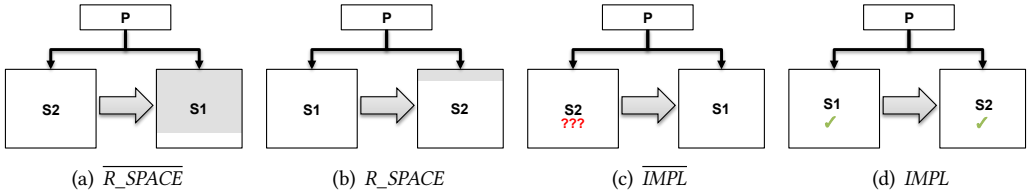
Definition 4.6 (Trade-off principle). Reduce modifications: Our assumption is that the more modifications are inserted to the original system by the error detection and correction approach, the higher is the overhead. This is a relatively crude approximation, but sufficient for our context, where only relative orderings matter at the higher layer top-down splits. Therefore, one subclass $S2$ can imply significantly more modifications than the other class $S1$. In order to reduce the number of modifications, the constraint propagation flows from $S1$ to $S2$. In the remainder of this paper, we link this principle to the term *R_MOD* and we use this term to annotate the relevant classification splits.

Here is an example based on the trade-off principle:

- *Reduce modifications:* Assume the split with respect to "Which online produced values can be used to acquire the additional value to be used for comparison". The two subclasses are "reuse already existing values", produced due to other computations reasons and not for fault tolerance ($S1$), and explicitly "create new values to be used for fault tolerance" ($S2$). The creation of new values requires higher modifications than reusing values that already exist in the system.

5 BINARY CLASSIFICATION TREE

By applying the methodology provided in Section 4 at the target domain and the problem under study, we derive the proposed binary classification tree for rigid error detection and correction



(a) $\overline{R\_SPACE}$ (b) R_SPACE (c) $\overline{IMPL}$ (d) $IMPL$

Fig. 6. Schematic of constraint propagation principles.

techniques. We present the classification tree in a way such that the unidirectional propagation is always from the left branch to the right branch of each split. In this section, we use parts of existing approaches in order to provide existing examples that illustrate each class. The full description and categorization of these approaches is provided in Section 6, while the description and the categorization of the complete set of analyzed techniques is given in the Appendix 1.

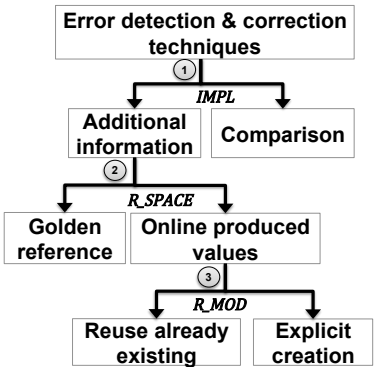


Fig. 7. High layers of the proposed binary tree classification.

Fig. 7 depicts the higher layers of the proposed classification. A split is applied to the design space, described by the attributes of the parent class. As the split is applied in the parent context, the attributes of the parent class are inherited to both subclasses. The split provides a further division of this design space, based on the single attribute that drives the split. The result is a subclass related to the part of the design space that includes the attribute, and a subclass referred to the remaining design space that does not include the attribute. The positive attribute and its complementary attribute of each split are presented in Table 1. We use the first split to illustrate in details the proposed top-down methodology. The initial parent class is the root of the binary tree and refers to the design space describing the conceptual aspects of any error detection and correction technique, as defined by the target domain under study. The root is further refined to the *additional information* (positive branch), i.e. the part of the design space dedicated to describe whatever is relevant with the generation of the additional information to be used in the error detection and correction technique. This attribute is related to the question "What is required to define an error detection and correction technique". The complementary branch of the attribute refers to the design space part that is left, if we exclude the part that is related to the generation of the additional information. So, the complementary branch has to be affirmatively reformulated to "what is still required to define an error detection and correction technique once the generation of the additional information is finalized and that information is available". The answer is given

Table 1. Characteristics of high layer splits

Split	Type	Question	Attribute	
			Affirmative branch	Complementary branch
1	What	What is required to define an error detection and correction technique	Additional information	Additional information → Comparison with the original information
2	How	How to obtain the additional information to be compared with the original value	Through a golden reference	golden reference → Through online produced values
3	How	How to obtain the additional information through online produced values	Reuse already existing values	Reuse existing values → Explicitly create the additional value

by the *comparison* branch, referring to the part of the design space related to what remains to be done in order to compare the original value and the generated additional information. During the exploration of the design space, we have to know the additional information that is generated by the technique in order to efficiently decide on how we can compare it with the original value (*IMPL*). The specific decisions over the lower layer branches of both the additional information branch and the comparison branch depend on the requirements of the technique to be designed. For instance, assume that the requirements are to design a technique that performs error correction for single errors with a success ratio of 100%. The decisions in the lower layers of the additional information branch have to be compatible with this design constraint. An example is to select as additional information two values that are identical with the original value (e.g. Triple Modular Redundancy - TMR). In a similar way, the decisions in the comparison branch must fulfill the need to always correct single errors. Due to the unidirectional propagation between the additional information branch and the comparison branch, the decisions taken in the additional information (left branch), e.g. value triplication, are also propagated as design constraints to the comparison branch. Due to this propagation, when the decisions are taken in the comparison branch, we already know that two additional values, identical to the original one are generated.

The additional information branch is refined by answering to the question "How to obtain the additional information to be compared with the original value". The result is to compare with a *golden reference* and to compare with a value that is *online produced*. The golden reference branch refers to the techniques that read an original value from the system and compare it with a reference, as schematically depicted in Fig. 8(a). The most common approach in this branch is to use monitors and compare the observed value with a predefined threshold. On the other hand, the techniques belonging to the online produced value branch compare the original value with a value produced during execution, as schematically depicted in Fig. 8(b). The most common approach in this branch is the creation of duplicated information used for error detection.

During the design space exploration, the golden reference is explored first, due to the space restriction principle (*R_SPACE*). To illustrate the reasoning, assume that the online produced value is explored first. Then, by propagating this decision to the golden reference, a part of the design space is unnecessarily removed. The way to produce the online value used for comparison has been already decided, so the use of a golden reference cannot further help. On the other hand, deciding first the golden reference can allow the combination with the online produced value, as the knowledge of the golden reference can be used to decide the online produced value.

By answering to the question "How to obtain the additional information through online produced values", the branch of the online produced value is refined into *reuse of already existing values* and the *explicit creation* of the additional information to perform error detection/correction. In the reuse category, the value to be compared with the original value already exists in the system, for other reasons than for error detection/correction. The error detection and correction technique has to

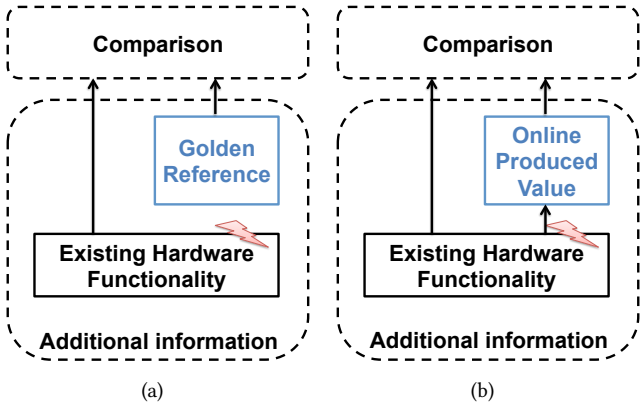


Fig. 8. Schematic of the difference between a) golden reference branch and b) online produced value branch.

slightly modify the existing system in order to be able to extract it and use it, e.g. using the redundant data already stored in the memory hierarchy, due to caches, to perform error correction [3]. On the other hand, in the explicit creation branch the value does not pro-exist by default in the system. It is explicitly created to perform error detection/correction. Examples of explicit creation are the insertion of a second function unit to compute a second value, such as reduced precision techniques [31], or the insertion of an additional register to store twice the output of a function unit, such as double sampling methods [27]. The hardware modifications of reusing the existing information are fewer than modifying the system to explicitly create the redundant value (*M_MOD*). The following sections further refine these higher layers of the proposed classification.

5.1 Golden reference

Table 2. Characteristics of golden reference splits

Split	Type	Question	Attribute	
			Affirmative branch	Complementary branch
4	What	What is required to obtain the golden reference	Type of information to be used as golden reference	How to generate this information
5	How	How an information can be used as golden reference	Use info relevant to transition (timing aspects)	Use info relevant to value (numerical aspects)
6	How	How the timing aspects can be used as golden reference	Use the moment that transition occurs	Use the duration that transition lasts
7	How	How the numerical aspects can be used as golden reference	Use the type of the value	Use the arithmetic value
8	How	How an arithmetic value can be described	Lossless	Lossy
9	How	How to obtain lossless information	Use value as it is (identical)	Encode the value
10	How	How to obtain lossy information	Approximate the value	Compress the value
11	What	What is required to generate the golden reference	Static computation	Further Processing

The golden reference subclasses are depicted in Fig. 9 and the characteristics of the splits are presented in Table 2. The annotation below each leaf correspond to existing error detection and correction techniques classified in Section 6. Through a first split, that replies to the question “What is required to obtain the golden reference”, the golden reference branch is divided into what type of *information* is the golden reference and the *generation* of the golden reference. The knowledge

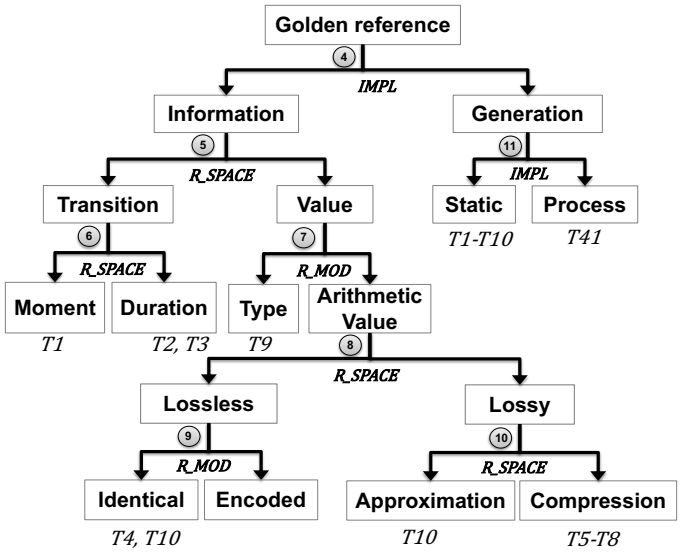


Fig. 9. Refinement of the golden reference branch annotated with the classified techniques.

about the type of information is required in order to decide on how to generate such information (*IMPL*). By answering to the question "How an information can be used as golden reference", the information branch is refined into using information relevant to timing aspects, i.e. *transitions* , and use information relevant to *values*. In the case of design space exploration, the monitoring of the transition is explored first. The combination of the branches in the other way around, i.e. monitoring first the value and, then, the transition, is not possible. The transition has to take place first in order to obtain the related value (*R_SPACE*). Exploring how the timing aspects can be used as golden reference, the transitions branch is further refined into the *moment* the transition takes place and how long it lasts, i.e. the *duration*. The moment is explored first, because the other way around, i.e. monitoring first the duration, and, then, the moment, is not possible. The first thing that occurs is the moment of the transition, and, then, the duration can have a meaning. A similar question on how the numerical aspects can be used as golden reference, the value branch is divided into the *type* of the value to be monitored, such as a circuit state characteristic (e.g. voltage supply rail), and the *arithmetic value*. The type is explored first, as it implies less modifications than using a specific arithmetic value.

Finally, the arithmetic value branch is further refined, based on the question "How an arithmetic value can be described", into *lossless* and *lossy* type. The lossless type describes the exact information, whereas in the lossy type some less essential information is lost and, thus only a part of the information is used. Although some information is lost, this leads to smaller values that can be used to potentially reduce the overhead of the fault tolerant approaches. The lossless way is explored first because the selections of the lossless can be combined with the selections of the lossy. The other way around is not possible. If the lossy has been selected first, we cannot undo this decision in the lossless class because the lost information cannot be retrieved. However, a lossless information can still become a lossy one (*R_SPACE*). A further refinement to the lossless information is if the information is *identical* or *encoded*. The identical information is explored first, as it does not insert additional modifications to the system, compared with the encoded class, that requires potentially

dedicated hardware to perform the encoding (*R_MOD*). A further refinement is performed by exploring the ways of implementing lossy information. The result is by *approximation* and by *compression*. In both cases, the obtained value is narrower than the original one. To achieve that, the approximation reduces the information by removing less important bits of the value. The compression leads to lossy information by combining the bits of the value in a more compact form. The approximation is explored first, as it adequately modifies the value, and, then, the compression can be applied on top of the approximation to further reduce the size. The other way around restricts the search space (*R_SPACE*), as no meaningful approximation can be applied over compressed values.

The *generation* branch is further split by answering to the question "What is required to generate the golden reference". The result is the part that is relevant to the *static computation* of the golden reference, before the execution of the application, and the potential *further processing* of the computed golden reference during execution. During the design space exploration, the static computation of the golden reference is explored first, since the processing requires to know this information in order to decide on how it could further process it (*R_MOD*). It should be stressed that this processing class refers only to the part of the design space that is responsible for any further processing applied only on the golden reference, and not any general processing applied by the error detection and correction approach. This branch can be further refined by applying similar splits to the implementation class of the explicit value creation, as described in Section 5.3. However, these splits are still different, as they depict only the part of the design space that describes how the additional value is online created.

To support the proposed classification, we classify a wide set of existing error detection and correction techniques, by decomposing them to primitive components and classifying the components using our classification. The complete decomposition of these techniques and the explication of the classification of a large subset of the studied techniques is presented in Section 6. In the following sections, we use the relevant part of some of these techniques to provide illustration examples for each of the presented classes.

T1: The transition detector with time-borrowing latch presented in [4] senses the input data transition, when the clock is logically high, and reports an error, when the input arrives late. Therefore, it has a primitive component that belongs to the transition/moment class.

T2: The approach of [17] uses fatal hardware traps to detect errors. More specifically the watchdog reset trap is thrown, when no instruction retires within a given number of ticks, and thus, it has a primitive component that belongs to the transition/duration class.

T9: The technique of [25] monitors the supply rail, and thus, the potential disturbance caused by a particle strike. Therefore, it has a primitive component that belongs to the value/type class.

T4: The detection approach of [37] monitors the target address of the branch instructions and compares it with the value estimated during compilation. Hence, it has a component that belongs to the identical class.

T10: ReStore framework [51] detects memory access and alignment exceptions by using the lower and upper bounds of the address space, which are determined before execution. Therefore, it has a component that belongs to the approximation class.

T5: The technique of [35] creates checksums during execution and compares them with a golden reference computed at design-time. Hence, it has a component that belongs to the compression class. In addition, it has a component that belongs to the computation/static, as the golden reference is computed before the execution.

T41: An Algorithmic Based Fault Tolerant (ABFT) approach [14] performs mathematical operations, during execution, on the initial encoded data (row and column checksums of the initial

matrix), thus it has a component that belongs to the generation/process, as the initial checksums are further processed during execution.

5.2 Reuse already existing values

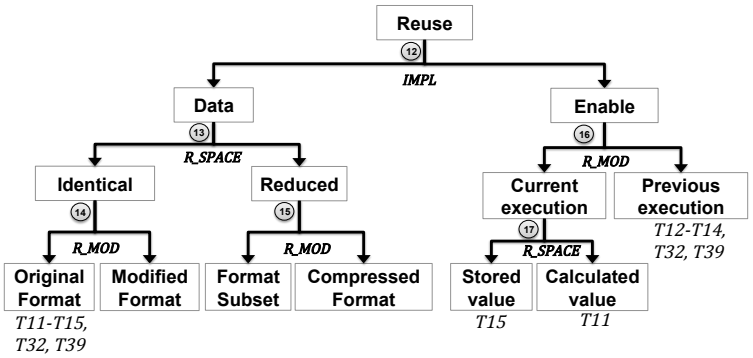


Fig. 10. Refinement of the reuse branch with the corresponding examples

The reuse subclasses are depicted in Fig. 10 and the characteristics of the splits are described in the Table 3. Similar to the previous section, the numbers below each leaf correspond to the techniques that have a primitive component that belongs to this leaf. The root is further refined through a split that replies to the question "What is required to generate the additional information by reusing already existing values". The result is the branch related to what type of *data* can be reused so as to serve as additional information and the *enabling* required to reuse this data. The knowledge about what type of the data is reused is needed in order to decide how to implement the reuse of this data (*IMPL*). The data branch refers to the design space part described by the complementary attribute of split 2 (use a online produced value) and the positive attribute of split 3 (a value that has been already created for other reasons). It is this inheritance of attributes from the ancestors that restricts the design space. The fact that we are reusing existing values for comparison prunes part of the options in the reuse/value branch. We are constrained by the fact that we can only use data values produced by the system execution, and not whatever value we may want, compared to the golden reference branch and the explicit creation branch. The options in reusing the existing data values are either to use the data value as it is created by the system (*identical*) or reuse only a part of it (*reduced*). The identical branch is explored first, because the design options under the identical branch can be combined with the design options under the reduced branch.

Table 3. Characteristics of reuse splits

Split	Type	Question	Attribute	
			Affirmative branch	Complementary branch
12	What	What is required to generate the additional information by reusing already existing values	The type of the data value that can be reused	From where the reuse can happen (Enable)
13	How	How a data value can be reused	As it is (identical)	Reuse only a part of it (Reduced)
14	How	How an identical value can be presented	Original format	Format modification
15	How	How a reduced value can be reused	Reuse a part of the format	Reuse a compressed format
16	How	How the reuse can be enabled	From current execution	From previous execution
17	How	How we reuse the values from the current execution	Stored values	Calculated values

The other way around is not possible. If the reduced design options have been selected first, we cannot undo this decision in the identical branch because the lost part cannot be retrieved. However, selecting first to reuse the identical data of the system, they can still be reduced due to a later decision (*R_SPACE*). A further refinement to the identical data is the reuse of the *original format* or a *modified format* provided already by the application execution. The original format is explored first since it does not require additional processing for the error detection and correction compared to the modified format (*R_MOD*). The reduced data is split into reusing a *subset* of the format and reusing a *compressed* format of the data. Similar to before, reusing only a subset of the format does not require modifications to the system for error detection and correction (*R_MOD*).

The enabling branch is further defined by answering to the question "How the reuse can be enabled". The result is to reuse the data from the *current execution* and from the *previous execution*. The current execution is explored first, as it describes a part of the design space with less modifications (*R_MOD*). By answering to the question "Which ways exist to reuse existing values from the current execution", the current execution is refined into reuse of *stored values* and reuse of *calculated values*. Exploring first the reuse of the calculated information and, then, the reuse of the stored information restricts the design space. The calculated info may affect the stored values, and, thus, removes a part of the design space (*R_SPACE*).

We provide one illustration example for most of the presented classes. In the best of our knowledge, we have not identified error detection and correction techniques that are reusing other data from the system than the original one.

T14: The extended history fault screener [34] verifies if the original value of an instruction is one of the 64 previously observed values. Hence, it has a component that belongs to data/identical/original format class and also a component that belong to enable/previous execution class.

T11: The technique [50] performs error detection by directly comparing the data at the input and at the output of a flip-flop. Therefore, one component belongs to the enable/calculated value class.

T15: Nostradamus [26], for errors in the execution stage, creates a signature by observing the registers at the decode stage and compares it with the real impact computed at the execute stage. Therefore, it has a component that belongs to the enable/stored value.

5.3 Explicit value creation

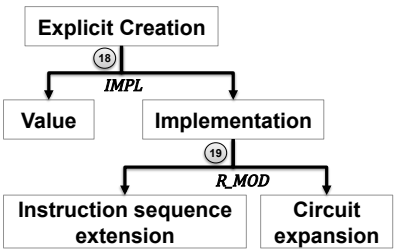


Fig. 11. Refinement of the explicit value creation branch

The higher layers of the explicit value creation branches are depicted in Fig. 11 and the corresponding splits in Table 4. By applying a split to the explicit value creation branch, it is further refined into two subclasses, i.e. the part that describes the type of the *value* to be created and the *implementation* on how to create this value. Although the split is similar to the reuse case, it is not the same, due to the inheritance of the parents attributes. The explicit creation/value refers to the part of the design space where the value, used for comparison, is created for fault tolerance, whereas

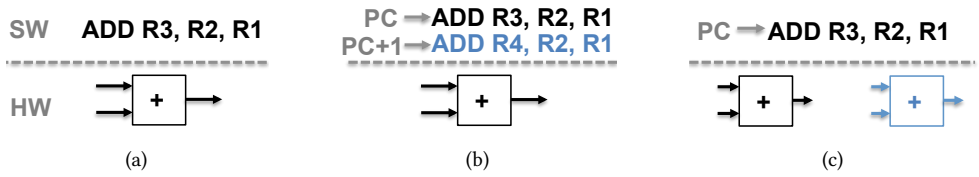


Fig. 12. Schematic representation of a) original instruction and circuit, b) instruction sequence extension and b) circuit expansion.

the reuse/value refers to the part of the design space, where already existing values are used for fault tolerance. What is the type of the value to be created is required to be known during the decision on how to implement the creation of this value (*IMPL*). The implementation branch is further refined into the creation of the value by inserting instructions and modifying the software-related part, i.e. the *instruction sequence extension* (which can include from one up to several instructions), and by inserting hardware components modifying the hardware-related part, i.e. the *circuit expansion*. The instruction sequence branch describes the case where the platform hardware components of the system are not modified in order to create the values, but the modifications are applied only to the software, which normally means changing the instruction sequence stored in the instruction memory. The right complementary branch describes the creation of the values by only modifying the hardware components of the system. The modifications in the instruction sequence are less than the modifications due to the extensions of the hardware (*R_MOD*). This difference can be schematically illustrated in Fig. 12, where Fig. 12(a) shows the original software, i.e. one instruction ADD, and the original hardware, i.e. one adder. In Fig. 12(b), a new instruction ADD is inserted in the program stored in the instruction memory to create the value to be used for the comparison, whereas the hardware is not modified. In Fig. 12(c) only one instruction exists, but it is used twice to control two different adders.

Table 4. Characteristics of higher layer of explicit value creation splits

Split	Type	Question	Attribute	
			Affirmative branch	Complementary branch
18	What	What is required to generate additional information by explicit online creation	Type of value to be created	Implementation of value creation
19	How	How we can create this value	Modify the instructions sequence	Modify the underlying hardware

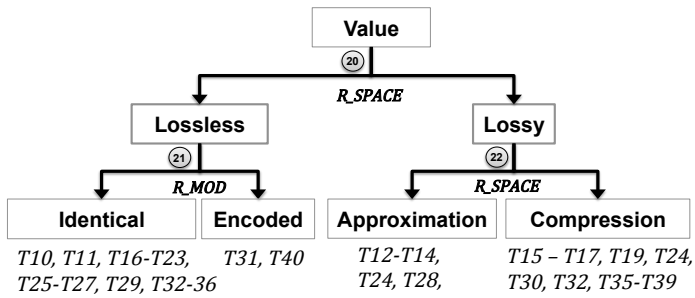


Fig. 13. Refinement of the explicit creation/value branch

5.3.1 Value. The value branches are depicted in Fig. 13 and the corresponding splits in Table 5. The splits are similar, but not the same, to the reuse branch. The differences derive from split 3. The value branch (split 18) inherits the complementary attribute of split 3, i.e. the value to be used for comparison is created during execution. On the contrary, the value branch of the reuse branch (split 12) inherits the positive attribute of split 3, i.e. reuse the values that have been already produced due to the other computations.

Table 5. Characteristics of explicit creation/value splits

Split	Type	Question	Attribute	
			Affirmative branch	Complementary branch
20	How	How a value can be described	Lossless	Lossy
21	How	How to create lossless information	Create identical value as original	Create encoded value compared to original
22	How	How to create lossy information	Create approximated value compared to original	Create compressed value compared to the original

T25: Double Modular Redundancy (DMR) in logic level [10] creates the same information with the original one and, thus, it has a primitive component that belongs to the identical class.

T31: The fault detection of shared directory entries in [15] reads the entry and stores it to a new register. The value is negated and stored back to the directory. Then, it is read again and compared with the initial value. Therefore, it has a primitive component that belongs to the encoded class.

T28: The Reduced Precision Redundancy (RPR) technique creates reduced precision values by truncation or rounding to be processed by arithmetic circuits [31]. Hence, the technique has a primitive component that belongs to the approximation class.

T16: The execution fingerprinting [19] compresses the architectural state of two processors into one fixed-size word using cyclic redundancy codes and compares them to detect potential changes. Hence, it has a primitive component that belongs to the compression class.

5.3.2 Instruction sequence extension. This branch refers to how the additional instructions are executed in order to create the values used for comparison. The splits are presented in Fig. 14 and

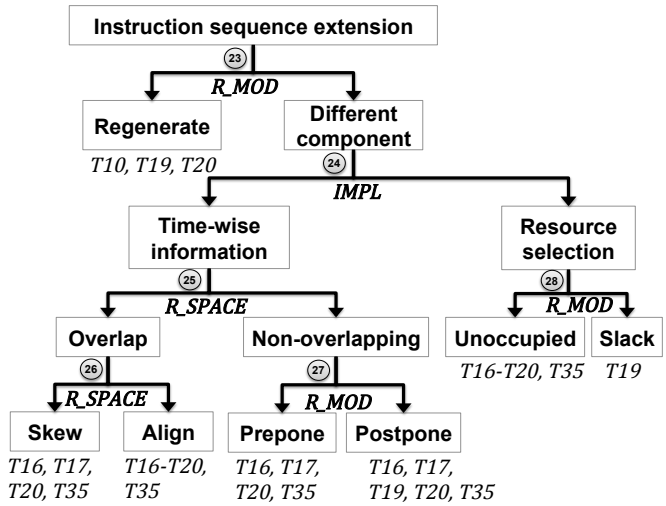


Fig. 14. Refinement of the instruction sequence extension branch

Table 6. The result of the first split is the case where the execution of the additional instructions occurs on the same component as the original ones, i.e. *regenerate*, and the case where the execution occurs on a *different existing component* of the system. As the attributes are inherited from the parents, the instruction sequence extension branch refers to the part of the design space, where the redundant values are created without modifying the hardware (positive attribute of split 19). Therefore, the execution on the same component has inherited this characteristic, and, thus, we know already that the additional instructions will be executed in the exact same way as the original instructions, since no hardware modification takes place. Therefore, this leaf can be better described by the term regenerate. The regenerate option is explored first as it implies less modifications on the original system (*R_MOD*). The different component branch is further refined through a split relevant to "what is required to define the execution of the additional instructions on another component". The result is the *time-wise information*, which describes the timing characteristics of the execution of the additional instructions on a different component with respect to the execution of the original instructions, and the *resource selection*, which describes how to select a component among the available ones. The time-wise information provides the timing aspects required in order to identify which components are available at the moment when the execution of the additional instructions takes place (*IMPL*).

Fig. 15 schematically depicts the differences between the subclasses of the time-wise information branch, where the black box is the original sequence and the blue box is the additional instruction sequence, which creates the redundant values. The A inside each box refers to one specific instruction. Based on the question "How the additional instructions can be executed in time with respect to the original execution", the time-wise information branch is further refined into *overlap* or *non-overlap* the execution of the additional and the original instructions. The propagation of the overlapping to the non-overlapping is motivated because the combination of these two options in the opposite way is not possible. If the non-overlapping option has been decided first, it cannot allow any overlapping of the individual instructions of the two sequences (*R_SPACE*), as depicted in Fig. 15(d) and Fig. 15(c). On the other hand, if the additional instruction sequence and the original sequence are overlapping, then some of the additional instructions can be executed in a non-overlapping way with the instructions of the original sequence, as depicted in Fig. 15(f) and Fig. 15(g). The overlapping branch is further refined into the branch where execution of the

Table 6. Characteristics of instruction sequence extension splits

Split	Type	Question	Attribute	
			Affirmative branch	Complementary branch
23	How	How the additional instructions (required to create the additional information) can be executed	On the same component as the original instructions (regenerate)	On a different existing component
24	What	What is required to define the execution of the additional instructions on another component	Define the time-wise execution of the additional instructions (relevant to the original execution)	Select on which component the additional instructions are executed
25	How	How the additional instructions can be executed (related to the original execution)	Additional execution overlaps with original one	Additional execution does not overlap with the original one
26	How	How the additional and the original instructions can be executed in an overlapped way	Additional execution is skewed compared to the original one	Additional and original instructions are executed in an aligned way
27	How	How the additional and the original instructions can be executed in an non-overlapped way	Additional execution is before the original one (Prepone)	Additional execution is after the original one (Postpone)
28	How	How to select a different component for the additional instruction execution	Select a component that is not currently used (unoccupied)	Select a component where slack can be created

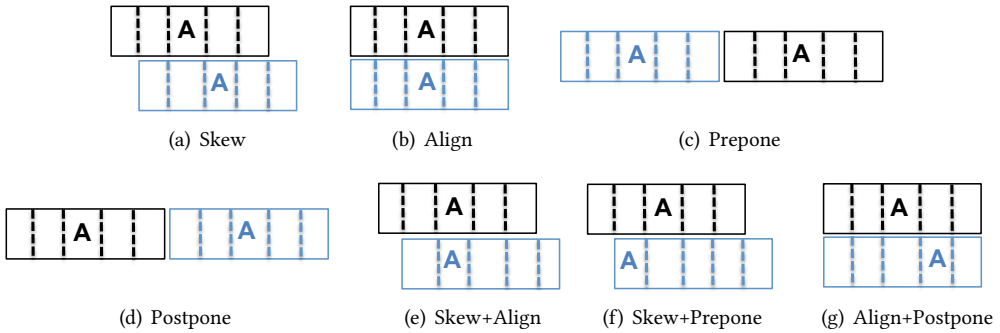


Fig. 15. Schematic of the difference of the classes of the time-wise information branch and their combinations.

additional instructions is *skewed* with respect to the original execution and the branch where both executions are *aligned*. The skew subclass is explored first as it restricts less the options in the aligned class. The propagation in the opposite way cannot lead to a valid combination, because if it is already decided that the sequence is aligned, no skewing can be performed. However, when skewing is decided first, some instructions can still be aligned inside the instruction sequence (Fig. 15(e)). The non-overlapping branch is divided into the case where the additional instructions are executed before the original ones (*prepone* class) and the case where the additional instruction are executed after the original ones (*postpone* class). The prepone is expected to have less impact in the scheduling of the instructions.

The branch of resource selection is further refined into the resources that are not used at the given time moment, i.e. *unoccupied* (which includes either idle resources after the scheduling or available resources during the scheduling), and the resources that can have a *time slack*. As the use of the unoccupied resources imply less modifications (*R_MOD*), they are explored first.

T10: The approach in [51] uses symptom detectors to trigger re-execution of the instructions for error detection in situations that are likely to occur in the presence of an error. Hence, they have a primitive component that belongs to the regenerate class.

T20: The approach of [41] executes with a delay a redundantly generated instruction stream after the start of the execution of the active stream on the same architecture. Hence, the technique has a component that belongs to the time-wise information/skew class.

T18: The technique of [12] has two processor units fully synchronized executing identical instruction streams. Therefore, the technique has a component that belongs to time-wise/align class and a component that belongs to resource selection/unoccupied class.

T16: The distributed temporal redundancy approach [19] allows the duplicated task to be executed on another core, without the fully synchronization requirement. Therefore, the technique has components that belong to the time-wise information/align class, time-wise information/skew class, time-wise information/prepone class and time-wise information/postpone class.

T19: The instruction duplication on VLIW datapaths can be performed by the compiler [9]. The compiler is aware of the configuration of the VLIW, in terms of number of issues and function units, and it can insert a new time slot, when no available function unit exists for the scheduling of the duplicated instructions. Therefore, it has a component that belongs to the resource selection/slack class.

5.3.3 Circuit expansion. The subclasses are depicted in Fig. 16 and the corresponding splits in Table 7. The branch is refined into the *time-wise information*, which defines when the additional

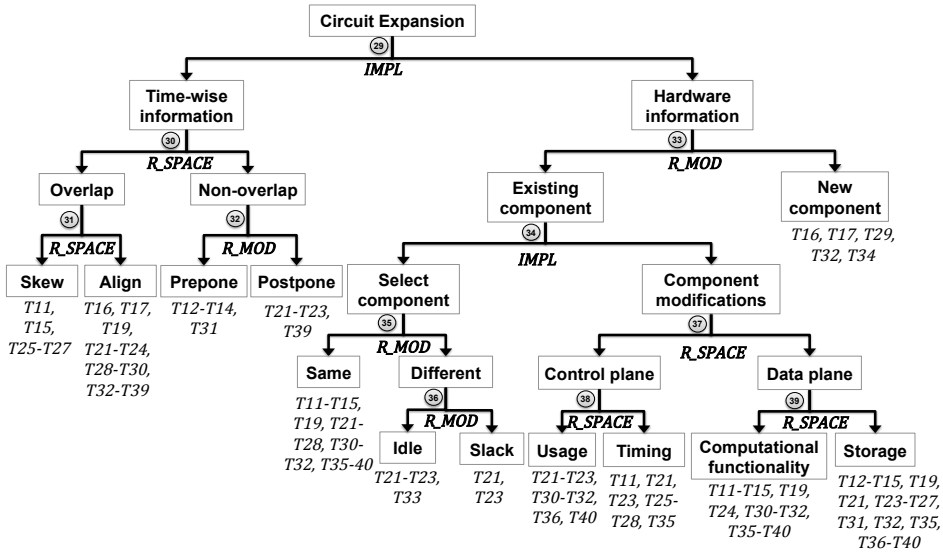


Fig. 16. Refinement of the circuit expansion branch

value is created with respect to the creation of the original value, and the *hardware information* required to describe the hardware expansion. The time-wise information is required in order to identify how the hardware expansion takes place (*IMPL*). The splits of the time-wise information are similar to the instruction sequence/time-wise information, but not the same, due to the inheritance of the complementary attribute of split 19, which implies modifications only in the hardware part. Therefore, the hardware is allowed to be modified for the creation of the additional values, which is not the case for the instruction sequence branch. The time-wise information is split into overlapping the creation of the additional values with the creation of the original ones (*overlap* branch) and creating the additional values without overlapping with the execution of the original instructions (*non-overlap* branch). The overlapping is further refined into *skew* or *align* and the non-overlapping branch into *prepone* class and the *postpone* class.

The hardware information is further refined into expanding the circuit by extending *existing components* and by inserting a *new component*, dedicated to the creation of the additional values. The expansion of an existing component has less hardware modifications than adding a completely new component to the platform (*R_MOD*). Further refinement of the existing component branch is the *selection* among the existing components and the potential *component modifications*. The selection of which component to extend is required to decide which modifications can be applied on this component (*IMPL*). The selected component can be the *same component*, as the one used for the original value creation, or another *different component*. The expansion of the same component implies less modifications to the original system, especially due to the required communications (*R_MOD*). The different components can be an *idle* component (including either a component of the platform that is currently not used by the application or spare resources already added to the platform for potential future workload increase), and a component with time *slack*. The use of a spare component implies less hardware modifications to the original system (*R_MOD*). The possible ways to extend the hardware of the selected component are by modifying the *control plane* and by modifying the *data plane*. The control plane decides how the data plane is used, and, thus, it is explored first. Modifying first the data plane may remove some of the promising

Table 7. Characteristics of circuit expansion splits

Split	Type	Question	Attribute	
			Affirmative branch	Complementary branch
29	What	What is required to define the extension of the circuit	Time-wise information of the creation of the additional values with respect to the original value	Hardware information with respect to the extension of the circuit
30	How	How the creation of additional values can occur in time (with respect to the creation of the original value)	Additional value creation overlaps with original one	Additional value creation does not overlap with the original one
31	How	How the additional values and original values can be created in an overlapping way	Additional value creation is skewed compared to the original one	Additional and original values are created in an aligned way
32	How	How the additional values and original values can be created in a non-overlapping way	Additional value is created before to the original one	Additional value is created after the original one
33	How	How the hardware can be expanded	Extend components already existing in the system	Explicitly add a component only for the creation of the additional values
34	What	What information is required to define the component expansion	Selection among the existing components	Potential modifications in the selected component hardware characteristics
35	How	How to select, among the existing components, which component to extend	Extend the same component where the original values are created	Extend a different component
36	How	How to select among the available different components	Components that are not currently used (Idle)	Components where slack can be created
37	How	How the selected component's hardware can be extended	By modifying the control plane	By modifying the data plane
38	How	How the control plane can be extended	Modify the usage of the data plane	Modify the timing aspects of the data plane
39	How	How the data plane can be extended	Extending functionality	Extending storage

options in the design space of the control plane (*R_SPACE*). The control plane can change the action that the data plane performs, i.e., *usage* class, and when the data plane performs this action, i.e., *timing* class. The way of using the existing functionalities of the data plane is required in order to define the timing aspect on when to perform this action (*R_SPACE*). For instance, using a part of the component that has longer delay prunes the use of specific frequencies. The data plane can be modified with respect to the *computational functionality* and the *storage* characteristics. The computational functionality includes the computation units and the relative interconnection of the computation units and the storage. The modification of the functionality is explored first because the other way around may unnecessary restrict the design space. If the storage characteristics are decided first, then the decision on the storage characteristics may prohibit some promising functionalities to be implemented, for instance due to the lack of storage means (*R_SPACE*). The schematic representation of the aforementioned modifications is depicted in Fig. 17.

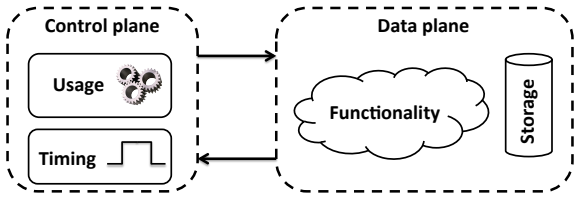


Fig. 17. Schematic of the difference possibilities to extend the component hardware.

T27: The Razor flip-flop [4] double-samples the input data by adding a shadow latch to the original data path that samples the input data at the falling clock edge. Therefore, it has a component that belongs to time-wise information/skew class (falling clock edge), a component that belongs to

select component/same class (extension of original data path), component modifications/timing class (falling clock edge) and component modifications/storage class (shadow latch).

T34: The triple modular redundancy, implemented by additional hardware, executes the same circuit three times in parallel [20]. Therefore, it has a component that belongs to time-wise information/align class and to the hardware information/new component class.

T12: The invariance-based fault screener [34] calculates and keeps a bitmask with the continually invariant bits across the instruction executions. Therefore, it has a component that belongs to the time-wise information/preprone class, as each time a value is produced, the bitmask is updated in order to be used for the next value, a component that belongs to the component modifications/functionality class and component modifications/storage class, as the original hardware is extended to include the hardware to compute the invariant bitmask and to store it.

T23: The hardware mechanism of [45] couples the pipelines of the VLIW processor. The second pipeline executes an original instruction, if it exists, otherwise it executes the duplicated instruction of the first pipeline. Hence, this technique has a component that belongs to select component/idle and component modifications/usage, as the use of the second pipeline is modified to execute the duplicated instructions. In case there are not enough idle slots, an additional time slot is inserted to create slack and accommodate the remaining replicated instructions. Therefore, it has components from time-wise information/postpone class and select component/slack class.

5.4 Comparison

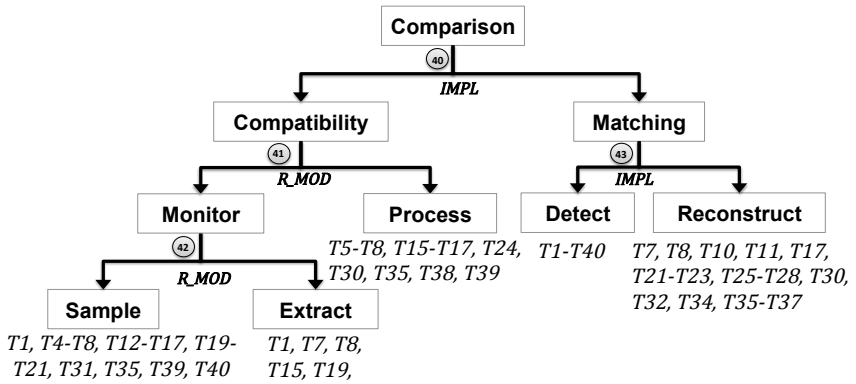


Fig. 18. Refinement of the comparison branch

The classes are depicted in Fig. 18 and the corresponding splits in Table 8. The comparison branch is refined based on what is required to define the comparison between the generated additional information and the original value. The left branch refers to the design space part that is relevant to making the original value *compatible* with the generated additional information. The right branch refers to the design space relevant to the *matching* between the generated additional information and the compatible original information. The compatibility is explored first, as first the additional information and the original information have to be made compatible before the real comparison (*IMPL*). The *compatibility* branch is further split by answering to the question "what is required to make the additional information and the original information compatible". The result are the branches of *monitoring* the compatible information and applying further *processing* to make the original information compatible. The monitoring refers to the case of obtaining the compatible information by performing simple modifications at the system, e.g. using probes to monitor the

voltage, and it incorporates the notion of sampling the compatible info. The processing class refers to the additional processing of the original information of the system. This processing class incorporates only the design space describing any further processing required to be applied in the original information, in order to be able to compare it with the additional information, and not any general processing applied by the error detection and correction approach. This class can be further refined through splits similar to the splits of the implementation class of the explicit value creation.

One such example is the case where a checksum is created for each basic block of instructions. The processing calculates the checksum, whereas the monitoring part samples the checksum by storing it into a register, whose value will be compared with a golden reference. The monitoring part is explored first, because the modifications of the monitoring part are fewer than the system modifications required for the processing (*R_MOD*). The monitoring branch is further divided into the *sampling* of the compatible information and the potential *extraction*, that must take place for the original value in order to enable the sampling. The sampling class refers to the use of the clock signal in order to sample the value either by storing it or by propagating it based on the clock. The extracting class refers to the potential combinational logic added to extract the original value from the system hardware. The class with the least modifications is the monitoring (*R_MOD*). The matching branch is further refined based on the question "What can be obtained by matching compatible additional information and original information". The result is to perform matching only to *detect* inconsistencies and to restore them to the correct value, i.e. *recontruct* class. First, the detection has to take place and this information is required to perform the replacement of the faulty value (*IMPL*).

T1: The transition detector with time-borrowing latch of [4] passes the signal through a small logic to detect if the signal changes (extract class, detect class) and it verifies if the change takes place when the clock is high (sample class).

T10: The symptom-based methods [51] create exceptions when an attempt is made to access an inappropriate address (detect class), whereas re-execution occurs for correction (reconstruct class).

T13: The dynamic range fault screener [34] verifies if the original value of an instruction belongs into a given range (detect class). As no further processing takes place to the original value, the component of this technique for the implementation branch belongs to the sampling leaf.

T16: The execution fingerprinting [19] requires further processing of the monitored original value, and thus, they have also a primitive component that belongs to the process class.

T7: The scrubbing techniques using the Cyclic Redundancy Code (CRC) [42, 43] require also a component that belongs to the process class for the re-calculation of the syndrome value.

Table 8. Characteristics of comparison splits

Split	Type	Question	Attribute	
			Affirmative branch	Complementary branch
40	What	What is required to define the comparison between the generated additional information and the original information	Make them compatible	Matching
41	What	What is required to make the additional and original information compatible	Monitor the original value	Apply further processing
42	What	What is required to implement a monitor of the original value	Hardware that samples the value	Hardware that extracts the value to enable monitoring
43	What	What can be obtained by matching the additional and original information	Detection of inconsistencies	Reconstruction of correct values

6 DECOMPOSITION OF EXISTING APPROACHES

The error detection and correction techniques are usually hybrid schemes, that combine more than one primitive class of the proposed classification. This occurs because designers aim at satisfying a variety of specifications, that usually cannot be met in an efficient way by following only one design direction. A technique always consists of several classes of the additional information branch and classes from the comparison branch. In this section, we provide the complete decomposition of 43 different techniques, by showing in which main branches they belong to. The techniques annotated with a "*" are hybrids of the main branches of the proposed classification. The decomposition of all the techniques is summarized in Fig. 20, where the lines correspond to the techniques and the columns correspond to classes of the proposed classification. The primitive components of each technique are marked with gray color in the corresponding column. Due to page limitations, in the following sections we describe a set of representative examples from the studied approaches, whereas the full decomposition of all studied techniques is provided as supplementary material in Annex 1. The complete classification is presented in Fig. 19. The main branches are shown in different colors and named using capital letters, whereas the classes are named using numbers.

6.1 Golden reference

T1: The Transition Detector with Time-Borrowing latch (TDTB) presented in [4] senses input data transitions, when the clock is logically high. As the input data transitions, a pulse is generated at the output of a XOR gate. If the input data arrives late during the logically high clock period, the pulse discharges the output node voltage of a dynamic gate, which reports the error. During the logically low clock cycle, the output node is precharged. The technique belongs to the golden reference branch. It is categorized as: 1) transition/moment, as TDTB senses transitions, 2) computation/static, as the golden reference is the upfront clock period, 3) compatibility/extract, for the part inserted to identify the transition, 4) compatibility/sample, as the dynamic gate verifies if the change takes place when the clock is high, and 5) matching/detect, as the technique performs only detection.
T2: One of the approaches presented in [17] uses fatal hardware traps to detect errors. More specifically, the approach uses a watchdog, where a reset trap is thrown when no instruction retires within an upfront given number of ticks. The technique belongs to the golden reference branch. It is categorized as: 1) transition/duration, as it is based on the number of ticks, 2) computation/static, as the threshold is the upfront given, and 3) matching/detect, as only detection occurs.
T4: The Control Flow Checking by Execution Tracing [37] monitors the target address of branch instructions and compares it with the address estimated during compilation. The technique belongs to the golden reference branch. It is categorized as: 1) value/identical and 2) computation/static, as an address is estimated before execution, 3) compatibility/sample, due to execution tracing, and 4) matching/detect, as only detection occurs.
T5: The Integrated Monitoring for Processor REliability and Security (IMPRES) [35] calculates a checksum for each basic block at compile time. It re-calculates the checksums at runtime to be compared with the value computed at compile time. The technique belongs to the golden reference branch. It is categorized as: 1) value/compression, as checksums are used as golden reference, 2) computation/static, as the checksums are upfront calculated, 3) compatibility/process and 4) compatibility/sample, as the checksums are run-time re-recomputed and stored, and 5) matching/detect class.
T9: The detection technique of [25] is based on monitoring the supply rail and, thus, the potential disturbance caused by a particle strike. The technique belongs to the golden reference branch. It is categorized as: 1) information/value/type, as the disturbance is monitored, 2) computation/static, as the decision of to detect an error is made if the disturbance exist or not, and 3) matching/detect.

T41: The Algorithmic Based Fault Tolerant (ABFT) approach [14] extends the input matrix with supplementary columns and rows containing checksums. Then, the matrix multiplication algorithm applies similar mathematical operations to both the original data and the checksum during execution to keep the checksum relationship invariant. Then, the new checksum is computed based on the original data, enabling error detection and error correction. Hence, this approach is categorized as: 1) golden reference/information/arithmetic value/lossy/compression class, as checksums are computed, 2) golden reference/generation/static class, as initial checksums are computed before execution, 3) golden reference/generation/process class, as the ABFT applies mathematical operations during execution that process the initial checksums, 4) comparison/compatibility/sample class and 5) comparison/compatibility/process class, as a new checksum has to be computed based on the original data, 6) comparison/detect and comparison/reconstruct, as the checksums allow to detect and also correct some errors.

T42: Capability checking approaches aim at the detection of malfunctions that cause illegal access to the memory system [24]. The capability of an object includes address, type and access rights. This information is used to pre-calculate offline a table with the access rights of each object towards another object. During execution, a low-cost processor translates the physical addresses of a specific memory reference and the physical address of the object it wants to access. Then, it verifies if the access rights from the main processor are compatible with the access-rights provided by the table. If they are not compatible, an error is signaled. Therefore, this approach belongs to: i) golden reference/arithmetic value/identical class and ii) golden reference/generation static class, as the exact access rights are computed offline for each pair of objects, iii) comparison/compatibility/sample class and iv) comparison/compatibility/process class, as memory accesses have to be translated in order to obtain the corresponding access rights from the table, and v) comparison/matching detect class, as an error is signaled when the access rights from the main processor and the golden reference do not match.

T43:* Consistency check methods use the knowledge of a transformation, which relates the inputs to the outputs of an algorithm, to perform error detection and correction. Consider an algorithm that computes the inverse matrix A^{-1} of the input matrix A and it uses the expression $A \times A^{-1} = I$, where I is the identity matrix [30], as consistency check. Therefore, this approach belongs to: i) golden reference/arithmetic value/identical class and ii) golden reference/generation static class, as the matrix I is used as a golden reference, with which we will compare the output of the algorithm and it is offline defined, iii) reuse/data/original format and reuse/enable/previous execution class, as the input matrix A is reused, iv) comparison/compatibility/sample class and v) comparison/compatibility/process class, as after the computation of the output matrix A^{-1}, it has to be processed through $A \times A^{-1}$ in order to be compared with the golden reference, and vi) comparison/matching detect class, as errors are detected.

6.2 Reuse existing values

T12:* A fault screener reports an error if the program's current behavior is inconsistent with the expected behavior given by a valid value space. The way to compute the valid value space determines the components of the technique. The invariance-based fault screener [34] calculates and keeps a bitmask for each static instruction representing which bits are continually invariant across the instruction executions. The technique belongs to the reuse (reuses the history of the instructions) and explicit creation (computation of invariant mask) branches. The technique is categorizes as: 1) data/identical/original format, since the complete value of the static instruction is re-used for the creation of the invariant, 2) enable/previous execution, since the values previously observed are re-used for the creation of the mask, 3) value/approximate, as the online computed value is an invariant mask, 4) time-wise information/preprone, as each time a value is produced, the

bitmask is updated in order to be used for the next value, 5) select component/same, 6) component modifications/functionality and 7) component modifications/storage, as the original hardware is extended with the hardware to compute and store the invariant, 8) compatibility/sample, for the original value, and 9) matching/detect, as only detection takes place.

T15:* Nostradamus [26] compares the instruction's expected impact on the architectural state with the actual impact that has the instruction execution. An expectation unit operates in parallel with the normal instruction decode logic to determine the instruction expectation, i.e. how the instruction will modify the architectural state compressed in a signature. The modification of the architectural state is expressed as the instruction expectation signature, which is a hashed value. During the execution stage, the real impact of the instruction execution is computed and compared with the expected one in order to detect an error. Therefore, the technique belongs to the reuse (reuse the information from the decode stage) and explicit creation branch (compute the signature mask). The technique categorized as: 1) data/value/original format, the value reused is the original value of the registers, 2) enable/current execution/stored value, as the same instruction is executed only in different pipeline stages and the value reused is stored in the registers, 3) value/compression, as a signature is created, 4) time-wise info/skew, as the signature is created at the decode stage during the execution of the original instruction, 5) select component/same, as the original pipeline is modified, 6) component modifications/functionality and 7) component modifications/storage, for the additional hardware for the calculation of the signature and the additional register to store it, 8) compatibility/extract, to observe the history of the instruction execution, 9) compatibility/process, to compute the real signature and 10) compatibility/sample, to store it, and 11) matching/detect, as the technique performs error detection.

6.3 Instruction sequence extension

T16:* The Execution Fingerprinting (EX) State Checkpointing [19] allows tasks to be executed at different times on redundant cores and compresses the changes into an external state, called fingerprint. EX is enabled periodically and pushes all register state to the fingerprinting unit. The technique belongs to the instruction sequence extension branch (redundant task execution) and to the circuit expansion (fingerprint computation). The technique categorized as: 1) value/identical, as the redundant task is the same as the original task, 2) time-wise information/skew, 3) time-wise information/align, 4) time-wise information/prepone and 5) time-wise information/postpone, as the redundant task is executed without time restrictions, 6) resource selection/unoccupied, as the second task is executed on a different core, 7) value/compression, as the register states are hashed to create the fingerprint, 8) time-wise information/align, as the fingerprint is created when the register state is pushed to the fingerprint unit, 9) new component, as a new core has been inserted for the execution of the redundant task,with the capabilities of creating the fingerprint, 10) compatibility/process, in order to create the original fingerprint, 11) compatibility/sample, to store the original fingerprint, 12) matching/detect, as the technique performs error detection.

T19:* The instructions are duplicated on VLIW datapaths by the compiler in [9]. The scheduling of the duplicated instructions is done along at a different function unit of the VLIW or after the original instructions. When this is not possible, a new time slot is added to create slack. The original values are stored to a register value queue and the addresses to the load/store queue. To find with which original value from the queue a duplicated value has to be compared to, the technique uses the output register address. The technique belongs to the instruction sequence extension as the duplication is performed in the instruction memory by the compiler. The technique is categorized as: 1) value/identical, as the same instructions are duplicated, and, thus, the same values are created, 2) time-wise information/align, and 3) resource selection/unoccupied, when the execution is done at the same time with the original instructions, 4) regenerate and 5) time-wise information/postpone,

when it the duplicated instructions are executed after the original ones, 6) resource selection/slack, when a new time slot is added, 7) compatibility/extract, to find out with which instruction to compare to, 8) compatibility/sample, due to the storing of the original values to the queue, and 9) matching/detect. In addition, the store/load queue is protected by a parity bit. This part of the approach belongs to: 10) value/compression, due to the parity, 11) time-wise align, as the parity is computed at the same time as the original value, 12) select component/same and 13) component modifications/storage, as the component where the original value is store is extended to keep the parity, and 14) component modifications/functionality, to compute the parity.

T20: The approach of [41] executes with a delay a redundantly generated instruction stream (R-steam) after the start of the execution of the active stream (A-stream) on a simultaneous multi-threaded architecture. Then, the execution of the individual instructions is decided based on a dynamic scheduler. The technique belongs to the instruction sequence extension. The technique is categorized as: 1) value/identical, as the streams are the same, 2) instruction sequence/time-wise information/skew, for the execution of the R-stream, 3) time-wise information/align, 4) time-wise information/prepone, 5) time-wise information/postpone, for the execution of the individual instructions of the R-stream, 6) resource selection/unoccupied FU, when the execution is performed on different FU than the ones used by the corresponding instructions of the A-stream, 7) regenerate, when the R-stream instructions are executed just after the corresponding A-stream instructions, 8) matching/sample, as the A-stream results are pushed onto the Delay Buffer, and 9) matching/detect.

6.4 Circuit expansion

T21: The technique of [32] uses a hardware mechanism that duplicates or triplicates the instructions inside the execution and decoding stages in order to take advantage of the idle slots in the current and next instruction bundle. The replicated instructions can be executed in the same time slot in another idle FU or in the next time slot in an idle FU (either same FU as the original or another FU). When there are not enough idle resources to exploit, the pipeline is stalled to add one or more additional time slots to create time slack. The technique belongs to the explicit creation branch. It is categorized as: 1) value/identical, as the same instructions are executed and, thus, the same values, 2) time-wise information/align and 3) select component/idle, when the replicated instructions are executed in the same slot as the original, 4) time-wise information/postpone and 5) select component/same, when the replicated instructions are executed in the next slot, 6) select component/slack, due to the inserted new time slot, 7) component modifications/usage, as the usage of the FUs is modified due to the run-time scheduling of instructions, 8) component modifications/timing, as stalls are inserted, 9) component modifications/data plane/storage, as additional registers are inserted to keep the values due to misaligned execution of the original and replicated instructions, 10) compatibility/sample, as the original result may be stored when needed due to the misaligned execution, 11) matching/detect, when duplication is used, 12) matching/reconstruct, when triplication is used.

T22: The phase-configurable VLIW processor [44] uses idle issue slots during a whole given program phase to execute duplicated instructions from the corresponding coupled pipeline and check their results. When a mismatch is found, the pipeline is flushed and the last instruction bundle is executed again. The technique belongs to the explicit creation branch. It is categorized as: 1) value/identical, as the coupled pipeline executes the same instructions as the original one, 2) circuit expansion/time-wise information/align and 3) select component/idle, as both duplicated and original values are executed at the same slot for error detection, 4) time-wise information/postpone and 5) select component/same, due to the re-execution of the original instruction for the error correction, 6) component modifications/usage, as the pipelines now executes original or duplicated instructions, 7) matching/detect, and 8) matching/reconstruct.

T24: Nostradamus [26] to deal with errors in functional units uses residue coding implemented in hardware. The technique belongs to the explicit creation branch. It is categorized as: 1) value/approximation, as the additional value is obtained by additional residue code operands, 2) time-wise information/align, as original and additional values are created at the same time, 3) select component/same and 4) component modifications/functionality, due to the additional residue code operand in the original data-path, 5) compatibility/process, to apply modulo operation to the original value, and 6) matching/detect.

T26: Double sampling methods [27] create the information in a moment that is skewed with respect to the original value. Therefore, the technique belongs to the circuit expansion branch. It is categorized as: 1) value/identical, as the same signal is latched by the shadow flip-flop, 2) time-wise information/skew, due to the delayed clock signal, 3) select component/same, since the extra shadow latch is introduced in parallel with the original flip-flop, 4) component modifications/timing, as the shadow latch is triggered by a delayed clock, 5) component modifications/storage, due to the use of the shadow latch, 6) matching/detect and 7) matching/reconstruct.

T28: The Reduced Precision Redundancy (RPR) creates additional values with reduced precision computed by two additional arithmetic circuits [31]. The technique belongs to the explicit creation branch. It is categorized as: 1) value/approximation, since the reduced precision values can be produced either by truncation or rounding, 2) time-wise information/align, as the additional values are created at the same time with the original one, 3) select component/same, as the original component is extended to include the RPR hardware, 4) component modifications/functionality, as the additional arithmetic circuits are inserted, 5) matching/detect, at the approach performs detection and 6) matching/reconstruct, for correction.

T31: The fault technique for shared directory entries [15] is as follows: the entry is read once and stored to a new register, then it is negated and stored back to the directory. Then, it is read again and stored to another register. The two registers are compared. The technique belongs to the explicit creation branch. It is categorized as: 1) value/encoded, as the information is negated, 2) time-wise information/prepone, as the original value is read after the creation of the additional information, 3) select component/same, as the original directory is modified to create the additional value, 4) component modifications/functionality, as the negation circuit is inserted, 5) component modifications/storage, for the new register to store the negated value, 6) component modifications/usage, as the way of using the data plane is modified to support the negation and storing back, 7) compatibility/sample, as the original value is stored to another register, and 8) matching/detect.

T40: The REcomputed with Shifted Operands (RESO) [29] shift the inputs of the ALU, reperforms the same instruction and the output is shifted back and compared with the original value. The technique belongs to the circuit expansion. It is categorized as: 1) value/encoded, since the operands and the result are shifted, 2) circuit expansion/time-wise information/postpone, as the execution of the second instruction is performed after the original one, 3) select component/same, as the original ALU is modified to support the RESO operation, 4) component modifications/usage, to modify the use of the ALU during the additional information creation, 5) component modifications/functionality, and 6) component modifications/storage, as the original ALU data-path is extended with shifters and one register, 7) compatibility/sample, for storing the original value and 8) matching/detect.

6.5 Observations

The proposed classification, combined with the categorization of the above techniques, provide several insights for the existing fault tolerance techniques The most crowded classes are the ones that have been more explored by the literature, whereas less crowded classes indicate design options that have been less used in the current state-of-the-art. By observing Fig. 20:

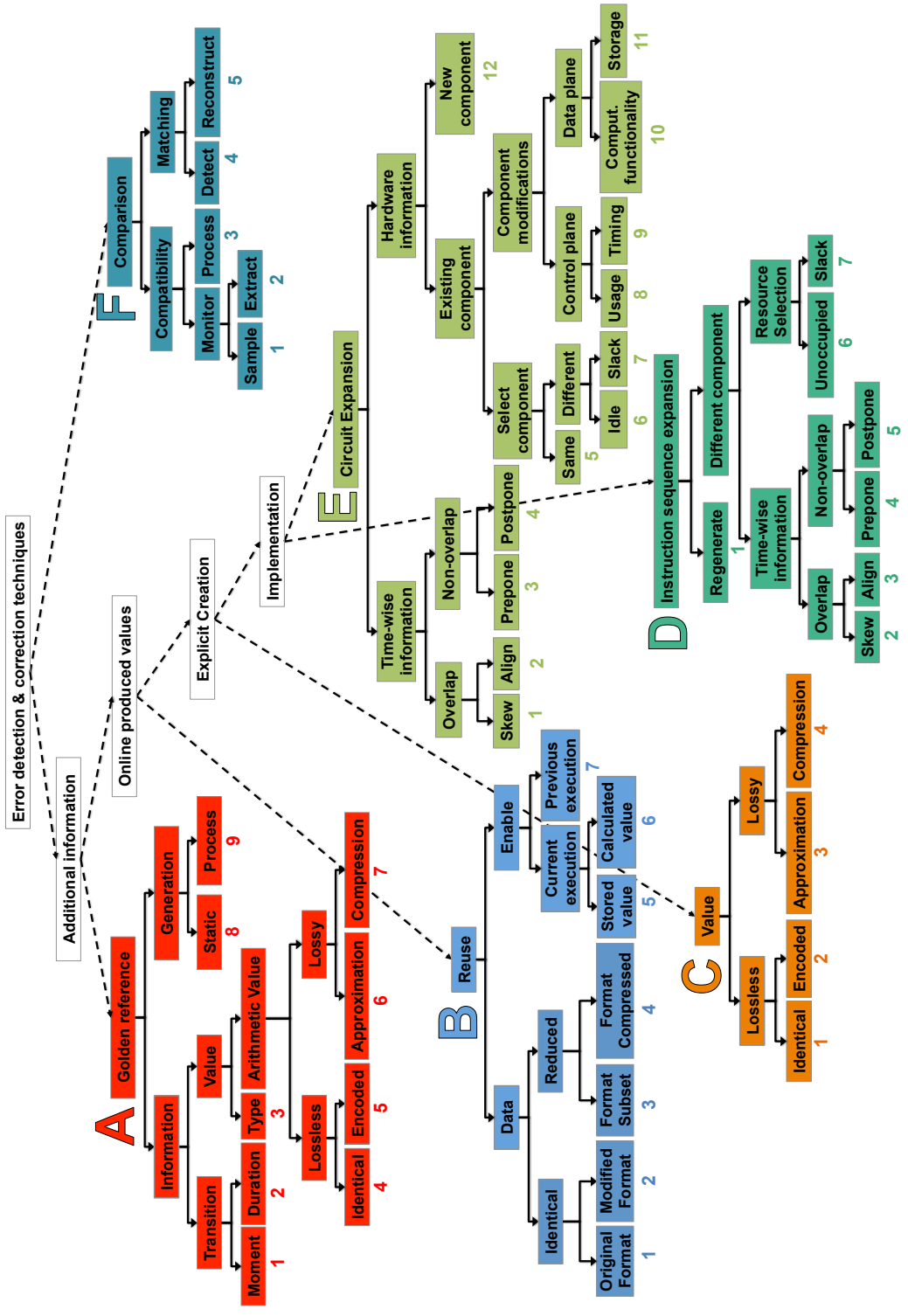


Fig. 19. Complete classification. A to F: main branches, numbers: classes inside each main branch.

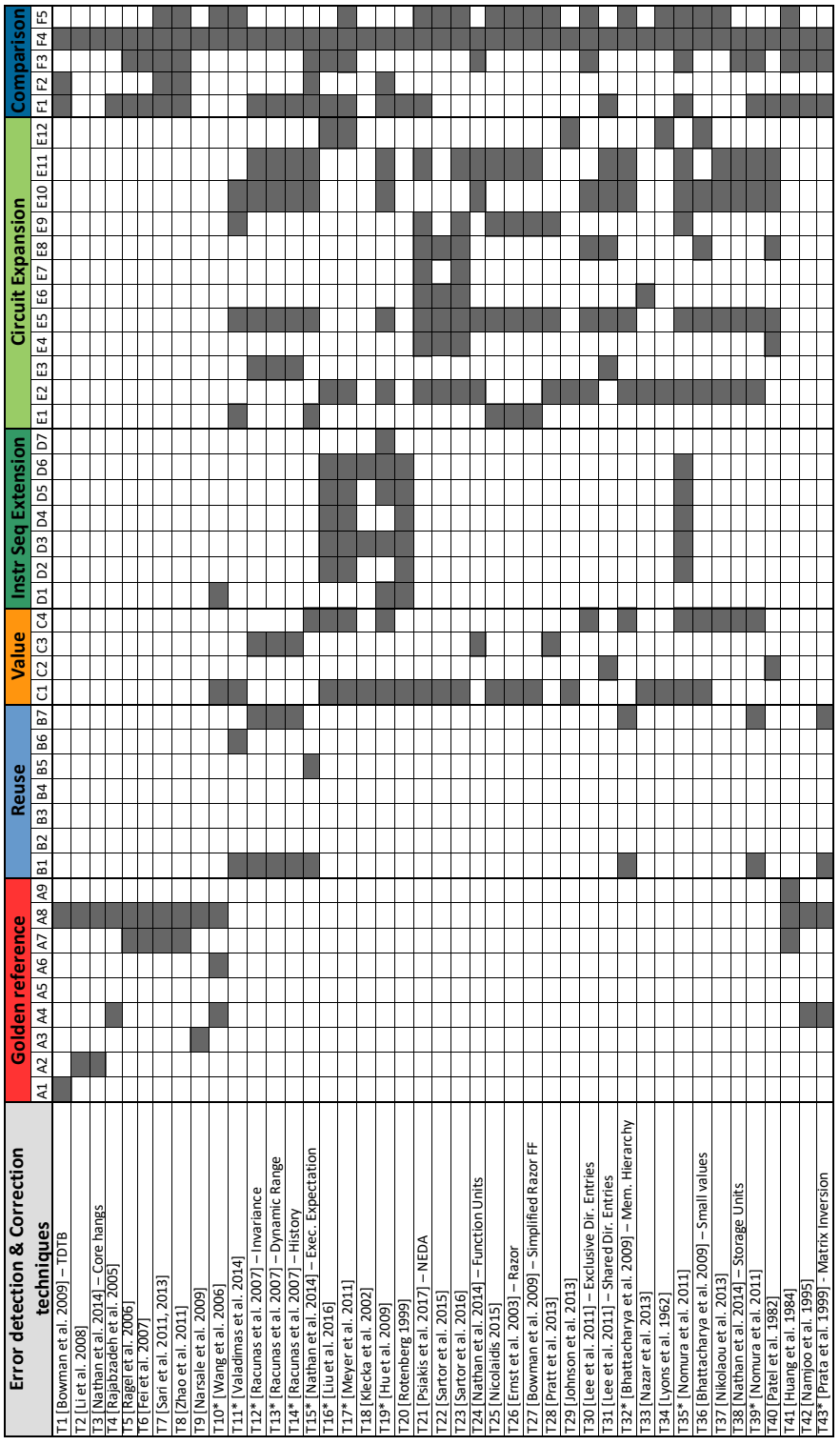


Fig. 20. Complete technique decomposition. * : hybrids of the main classification branches.

- A high majority of the classified techniques (69.77%) produce online (during execution) the additional information to be used for comparison.
- All studied reuse techniques are hybrids with other classes. Reuse techniques are usually combined with classes from the circuit expansion branch. In this way, the reused value is further processed through hardware in order to provide the value to be used for comparison.
- Many golden reference techniques (46.15%) use a lossy value, so as to reduce the overhead.
- A high majority of the classified techniques (60.44%) of the explicit creation branch use an additional information that is the same as the original one.
- Most of the instruction sequence extension techniques are exploring the use of components different than the original one (83.3%).
- Several circuit expansion techniques create the redundant information at the same time (62.07%) or at the same component (79.31%) or both at the same time and at the same component (44.83%) used by the original value.
- The second most popular option in the explicit creation branch is to create a lossy value for comparison, so as to reduce the overhead (46.67%).
- Several approaches require to process the original value, before performing the comparison (34.88%).
- Usually, the extraction of the original data is straightforward, as most of the approaches do not require any specific extraction component (81.40%).
- Few techniques use encoded information (4.65%) and they belong to the circuit expansion branch.
- Few techniques perform online processing of the golden reference (2.33%).

The proposed classification provides to the readers insight on the existing techniques and supports the design of new error detection and correction approaches. Due to its single attribute splits, the differences and the similarities of the techniques can be pointed out compared with traditional classifications that are driven by multi-way splits. We provide a few illustrative examples based on the presented techniques to illustrate how the proposed classification is capable of providing such a comparison. Fig. 20 presents the similarities and differences among all studied techniques.

The first example is between the techniques T21 and T22. Both approaches duplicate the instructions at run-time through a hardware mechanism and execute the duplicated instructions on the idle FUs of a VLIW processor. In traditional classifications both techniques belong to the same class. For instance, in the classification presented in [18], both approaches are classified under spatial redundancy, as the instructions are executed in parallel in idle FUs. Though, as an additional time slot is inserted in T21, that could be also considered as temporal redundancy. For the classification of [16], T21 and T22 belong to the execution redundancy class and for the classification of [22], they belong to the RTL level. Using the proposed binary tree classification, we can point out the relevant information for the similarities and the differences of these techniques. The similarities of the technique T21 and T22 are that they are using the same value type as the original instructions, redundant and original instruction can be executed in an aligned way, the redundant instructions can be executed after the original ones, the extension of the circuit is applied to the component where the original instructions are executed and the usage of the FU is modified to support the execution of the redundant instructions. In contrast to T22, the technique T21 also modifies the timing aspects of the execution by inserting a new time slot and requires to modify the storage part of the component in order to store the results of misaligned executions.

Another example is between the T12 [34] (similar for T13 and T14) and T15 [26]. Both techniques create a signature with which the original value is compared to. T12 calculates and keeps a bitmask for each static instruction representing which bits are continually invariant across the instruction executions. T15 computes the expected impact of an instruction execution on the architectural

state, when the instruction is at the decode stage. In traditional classifications both techniques belong to the same class. The classification of [18] could classify the approaches under redundant circuit design class due to the signature creation. The classification of [16], T12 and T15 can belong to the execution redundancy class and for the classification of [22], they belong to the RTL level. The proposed classification can identify both the similar and the different parts between T12 and T15. The similarities are that both approaches reuse data in order to generate the additional information (reuse branch) and they reuse the complete data from the system in order to create a signature. However, the T12 uses values observed during previous executions, whereas T15 uses values observed during the current instruction execution. In addition, the signature of T12 is an approximation of the expected value, whereas the signature of T15 is a compression of the expected impact. T12 creates the signature upfront (before the execution of the original information) whereas T15 during the execution of the original instruction. Both techniques modify the original component where the original value is created in order to compute the signature and store it.

7 CONCLUSIONS

In this work we proposed a binary tree classification for error detection and correction techniques without adaptation of their functionality during execution. The creation of our classification relies on top-down splits driven by single attributes coming from the main characteristics of the design space. The top-down split creates two complementary and non-overlapping subbranches clearly dividing the design space in two parts. In contrast with previous works, the proposed classification allows the combinations of the classes. Hence, the proposed classification categorizes a technique by decomposing it into a set of primitive components, where each component belongs to one of the proposed classes. During the design space exploration, the proposed classification follows a unidirectional propagation of design constraints from the left subbranch to the right subbranch. To validate our classification and to provide further insight, we present illustration examples for each of the proposed classes. Last, but not least, we present the decomposition of 43 different error detection and correction techniques using the proposed binary classification. The obtained results highlight which classes are highly populated, indicating that this part of the design space has been largely studied by the literature (such as classes in circuit expansion branch), and which classes have less members, which tends to indicate that this part of the design space remains to be further explored (such as classes in the reuse branch).

ACKNOWLEDGMENTS

This work was partially funded by RAPID (Regime d'Appui pour l'Innovation Duale) research and innovation program from DGA (Direction Generale de l'Armement) and DGE (Direction Generale des Entreprises) under grant agreement for the project FLODAM.

REFERENCES

[1] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (Jan 2004), 11–33. https://doi.org/10.1109/TDSC.2004.2

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

[3] K. Bhattacharya, N. Ranganathan, and S. Kim. 2009. A Framework for Correction of Multi-Bit Soft Errors in L2 Caches Based on Redundancy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 2 (Feb 2009), 194–206. https://doi.org/10.1109/TVLSI.2008.2003236

[4] K. A. Bowman, J. W. Tschanz, Nam Sung Kim, J. C. Lee, C. B. Wilkerson, S. L. L. Lu, T. Karnik, and V. K. De. 2009. Energy-efficient and metastability-immune timing-error detection and recovery circuits for dynamic variation tolerance. In *Proceedings of the 2008 IEEE International Conference on Integrated Circuit Design and Technology and Tutorial*. 155–158. https://doi.org/10.1109/ICICDT.2008.4567268

[5] H. Cho, L. Leem, and S. Mitra. 2012. ERSA: Error Resilient System Architecture for Probabilistic Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 4 (April 2012), 546–558. https://doi.org/10.1109/TCAD.2011.2179038

[6] S. Girbal, M. Moreto, A. Grasset, J. Abella, E. Quinones, F. J. Cazorla, and S. Yehia. 2013. On the convergence of mainstream and mission-critical markets. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–10.

[7] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera. 2011. Architectures for online error detection and recovery in multicore processors. In *Proceedings of the 2011 Design, Automation Test in Europe*. 1–6. https://doi.org/10.1109/DATE.2011.5763096

[8] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. 2013. Reliable On-chip Systems in the Nano-era: Lessons Learnt and Future Trends. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 99, 10 pages. https://doi.org/10.1145/2463209.2488857

[9] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. 2009. Compiler-assisted Soft Error Detection Under Performance and Energy Constraints in Embedded Systems. *ACM Transactions on Embedded Computing Systems* 8, 4, Article 27 (July 2009), 30 pages. https://doi.org/10.1145/1550987.1550990

[10] J. Johnson, W. Howes, M. Wirthlin, D. L. McMurtrey, M. Caffrey, P. Graham, and K. Morgan. 2008. Using Duplication with Compare for On-line Error Detection in FPGA-based Designs. In *Proceedings of the 2008 IEEE Aerospace Conference*. 1–11.

[11] Rajshekar Kalayappan and Smruti R. Sarangi. 2013. A Survey of Checker Architectures. *ACM Comput. Surveys* 45, 4, Article 48 (Aug. 2013), 34 pages.

[12] J.S. Klecka, W.F. Bruckert, and R.L. Jardine. 2002. Error self-checking and recovery using lock-step processor pair architecture. (May 21 2002). http://www.google.ch/patents/US6393582 US Patent 6,393,582.

[13] A. Kritikakou, F. Catthoor, V. Kelefouras, and C. Goutis. 2013. A Systematic Approach to Classify Design-time Global Scheduling Techniques. *ACM Comput. Surveys* 45, 2, Article 14 (March 2013), 30 pages. https://doi.org/10.1145/2431211.2431213

[14] Kuang-Hua Huang and J. A. Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. Comput.* C-33, 6 (June 1984), 518–528.

[15] H. Lee, S. Cho, and B. R. Childers. 2010. PERFECTORY: A Fault-Tolerant Directory Memory Architecture. *IEEE Trans. Comput.* 59, 5 (May 2010), 638–650. https://doi.org/10.1109/TC.2009.138

[16] I. Lee, M. Basoglu, M. Sullivan, D. H. Yoon, L. Kaplan, and M. Erez. 2011. *Survey of error and fault detection mechanisms*. Technical Report. LPH Group, Department of Electrical and Computer Engineering, The University of Texas at Austin.

[17] M. L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. 2008. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 265–276. https://doi.org/10.1145/1346281.1346315

[18] T. Li, J. A. Ambrose, R. Ragel, and S. Parameswaran. 2016. Processor Design for Soft Errors: Challenges and State of the Art. *ACM Comput. Surv.* 49, 3, Article 57 (Nov. 2016), 44 pages. https://doi.org/10.1145/2996357

[19] M. Liu and B. H. Meyer. 2016. Bounding error detection latency in safety critical systems with enhanced Execution Fingerprinting. In *Proceedings of the 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 47–52. https://doi.org/10.1109/DFT.2016.7684068

[20] R. E. Lyons and W. Vanderkulk. 1962. The Use of Triple-modular Redundancy to Improve Computer Reliability. *IBM J. Res. Dev.* 6, 2 (April 1962), 200–209. https://doi.org/10.1147/rd.62.0200

[21] D. Markovic, V. Stojanovic, B. Nikolic, M. A. Horowitz, and R. W. Brodersen. 2004. Methods for true energy-performance optimization. *IEEE Journal of Solid-State Circuits* 39, 8 (Aug 2004), 1282–1293. https://doi.org/10.1109/JSSC.2004.831796

[22] A. Martínez-Álvarez, S. Cuenca-Asensi, and F. Restrepo-Calle. 2016. Soft Error Mitigation in Soft-Core Processors. In *FPGAs and Parallel Architectures for Aerospace Applications*. Springer, 239–258.

[23] J. W. McPherson. 2006. Reliability challenges for 45nm and beyond. In *Proceedings of the 2006 43rd ACM/IEEE Design Automation Conference*. 176–181. https://doi.org/10.1145/1146909.1146959

[24] M. Namjoo and E. HcCluskey. 1995. WATCHDOG PROCESSORS AND CAPABILITY CHECKING. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. IEEE Computer Society, Los Alamitos, CA, USA, 94.

[25] A. Narsale and M. C. Huang. 2009. Variation-tolerant hierarchical voltage monitoring circuit for soft error detection. In *Proceedings of the 2009 10th International Symposium on Quality Electronic Design*. 799–805. https://doi.org/10.1109/ISQED.2009.4810395

[26] R. Nathan and D. J. Sorin. 2014. Nostradamus: Low-cost hardware-only error detection for processor cores. In *Proceedings of the 2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6. https://doi.org/10.7873/DATE.2014.173

[27] M. Nicolaidis. 2015. Double-Sampling Design Paradigm- A Compendium of Architectures. *IEEE Transactions on Device and Materials Reliability* 15, 1 (March 2015), 10–23. https://doi.org/10.1109/TDMR.2014.2388358

[28] F. Oboril and M. B. Tahoori. 2012. 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)*. 1–12. https://doi.org/10.1109/DSN.2012.6263957

[29] J. H. Patel and L. Y. Fung. 1982. Concurrent Error Detection in ALU's by Recomputing with Shifted Operands. *IEEE Trans. Comput.* 31, 7 (July 1982), 589–595. https://doi.org/10.1109/TC.1982.1676055

[30] P. Prata and J. G. Silva. 1999. Algorithm based fault tolerance versus result-checking for matrix computations. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*. 4–11. https://doi.org/10.1109/FTCS.1999.781028

[31] B. Pratt, M. Fuller, and M. Wirthlin. 2011. Reduced-precision redundancy on FPGAs. *International Journal of Reconfigurable Computing* 2011 (2011), 12. https://doi.org/10.1155/2011/897189

[32] R. Psiakis, A. Kritikakou, and O. Sentieys. 2017. Run-time Instruction Replication for permanent and soft error mitigation in VLIW processors. In *Proceedings of the 2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)*. 321–324. https://doi.org/10.1109/NEWCAS.2017.8010170

[33] G. Psychou, D. Rodopoulos, M. M. Sabry, T. Gemmeke, D. Atienza, T. G. Noll, and F. Catthoor. 2017. Classification of Resilience Techniques Against Functional Errors at Higher Abstraction Layers of Digital Systems. *ACM Comput. Surveys* 50, 4, Article 50 (Oct. 2017), 50:1–50:38 pages.

[34] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. 2007. Perturbation-based Fault Screening. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 169–180. https://doi.org/10.1109/HPCA.2007.346195

[35] R. G. Ragel and S. Parameswaran. 2006. IMPRES: integrated monitoring for processor reliability and security. In *Proceedings of the 2006 43rd ACM/IEEE Design Automation Conference*. 502–505. https://doi.org/10.1145/1146909.1147041

[36] A. Rahimi, L. Benini, and R. K. Gupta. 2016. Variability Mitigation in Nanometer CMOS Integrated Systems: A Survey of Techniques From Circuits to Software. *Proc. IEEE* 104, 7 (July 2016), 1410–1448. https://doi.org/10.1109/JPROC.2016.2518864

[37] A. Rajabzadeh and S. G. Miremadi. 2005. A hardware approach to concurrent error detection capability enhancement in COTS processors. In *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing (PRDC'05)*. 8 pp.–. https://doi.org/10.1109/PRDC.2005.7

[38] P. Ramanathan, K. G. Shin, and R. W. Butler. 1990. Fault-tolerant clock synchronization in distributed systems. *Computer* 23, 10 (Oct 1990), 33–42. https://doi.org/10.1109/2.58235

[39] ESA-ECSS Requirements and Standards Division. 2016. ECSS-Q-HB-60-02A, Space product assurance: Techniques for radiation effects mitigation in ASICs and FPGAs handbook. (2016).

[40] D. Rodopoulos, G. Psychou, M. M. Sabry, F. Catthoor, A. Papanikolaou, D. Soudris, T. G. Noll, and D. Atienza. 2015. Classification Framework for Analysis and Modeling of Physically Induced Reliability Violations. *ACM Comput. Surveys* 47, 3, Article 38 (Feb. 2015), 33 pages. https://doi.org/10.1145/2678276

[41] E. Rotenberg. 1999. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*. 84–91. https://doi.org/10.1109/FTCS.1999.781037

[42] A. Sari and M. Psarakis. 2011. Scrubbing-based SEU mitigation approach for Systems-on-Programmable-Chips. In *Proceedings of the 2011 International Conference on Field-Programmable Technology*. 1–8. https://doi.org/10.1109/FPT.2011.6132703

[43] A. Sari, M. Psarakis, and D. Gizopoulos. 2013. Combining checkpointing and scrubbing in FPGA-based real-time systems. In *Proceedings of the 2013 IEEE 31st VLSI Test Symposium (VTS)*. 1–6. https://doi.org/10.1109/VTS.2013.6548910

[44] A. L. Sartor, A. F. Lorenzon, L. Carro, F. Kastensmidt, S. Wong, and A. C. S. Beck. 2015. A Novel Phase-Based Low Overhead Fault Tolerance Approach for VLIW Processors. In *Proceedings of the 2015 IEEE Computer Society Annual Symposium on VLSI*. 485–490. https://doi.org/10.1109/ISVLSI.2015.19

[45] A. L. Sartor, S. Wong, and A. C. S. Beck. 2016. Adaptive ILP control to increase fault tolerance for VLIW processors. In *Proceedings of the 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 9–16. https://doi.org/10.1109/ASAP.2016.7760767

[46] S. Sengupta, K. Saurabh, and P. E. Allen. 2004. 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)*, Vol. 1. I–325–I–328 Vol.1. https://doi.org/10.1109/ISCAS.2004.1328197

[47] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam. 2015. Mitigation of Radiation Effects in SRAM-Based FPGAs for Space Applications. *ACM Comput. Surv.* 47, 2, Article 37 (Jan. 2015), 34 pages. https://doi.org/10.1145/2671181

[48] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. 2004. Temperature-aware Microarchitecture: Modeling and Implementation. *ACM Transactions in Architecture and Code Optimisation* 1, 1 (Mar 2004), 94–125. https://doi.org/10.1145/980152.980157

[49] T. Sudo, H. Sasaki, N. Masuda, and J. L. Drewniak. 2004. Electromagnetic interference (EMI) of system-on-package (SOP). *IEEE Transactions on Advanced Packaging* 27, 2 (May 2004), 304–314. https://doi.org/10.1109/TADVP.2004.828817

[50] S. Valadimas, A. Floros, Y. Tsiatouhas, A. Arapoyanni, and X. Kavousianos. 2014. The Time Dilation Technique for Timing Error Tolerance. *IEEE Trans. Comput.* 63, 5 (May 2014), 1277–1286. https://doi.org/10.1109/TC.2012.289

[51] N. J. Wang and S. J. Patel. 2006. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (July 2006), 188–201. https://doi.org/10.1109/TDSC.2006.40

[format=acmsmall, review=false, screen=true]acmart

booktabs

subfigure float

tabularx

enumitem multirow

xr rsec:PROB23Target Domain and Problem Formulationsection.2rrelated$_{w}ork34RelatedWorksection.3rGiz1$

Received September 2018

Appendix of "Binary Tree Classification of Rigid Error Detection and Correction Techniques"

ANGELIKI KRITIKAKOU, Univ Rennes, Inria, CNRS, IRISA, France
RAFAIL PSIAKIS, Univ Rennes, Inria, CNRS, IRISA, France
FRANCKY CATTHOOR, IMEC, KU Leuven, Belgium
OLIVIER SENTIEYS, Univ Rennes, Inria, CNRS, IRISA, France

ACM Reference Format:
Angeliki Kritikakou, Rafail Psiakis, Francky Catthoor, and Olivier Sentieys. 2020. Appendix of "Binary Tree Classification of Rigid Error Detection and Correction Techniques". *ACM Comput. Surv.* 1, 1, Article 1 (January 2020), 12 pages. https://doi.org/10.1145/3397268

1 FULL DECOMPOSITION OF THE TECHNIQUES UNDER STUDY

In this annex, we provide the complete explanation of the decomposition of the techniques presented in Table 20 of manuscript entitled "Binary Tree Classification of Rigid Error Detection and Correction Techniques".

1.1 Golden reference

Technique 1: The Transition Detector with Time-Borrowing latch (TDTB) presented in [2] senses input data transitions when the clock is logically high. As the input data transitions, a pulse is generated at the output of a XOR gate. If the input data arrives late during the logically high clock period, the pulse discharges the output node voltage of a dynamic gate, which reports the error. During the logically low clock cycle, the output node is precharged. Therefore, the technique belongs to the golden reference branch and it is categorized in the following classes: 1) transition/moment, as TDTB senses the transitions, 2) computation/static, as the golden reference is determined by the upfront clock period, 3) compatibility/extract, for the XOR gate that detects if the signal changes, 4) compatibility/sample, as the dynamic gate verifies if the change takes place when the clock is high, and 5) matching/detect, as the technique performs only detection.

Technique 2 and 3: One of the approaches presented in [10] (T2) uses fatal hardware traps to detect errors. More specifically, the approach uses a watchdog, where a reset trap is thrown when no instruction retires within an upfront given number of ticks. Therefore, the technique belongs to the golden reference branch and it is categorized in the following classes: 1) transition/duration, as it is based on the number of ticks, 2) computation/static, as the threshold on the number of ticks is given upfront, and 3) matching/detect, as only detection occurs. Similar is the classification for the

Authors' addresses: Angeliki Kritikakou, Univ Rennes, Inria, CNRS, IRISA, Campus de Beaulieu, 263 Avenue General Leclerc, Rennes, 35042, France, angeliki.kritikakou@inria.fr; Rafail Psiakis, Univ Rennes, Inria, CNRS, IRISA, Campus de Beaulieu, 263 Avenue General Leclerc, Rennes, 35042, France, rafail.psiakis@inria.fr; Francky Catthoor, IMEC, KU Leuven, Kapeldreef 75, Leuven, 30001, Belgium, Francky.Catthoor@imec.be; Olivier Sentieys, Univ Rennes, Inria, CNRS, IRISA, Campus de Beaulieu, 263 Avenue General Leclerc, Rennes, 35042, France, olivier.sentieys@inria.fr.

0360-0300/2020/1-ART1 $15.00
https://doi.org/10.1145/3397268

Nostradamus [17] (T3) for the errors that cause the core to hang. A watchdog timer considers that an error has occurred, if no instruction has committed in the past ten thousand cycles.

Technique 4: The Control Flow Checking by Execution Tracing [28] monitors the target address of branch instructions and compares it with the address estimated during compilation. Therefore, the technique belongs to the golden reference branch and it is categorized in the following classes: 1) value/identical and 2) computation/static, as an address is estimated before execution, 3) compatibility/sample, due to execution tracing, and 4) matching/detect, as only detection occurs.

Technique 5 and 6: The Integrated Monitoring for Processor REliability and Security (IMPRES) [27] (T5) calculates a checksum for each basic block at compile time and re-calculates the checksums at runtime to be compared with the value computed at compile time. Therefore, the technique belongs to the golden reference branch and it is categorized in the following classes: 1) value/compression, as checksums are used as golden reference, 2) computation/static, as the checksums are calculated at compilation, 3) compatibility/process and 4) compatibility/sample, as the checksums are run-time recomputed and stored 5) matching/detect, as error detection is performed. Similar classification has the technique of [4] (T6), which observes the processor's execution trace through hashed values of basic blocks at run-time, checks whether the execution trace aligns with the expected program behavior produced at compile time.

Technique 7 and 8: The scrubbing techniques for the configuration memory belong also to the golden reference branch. More precisely, the technique of [30, 31] (T7) uses Error Correction Code (ECC) for the configuration memory frames and Cyclic Redundancy Check (CRC) for the entire memory. An each read-back, the corresponding syndrome is calculated and compared with the golden pre-calculated value. If a damaged frame is found, it is corrected. Therefore, the technique belongs to the golden reference branch and it is categorized in the following classes: 1) value/compression and 2) computation/static, due to the pre-calculated ECC and CRC, 3) compatibility/extract, as a readback has to occur, 4) compatibility/process and 5) compatibility/sample, to recompute and store the ECC and CRC, 6) matching/detect, 7) matching/reconstruct, as both detection and correction occur. A similar classification has the technique of [36] (T8), which uses a hamming code based error detection and correction. A syndrome memory holds the upfront calculated syndromes. During execution, the original syndrome is calculated and compared for error detection and correction.

Technique 9: The detection technique of [16] is based on monitoring the supply rail and, thus, the potential disturbance caused by a particle strike. Therefore, the technique belongs to the golden reference branch and it is categorized in the following classes: 1) information/value/type, as the disturbance is monitored, 2) computation/static, as the decision of to detect an error is made if the disturbance exist or not, and 3) matching/detect, as only error detection takes place.

Technique 10:* ReStore framework [35] is based on symptom-based methods that take advantage of inherent states of the processor architecture in order to detect the presence of soft errors, such as the ISA-defined exceptions. When a symptom occurs, re-execution takes place. The detection of memory access and alignment exceptions is performed by using the lower and upper bounds of the address space, which is determined before execution. Therefore, the technique belongs to the golden reference branch and it is categorized in the following classes: 1) information/value/approximation, due to the use of upper and lower bounds, 2) computation/static, as the address space is defined compile time, 3) matching/detect, the technique performs detection. The part related to the detection of incorrect control flow is based on confidence predictors. When a mis-speculation is discovered, it is considered that the predictor is correct and the mispeculation implies an error. Therefore, this part of the technique belongs also to the golden reference branch and it is categorized in the following classes: 4) value/identical, as an error is detected if mispeculation takes place, 2) computation/static, as it is upfront decided that a mispeculation means an error, and 5) matching/detect, as this part of

the technique performs detection. The part of the technique relevant to the event logs, re-executes the task on the same core and tracks and records events during the original and the redundant execution . In case an error is detected, a third execution takes place to identify if the error was on the original execution. Therefore, this part of the technique belongs to the explicit creation branch and it is categorized in the following classes: 6) value/identical, as the same sequence of instructions is executed, 7) instruction sequence/regenerate, as the sequence of instructions is re-executed on the same component and with the same way as the original sequence, 8) compatibility/sample, due to the creation of the log file for the original execution, 9) matching/detect, and 10) matching/reconstruct, as both detection and correction occur.

Technique 41: The Algorithmic Based Fault Tolerant (ABFT) approach [8] extends the input matrix with supplementary columns and rows containing checksums. Then, the matrix multiplication algorithm applies similar mathematical operations to both the original data and the checksum during execution to keep the checksum relationship invariant. Then, the new checksum is computed based on the original data, enabling error detection and error correction. Hence, this approach is categorized as: 1) golden reference/information/arithmetic value/lossy/compression class, as checksums are computed, 2) golden reference/generation/static class, as initial checksums are computed before execution, 3) golden reference/generation/process class, as the ABFT applies mathematical operations during execution that processes the initial checksums, 4) comparison/compatibility/sample class and 5) comparison/compatibility/process class, as a new checksum has to be computed based on the original data, 6) comparison/detect and comparison/reconstruct, as the checksums allow to detect and also correct some errors.

Technique 42: Capability checking approaches aim at the detection of malfunctions that cause illegal access to the memory system [15]. The capability of an object includes address, type and access rights. This information is used to pre-calculate offline a table with the access rights of each object towards another object. During execution, a low-cost processor translates the physical addresses of a specific memory reference and the physical address of the object it wants to access. Then, it verifies if the access rights from the main processor are compatible with the access-rights provided by the table. If they are not compatible, an error is signaled. Therefore, this approach belongs to: i) golden reference/arithmetic value/identical class and ii) golden reference/generation static class, as the exact access rights are computed offline for each pair of objects, iii) comparison/compatibility/sample class and iv) comparison/compatibility/process class, as memory accesses have to be translated in order to obtain the corresponding access rights from the table, and v) comparison/matching detect class, as an error is signaled when the access rights from the main processor and the golden reference do not match.

Technique 43:* Consistency check methods use the knowledge of a transformation, which relates the inputs to the outputs of an algorithm, to perform error detection and correction. Consider an algorithm that computes the inverse matrix A^{-1} of the input matrix A and it uses the expression $A \times A^{-1} = I$, where I is the identity matrix [23], as consistency check. Therefore, this approach belongs to: i) golden reference/arithmetic value/identical class and ii) golden reference/generation static class, as the matrix I is used as a golden reference, with which we will compare the output of the algorithm and it is offline defined, iii) reuse/data/original format and reuse/enable/previous execution class, as the input matrix A is reused, iv) comparison/compatibility/sample class and v) comparison/compatibility/process class, as after the computation of the output matrix A^{-1}, it has to be processed through $A \times A^{-1}$ in order to be compared with the golden reference, and vi) comparison/matching detect class, as errors are detected.

1.2 Reuse existing values

Technique 11:* The technique of [34] performs error detection by directly comparing the input and output data of a flip-flop. After error detection the logic evaluation time is extended by a clock cycle for error correction. A multiplexer (MUX) with a feedback configuration to capture delayed valid data is inserted to the original data path. Correction is done by re-feeding the flip-flop with the correct data of the MUX-latch. Therefore, the technique belongs to the explicit creation branch. The error detection part belongs to the reuse branch, as no new value is explicitly created, and it is categorized in the following classes: 1) data/identical/original format and 2) enable/calculated value, as the same original value is reused for detection coming from the input of the flip-flop, and 3) matching/detect. For the error correction, the technique modifies the hardware, so it belongs to the circuit expansion branch. This part is categorized in the following classes: 4) value/identical, as the same data are captured, 5) select component/same, as the hardware modifications take place at around the original flip-flop, 6) component modifications/functionality, due to the inserted MUX, 7) component modifications/control plane/timing and 8) time-wise information/skew, as the MUX is controlled by an error flip-flop driven by a delayed clock, and 9) matching/reconstruct.

Technique 12, 13* and 14*:* A fault screener reports an error if the program's current behavior is inconsistent with the expected behavior given by a valid value space. The way to compute the valid value space determines the components of the technique. The invariance-based fault screener [26] (T12) calculates and keeps a bitmask for each static instruction representing which bits are continually invariant across the instruction executions. Therefore, the technique belongs to the explicit creation branch and it has a part that belongs to reuse branch, since it reuses the history of the instructions to compute the invariant mask, and to the circuit expansion, as the invariant mask is calculated by the hardware. The technique categorized in the following classes: 1) data/identical/original format, since the complete value of the static instruction is re-used for the creation of the invariant, 2) enable/previous execution, since the values previously observed are re-used for the creation of the mask, 3) value/approximate, as the online computed value is an invariant mask, 4) time-wise information/preprone, as each time a value is produced, the bitmask is updated in order to be used for the next value, 5) select component/same, 6) component modifications/functionality, and 7) component modifications/storage, as the original hardware is extended to include the hardware to compute the invariant and to store it, 8) compatibility/sample, as the original value is sampled, and 9) matching/detect. The same classification has the dynamic-range fault screener [26] (T13) that detects an error when a static instruction gives a value that it is outside of its normal range, computed run-time based on the observed values. Similarly, the extended history fault screener [26] is decomposed in the same way since it keeps a history of 64 unique values and 64 unique deltas between successive values for each static instruction.

Technique 15:* Nostradamus [17] for each instruction, it compares the instruction's expected impact on the architectural state with the actual impact that has the instruction's execution. An expectation unit operates in parallel with the normal instruction decode logic to determine the instruction expectation, i.e. how the instruction will modify the architectural state compressed in a signature. The modification of the architectural state is expressed as the instruction expectation signature, which is a hashed value. During the execution stage, the real impact of the instruction execution is computed and compared with the expected one in order to define an error. Therefore, the technique belongs to the explicit creation branch and it has a part that belongs to reuse branch, since it reuses the information from the decode stage to compute the signature mask, and to the circuit expansion, as the signature is calculated by the hardware. The technique categorized in the following classes: 1) data/value/original format, the value reused is the original value of the registers, 2) enable/current execution/stored value, as the same instruction is executed only in

different pipeline stages and the value reused is stored in the registers, 3) value/compression, as a signature is created, 4) time-wise info/skew, as the signature is created at the decode stage during the execution of the original instruction, 5) select component/same, as the original pipeline is modified, 6) component modifications/functionality and 7) component modifications/storage, for the additional hardware for the calculation of the signature and the additional register to store it, 8) compatibility/extract, to observe the history of the instruction execution, 9) compatibility/process, to compute the real signature and 10) compatibility/sample, to store it, and 11) matching/detect, as the technique performs error detection.

1.3 Instruction sequence extension

Technique 16:* The Execution Fingerprinting (EX) allows tasks to be executed at different times on redundant cores and compresses the changes into an external state, called fingerprint. At regular intervals, the original and redundant fingerprints are compared. EX State Checkpointing [11] is enabled periodically and pushes all register state to the fingerprinting unit. Therefore, the technique belongs to the explicit creation branch and it has a part that belongs to instruction sequence extension branch, since a redundant task is executed, and to the circuit expansion, as the fingerprint is calculated by the hardware. The technique categorized in the following classes: 1) value/identical, as the redundant task is the same as the original task, 2) time-wise information/skew, 3) time-wise information/align, 4) time-wise information/prepone and 5) time-wise information/postpone, as the redundant task is executed without time restrictions, 6) resource selection/unoccupied, as the second task is executed on a different core, 7) value/compression, as the register states are hashed to create the fingerprint, 8) time-wise information/align, as the fingerprint is created when the register state is pushed to the fingerprint unit, 9) new component, as a new core has been inserted for the execution of the redundant task,with the capabilities of creating the fingerprint, 10) compatibility/process, in order to create the original fingerprint, 11) compatibility/sample, to store the original fingerprint, 12) matching/detect, as the technique performs error detection.

Technique 17:* The distributed temporal redundancy technique [14] is similar to the technique 16. The main difference is 1) matching/reconstruct, if the fingerprints do not match, a third copy is executed on a third resource to determine the correct value.

Technique 18: The technique of [7] proposes a lock-step processor pair architecture, where the two processor units are fully synchronized executing identical instruction streams. Therefore, the technique belongs to the explicit creation branch, and more precisely to the instruction sequence extension as two identical streams are used, and it is categorized in the following classes: 1) time-wise information/align, as the cores are executed in lock-step so the task execution is fully synchronized, 2) resource selection/unoccupied class, as an idle core of the platform is used for the duplicated instruction stream, 3) matching/detect.

Technique 19:* The instructions are duplicated on VLIW datapaths by the compiler in [5]. The scheduling of the duplicated instructions is done along at a different function unit of the VLIW or after the original instructions. When this is not possible, a new time slot is added to create slack. The original values are stored to a register value queue and the addresses to the load/store queue. To find with which original value from the queue a duplicated value has to be compared with, the technique uses the output register address. Therefore, the technique belongs to the explicit creation branch, and more precisely to the instruction sequence extension as the duplication is performed in the instruction memory by the compiler. The technique is categorized in the following classes: 1) value/identical, as the same instructions are duplicated, and, thus, the same values are created, 2) time-wise information/align, and 3) resource selection/unoccupied, when the execution is done at the same time with the original instructions, 4) regenerate and 5) time-wise information/postpone, when it the duplicated instructions are executed after the original ones, 6) resource selection/slack,

when a new time slot is added, 7) compatibility/extract, to find out with which instruction to compare to, 8) compatibility/sample, due to the storing of the original values to the queue, and 9) matching/detect. In addition, the store/load queue is protected by a parity bit. This part of the approach belongs to: 10) value/compression, due to the parity, 11) time-wise align, as the parity is computed at the same time as the original value, 12) select component/same and 13) component modifications/storage, as the component where the original value is store is extended to keep the parity, and 14) component modifications/functionality, to compute the parity.

Technique 20: The approach of [29] executes with a delay a redundantly generated instruction stream (R-steam) after the start of the execution of the active stream (A-stream) on a Simultaneous Multi-Threaded (SMT) architecture. Then, the execution of the individual instructions is decided based on a dynamic scheduler. Therefore, the technique belongs to the explicit creation branch, and more precisely to the instruction sequence extension as two streams are executed. The technique is categorized in the following classes: 1) value/identical, as the streams are the same, 2) instruction sequence/time-wise information/skew, for the execution of the R-stream, 3) time-wise information/align, 4) time-wise information/prepone, 5) time-wise information/postpone, for the execution of the individual instructions of the R-stream, 6) resource selection/unoccupied FU, when the execution is performed on different FU than the ones used by the corresponding instructions of the A-stream, 7) regenerate, when the R-stream instructions are executed just after the corresponding A-stream instructions, 8) matching/sample, as the results of each A-stream instruction are also pushed onto the Delay Buffer, and 9) matching/detect.

1.4 Circuit expansion

Technique 21: The technique of [25] a hardware mechanism that duplicates or triplicates the instructions inside the execution and decoding stages in order to take advantage of the idle slots in the current and next instruction bundle. The replicated instructions can be executed in the same time slot in another idle FU or in the next time slot in an idle FU (either same FU as the original or another FU). When there are not enough idle resources to exploit, the pipeline is stalled to add one or more additional time slots to create time slack. Therefore, the technique belongs to the explicit creation branch and it is categorized in the following classes: 1) value/identical, as the same instructions are executed and, thus, the same values, 2) time-wise information/align and 3) select component/idle, when the replicated instructions are executed in the same slot as the original, 4) time-wise information/postpone and 5) select component/same, when the replicated instructions are executed in the next slot, 6) select component/slack, due to the inserted new time slot, 7) component modifications/usage, as the usage of the FUs is modified due to the run-time scheduling of instructions, 8) component modifications/timing, as stalls are inserted, 9) component modifications/data plane/storage, as additional registers are inserted to keep the values due to misaligned execution of the original and replicated instructions, 10) compatibility/sample, as the original result may be stored when needed due to the misaligned execution, 11) matching/detect, when duplication is used, 12) matching/reconstruct, when triplication is used.

Technique 22: The phase-configurable VLIW processor [32] uses idle issue slots during a whole given program phase to execute duplicated instructions from the corresponding coupled pipeline and check their results. When a mismatch is found, the pipeline is flushed and the last instruction bundle is executed again. Therefore, the technique belongs to the explicit creation branch and it is categorized in the following classes: 1) value/identical, as the coupled pipeline executes the same instructions as the original one, 2) circuit expansion/time-wise information/align and 3) select component/idle, as both duplicated and original values are executed at the same slot for error detection, 4) time-wise information/postpone and 5) select component/same, due to the re-execution of the

original instruction for the error correction, 6) component modifications/usage, as the pipelines now executes original or duplicated instructions, 7) matching/detect, and 8) matching/reconstruct.

Technique 23: An improved version of [32] is presented in [33]. In case there are not enough idle slots, an additional time slot is inserted to create slack and accommodate the remaining replicated instructions. Hence, this technique has a similar categorization to the previous technique except that: 1) time-wise information/postpone, 2) select component/slack, and 3) component modifications/timing are included, due to the inserted additional time slot.

Technique 24: Nostradamus [17] to deal with errors in functional units uses residue coding implemented in hardware. Therefore, the technique belongs to the explicit creation branch and it is categorized in the following classes: 1) value/approximation, as the additional value created during the execution is obtained by additional residue code operands, 2) time-wise information/align, as original and additional values are created at the same time, 3) select component/same and 4) component modifications/functionality, due to the inserted additional residue code operand in the original data-path, 5) compatibility/process, since the original value has also to be reduced to the modulo operation before comparing with the additional value, and 6) matching/detect.

Technique 25, 26 and 27: Double sampling methods [19] (T23) create the information in a moment that is skewed with respect to the original value. In Razor [3] (T24) the pipeline flip-flop is modified by inserting a shadow latch triggered by a delayed clock signal and latches the same signal as the main flip-flop. The value in the shadow latch, which is guaranteed to be correct, is utilized to correct the delay failure, when an error is detected. A similar approach is the simplified Razor Flip-Flop (RFF) [2] (T25), where the metastability detector is omitted. Therefore, these techniques belong to the explicit creation branch, and more precisely to circuit expansion branch. They are categorized in the following classes: 1) value/identical, as the same signal is latched by the shadow flip-flop, 2) time-wise information/skew, due to the delayed clock signal, 3) select component/same, since the extra shadow latch is introduced in parallel with the original flip-flop, 4) component modifications/timing, as the shadow latch is triggered by a delayed clock, 5) component modifications/storage, due to the use of the shadow latch, 6) matching/detect and 7) matching/reconstruct.

Technique 28: The Reduced Precision Redundancy (RPR) technique creates additional values with reduced precision computed by two additional arithmetic circuits [24]. Therefore, the technique belongs to the explicit creation branch and it is categorized in the following classes: 1) value/approximation, since the reduced precision values can be produced either by truncation or rounding, 2) time-wise information/align, as the additional values are created at the same time with the original one, 3) select component/same, as the original component is extended to include the RPR hardware, 4) component modifications/functionality, as the additional arithmetic circuits are inserted, 5) matching/detect, at the approach performs detection and 6) matching/reconstruct, as the approach performs correction.

Technique 29: Duplication with comparison inserts an identical copy of a circuit and compares its output with the original one [6]. Therefore, the technique belongs to the explicit creation branch and it is categorized in the following classes: 1) value/identical, as exactly the same circuit is used and, thus, the same values are created, 2) time-wise information/align, as it creates the duplicated value at the same time with the original, 3) hardware information/new component, as a new identical circuit is inserted, 4) matching/detect, as the approach performs detection.

Technique 30: The fault technique for exclusive directory entries in a cache coherency directory of [9] modifies the original sharer field to a binary representation to include Single Error Correction Double Error Detection (SECDED). Therefore, the technique belongs to the explicit creation branch and it is categorized in the following classes: 1) value/compression, due to the SECDED used, 2) time-wise information/align, as SECDED and original value are stored at the same time, 3) hardware information/select component/same, as SECDED and original value are stored at the same entry, 4)

component modifications/usage, as the original entry is modified to binary representation to create space for the SECDED, 5) component modifications/functionality, so as to compute the SECDED, 6) compatibility/process, to compute the SECDED for the original value, 7) matching/detect, as double error detection is possible, and 8) matching/reconstruct, due to single error correction.

Technique 31: The fault technique for shared directory entries [9] is as follows: the entry is read once and stored to a new register, then it is negated and stored back to the directory. Then, it is read again and stored to another register. The two registers are compared for error detection. Therefore, the technique belongs to the explicit creation branch and it is categorized in the following classes: 1) value/encoded, as the information to perform the comparison is negated, 2) time-wise information/prepone, as the original value is read after the creation of the additional information, 3) select component/same, as the original directory is modified to create the additional value, 4) component modifications/functionality, as the negation circuit is inserted, 5) component modifications/storage, for the new register to store the negated value, 6) component modifications/usage, as the way of using the data plane is modified to support the negation and storing back, 7) compatibility/sample, as the original value is stored to another register, and 8) matching/detect.

Technique 32:* The hardware technique of [1] multi-bit errors are detected using simple Error Detection Codes (EDC), like hamming distance or cyclic redundancy codes (CRC) for the cache lines. When an error is detected, it is corrected using the data redundancy in the memory hierarchy, i.e. between the write-through L1 cache and the L2 cache and the clean data lines of the L2 cache and the main memory. Errors become affective when a L2 cache line is replaced, and, thus, written back to the memory. So, in a L2 cache replacement, the entry has to checked for soft errors. In case an error exist, the correct data are fetched from L1 cache. Errors in clean L2 cache lines can be corrected by re-fetching them from the main memory. Therefore, the technique belongs to the explicit creation branch, and more precisely to the reuse branch (for the error correction) and the circuit expansion (for the error detection codes). The technique is categorized in the following classes: 1) reuse/data/original format, as the same data is reused from L1 cache (main memory), 2) enable/previous execution, as the values have been written to L1 cache (main memory) before the L2 cache replacement. The second part is the error detection codes applied to the cache lines and it belongs to: 3) explicit creation/value/compression, for the EDC, 4) circuit expansion/time-wise information/align, as the EDC are created at the same time with the original value, 5) select component/same, as the EDC is stored at the same cache line with the original value, 6) component modifications/functionality, for the computation of EDC, 7) component modifications/storage, to store the EDC bits. The approach performs both 8) matching/detect and 9) matching/reconstruct.

Technique 33: The technique of fine-grained error detection with carry propagation chains of [18] uses the carry chain already included in FPGA slices to generate an error indication signal. The duplicated information is stored in currently unused LUTs. Therefore, the technique belongs to the explicit creation branch and it is categorized in the following classes: 1) value/identical, as the same value as the original one is created, 2) circuit expansion/time-wise information/align, as the duplicated value is computed at the same time as the original, 3) select component/idle, as unused LUTs are used for the duplication, 4) matching/detect.

Technique 34: Triple modular redundancy where the same circuit is executed three times in parallel [12] belongs to circuit expansion branch: 1) value/identical, 2) circuit expansion/time-wise information/align, 3) hardware information/new, 4) matching/detect and 5) matching/reconstruct.

Technique 35:* The technique in [21] explores the effect that has the enabling of the error detection technique only for a small period of time to the number of undetected errors. The execution time is divided into epochs, and each epoch to a singular period and a DMR period. When an error occurs, it is detected in a DMR period and the system rolls back to the previous checkpoint. For the detection, each core is enhanced with a reliability manager with control logic and two buffers.

Two cores are coupled for the DMR period to execute the duplicated instructions. At commit-time, the architectural state updates are sent to the checker core in a form of fingerprint. If an error is detected, the application is re-executed. Therefore, the technique belongs to the explicit creation branch, more precisely in the instruction sequence extension (instructions running on two different cores) and in the circuit expansion (fingerprint and synchronization). The technique is categorized in the following classes: 1) value/identical, as the same instructions are required, 2) instruction sequence/postpone and 3) resource selection/unoccupied, when an error is found, 4) instruction sequence/time-wise information/skew, 5) instruction sequence/time-wise information/align, 6) instruction sequence/time-wise information/prepone, 6) instruction sequence/time-wise information/postpone, as the duplicated instructions are executed without time restriction and the cores may not be synchronized, 7) resource selection/unoccupied, for the coupled core, 8) value/compression, for the use of fingerprint, 9) circuit expansion/time-wise information/align, as the fingerprint is created at the commit stage of the original instruction, 10) select component/same, as the core is modified with the manager to create the fingerprint, 11) component modifications/timing, for the synchronization performed by the manager, 12) component modifications/functionality, for the computation of the fingerprint, 13) component modifications/storage, for the use of the buffers to store the fingerprint, 14) compatibility/process and 15) compatibility/sample, for the computation and storage of the fingerprint of the original value, 16) matching/detect, and 17) matching/reconstruct.

Technique 36: The hardware mechanism of [1] identifies small data values, which can fit in the half of the word in the memory. Then, the upper half of the memory word is used for duplicating the data value. Multi-bit errors in the lower half of the word is corrected using the duplicate copy in the upper half. The multi-bit errors which cannot be corrected using the inherent redundancy are corrected by using a small Error Correction Code (ECC) cache. Therefore, the technique belongs to the explicit creation branch, and more precisely to the circuit expansion. It is categorized in the following classes: 1) explicit creation/identical, for the small data values as duplication is applied in this case, 2) circuit expansion/time-wise information/align, as both original small value and duplicated small value are stored at the same time, 3) select component/same, as both original small value and duplicated small value are stored in same word in the memory, 4) component modifications/usage, to use the same word for the original and the duplicated value, 5) component modifications/functionality, to identify the small data values. The part related to the error correction codes and it is categorized in: 6) explicit creation/value/compression, for the ECC, 7) circuit expansion/time-wise information/align, as they are created at the same time with the original value, 8) select component/new, as a new cache is inserted for the ECC. The approach performs both 9) matching/detect and 10) matching/reconstruct.

Technique 37: The check-on-write approach in caches [20] performs error detection and correction of a value before it is overwritten based on Error Correction Codes. Therefore, the technique belongs to the explicit creation branch, and more precisely to the circuit expansion. It is categorized in the following classes: 1) value/compression, for the ECC, 4) circuit expansion/time-wise information/align, as the ECC is created when the value is created, 5) select component/same, as the ECC is stored at the same cache line with the original value, 6) component modifications/functionality, for the computation of ECC, 7) component modifications/storage, to store the ECC bits. The approach performs both 8) matching/detect and 9) matching/reconstruct.

Technique 38: For the errors in storage, Nostradamus [17] uses error detection codes, such as parity based on XOR among the bits. Therefore, the technique belongs to the circuit expansion. It is categorized in the following classes: 1) value/compression, for the parity, 2) circuit expansion/time-wise information/align, as the parity is created when the value is created, 3) select component/same, as the parity is stored with the original value, 4) component modifications/functionality, for the computation of parity, 5) component modifications/storage, as the size is extended to include the

parity bit. The approach belongs to 6) compatibility/process, since the parity of the original value has to be recomputed, and 7) matching/detect, as only error detection occurs.

Technique 39:* The token coherence signature checker [13] checks for errors in cache coherency. It is based on computing signatures that represent recent histories of coherence events at memory and cache controllers. Periodically, these signatures are sent to a verifier to check for errors. Therefore, the technique belongs to the reuse branch (observing the events) and in the circuit expansion (signature creation). The technique is categorized in the following classes: 1) data/original and 2) enable/previous execution, since recent history of occurring events during execution are observed, 3) value/compression, since signatures are used, 4) circuit expansion/time-wise information align, as the signatures are updated every time an event occurs, 5) select component/same, as the memory controller is extended to compute the signature, 6) component modifications/functionality, and 7) component modifications/storage, for the signature computation and storage, 8) compatibility/process and 9) compatibility/sample, for the second signature, and 10) matching/detect.

Technique 40: The REcomputed with Shifted Operands (RESO) [22] shift the inputs of the ALU, reperforms the same instruction and the output is shifted back and compared with the original value. Therefore, the technique belongs to the circuit expansion. The technique is categorized in the following classes: 1) value/encoded, since the operands and the result are shifted, 2) circuit expansion/time-wise information/postpone, as the execution of the second instruction is performed after the original one, 3) select component/same, as the original ALU is modified to support the RESO operation, 4) component modifications/usage, to modify the use of the ALU during the additional information creation, 5) component modifications/functionality, and 6) component modifications/storage, as the original ALU data-path is extended with shifters and one register, 7) compatibility/sample, for storing the original value and 8) matching/detect.

REFERENCES

[1] K. Bhattacharya, N. Ranganathan, and S. Kim. 2009. A Framework for Correction of Multi-Bit Soft Errors in L2 Caches Based on Redundancy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 2 (Feb 2009), 194–206. https://doi.org/10.1109/TVLSI.2008.2003236

[2] K. A. Bowman, J. W. Tschanz, Nam Sung Kim, J. C. Lee, C. B. Wilkerson, S. L. L. Lu, T. Karnik, and V. K. De. 2009. Energy-efficient and metastability-immune timing-error detection and recovery circuits for dynamic variation tolerance. In *Proceedings of the 2008 IEEE International Conference on Integrated Circuit Design and Technology and Tutorial*. 155–158. https://doi.org/10.1109/ICICDT.2008.4567268

[3] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. 2003. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 7–. http://dl.acm.org/citation.cfm?id=956417.956571

[4] Y. Fei and Z. J. Shi. 2007. Microarchitectural Support for Program Code Integrity Monitoring in Application-specific Instruction Set Processors. In *Proceedings of the 2007 Design, Automation Test Exhibition*. 1–6. https://doi.org/10.1109/DATE.2007.364391

[5] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. 2009. Compiler-assisted Soft Error Detection Under Performance and Energy Constraints in Embedded Systems. *ACM Transactions on Embedded Computing Systems* 8, 4, Article 27 (July 2009), 30 pages. https://doi.org/10.1145/1550987.1550990

[6] J. Johnson, W. Howes, M. Wirthlin, D. L. McMurtrey, M. Caffrey, P. Graham, and K. Morgan. 2008. Using Duplication with Compare for On-line Error Detection in FPGA-based Designs. In *Proceedings of the 2008 IEEE Aerospace Conference*. 1–11.

[7] J.S. Klecka, W.F. Bruckert, and R.L. Jardine. 2002. Error self-checking and recovery using lock-step processor pair architecture. (May 21 2002). http://www.google.ch/patents/US6393582 US Patent 6,393,582.

[8] Kuang-Hua Huang and J. A. Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. Comput.* C-33, 6 (June 1984), 518–528.

[9] H. Lee, S. Cho, and B. R. Childers. 2010. PERFECTORY: A Fault-Tolerant Directory Memory Architecture. *IEEE Trans. Comput.* 59, 5 (May 2010), 638–650. https://doi.org/10.1109/TC.2009.138

[10] M. L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. 2008. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 265–276. https://doi.org/10.1145/1346281.1346315

[11] M. Liu and B. H. Meyer. 2016. Bounding error detection latency in safety critical systems with enhanced Execution Fingerprinting. In *Proceedings of the 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 47–52. https://doi.org/10.1109/DFT.2016.7684068

[12] R. E. Lyons and W. Vanderkulk. 1962. The Use of Triple-modular Redundancy to Improve Computer Reliability. *IBM J. Res. Dev.* 6, 2 (April 1962), 200–209. https://doi.org/10.1147/rd.62.0200

[13] A. Meixner and D. J. Sorin. 2007. Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 145–156. https://doi.org/10.1109/HPCA.2007.346193

[14] B. H. Meyer, B. H. Calhoun, J. Lach, and K. Skadron. 2011. Cost-effective safety and fault localization using distributed temporal redundancy. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. 125–134. https://doi.org/10.1145/2038698.2038719

[15] M. Namjoo and E. HcCluskey. 1995. WATCHDOG PROCESSORS AND CAPABILITY CHECKING. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. IEEE Computer Society, Los Alamitos, CA, USA, 94.

[16] A. Narsale and M. C. Huang. 2009. Variation-tolerant hierarchical voltage monitoring circuit for soft error detection. In *Proceedings of the 2009 10th International Symposium on Quality Electronic Design*. 799–805. https://doi.org/10.1109/ISQED.2009.4810395

[17] R. Nathan and D. J. Sorin. 2014. Nostradamus: Low-cost hardware-only error detection for processor cores. In *Proceedings of the 2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6. https://doi.org/10.7873/DATE.2014.173

[18] G. L. Nazar, P. Rech, C. Frost, and L. Carro. 2013. Radiation and Fault Injection Testing of a Fine-Grained Error Detection Technique for FPGAs. *IEEE Transactions on Nuclear Science* 60, 4 (Aug 2013), 2742–2749. https://doi.org/10.1109/TNS.2013.2261319

[19] M. Nicolaidis. 2015. Double-Sampling Design Paradigm- A Compendium of Architectures. *IEEE Transactions on Device and Materials Reliability* 15, 1 (March 2015), 10–23. https://doi.org/10.1109/TDMR.2014.2388358

[20] P. Nikolaou, Y. Sazeides, L. Ndreu, E.Ozer, and S. Idgunji. 2013. Memory array protection: Check on read or Check on Write?. In *Proceedings of the 2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 214–219. https://doi.org/10.7873/DATE.2013.057

[21] S. Nomura, M. D. Sinclair, Chen-Han Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. 2011. Sampling + DMR: Practical and Low-overhead Permanent Fault Detection. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 201–212. https://doi.org/10.1145/2000064.2000089

[22] J. H. Patel and L. Y. Fung. 1982. Concurrent Error Detection in ALU's by Recomputing with Shifted Operands. *IEEE Trans. Comput.* 31, 7 (July 1982), 589–595. https://doi.org/10.1109/TC.1982.1676055

[23] P. Prata and J. G. Silva. 1999. Algorithm based fault tolerance versus result-checking for matrix computations. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*. 4–11. https://doi.org/10.1109/FTCS.1999.781028

[24] B. Pratt, M. Fuller, and M. Wirthlin. 2011. Reduced-precision redundancy on FPGAs. *International Journal of Reconfigurable Computing* 2011 (2011), 12. https://doi.org/10.1155/2011/897189

[25] R. Psiakis, A. Kritikakou, and O. Sentieys. 2017. Run-time Instruction Replication for permanent and soft error mitigation in VLIW processors. In *Proceedings of the 2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)*. 321–324. https://doi.org/10.1109/NEWCAS.2017.8010170

[26] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. 2007. Perturbation-based Fault Screening. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 169–180. https://doi.org/10.1109/HPCA.2007.346195

[27] R. G. Ragel and S. Parameswaran. 2006. IMPRES: integrated monitoring for processor reliability and security. In *Proceedings of the 2006 43rd ACM/IEEE Design Automation Conference*. 502–505. https://doi.org/10.1145/1146909.1147041

[28] A. Rajabzadeh and S. G. Miremadi. 2005. A hardware approach to concurrent error detection capability enhancement in COTS processors. In *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing (PRDC'05)*. 8 pp.–. https://doi.org/10.1109/PRDC.2005.7

[29] E. Rotenberg. 1999. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*. 84–91. https://doi.org/10.1109/FTCS.1999.781037

[30] A. Sari and M. Psarakis. 2011. Scrubbing-based SEU mitigation approach for Systems-on-Programmable-Chips. In *Proceedings of the 2011 International Conference on Field-Programmable Technology*. 1–8. https://doi.org/10.1109/FPT.

2011.6132703

[31] A. Sari, M. Psarakis, and D. Gizopoulos. 2013. Combining checkpointing and scrubbing in FPGA-based real-time systems. In *Proceedings of the 2013 IEEE 31st VLSI Test Symposium (VTS)*. 1–6. https://doi.org/10.1109/VTS.2013.6548910

[32] A. L. Sartor, A. F. Lorenzon, L. Carro, F. Kastensmidt, S. Wong, and A. C. S. Beck. 2015. A Novel Phase-Based Low Overhead Fault Tolerance Approach for VLIW Processors. In *Proceedings of the 2015 IEEE Computer Society Annual Symposium on VLSI*. 485–490. https://doi.org/10.1109/ISVLSI.2015.19

[33] A. L. Sartor, S. Wong, and A. C. S. Beck. 2016. Adaptive ILP control to increase fault tolerance for VLIW processors. In *Proceedings of the 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 9–16. https://doi.org/10.1109/ASAP.2016.7760767

[34] S. Valadimas, A. Floros, Y. Tsiatouhas, A. Arapoyanni, and X. Kavousianos. 2014. The Time Dilation Technique for Timing Error Tolerance. *IEEE Trans. Comput.* 63, 5 (May 2014), 1277–1286. https://doi.org/10.1109/TC.2012.289

[35] N. J. Wang and S. J. Patel. 2006. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (July 2006), 188–201. https://doi.org/10.1109/TDSC.2006.40

[36] Q. Zhao, Y. Ichinomiya, M. Amagasaki, M. Iida, and T. Sueyoshi. 2011. A Novel Soft Error Detection and Correction Circuit for Embedded Reconfigurable Systems. *IEEE Embedded Systems Letters* 3, 3 (Sept 2011), 89–92. https://doi.org/10.1109/LES.2011.2167213

Received September 2018