Classification of Resilience Techniques Against Functional Errors at Higher Abstraction Layers of Digital Systems

GEORGIA PSYCHOU[1] , DIMITRIOS RODOPOULOS[2], MOHAMED M. SABRY[3],
TOBIAS GEMMEKE[4], DAVID ATIENZA[3], TOBIAS G. NOLL[1], FRANCKY CATTHOOR[5],
[1]EECS, RWTH Aachen, [2]IMEC, [3]ESL, EPFL, [4]IDS, RWTH Aachen; formerly Holst Center/IMEC,
[5]IMEC & KU Leuven

Nano-scale technology nodes bring reliability concerns back to the center stage of digital system design. A systematic classification of approaches that increase system resilience in presence of functional hardware-induced errors is presented, dealing with higher system abstractions: i.e. the (micro-) architecture, the mapping and platform software. The field is surveyed in a systematic way based on non-overlapping categories, which add insight into the ongoing work by exposing similarities and differences. Hardware and software solutions are discussed in a similar fashion, so that interrelationships become apparent. The presented categories are illustrated by representative literature examples to illustrate their properties. Moreover, it is demonstrated how hybrid schemes can be decomposed into their primitive components.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*fault tolerance*; *reliability, availability, and serviceability*; B.8.1 [**Hardware**]: Performance and Reliability—*reliability, testing, and fault tolerance*

General Terms: Reliability, Design

Additional Key Words and Phrases: Resilience, Reliability, Mitigation, Fault Tolerance

1. INTRODUCTION

The early concerns of John von Neumann [Von Neumann 1956] regarding building reliable computing entities out of unreliable components were largely forgotten with the gradual replacement of vacuum tubes by transistors and the following high-scale transistor integration [Palem and Lingamneni 2012]. Now, after some decades, reliability has come back to the forefront in the context of modern CMOS technology. The

---

current reliability concerns originate from mechanisms, that manifest both during the manufacturing process and during the system's operational lifetime. Inherent time-zero and time-dependent device variability, noise (e.g. supply voltage fluctuations) and particle strikes are some of the most prevalent causes of such concerns [Borkar 2005], [McPherson 2006], [Kuhn et al. 2011], [Aitken et al. 2013]. The anomalous physical conditions that are created from those effects, are called *faults*. Depending on various conditions, faults can manifest as bit-level corruptions in the internal state or at the outputs of a digital system. The term *functional errors* is used to capture this class of errors, with the worst case manifestation toward the end user being a complete failure on the expected system service.

The manifested errors can be temporary or permanent [Bondavalli et al. 2000], [Borkar 2005]. Temporary errors include transient and intermittent errors. Transient errors are non-deterministic (concerning time and location), e.g. as a result of a fault due to a particle strike. Intermittent errors occur repeatedly but non-deterministically in time at the same location and last for one cycle or even for a long (but finite) period of time. Main causes for intermittent errors are design weaknesses, aging and wear-out, like Bias Temperature Instability (BTI), Hot Carrier Injection (HCI) etc. In contrast, permanent errors after their first occurrence persist forever. Causes for permanent errors are fabrication defects and aging.

The current work presents a classification scheme for organizing the research domain on mitigation of functional errors at the higher abstraction layers that manifest themselves during the operational lifetime, and discusses representative work for each category. Given the multitude of reliability issues in modern digital systems, it is vital to set the boundaries of the current survey: This survey discusses resilience schemes at the architectural/microarchitectural layer and platform software, which have increased in diversity during the last decades, following the evolution of computer architecture, parallel processing, software stack and general system design. Techniques at application, circuit and device layers, can potentially act complementary to the techniques presented here, but are not part of the current scope. Reliability-related errors that occur due to hardware-design errors, *insufficiently specified systems* or *malicious attacks* [Avizienis et al. 2004] or erroneous software interaction (i.e. manifestation of software bugs due to software of reduced quality [Lochmann and Goeb 2011]) are beyond the current scope. Techniques to mitigate permanent errors that have been detected during testing in order to improve yield or lifetime are not included. Techniques to tackle permanent errors due to device and wire wear-out are incorporated though. The main contributions of this work are:

 (i) An integrated overview of the domain of functional reliability techniques (at the higher system level stack) is presented, using a systematic, hierarchical top-down splitting into sub-classes.
 (ii) Multiple representative prior and state-of-the-art publications are mapped to these categories to illustrate the concepts involved.
(iii) Hardware and software solutions are discussed using a similar reasoning, to allow interrelations to become more visible.
(iv) The complementary nature of the splits allows hybrid schemes to be effectively decomposed and better understood. That is especially important in the era of growing cross-layer resilience design.

The current paper is organized as follows: Section 2 presents terminology regarding reliable system design, the abstraction layers that are addressed in this work and information on the rationale of the proposed classification. The classification along with the presentation of published literature begins in Section 3 for techniques that operate at the (micro-) architectural layers of the system and continues in Section 4

with techniques at the mapping and software part of the platform. Section 5 illustrates ways of using the proposed framework and Section 6 discusses observations and trends in the domain. Finally, related work is presented in Section 7 and Section 8 concludes the paper. Moreover, from this point on, the symbol ⓢ will be used to refer the reader to the supplementary material (see ACMCSUR website) for additional information.

2. CONTEXT AND USEFUL TERMINOLOGY

2.1. Resilient Digital System Design

This survey presents an organization of techniques that can be used to make a digital system more reliable at functional level. **Reliability** is defined as the probability that over a specific period the system will satisfy its **specification**, i.e. the total set of requirements to be satisfied by the system. **Functional reliability** is defined as the probability that over a specific period of time the system will fulfill its **functionality**, i.e. the set of functions that the system should perform [IEEE_Std 1990]. Functional reliability is related with correcting binary digits as opposed to parametric reliability that deals with aspects of variations in operation margins [Rodopoulos et al. 2015]. Functionality is one of the major elements of the specification set. Others may be minimum performance (e.g. throughput [ops/s], computational power [MIPS]), maximum costs (e.g. silicon area [mm^2], power [W], energy [J/op], latency [s/op]). In the following, the term reliability will be used to denote the functional reliability. The term **resilience** describes the ability of a system to defer or avoid (functional) system failures in the presence of errors. When a system becomes more resilient, its reliability is increased. The terms reliable and resilient (system design) will be used interchangeably in this paper ⓢ.

2.2. Computing Terminology

2.2.1. Terminology on Abstraction Layers.

This survey includes techniques implemented at the microarchitecture and architecture layers, as well as at the mapping & software (SW) of the system, as shown in Figure 1. The device, circuit and application layers are not considered. In this survey, the term **platform** denotes a system composed of architectural and microarchitectural components together with the software required to run applications. When the system is not SW-programmable, like some small embedded systems are, the term platform denotes only the hardware part.

Platform HW. **Microarchitecture** describes how the HW constituent parts are connected and inter-operate to implement the operations that the HW supports. It includes the memory system, the memory interconnect and the internals of processors [Hennessy and Patterson 2011]. This applies both to very flexible SW-programmable processors, where an instruction-set is present to control the operation sequence, and to dedicated HW processing components. Dedicated HW processors feature minimum to limited flexibility. Both SW-programmable and dedicated components can be mapped on highly reconfigurable fabrics, like field-programmable gate arrays (FPGAs). The primary difference compared with the SW-programmable processors is that not only the control flow but also the data flow can be substantially changed/reconfigured. The microarchitecture together with the Instruction Set Architecture (ISA) constitute the computer **architecture** (although the term has been recently used to include also other aspects of the design [Hennessy and Patterson 2011]).

In general, the term **HW module** denotes a subset of the digital system's HW, the internals of which cannot be observed (or it is chosen that they are not observed), correspondingly to the term *black box* [Rodopoulos et al. 2015]. To define a HW module, its functionality and its interface with the external world must be described. At the

microarchitectural and architectural layer, examples of HW modules are a multiprocessor system, a single core, a functional unit, the row of a memory array, a pipeline stage, a register (without exposing the internal circuit implementation though). In the context of this survey, the term **platform HW** is an umbrella term, that encompasses the microarchitecture and architecture layers of a system.

Mapping. During **mapping**, the algorithmic level specification is mapped into a pre-selected datapath and control path that implements the required behaviour Ⓢ. Nowadays, the term is also used to denote how an application or an application set is split, distributed and ordered in order to run in a multiprocessor design.

Platform SW. In order to enable software-hardware interaction, an instruction set is selected initially. The instruction set defines the hardware-software interface [Hennessy and Patterson 2011]. Many application instances sharing specific characteristics (a "domain") can be mapped on the same instruction set. Each of the instructions in that set can then be implemented in the hardware in different ways.

Platform SW includes several sublayers that interpret or translate high level operations (derived from the algorithmic description) into "primitive" instructions, which correspond to the instruction set and are ready to be executed by the hardware. Examples include: system libraries, operating systems and run-time managers Ⓢ.

Fig. 1: Scope of the current paper

2.2.2. Additional Terminology.

A **Control Data Flow Graph (CDFG)** is a graph representing all possible paths the flow of data can follow during execution. An application corresponds to a separate CDFG in the system. **A process** is an instantiation of a program, or a segment of code, under execution consisting of "own" memory space, containing an image of the executable code and data, resource descriptions, security attributes, and state information (register content, physical memory addressing etc.), i.e. all the information necessary to execute the program Ⓢ. **Threads** are sequences of instructions, or a flow of control, in a program which can be executed concurrently. All threads in a given process share the private address space of that process Ⓢ. The term **task** is used quite ambiguously in the literature: On the one hand, the terms task and process are used synonymously. On the other hand, the terms process and thread are considered as "mechanic" while the term task is considered as being more conceptual and used in the context of scheduling as a set of program instructions loaded in memory for execution. The term task in this paper is used as an umbrella term, which can denote
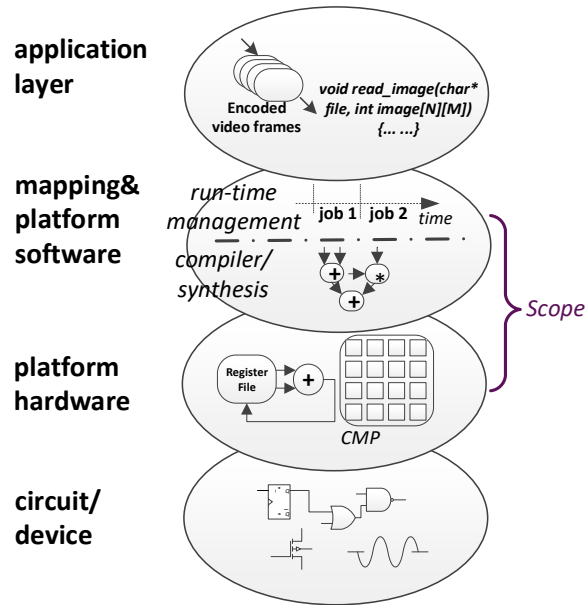
complete applications, sub-parts of the CDFG like processing kernels (e.g. for-loops) or even single computations (e.g. instructions) depending on the context.

2.3. Rationale of the classification and its presentation

The proposed classification tree is organized using a top-down splitting of the types of techniques that increase the system resilience. It is accompanied by a mapping of related work (see Figure 2). The top-down splitting allows to reach a comprehensive list of types of techniques, which can always be expanded further on demand. Splits are created based on properties of the techniques, which allow them to be grouped together. More specifically, the properties in the proposed framework regard: (1) the effect that the techniques have on the execution and (2) the changes that are required on the system design for a technique to be implemented. The properties will be elaborated as the tree is being presented.
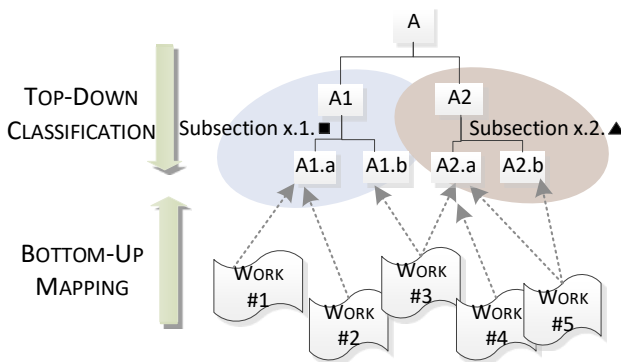
Fig. 2: Top down splitting to create the classification tree and mapping of the related work

Other organizations are also possible, like organizing the splits around the system functionality, hardware components, types of errors (transient, intermittent, permanent), types of resilience metrics or the application domains. The aforementioned organization is chosen in order to stress the reusability of techniques but also to enable the better understanding of hybrid combinations. This is especially supported through the complementarity of the categories. It is important to note that many actual approaches that increase resilience typically represent hybrids and do not fall strictly into only one of the categories.

For the presentation of the classification tree, the following structure is followed for each of the abstraction layers (platform hardware, mapping and platform software). First, the main classes are presented for the different techniques. Within each class, subcategories are presented which are illustrated with the help of a figure. Groups of nodes are chosen to be discussed together. For the visualization of the groups, bubbles with different colors are used, along with the subsection number and a small geometrical shape (see Figure 2). The colors and the geometrical shapes are used to enable a more explicit link with the corresponding subsections in the text. Especially the geometrical shapes are used for the facilitation of the reader in the black-white printed version. The order of the leaves, the colors and the geometrical shapes do not indicate the significance or the maturity of the techniques. For each of the classes, pros and cons are discussed, based on general properties bound to each class. Among the aspects considered are: area and power overhead, performance degradation (in terms of additional execution cycles), mitigation latency (delay until the scheme fulfils the intended mitigation function), error protection, general applicability, storage overhead. An overview of those for the different classes can be found in Tables II-VII in the Appendix (see supplementary material). In parallel, representative related work is discussed to further illustrate the subcategory concept and demonstrate the usefulness of the proposed classification scheme for classifying existing (and future) literature [S]. Moreover, in Ta-

bles II-VII in the Appendix, a crude indication of the amount of literature for each of the classes is performed.

Finally, the notion of **non-determinism** is introduced and will be discussed in the paper whenever appropriate. A common technique to mask the effect of errors is by employing replication. During replication, an algorithmic function is executed again, often by using extra hardware or software. However, deterministic execution is required for replicas to work. Determinism ensures that different runs of the same function under the same input will produce identical outcomes. In practice, deterministic execution is challenged by a multitude of non-deterministic events [Poledna 1996], [Slember and Narasimhan 2006], [Poledna 2007]. Examples include non-predictable user or sensor inputs, timers, random numbers, system calls and interrupts Ⓢ.

3. PLATFORM HARDWARE

To make digital systems more robust, functional capabilities need to be provided that would be unnecessary in a fault-free environment. This section focuses on techniques that modify the hardware capabilities for reliability purposes. The goal is to provide non-overlapping categories that cover the broad range of error mitigation and resilience techniques. The complete classification scheme is shown in Figure 12 in Subsection 3.5. A high level split for the proposed classification tree is shown in Figure 3.

Techniques are first classified into techniques that continue the execution forward
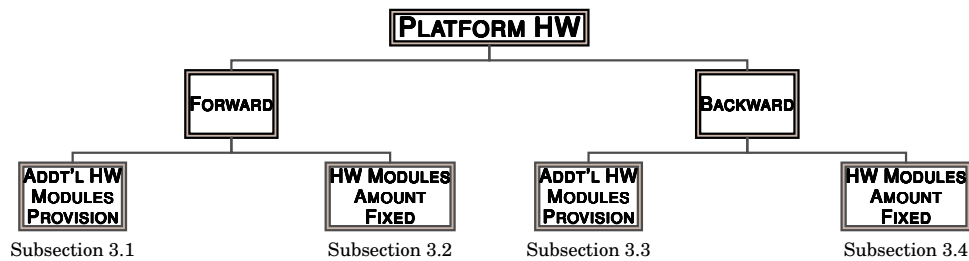
Fig. 3: Basic classification[1] for techniques at the platform HW

(**forward**) and those that move the execution to an earlier point (**backward**). Both categories are further split into techniques that require the addition of HW modules in the platform at design time (**additional HW modules provision**) and techniques that keep the amount of modules the same (**HW modules amount fixed**). In the latter case, only a HW or SW controller may be needed. These four classes are discussed in the following subsections, as shown in Figure 3. Main criteria for further categorization include whether modifications are required in: existing functionalities, existing design implementations, resource allocation, operating conditions, the interaction with neighbouring modules, storage overhead. Leaves of the tree have an accompanying simple ordinal number for identification. The numbers (together with the leaves) are collectively shown in Figure 12.

3.1. Forward execution - Additional HW modules provision

This subsection discusses techniques that increase the resilience through adding HW modules on the platform. The added modules may have either the same (**same**

---

[1]The boxes in the classification figures include hyperlinks to the text. By clicking on each of the boxes, the reader will be transferred to the corresponding section in the text.

functionality) or different (**different functionality**) functionality. The structure of this subtree along with the corresponding subsections is illustrated in Figure 4.

3.1.1.   Same functionality ■ . This group includes techniques that add hardware modules of the same functionality as the one(s) that should be protected. Some of the most known and well-established fault tolerant techniques are found in this category. The **provision of additional HW modules** can be further categorized into modules that are used in parallel execution mode and modules used as spares. **Parallel execution** denotes that the modules are all active and processing operations (or hold/transfer data and instructions for processing). The term **spares** denotes that

Fig. 4: Classification for forward techniques that require additional HW modules

the added modules are not all executing in parallel with the default ones. They will only start executing upon certain conditions.

Parallel execution ①[2]. In general, parallel execution implies that the modules are all actively used for the intended functionality, or at least potentially when the workload is very high ⑤. The term *lockstep* denotes a mode of operation, according to which, HW modules execute the same operations regarding the same program at the same time. Generally, lockstep processing can be "tight" or "loose" depending on whether the outputs of the modules are synchronized at the operation level or only selectively, for example at the I/O level [Aggarwal 2008]. Lockstep processing is used to make a system more robust either by masking an error, i.e. by allowing the correct output to be produced independent of which module caused the error, or by using explicit knowledge of the faulty module.

In the first case, multiple modules (*N* modules in the general case) with the same specification as the primary module are provided and majority voting is applied at their output. No error detection is required as the **error is masked** through the voting. This results in a well-known technique called *N-modular redundancy* or *NMR*. Typically *N* is an odd number to avoid uncertain output votes. Most often, the scheme has been employed in the form of *triple modular redundancy (TMR)* so that a correct output is produced with a two out of three vote.

Lockstep processing can be combined also with system **awareness of the faulty module**. In this case, a separate detection scheme is employed for the identification of the faulty module. Majority voting is not required, as after the detection, the faulty module is considered *not valid* any more. Only the output of the other module(s) is considered valid. So in this case, only two modules operating in lockstep suffice for producing a correct output.[3] One technique commonly found in literature, belonging to
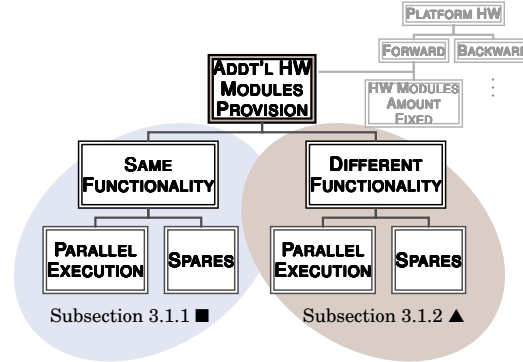
---

[2]The circled numbers refer to the corresponding leaves in the overall classification tree (in this case Figure 12). By clicking on these numbers the reader will be transferred to this figure.

[3]However, detection of the faulty module can also be employed in *NMR* schemes [Siewiorek and Swarz 1982]. Even though the output would be correct also without it, this knowledge can be used in order to have faulty module replaced.

this category, is the so-called **_pair-and-spare_**[4] technique. In pair-and-spare, two pairs of replicas operate in lockstep, as illustrated in Figure 5. Within each pair, error detection is performed through a comparison circuit. In presence of an error, the faulty pair declares itself as faulty. Then, the output of the other pair is selected as the valid one ⓢ.
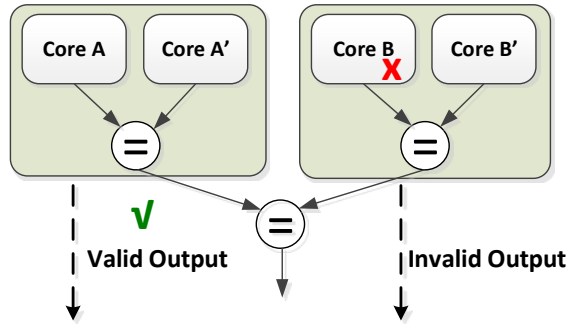
Fig. 5: Lockstep execution in a _pair-and-spare_ structure

Replica **determinism** is not an issue here as the processors perform their operations simultaneously and they operate on identical inputs [Poledna 1996]. **Pros** in this class include the high error protection, the lack of latency and performance overhead and the general applicability. **Cons** include the very high area (e.g. 200% for TMR) and power overhead. Literature **examples** on the aforementioned concepts include: [Dickinson et al. 1964], [Jewett 1991], [May et al. 2008] on _TMR_, Stratus computers and the VAXft 3000 minicomputer [Siewiorek 1990] on _pair and spare_ ⓢ.

Spares ②. In this category, the added modules, which deliver the same functionality as the original ones, act as spares. The role of spare modules can be potentially dual: The first use of spares is to remain in standby mode and **take over execution when the primary module fails** ⓢ. The second use of spares is to take over execution (or be included in the system operation) for part of the time, without the primary module experiencing some failure. That means that the **execution can potentially alternate between the spare and the primary** module. Several reasons can motivate the undertaking of such a scheme. One possibility is related to the benefits coming from sharing the workload (in time). For example, it is known [He et al. 2011] that the device stress, which contributes to the system aging, is increased when there is a full workload operation compared to when there is alteration of active and inactive periods. Through alternating the execution between a primary and a spare the lifetime of the system could be expanded. Another possibility is that the modules (original and spare) have partially different internal implementation, which gives them characteristics that fit better for certain conditions. In this case, the execution may alternate depending on the changing application requirements, for example, due to changes in the input workload or in environmental parameters (e.g. noise or temperature) ⓢ. **Pros** include the high error protection and lack of performance overhead. **Cons** include the area overhead. The power overhead can be avoided depending on whether the spares are powered or not and this is a trade-off with latency (see supplementary material). The approach is generally applicable, except if spares are tailored to fit changing application requirements. Literature **examples** on the aforementioned concepts include: [Chean and Fortes 1990], [Srinivasan et al. 2005] on spares with failing modules, [Shin et al. 2008], [Narayanan et al. 2010] on spares with working modules ⓢ.

3.1.2.   _Different functionality_ ▲ . This group includes techniques that add hardware modules of different functionality than the one(s) that should be protected or become

---

[4]The part "spare" of the term is misleading as in fact all the modules involved operate in lockstep.

Fig. 6: Read out (7,4) Hamming codeword and syndrome generation for zero and one error with correction

more robust. Again, a distinction can be made between modules that are in **parallel execution** mode and modules that act as **spares**.

Parallel execution ③. Here, the added module performs different functions than the original module. Several possibilities exist: A category includes **hybrid** schemes, according to which, the added modules that are **designed to be more robust** (by employing for example circuit-level techniques). The added module can perform only a subset of the operations of the original module for verification purposes, i.e. it is a module with **reduced functionality**. For example, the most crucial operations or the ones that cannot be performed (repeated) by any other of the already existing modules on the platform, maybe be performed by the added module. Since it is designed to be more robust, its output is assumed as the correct one. Another possibility is that the added module performs a super-set of the operations of the original module, namely it is a module of **increased functionality**. That means that it performs the operations of the original module plus additional operations, which are normally performed by other modules on the platform. That would be the case when the added module would act like a supervisor for several modules. An additional possibility is that the added module performs **different types of functions**. For example, it may perform some error correction. Given that the HW module granularity can go down to a register, the error correction codes (ECC) are placed in this category. They are typically implemented in memory structures, but also in buses, state machines and arithmetic units. Figure 6 shows an example of a single bit correction with the Hamming code [Hamming 1950]. Syndrome bits are created during the read operation. If a single error occurs, the syndrome identifies the erroneous bit. **Pros** include the flexibility to trade-off area, power, performance overhead and latency with the error protection by selecting a fitting functionality to be added. **Cons** include that this class generally requires system-specific solutions (although for ECC reusable concepts are typically applied). Literature **examples** on the aforementioned concepts include: the Algorithmic Noise Tolerance (ANT) [Hegde and Shanbhag 2001] on modules with reduced functionality, [Hamming 1950], [Dutt et al. 2014] on ECC ⑤.

Spares ④. As already discussed, spare modules can be present in order to take over execution in case the primary module fails or to take over execution for part of the time, even if no failure is present. A *reduced functionality* spare module is able to continue execution at a reduced power and area overhead but also at a degraded performance (since only part of the functionality is available). An *increased functionality* spare module is able to continue execution in an environment that the primary module has been shown to be not good enough. By using its additional functionality, it keeps or improves the reliability target (at extra area and power cost) ⑤. Similarly to the earlier

category, **pros** include the flexibility to trade-off area, power, latency and performance with error protection by selecting the appropriate solution. **Cons**, generally in this class, include that system-specific solutions are required. In the literature, techniques that employ spares with reduced functionality have been identified. **Examples** can be found in [Tomayko 1986] ⓢ.

3.2. Forward execution - HW modules amount fixed

This subsection discusses techniques that use only the same amount of modules on the platform as the original system (before reliability related countermeasures are added). Hardware modifications (like adding interconnects) may be required but no additional HW module is added. A HW or SW controller is often needed to coordinate the actions. These techniques are split into techniques that reuse the existing HW modules (**existing HW modules**) and those that replace one (or more) module with an alternate in order to make the system more robust (**alternate HW modules**). The first category is further split into techniques that either change the way of operation of the HW modules (**HW modules operation mode**) or leave the operation unaltered and change the way the work-

Fig. 7: Classification for forward techniques that keep the amount of HW modules fixed

load is mapped on these HW modules (**resource allocation**). Changing the operation of the HW modules means that the changes have as focus either the functionality (**functionality control**) or the operating conditions (**operating conditions control**). Functionality-oriented modifications either focus on the internals of a HW module so that the intended module usage is exploited for reliability purposes (**internal functionality reuse**) or on the input-output behavior of the module and how it interacts with the other modules (**I/O configuration modification**). Figure 7 shows the proposed subtree and its division into subsections.
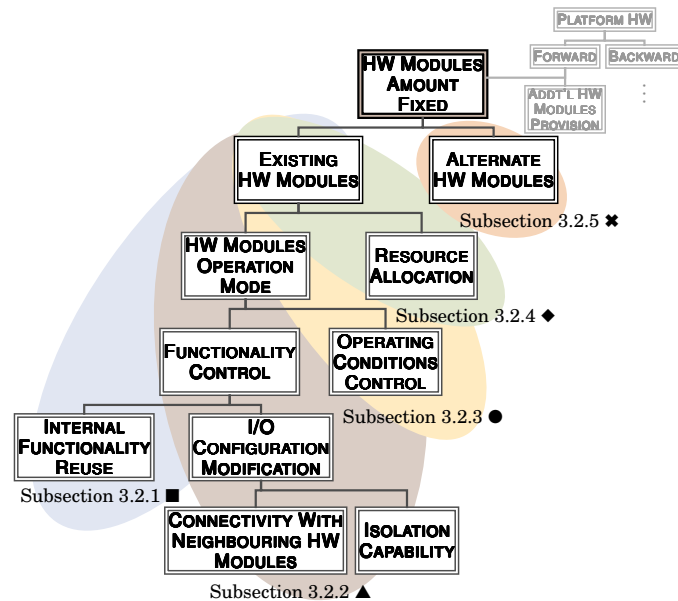
3.2.1.   Internal functionality reuse ⑤ ■ . Techniques belonging to this category are very **system/application dependent**. For example, communication or signal processing systems typically have blocks that perform channel or source coding. Channel decoders mitigate errors introduced by the channel and can be potentially reused in order to mitigate hardware-induced errors. **Pros** include the lowest possible area and power overhead due to the reuse. **Cons** include the lack of general applicability, latency, possible performance costs and the limited error protection. Literature **examples** that reuse the channel decoder include: [Khajeh et al. 2012], [Brehm et al. 2012] ⓢ.

3.2.2. I/O configuration modification ▲ . This group of techniques re-organizes the interaction of a module with the other modules. This can potentially mean a different way of connecting or communicating (**connectivity with neighbouring HW modules**) or even an isolation action (**isolation capability**), during which, an erroneous module is bypassed from the system.

Connectivity with neighbouring HW modules ⑥. Inter-module techniques can exploit **inherent redundancy** typically present in regularly structured systems, like arrays of processing elements (PEs), to increase the masking and correction capability of the system. Nowadays, high-performance is achieved primarily by chip multiprocessors (CMPs). CMPs are composed of multiple cores located in a single die or on multiple dies in a single package. The types of cores may vary: from simple, in-order processors up to more complex, superscalar ones. They enable high performance through parallel computation. The CMPs are used here as driver, but the ideas can be applied to other regularly structured systems, where reuse is possible. The availability of the cores can be exploited to create masking capability by, for example, running a process in three cores in parallel in a *TMR* structure. Or the hardware itself can be built as **reconfigurable**, so that, the modules can be connected in a different way depending on run-time conditions. Typically, this last possibility is found in the form of a hybrid; for example, it is often found together with spare modules. **Pros** include the low area and power overhead (due to the reuse of existing modules but with additional cross-links), the general applicability (for systems with inherent redundancy), the potentially high error protection. **Cons** include the latency and blockage of resources for reliability that could be used to improve performance. A literature **example** that employs a modified connection network in CMPs is found in [Aggarwal et al. 2007] ⓢ.

Isolation capability ⑦. To prevent erroneous results from corrupting the system output, faulty components can be **bypassed (through a switch) or powered off**, in case such an isolation capability has been added in the system. The system continues operating but at a degraded performance. These schemes exploit inherent redundancy in regularly structured systems such as arrays of PEs, memories and interconnection networks or even processors.[5] **Pros** include low area and power overhead, general applicability (for systems with inherent redundancy). **Cons** include latency, degraded performance, limited error protection. Literature **examples** of the concept include: [Srinivasan et al. 2005], [Bower et al. 2005] on structures within processors, [Gupta et al. 2008], [Romanescu and Sorin 2008] on pipeline stages in CMPs ⓢ.

3.2.3. Operating conditions control ⑧ ● . Operating conditions represent the interference caused to a digital system by its environment [Rodopoulos et al. 2015]. This covers a broad range of effects like radiation, temperature, humidity but also electrical stimuli. This category includes all actions that influence the operating conditions of the digital system, beyond changing the system's functionality.

Typically, **operating parameters, such as the supply voltage and the clock frequency**, are controlled to manage the performance, power and reliability tradeoffs. Scaling the voltage beyond a critical limit can lead to excessive error rates. On the other hand, using conservative guard bands for the voltage setting can lead to significant power overhead. **Pros** include lack of area overhead, general applicability (assuming that knobs are present in the system for power management). **Cons** include the latency and limited error protection. Power and/or performance will typically be

---

[5]Quite often, this isolation functionality is combined with techniques presented in *Additional HW modules provision/same functionality* or the previous category so that a different error-free module is used instead. In these cases, these are hybrid combinations. Note that techniques that employ additional modules that run in parallel or act as spares do not necessarily isolate the faulty component.

affected depending on the knob being used. Here, works that implement control algorithms that change operational parameters are classified, like the **examples** of [Karl et al. 2006], [Rosing et al. 2007] ⑤.

3.2.4.   *Resource allocation* ⑨ ♦ . Here, the way the hardware resources are assigned is modified without changing the way of operation of the HW modules. Simply the task is **migrated or swapped with another task**. **Pros** include the limited area, power, performance overhead due to the modules reuse (with the exception of adding specialized interconnects) and the rather general applicability (for systems with inherent redundancy). **Cons** include latency during migration and limited error protection. Literature **examples** on hardware-based task migration include: [Powell et al. 2009], [Venkataraman et al. 2015] ⑤.

3.2.5.   *Alternate HW modules* ⑩ ✖ . This category includes schemes that replace an existing HW module with another **more robust implementation** for the system context (without employing circuit or lower layer techniques). **Pros** include the limited area and power, performance overhead as the new implementation will typically satisfy the system requirements, while minimizing additional cost. **Cons** include that system-specific solutions are required (if existing at all) and typically only limited error protection will be possible. Literature **examples** in this category include: [Hussien et al. 2010], [Hussien et al. 2011] on alternate channel decoders ⑤.

3.3. Backward execution - Additional HW modules provision

This subsection discusses techniques that increase the resilience of systems through rollback to an earlier point of execution and repetition of the execution.

Fig. 8: Classification for backward techniques that require the provision of additional HW modules

Just like in the *forward* execution category, the added modules can have either the same (**same functionality**) or different (**different functionality**) functionality. The corresponding categories and subsections are shown in Figure 8.[6]
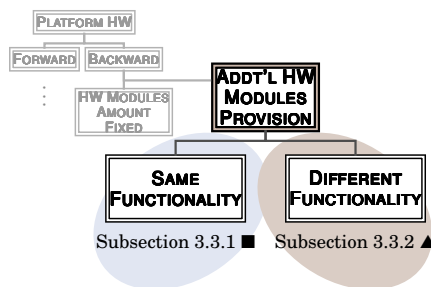
3.3.1.   *Same functionality* ⑪ ■ . This category discusses techniques that provide additional HW modules with the same functionality as the original ones. The recovery is achieved by **repeating part (or the whole) of the execution**, when an error is detected. In this category, the second module plays an active role in the recovery. For example, it can activate the execution repetition or provide necessary information to the first module so that the execution is repeated successfully. When the second module executes the same instruction sequence, **non-deterministic** execution is not a concern, as long as identical inputs can be provided to both modules. **Pros** include the potentially high error protection (at the expense then of performance and latency). Moreover, the

---

[6]Note that lower level splits like a split between modules that are in **parallel execution** mode and modules that act as **spares** are also possible (like in the *forward* category). Spare modules would, for example, not only take over the execution after the primary module has failed but also repeat the failed execution. However, hardware-based techniques that retry the execution using spare modules have been less explored in the literature. So, this split is left as implied for this tree.

technique is generally applicable. **Cons** include the high area and power overhead. A literature **example** in this category is [Pflanz and Vierhaus 2001] ⓢ.

3.3.2.   *Different functionality* (12) ▲ . Instead of adding modules with the same functionality, modules with different functionality can be added; the added modules play an active role in the recovery as in the previous category. The added modules can be with **reduced or increased functionality** as in the corresponding *forward* category for similar reasons. **Pros** include the flexibility to trade-off area, power, performance, latency with error protection depending on the selected functionality. **Cons** include that the solutions are rather system-specific. A literature **example** in this category is [Austin 1999] ⓢ.

3.4. Backward execution - HW modules amount fixed

The majority of the techniques proposed in the literature that employ backward execution, reuse the already existing HW modules as the additional area overhead of the previous category is avoided. This can be achieved by techniques that retry the execution without explicit storage (**retry without state storage**) and techniques that retry by storing some (redundant) system information at intermediate execution points to be used for system recovery (**retry with state storage**).[7] *Checkpointing* is a term that refers to the intermediate storing of the application's state (or of part of it), like register and memory contents. Additional events may be registered as part of the state, which are called *logs*. The corresponding categories and subsections are shown in Figure 9.

3.4.1.   *Retry without state storage* ■ . This category includes the schemes that move back the execution to an earlier point and repeat it, upon error detection. The execution can be successfully repeated without explicitly storing the system state either because the **state information is not really needed** or because it is **provided indirectly** by executing another task, which produces the required information. In the degenerate case, a hardware-driven restart/reboot procedure can be triggered to remedy transient errors. The techniques can be further distinguished into techniques that take place within the boundaries of a single module, i.e. **intra-module** and techniques that operate across modules, i.e. **inter-module**, as shown in Figure 9.
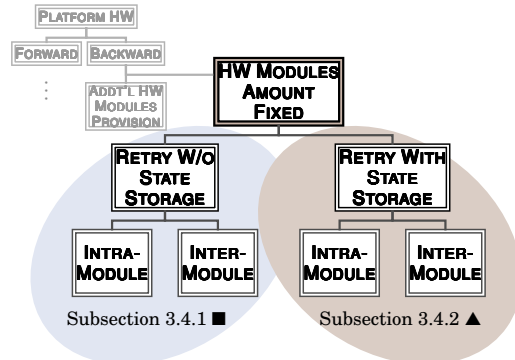

Fig. 9: Classification for backward techniques that reuse existing HW modules

Intra-module (13). In this category belong schemes that either exploit **inherent features** of processors to retry a task execution, like instruction retry or cache refetch, or employ additional hardware-based tasks.

For example, Ray et al. [Ray et al. 2001] propose to use the pre-existing instruction rewind mechanism present in superscalar machines for branch mispredictions in order to handle error recovery. After detecting an error (by duplicating the instruction during

---

[7]Note that the *Backward/additional HW modules provision* subtree could also be split further in similar categories, depending on whether intermediate state storage is involved. However, in that case, because the biggest overhead comes from additional HW modules, this split is left as an implied lower level split.

the decode stage and comparing the results before committing), the contents of the ReOrder Buffer (ROB) are flushed and the instruction is re-executed, similarly to what happens upon a branch misprediction event (see Figure 10a). In case the results agree after cross-checking, a single instruction retires and execution proceeds.

Hardware-based tasks in the literature are implemented by **simultaneous multithreading** to increase on-chip parallelism. Simultaneous multithreading (SMT) is a technique that allows multiple threads to issue multiple instructions each cycle on a superscalar processor [Tullsen et al. 1996]. The threads can be separate from each other or coupled to each other. Separate threads could be potentially created to execute the same program in a *TMR* structure, assuming that care is taken so that the threads use identical shared resources. The literature focuses on employing hardware-based threads in *coupled execution* mode. According to this mode of operation, the threads communicate with each other, i.e. one thread uses some knowledge from the other thread(s) in order to execute the program. Coupled execution has been used with processors in order to speed-up execution and the idea has been reused for fault tolerance [Sundaramoorthy et al. 2000]. The concept is as follows: Two streams of the same program run in parallel but with a time lag (see Figure 10b). The first stream is a less accurate one as it processes less instructions than a complete stream would. It bypasses certain computations and branch instructions as indicated by a hardware monitor which has observed past instances. Thus, it can run faster than a complete stream. Its results are stored in a delay buffer. The second stream is an accurate one, as it executes all the instructions. However, it receives information from the first stream through a delay buffer, which allows it to run also faster. For example, it uses memory load values and thus it can avoid memory latencies. When the second thread commits (writes its results to the registers), the results from both threads are compared. If they are not identical, the results of the second one are used to restore the system state.
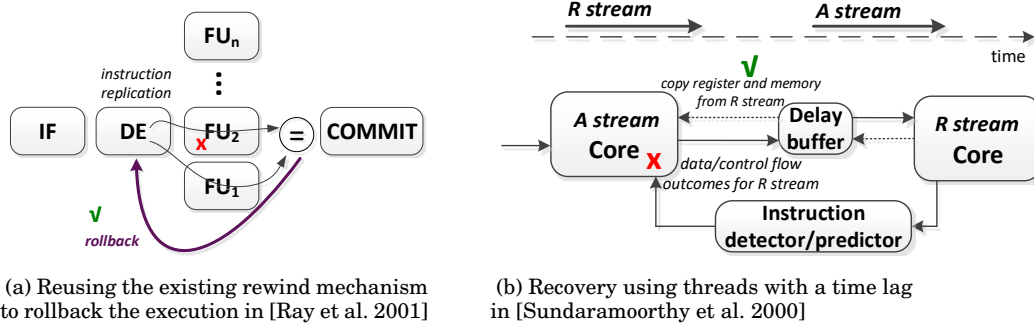

(a) Reusing the existing rewind mechanism to rollback the execution in [Ray et al. 2001]

(b) Recovery using threads with a time lag in [Sundaramoorthy et al. 2000]

Fig. 10: Illustration of concepts in the platform-HW *backward* category

Non-deterministic events like traps and exceptions are handled with some minimal support from the operating system. The first stream stalls until the delay buffer completely empties and the second stream is terminated. The first stream is serviced (by the operating system) and execution resumes. **Pros** include the low area and power overhead, potentially high error protection (but only for transient errors) and rather general applicability. **Cons** include the latency, performance overhead and the limitation to transient errors. Literature **examples** include: [Ray et al. 2001] on re-executing instructions, [Rotenberg 1999] on tasks with a time lag, [Cho et al. 2012] on restarting a core [S].

Inter-module (14). This category has similar properties with the earlier but requires the **cooperation** of modules. **Pros** and **cons** are similar with the previous category, but here also permanent errors can be handled and synchronization issues have to be addressed. In the literature, mainly **examples** that include tasks with time lag have been identified: [Sundaramoorthy et al. 2000], [Gomaa et al. 2003] [S].

3.4.2. *Retry with state storage* ▲ . This group of techniques employs the storage of a complete or partial error-free state and the rollback to that state upon detection of an error. Afterwards, the execution is repeated to acquire error-free results, assuming that the error was transient. The techniques are also distinguished into **intra-module** and **inter-module**. In the latter category issues that have to do with the state synchronization among several modules have to be addressed. Checkpointing/rollback refers to a widespread concept, according to which, the state of a process is proactively stored at certain intervals during execution so that the correct state is restored in case of an error. The majority of prior work realizes software-based checkpointing/rollback schemes. However, groups both in industry and academia have provided fully hardware-based implementations. Typically, when these schemes address **non-determinism**, this is done by synchronizing the checkpoints with the external events (e.g. interrupts). Namely, when an external event takes place, a checkpoint is forced.

Intra-module (15). This category includes schemes that **store the whole state or a subpart** of the state of a module in order to restart the execution from that stored point if an error occurs. The storage can take place in the main memory, hard disk, register file or memory buffers, and is often complemented by another error resilience technique, like ECC, in order to be more robust. A broad range of checkpointing techniques exist, from techniques that store checkpoints very rarely (every thousands up to billions of cycles) assuming low error rates up to techniques that perform checkpointing very often (every few cycles) assuming high error rates. **Pros** include the high error protection (for transient

Fig. 11: A local error can trigger all the CMP cores to roll-back in global checkpointing schemes

errors only), the general applicability. **Cons** include latency (depending on the checkpointing granularity), performance (depending also on whether checkpointing is overlapped with normal execution) and the limitation to transient errors. Area and power overhead is medium. Literature **examples** include: [Ahmed et al. 1990] on cache-based checkpoints, [Wang and Patel 2006], [Gupta et al. 2009], [Li et al. 2013b] on register-based checkpoints [S].

Inter-module (16). Such schemes are typically found in multicore architectures. Here, on top of the external **non-deterministic events**, like interrupts, also internal events have to be taken care of, like the accesses to the shared memory. These checkpointing schemes can be characterized as global and local. In the **global** schemes, common checkpoints are created among all modules and upon detection all modules have to roll back to an earlier state (even when many of them are error-free). Figure 11 illustrates the concept. A challenge with this approach is the scalability as the number of cores increases. On the other hand, **local** checkpointing schemes allow such actions
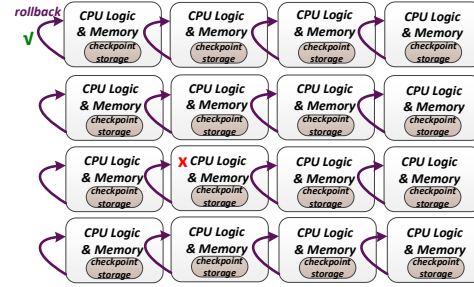
to be made by a subset of the modules, performing only local synchronization and information storage. A taxonomy of hardware-based checkpointing schemes for CMPs can be found in [Prvulovic et al. 2002]. **Pros** and **cons** are similar with the previous category, but with extra synchronization costs. Global schemes induce more overhead during checkpointing, but have a simpler recovery, compared to local schemes. Literature **examples** include: [Wu et al. 1990], [Agarwal et al. 2011] on local and [Sorin et al. 2002] on global schemes ⓢ.

3.5. Overall platform hardware classification

The sub-trees presented in the previous subsections are combined to form the overall classification tree for platform HW techniques, as shown in Figure 12. Starting from the top-level split of Figure 3, the intermediate nodes (colored by pale green) are followed when necessary, to reach the final classes (colored by darker green and numbered).
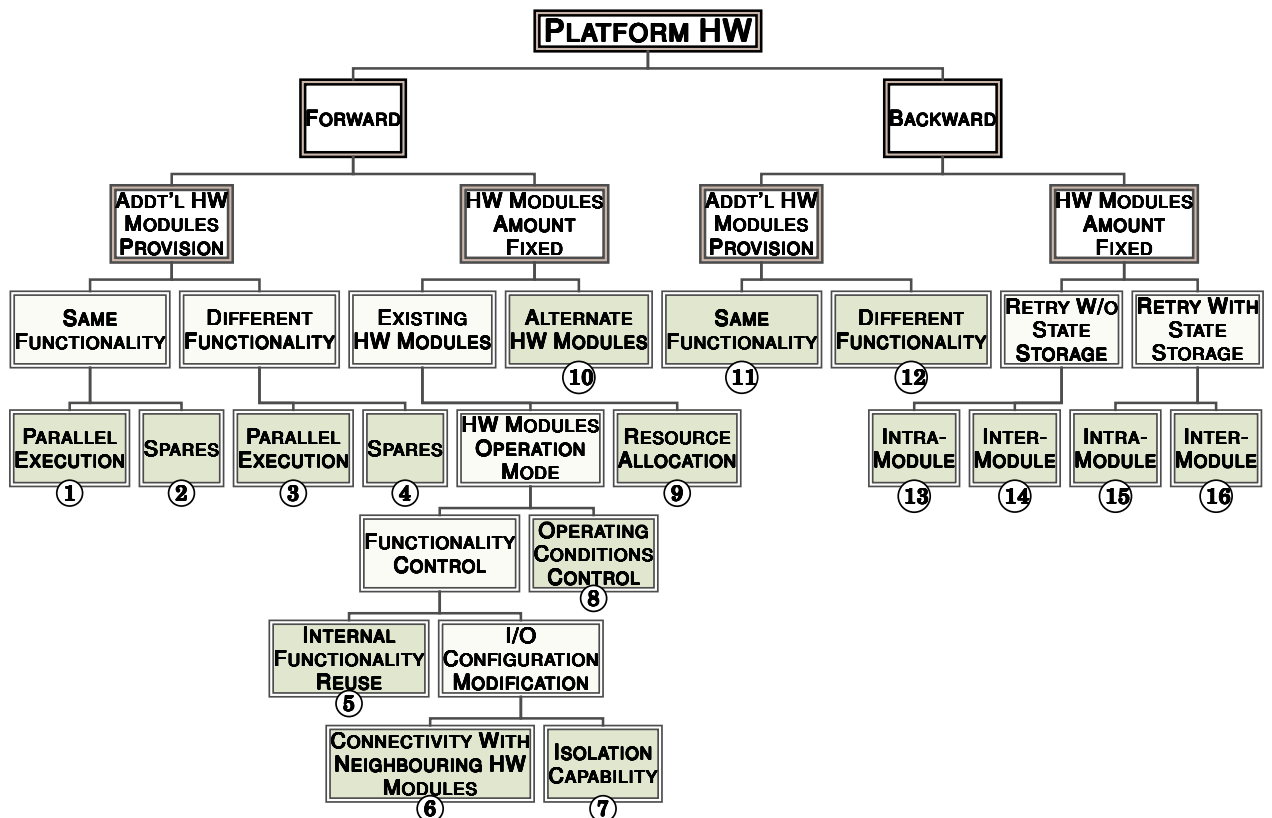

Fig. 12: Overall proposed classification for techniques at the platform HW

4. PLATFORM SOFTWARE

In this section, techniques that extend the platform software capabilities for reliability purposes are presented. The proposed classification is built around the notion of *tasks* in a similar way that the earlier section was built around the notion of *HW modules*. The complete classification scheme is shown in Figure 22 in Subsection 4.5. As in the platform hardware section, the first split in the platform software techniques
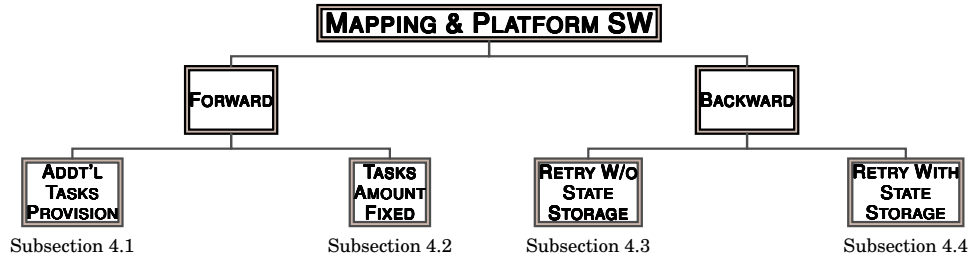
Fig. 13: Basic classification for techniques at the mapping and platform software

is between (**forward**) and (**backward**) techniques. *Forward* techniques are then further split into techniques that require additional tasks (**additional tasks provision**) and techniques that cope with the existing amount of tasks (**tasks amount fixed**). *Backward* techniques are split into techniques that require the re-execution of tasks without storing state information (**retry without state storage**) and techniques that do require such intermediate state storage (**retry with state storage**). Since re-execution of tasks is a prerequisite for a technique to be characterized as backward technique, backward techniques always include additional tasks (a re-execution can be seen as providing an additional identical task in time). These four classes are discussed in the following subsections, as shown in Figure 13 Ⓢ. Main criteria for further categorization into classes include whether modifications are required in: existing functionalities, existing task implementations, the resource allocation, the interaction with neighbouring tasks, execution mode (of additional tasks), cooperation among HW modules. Here, also, end nodes are accompanied by a ordinal number.

4.1. Forward execution - Additional tasks provision

This subsection discusses techniques that increase the resilience of systems through providing additional tasks using the platform's software without moving the execution to an earlier point. The added tasks may have either the (**same functionality**) or (**different functionality**). The structure of this subtree along with the corresponding subsections is illustrated in Figure 16. These techniques can be differentiated according to the granularity of the replicated task (instruction, thread, process) and/or according to the abstraction layer in the software stack (see Subsection 2.2.1) that the replication takes place.
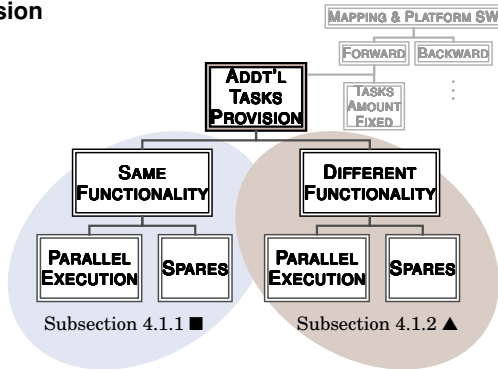

Fig. 14: Classification for forward techniques that require additional tasks

4.1.1. Same functionality ■ . As in the corresponding platform HW category, this category can be further split into tasks that are in **parallel execution** mode and tasks that act as **spares**.

Parallel execution ①. The parallel execution implies that the HW resources are available, like in the case of a multicore architecture. In case additional HW resources are provided but the tasks run at the higher software stack, then this category is a hybrid with the (*Additional HW modules provision/same functionality*) category. Here,

similar concepts can be applied as in the corresponding platform HW category but differences exist as well. Parallel tasks can run in lockstep. They may be configured in an execution mode, according to which, awareness of the faulty task is not required, since **the error is masked**; like in an NMR or TMR structure combined with voting. The alternative is that the **faulty module is identified** (some explicit detection/diagnosis scheme is present) and then only the outputs of the other running task(s) are considered valid. The concept of using multiple tasks in parallel and applying majority voting at their outputs has been presented in the literature since a long time. Execution among the redundant processes/threads must be **deterministic**. The literature has focused mostly on employing such schemes on multicore architectures. Thus, the first source of non-determinism that has to be tackled is the shared memory accesses.

When the parallel execution is combined with fault awareness, majority voting is not necessary. Two tasks are typically sufficient to have a robust execution. In the presence of non-deterministic events, like I/O operations, the literature deals with this category typically in the following manner: One of the two tasks (*primary*) executes I/O operations and the other task (*backup*) gets informed about the results. If the primary task is declared faulty, then the **backup takes the role of the primary** and continues operation, including I/O operations.

Pros include the high error protection (for the task that is protected), the lack of latency (except if there is lack of empty slots during scheduling), storage and power overhead (assuming the HW resources would be used anyway). Depending on the software stack level implementation, different degrees of transparency can be achieved. **Cons** include the blockage of resources for replicating functions leading indirectly to performance overhead. Literature **examples** include: [Avizienis 1985], [Fiala et al. 2012] on TMR at user-level, [Döbel et al. 2012] on TMR at OS-level, [Jeffery and Figueiredo 2012] on primary/backup set-up at VM level. For example, the authors in [Döbel et al. 2012] base their technique on a microkernel. A master process (see Figure 15) generates the replica threads and performs the output checks ⑤.

Spares ②. As already discussed in Subsection 3.1 in the HW category, a spare can have a dual role: to take over execution when the primary module fails or, potentially, to take over execution for part of the time so that the execution alternates between the primary and the spare, depending on the objectives. Having a task with the same functionality as a spare could make sense, for example, if, upon failure of the primary task, the spare task is loaded from a different instruction memory location (which may be considered more robust). The execution would continue forward (past errors would not be corrected) but the loading action could prevent to have future error manifestations due to corrupted instruction memory ⑤. **Pros** would include the limited storage, area, performance overhead, latency, high error protection (only for instruction memory errors) and general applicability. **Cons** would include the limitation to instruction memory errors. Literature schemes that use alternate tasks as spares in a forward mode have not be found.

4.1.2. Different functionality ▲ . In this case, the added task(s) deliver a different functionality than the one that should be made more robust. This category can also be distinguished into tasks in **parallel execution** mode and tasks that act as **spares**.

Parallel execution ③. In principle, similar types of techniques can be applied as in the corresponding platform HW category. The added task may perform a **subset or a super-set** of the functions of the original task but because of some complementary technique (i.e. it is a **hybrid**) it is assumed to be more robust. The complementary technique may be that the processing element on which the added task runs is configured to be more robust. Or the added task performs some **different function**, like error correction. It must be noted that parallel execution in this context, does not nec-

essarily imply that the additional task runs at the exact same moment as the original one, but it is **active in parallel** with the original task during the system lifetime. For example, it may be executed more sporadically, e.g. periodically. **Pros** include the flexibility to balance among storage, power, performance, latency and error protection through appropriately selecting the added function. **Cons** include the need for system-specific solutions and the blockage of resources. A literature **example** is [Shirvani et al. 2000] Ⓢ.

Spares ④. Tasks that have a different functionality can also be potentially used as spares. For example, the backup task may have some inherent error correction coding, which makes it more appropriate than the original task for given external conditions. **Pros** and **cons** are similar to the previous category; however, in this case the added task runs instead of the original task, leading to partially different costs depending on the exact implementation. Literature schemes that use spare tasks with different functionality in a forward execution mode have not been found.
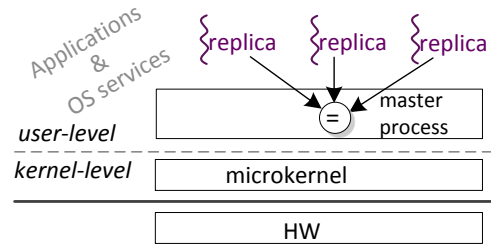

Fig. 15: Redundant multithreading at the OS level in [Döbel et al. 2012]

4.2. Forward execution - Tasks amount fixed

This subsection discusses techniques that do not provide additional tasks in the system in order to make it more reliable. Initially these techniques can be distinguished between techniques that make use of the existing tasks on the platform (**existing tasks**) and techniques that replace the implementation of existing tasks with an alternate implementation (**alternate tasks**). The techniques based on the *existing tasks* can be split into techniques that manipulate the functionality of the tasks (**functionality control**) and techniques that re-arrange the allocation of the tasks into the hardware resources (**resource allocation**). Figure 16 shows the proposed subtree and its division into subsections.

4.2.1. Functionality control ■ . Techniques that are focused around the functionality of tasks can either operate within the task boundaries, by reusing the task functionality (**internal functionality reuse**) or operate outside the task boundaries by rearranging its interaction with the other tasks (**I/O configuration modification**). The latter can be further split into techniques that reorganize the execution sequence (**scheduling-ordering**) and techniques that isolate results from corrupted tasks (**isolation capability**).

Internal functionality reuse ⑤. In this category some knowledge of the inherent task functionality is exploited in a useful way to increase resilience. By **configuring some parameters** through the platform SW, a more resilient execution is achieved. Although a literature example in the present context has not been found, such a scheme could be similar to the following example. In [Pant et al. 2012] the

authors consider a number of parameters for the H.264 encoding: sub-pixel motion estimation, size of motion estimation search window, DCT window size and run-length encoding mechanism. They show that adapting those parameters can utilize the application algorithm's quality or performance tradeoffs to achieve error free operation in presence of permanent manufacturing variations. **Pros** include the low storage, power overhead. **Cons** include the need for very system-specific solutions, the limited error protection. Performance and latency may be affected depending on the exact implementation.

I/O configuration modification/scheduling-ordering ⑥. These schemes reorganize the application or instruction profile so that the **re-ordered execution is more robust**. Typically, additional information is used regarding the vulnerability of certain instructions or operands (and thus registers). By re-scheduling the flow, the interval that the most vulnerable operations and operands are used is minimized. **Pros** include the very limited storage, power overhead and rather general applicability. **Cons** include the need for additional information to guide the re-scheduling, the limited error protection. Performance and latency may be affected. Literature **examples** at the instruction-level include: [Yan and Zhang 2005], [Rehman et al. 2012] ⑤.

Fig. 16: Classification for forward techniques that keep the amount of tasks fixed

I/O configuration modification/isolation capability ⑦. This class includes techniques which isolate tasks so that errors **do not propagate to subsequent tasks or the output**. To be more accurate, the focus is on the results of the tasks. This requires some application knowledge to ensure that the impact of the discarded computations is not destructive. **Pros** include the lack of storage, power overhead and latency. **Cons** include the need for system-specific solutions, the low error protection (through isolation), the potential performance degradation. A hybrid **example** is [de Kruijf et al. 2010] ⑤.

4.2.2. Resource allocation ⑧ ▲ . This category includes techniques that change the assignment of tasks and data to hardware components in a way that the most **reliable match between task and hardware module** is found. This can be in the context of a single processor or a multiprocessor system. To find the best match, information regarding the vulnerability (or robustness) of the tasks and of the HW modules has first to be identified. The category includes also the case where although the task was initially running in one core, is **migrated** to another compatible core (task migration). **Pros** include the very limited storage, power overhead and rather general applicability. **Cons** include the need for additional information to guide the re-allocation, the limited error protection. Performance and latency may be affected. Literature **examples** include: [Yan and Zhang 2005] on register allocation, [Rahimi et al. 2013], [Chakravorty et al. 2006] on task-processor mapping ⑤.
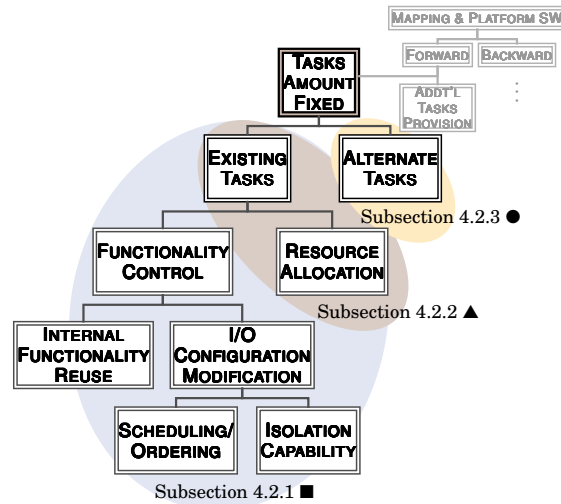
4.2.3. Alternate tasks ⑨ ● . Here, an existing task is modified and **replaced** by an alternate one with a **more robust** implementation, without providing additional tasks and without implementing a different algorithm. Typically, this is driven by information regarding vulnerable parts of the HW and is fed into the compiler, which performs the alterations. **Pros** include the lack of storage and latency. **Cons** include the limited error protection and rather system-specific applicability. Depending on the applied scheme, power and performance may be affected. Literature **examples** include: altered code that bypasses faulty HW [Meixner and Sorin 2008], altered code that reduces the amount of critical instructions [Rehman et al. 2011] ⑤.

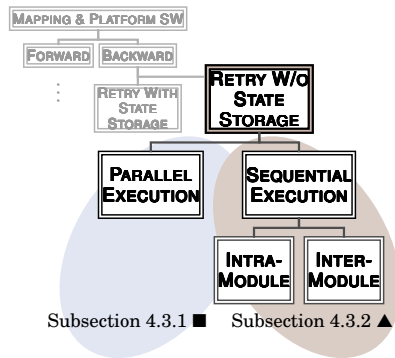## 4.3. Backward execution - Retry without state storage

Fig. 17: Classification for backward techniques that do not require state storage

This subsection discusses techniques that increase the resilience of systems by retrying the execution using the software stack without the need for storing intermediate state. Such schemes move back the execution to a point that information has been stored by the system itself due its normal functionality (without extra reliability-related storage overhead). In the degenerate case, a system can be re-started, e.g. the operating system can reboot in presence of an error. For a long time, such retry mechanisms have been integrated in correcting transient disk and memory read errors [Siewiorek and Swarz 1982]. But also in recent works, existent system features are exploited allowing re-execution without additional storage actions. A task can be re-executed without requiring extra storage either because there is another task running in parallel that keeps the state updated but potentially with a delay (**parallel execution**) or the task is being started from the beginning (**sequential execution**), so intermediate state information is not needed. Figure 17 shows the corresponding subtree.

4.3.1. Parallel execution ⑩ ■ . This category assumes that additional HW resources are available. When HW resources are explicitly added for the implementation, the technique is a hybrid with the (*Additional HW modules provision/same functionality*) category. The so-called ***primary/backup*** technique assumes that two tasks are executing in parallel, with one of them lagging behind the other. However, only the primary processor handles communication with external devices. The backup will take up this role once the primary processor fails. As the execution between the copies should be **deterministic**, extra care has to be taken for the handling of non-deterministic events. Typically, these are handled by the primary module and the result is passed afterwards to the backup module. Implementation of the primary/backup technique can take place at several layers of the software stack. **Pros** include the high error protection at minimum storage and rather general applicability. **Cons** include the power, performance overhead, latency and blockage of resources. A literature **example** in this category is [Bressoud and Schneider 1996]. The replicas are implemented at the virtual machine level and are running on different physical processors. The primary task handles I/O communication and the backup task lags a few instructions behind the primary. After a specific number of instructions have been executed, called *epoch*

(see Figure 18), the hypervisor communicates the interrupts the primary received and accompanying data to the backup ⓢ.

4.3.2. Sequential execution ▲ . This group includes techniques that restart the execution of the task from the beginning. As typical in the current framework, tasks at different granularities can exist, like instruction, thread, process. But the focus here will be on the task notion as it is meant in a broad category of techniques that is covered here, called **fault tolerant scheduling** techniques. These techniques are primarily employed for real-time systems. Such a system processes a set T of tasks. A task can be a unit of work, such as a granule of computation, a unit of data transmission or a file transfer [Liu et al. 1994] or in general a thread or a process that has to be ready by a certain deadline. Each task has a release time (and a deadline). If the execution finishes before the deadline for all the tasks, then the application is successfully executed. Resilience is achieved by re-executing a given task when it fails. The goal of a fault tolerant scheduling is to guarantee that the total execution time of the task set, including possible delays from re-execution due to (transient or permanent) faults, meets the system (hard) deadline. If this is not possible, the task set is rejected.

The tasks may be **periodic (with equal or varying periods), aperiodic or sporadic** [Sprunt et al. 1989]. Periodic tasks are released every Pi seconds (task period), while the release time in aperiodic tasks varies. Sporadic tasks are aperiodic tasks that have hard deadlines. Each release of a task is called an *instance*. Sporadic tasks may have an upper limit in their rate of arrival. For sporadic tasks, the system has to create the schedule as the tasks arrive and not offline ⓢ. The task can be re-executed either on a single processor so that transient errors are removed or on a different processor so that permanent errors are avoided. Therefore, this category is further split into **intra-module** and **inter-module** techniques. In intra-module techniques, the literature does not generally address tasks that are amenable to **non-deterministic events**. In inter-module techniques, this issue is partially addressed, especially concerning the shared memory accesses.

Fig. 18: The primary and backup tasks synchronize at each epoch in [Bressoud and Schneider 1996]

Intra-module ⑪. In order to deal with transient faults, a task is re-executed from the start on the same (single) processor; either the same task or a different (e.g. lighter) version of the task. To make a fault tolerant scheduling, **extra slack is inserted** in the schedule enough to allow the task re-execution upon fault occurrence. This extra slack is also referred to -symbolically- as ***backup task*** ⓢ. **Pros** include the limited storage, medium power overhead, rather general applicability, potentially high error protection (only transient errors). **Cons** include the performance overhead, latency and limitation to transient errors. Literature **examples** include: [Reis et al. 2007], [Rehman et al. 2013] on executing additional instructions, [Pandya and Malek 1998], [Ghosh et al. 1998] on fault-tolerant scheduling of periodic tasks and [Liberato et al. 2000] on aperiodic tasks ⓢ.

Inter-module ⑫. The following group of techniques is applied on a multiprocessor system and addresses not only transient but also permanent errors. A different processor is used to **re-assign** the task execution. Typically, the alternate processors have private memory. Each task has a backup copy. In general, the backup tasks may be
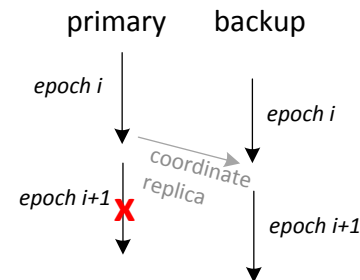
passive or active. **Active** tasks are scheduled and execute independent of whether an error occurs. The **passive** copies of the tasks are assigned and scheduled on different processors and will execute only if an error occurs. The copies may be pre-loaded in the memory of each processor (before execution time). Alternatively, only the required copies are loaded at run-time in a demand-driven way. The tasks may be periodic (with equal or varying periods) or sporadic. In these techniques two policies need to be derived: one for the task allocation to each processor and one for the scheduling of the tasks within each processor. **Pros** and **cons** are similar as in the previous category. However, in this case potentially also permanent errors can be addressed at the expense of resource blockage and extra synchronization. Moreover, in task scheduling schemes active tasks cause more latency and power overhead but are simpler to schedule. Literature **examples** include: [Krishna and Shin 1986], [Bertossi et al. 1999] on periodic tasks and [Ghosh et al. 1994] on aperiodic tasks. For example, Figure 19 shows an example of scheduling 4 primary tasks (P1, P2, P3, P4) and their backups (B1, B2, B3, B4) on three processors, as indicated in [Ghosh et al. 1994]. When the original tasks complete execution, their backups are deallocated and the space can be used for scheduling other tasks ⑤.
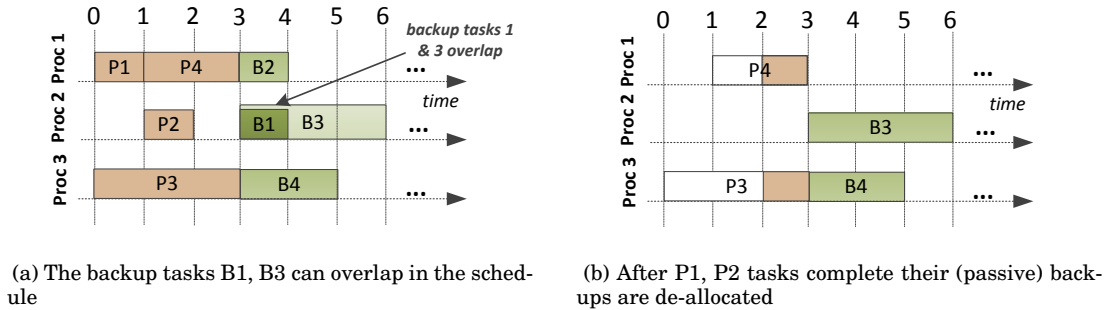

(a) The backup tasks B1, B3 can overlap in the schedule

(b) After P1, P2 tasks complete their (passive) backups are de-allocated

Fig. 19: Scheduling backup tasks in a multiprocessor system in [Ghosh et al. 1994]

4.4. Backward execution - Retry with state storage

The other group of backward techniques includes the techniques that retry the execution by storing the state of the system at intermediate points. The concept of proactively storing some part of the system state (and potentially additional information) in order to be able to restore the state and re-execute in the presence of errors has already been presented in Section 3.4.2. Here, checkpointing/rollback implementations at the mapping layer and the lower layers of the software stack are discussed. These can be differentiated between techniques that operate within a single HW module (**intra-module**) and techniques that operate across HW modules (**inter-module**). The proposed subtree is shown in Figure 20 ⑤.

   Different checkpointing schemes can be implemented for systems that do not have to deal with non-deterministic events and those that do. In applications of deterministic nature more design time knowledge can be exploited since the execution is more predictable due to the lack of non-deterministic events. For example, design time **knowledge of the control and data flow graph (CDFG)** can be exploited for optimizing the placement of the checkpoints. Rather than saving checkpoints at fixed intervals, checkpoints can be stored in a customized way so that the amount of stored data is minimized.

Using the CDFG to decide on the placement of checkpoints is not really practical in the *non-deterministic* applications due to the fact that the execution flow is decided in a non-deterministic way at run-time. For such applications, typically **extra effort** has to be invested to save relevant information. For example, the inter-process dependencies are often recorded, so that the execution can be accurately repeated during the recovery phase.

Fig. 20: Classification for SW-based backward techniques that require intermediate state storage

4.4.1. Intra-module (13) ■ . The techniques can be further characterized according to **when the checkpoint placement (location and/or frequency)** is decided. If the decision is made at design-time, the techniques belong to the *offline* category. Techniques that are applied during behavioral system synthesis or incorporate the checkpoint placement at the compilation or even combined with a design-time scheduling algorithm are offline techniques. When the decision is made at run-time, during program execution, they belong to the *online* category. These techniques are typically combined with a scheduling algorithm.
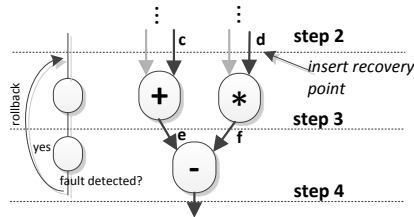

Fig. 21: The insertion of a checkpoint in the CDFG turns the number of required hardware registers to four instead of two at control step 3 (adapted from [Blough et al. 1997])

The techniques at **behavioral system synthesis level** typically use the application CDFG and identify the optimal number and locations of checkpoints under some optimization constraints. Such optimization constraints may include the amount of expected rollbacks, the additional execution time due to the rollback and the additional hardware resources, like registers needed to store the lifetime-extended variables [Blough et al. 1997]. An example that illustrates the usage of extra registers can be seen in Figure 21. Without the recovery facility, the hardware registers used for storing intermediate variables *c, d* can be reused for variables *e, f*. Therefore, at control step 3 only two registers are needed for the storage of the variables *e, f*. However, when a recovery point is added at control step 2, two additional registers are required for storing temporary variables *c, d*.

Techniques at compiler level **employ the compiler** to identify the optimal checkpoint locations. For example, the compiler can identify variables that are dead. These variables do not need to be included in the checkpoint. In the latter case, they can be used as assistance/guidance since the actual decisions on the checkpointing have to be made at run-time due to the need to handle non-deterministic events ⑤. Beyond the earlier discussed types of systems, intra-module schemes may address applications that are **amenable to numerous non-deterministic** events: uncertain functions (like human input functions), interrupts, system calls, I/O operations due to communication with external devices. Schemes, that incorporate the impact of such events in their rollback techniques, store information beyond the state of the participating processes, like interactions with external devices. Such additional pieces of information
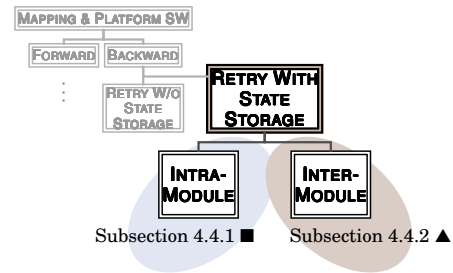
are typically called *logs*. Sometimes logs can be sufficient for the accurate re-execution of the process, without storing the process state. In the literature, schemes have been developed tailored to address one specific type of non-deterministic events or more of them together. Here, a few examples are provided and the reader is referred to specialized surveys on the topic. Such concepts are elaborated in [Elnozahy et al. 2002], [Sancho et al. 2005], [Chen et al. 2015], [Egwutuoha et al. 2013]. These surveys address both single-threaded and multithreaded/multi-process applications Ⓢ.

A number of techniques have been developed that do not explicitly bring the handling of non-deterministic events to the forefront. Some of them do not take care of them at all and some of them address them partially. They focus on performing checkpointing in a -to a large extent- transparent way for the user. These techniques can be differentiated depending on the abstraction level. Typically they are implemented either at kernel-level or user-level (see Section 2.2 and supplementary material). For **kernel-level** implementations, either the OS source code is available for modification or the user installs developed packages. The packages are only available for specific operating systems though. This implies that OS updates will require modifications of the packages. However, compared to user-level schemes no extra compilation or linking has to be performed by the user and the application is fully unaware of the checkpointing. Checkpointing at **user-level** utilizes run-time libraries that are linked to the application program. Minimal changes in the source code may be required. Schemes implemented at user-level improve portability and offer the possibility to the user to identify program points at which state data are essential for restarting the execution.

Pros include the high error protection and general applicability. **Cons** include the potentially high storage and power overhead, the potentially very high latency and performance (depending also on whether checkpointing is overlapped with normal execution). These costs vary depending on the exact implementation and selected granularity. Literature **examples** include: [Chandy and Ramamoorthy 1972], [Orailoglu and Karri 1994] on CDFG checkpoint placement, [Li et al. 1994], [Ramkumar and Strumpen 1997] on employing the compiler for the checkpointing. Examples that address non-deterministic applications include: [Hendriks 2002], [Duell 2005] at kernel-level, [Plank et al. 1994], [Slye and Elnozahy 1996] at user-level Ⓢ.

4.4.2. Inter-module ⑭ ▲ . While the multithreaded applications are also amenable to non-deterministic events which go beyond the process and thread concurrency issues, the focus of the related literature is **on handling these process/thread dependencies** so that rollback takes place effectively. Nevertheless, system-specific strategies have been developed which deal with **events coming from the external environment**, especially events due to communication with external devices Ⓢ. Online multiprocessor checkpointing can be broadly characterized as local and global. **Local** schemes require that only a single process or a subset of the processes that have interacted save independently a checkpoint and no global coordination takes place. That means, the rest of the processes do not have to perform any action at that moment. Typically, in such schemes, when a process fails, then all processors have to coordinate to create a consistent system-wide state. The inter-thread data dependencies of the processors that have communicated have to be recorded so that the inter-thread communication is correctly reconstructed at recovery time. Compared to global schemes, local schemes reduce the amount of data to be stored during checkpointing but require typically a more complicated recovery algorithm. Moreover, local schemes can potentially cause that one process rolls back after the other until the system goes back to a consistent state, potentially even back to the beginning. This type of rollback propagation is also called *domino effect*. On the contrary, **global** schemes require that

all processes take actions in order to take a single, global, system-wide checkpoint at distinct times. One disadvantage is that they become less scalable as the number of processes increase. **Pros** and **cons** are similar to the previous category with the extra overhead required to handle inter-process and inter-thread dependencies. As in the corresponding HW case, local schemes require less storage during checkpointing but need typically a more complicated recovery, compared to global schemes. Literature **examples** include: [Wu and Fuchs 1990], [Elnozahy 1994] on local and [Duell 2005], [Batchu et al. 2004] on global schemes ⓢ.

4.5. Overall mapping and platform software classification

By combining the sub-trees of the previous subsections, the overall mapping and platform software classification tree is built, as shown in Figure 22. Starting from the top-level split of Figure 13, the intermediate nodes (colored by pale yellow) are followed when necessary, to reach the final classes (colored by darker yellow and numbered).
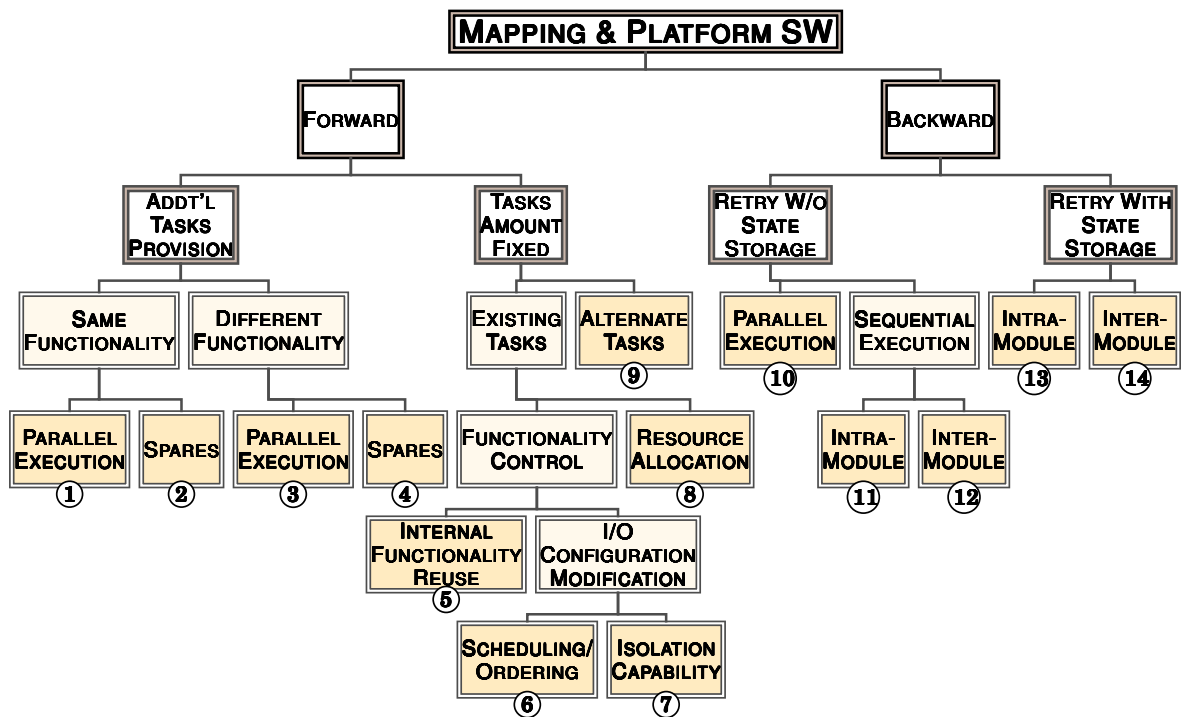

Fig. 22: Techniques that rely on mapping and platform software

5. USAGE OF THE CLASSIFICATION FRAMEWORK

Identifying the primitive components (corresponding to a primitive category) and their position in the framework first, allows to handle the complexity of the sometimes highly sophisticated mitigation schemes. A "divide and conquer" view of the publication enables the reader to delve into the most relevant implementation details (when that is necessary) in a much more controlled way.
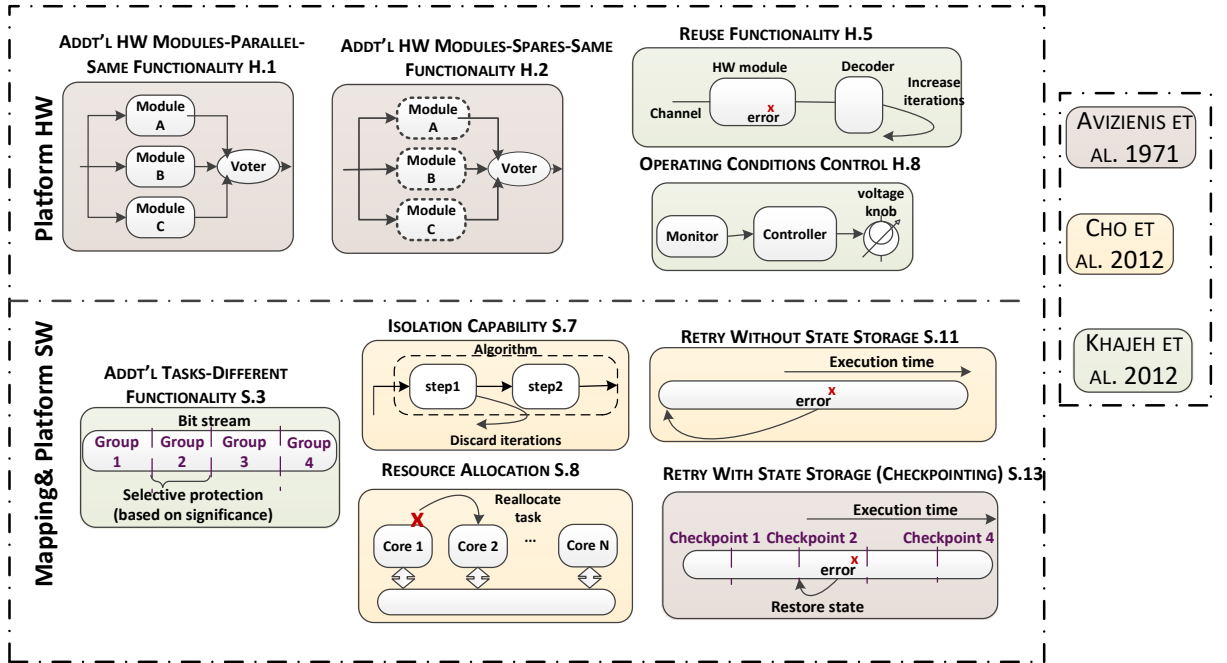
Fig. 23: Mapping of hybrid schemes according to the proposed classification

5.1. Mapping of hybrid schemes

In reality, the resiliency and mitigation approaches, which are present in research papers, rarely belong to a single leaf of the previous, and indeed any, classification. The majority of the published work consists of hybrid combinations of the leaves. Positioning these works in the classification framework above, allows to obtain a better understanding of the techniques.

In this subsection, the previous classification is applied on selective works, from a range of older and more recent literature, to illustrate how works can be classified by using the appropriate combination of leaves. For the mapping capital letters are used to indicate the major group the technique belongs to (**H** for platform HW and **S** for mapping & platform SW) and the number of the leaf from the corresponding figure (Figure 12 for platform HW, Figure 22 for mapping & platform SW). The selection of these papers is based on several criteria, the main ones being that they include hybrid schemes, that they cover a broad range and that combinations from many different leaves are present to provide a better illustration.

(1) A cross-layer co-exploration approach for reliability/energy optimization for video applications over wireless channels is presented in [Khajeh et al. 2012].

Platform HW: Reusing the Viterbi (by increasing the trace back depth) and Turbo (by increasing the number of iterations) channel decoders is discussed (because they are primarily used by WCDMA systems) as a complementary approach to reinforce the correcting capabilities (**H.5**). Aggressive Voltage Scaling (AVS) is applied on sections of the WCDMA modem. The authors focus on the embedded memory organization for which a specific analysis of the errors is performed to characterize the read and write

failures statistically, resulting in an analytic model. Based on this model, power savings can be maximized while maintaining a macro level quality metric such as the Peak Signal to Noise Ratio (PSNR) of the image sequence (**H.8**).

Mapping & platform SW: The difference in the significance of groups of bits is used to perform selective protection by proper encoding at the middleware layer (**S.3**). In particular, UDP-Lite packetization is (re-)designed in such a way that bits which are very sensitive to errors are better protected. [8]

(2) The ERSA architecture [Cho et al. 2012] is a multicore architecture with the characteristic that some cores are super reliable (SRC), while the majority are relaxed reliability cores (RRC). The super reliable cores execute operations that are less resilient to errors. [9]

Platform HW: The RRCs can be restarted by using a watchdog timer (**H.13**).

Mapping & platform SW: A run time scheduler reassigns a task that has failed on a particular RRC to another RRC (**S.8**). Moreover, the SRC can terminate the execution of a RRC task and reboot the RRC (**S.12**). Computations are discarded when excessively large fluctuations are present (**S.7**). This requires knowledge of the applications algorithm so it is a hybrid itself.

(3) The JPL-STAR was a fault tolerant computer designed to cope with many kinds of hardware faults [Avizienis et al. 1971].

Platform HW: Triple Modular Redundancy with voting is applied for the test and repair processor (TARP). This processor monitors the operation of the main computer and implements the recovery (**H.1**). One or more unpowered spares are provided for each functional unit and the TARP itself (**H.2**). All machine words (data and instructions) are encoded using error-detecting codes (**H.3**).

Mapping & platform SW: Checkpointing and rollback of programs to a previous state is also used when an error is detected (**S.13**). The system uses also a restart (named "cold start") procedure in the TARP and the resident executive (**S.11**).

5.2. Comparison of closely related schemes

Academic Works at the Same Abstraction Level. Research trends undergo a proliferating application of hybrid mitigation mechanisms. The work by Li et al. [Li et al. 2013a] provides a hybrid error mitigation mechanism, called DHASER, that exploits several mapping & platform SW approaches. In particular, this scheme adopts the following mitigation approaches:

- First, a task mapping approach (**S.8**) is employed, by analyzing the impact of single-event upset on the overall correctness of a running task, without correcting this error. The outcome is a generated parameter per task (based on the masking capability of the task). Thus, the tasks can be appropriately allocated to the processing elements with the required level of resiliency for a reliable operation. For example, highly vulnerable tasks are executed on cores that are more robust and less vulnerable ones on cheaper and more power-efficient cores without recovery functionalities.

- A second mechanism is applied to select the appropriate error protection mechanism to each processing element. The choices are between re-executing instructions inserted by the compiler (**S.11**) or a hybrid SW/HW checkpointing scheme (**H.15**, **S.13**).

---

[8]An additional technique implemented at the paper is the modification of the video compression encoding mode at the algorithmic layer

[9]The authors also employ circuit-layer techniques to implement the asymmetric reliability among the SRCs and the RRCs.

ERSA		DHASER		dTune	
Feature	**Class**	**Feature**	**Class**	**Feature**	**Class**
Distributing tasks to SRC and RRC cores	**S.8**	Task allocation based on parameter selection	**S.8**	Task allocation based on parameter selection	**S.8**
RRC core reboots	**H.13**	Re-execute instructions	**S.11**	Redundant Multi-threading (TMR)	**H.9**
SRC reboots RRC	**S.12**	HW/SW checkpointing (functionality mapped into micro-operations)	**S.13**, **H.15**	Alternate reliable code versions	**S.9**
Discard computations	**S.7**	–	–	–	–

Table I: Classification of run time task mapping by platform SW solutions

ASER		Hayat	
Feature	**Class**	**Feature**	**Class**
Task allocation on subset of cores given application properties and constraints (e.g. dark silicon area)	**S.8**	Task allocation on subset of cores based on aging estimation mechanism	**S.8**
TMR on processor parts	**H.1**	Operating conditions control	**H.8**

Table II: Resilience schemes in the dark silicon constraint

This selection is based on the expected to-be-mapped task from the previous technique, and the characteristics of the targeted processing element.

This classification also allows to make a clear distinction with the platform SW part of the ERSA approach [Cho et al. 2012] discussed earlier (see 2 first columns of Table I). That approach is intended to deal with the same problem formulation, namely reliable run time mapping of tasks on a heterogeneous multicore platform. ERSA formulates this as "distributing tasks to SRCs and RRCs, i.e. Super Reliable and Relaxed Reliability Cores". It combines this also with algorithmic layer resilience but this part will not be discussed here. In order to achieve this reliable task mapping, both approaches use partly similar and partly different techniques and the systematic classification provides effective insight in this. The **(S.8)** techniques of both ERSA and DHASER are used in a largely similar way but ERSA uses an instance of the **(S.7**, **S.12**, **H.13)** option on top of this, whereas the presently discussed approach uses instances of the **(S.11**, **S.13**, **H.15)** leaves in combination with it (see above). To illustrate further the point made, in the third column, dTune [Rehman et al. 2014] has been added, which performs reliable task mapping **(S.8)** by combining reliable code versions **(S.9)** with HW-based Redundant Multithreading **(H.9)**.

Finally, in Table II, two schemes in the so-called dark silicon constraint are briefly compared, ASER [Kriebel et al. 2014] and Hayat [Gnad et al. 2015] without providing a detailed explanation. In the supplementary material, a comparison of two industrial RAS schemes is elaborated, from Intel [Intel 2011] and IBM [Mitchell et al. 2009] ⑤.

6. DISCUSSION AND FUTURE CHALLENGES

The proposed classification was illustrated through a representative list of schemes, to better absorb the related ideas and support the validity of the tree. Although the

focus of the current paper was not to present a comprehensive list of state-of-the-art schemes, some observations become clear regarding the evolution of techniques.

Observations derived from the proposed framework and literature. The literature on fault tolerance and resilience techniques has evolved in accordance with the **trends in computer architecture and (platform) software design** development. Due to the power density issues that came along with technology scaling, there was a shift towards multicore designs and in general parallel processing. The inherent regularity and abundance of such designs offered opportunities for fault-tolerant techniques. The software had to evolve as well to make use of these new designs (especially through multithreading). Another evolution has to do with adding more custom processing elements (like GPUs) on the platform, together with general-purpose components, in order to accelerate part of the functionality. Of course, fault tolerant techniques did not omit to exploit more ad-hoc features of certain platforms, like features that enable the out-of-order execution on superscalar platforms.

Commonalities and differences between HW-based and SW-based schemes can be observed. Traditional fault-tolerance was based on a **few basic types of techniques**: hardware replication, re-execution starting from a previously saved state (typically at the OS level), hardware-based error coding schemes. While these techniques have not been abandoned, over time there is a clear boost in platform software and mapping approaches. This does not reduce complexity for the designer but allows to have the cost reduced. Employing mapping and platform software techniques even creates new flavors of the aforementioned techniques ending still in a big set of instantiated techniques. For example, *TMR* can be implemented at the MPI library [Fiala et al. 2012], error coding can be implemented through a software task [Shirvani et al. 2000], checkpointing can be implemented at almost all layers of the software stack (see Section 4.4). But this is not the only commonality. Common concepts can be identified behind the techniques both in the forward category (with the exception of the operating conditions control for the SW) and the backward category (with the exception of having the amount of tasks fixed in SW since there is always some repetition of execution). Nevertheless differences also exist. The most prominent being that mapping and SW provides a lot of flexibility due to the re-mapping possibilities of a given task sequence onto the "fixed" HW. This leads to a number of techniques that are not possible in the HW-based approaches: re-arranging the instruction profile, fault tolerant task scheduling, fault tolerant mapping on multi/many-core architectures [S].

Trends and new directions. **Applications** themselves have been evolving. Beyond the advance in parallelizable applications, enabled by the architecture changes, there has been an explosion in the types of embedded applications, covering many different aspects of the daily life. Fault-tolerant design has invaded such areas, even lifestyle applications, as multimedia applications [Andreopoulos 2013]. Networked applications expanded further the deliverable functionality possibilities. There is a clear tendency to exploit more of the application knowledge in order to minimize the cost; especially, when error tolerance is possible, like in approximate computing [Sampson et al. 2015]. Other examples of emerging error-tolerant application domains are Recognition, Mining and Synthesis (RMS) [Dubey 2005] as well as artificial neural networks (ANNs) [Temam 2012]. It is important to note that as applications and systems keep on evolving, **new combinations of requirements** that have to be satisfied are created. For example, enterprise distributed real-time and embedded (DRE) systems combine, among others, the requirement for high availability, with real-time response, resource-constraints and certain quality of service (QoS) requirements, like low-latency [Tambe 2010], [He and Da Xu 2014].

Adaptivity is another notable characteristic of more recently developed techniques. Techniques incorporate elements that allow them to be differentiated during run time depending on the changing conditions. Knobs that allow fine-grained control, enable cost reduction by satisfying the minimum necessary requirements. This is strongly enabled by the evolution and availability of monitor and sensor systems [Chandra 2014]. The system behavior can be adapted at run time whenever significant environmental changes take place, or according to varying error rates.

As indicated by several authors during the last years [DeHon et al. 2010], [Carter et al. 2010], [Reddi et al. 2012], [Henkel et al. 2014], **synergistic reliability approaches** that combine several techniques across the same layer or cross-layer can lead to near-optimal solutions. This is especially so, as errors can be masked as they propagate through the different hardware and software layers (including the application itself). Therefore, by properly propagating information among the different layers and providing a suitable degree of adaptivity (with design time and run time knobs), the most cost-effective solutions can be achieved. This has brought on challenges as well, as knowledge and expertise from different domains has to be combined.

Further technology trends like **3D integration, incorporating heterogeneous technologies on a single platform and dark silicon** pose new challenges and opportunities for the fault tolerance techniques. In 3D designs the outer dies offer a shield against radiation particles. So, for example, part of the cache can be mapped in the inner dies without ECC protection to reduce energy and latency [Sun et al. 2011]. The authors in [Shafique et al. 2014] discuss the challenges coming from the dark silicon era. For a given (maximum) thermal design power (TDP), not all transistors can be simultaneously powered on at full performance, so that the chip operates below the thermal safe temperature. For example, during a particular TDP (mode 5), the authors have observed some natural trade-offs between transient fault rates and lifetime reliability (through aging). The TDP mode 5 corresponds to a mode, where cores are operated in near threshold voltages. At this mode, 3x-30x higher soft error rates have been observed. However, since the cores can be operated at reduced temperatures at this mode, they are exposed to reduced aging. This information can be provided to the run time manager of the system, which can then make appropriate dynamic decisions depending on the system quality targets.

Moreover, as many systems nowadays employ **commodity-off-the-shelf (COTS)** hardware and/or software components, the case of building a system's hardware and software from the ground up becomes rare. This leads to partial blackbox-based design and the resulting lack of internal design knowledge adds an additional challenge on deriving appropriate reliability-driven approaches.

The aforementioned challenges require the designers to come up with innovative solutions in order to ensure reliable digital system operation. This makes the request for a global view on the domain of reliability-improvement techniques more necessary. This survey is a contribution in that direction.

7. RELATED WORK ON CLASSIFICATION SCHEMES

During the last decades, both academia and industry have invested effort to describe fault tolerance and mitigation techniques in a structured way. Here, some representative examples are discussed, that act complementary to this work.

In the extended past, works like the one by Randell [Randell et al. 1978], Siewiorek and Swarz [Siewiorek and Swarz 1982] (in their book) have described the principles of reliable system design, including terminology, metrics and models. In addition, they explained in detail resilience features of highly reliable and highly available systems, like commercial computers, spacecraft and avionics systems. What is most relevant for the current work, is that these works include a categorization of the broad literature

at that time in the domain of reliability mitigation including (micro-) architectural and software layers. Rivers et al. [Rivers et al. 2011] give an overview of the current state-of-the-art practices for error tolerance in server class microprocessors: A basic discussion is performed regarding the abstraction layers and the different forms of redundancy (information, space, time), based on which error tolerance is achieved. The aim of the paper is to give a review of current schemes and discuss approaches of promise for the future such that they do not present an elaborate classification scheme. Gizopoulos et al. [Gizopoulos et al. 2011] present a taxonomy of error recovery and repair techniques for multicore processor architectures. In this paper, a basic split of mitigation techniques is made between error recovery and error repair techniques. Error recovery is further split into forward error recovery (FER), which includes redundancy, like for example triple modular redundancy, and backward error recovery (BER), which includes rolling back to a previously saved correct state of the system. They consider that error repair techniques include basically reconfiguration and graceful performance degradation. Abdallah et al. [Abdallah et al. 2012] present a survey on designs of stochastic hardware with relaxed guard-bands in order to achieve reliable operation and satisfactory performance. Such designs address applications that are inherently error tolerant. Mittal and Vetter [Mittal and Vetter 2015] present a survey of techniques that improve resilience as well as reliability metrics, during the last 10-15 years. They use some rough categories for their presentation (e.g. redundancy-based, compiler-based etc.) and they classify the presented schemes based on a few criteria, like the processor component that is addressed, key approach/feature of the technique and evaluation platform. The survey in [Saha 2006] lists a number of software-level techniques to counteract hardware-induced errors but also software bugs.

Many of the aforementioned survey papers follow a different approach: They present selected works in detail or they select techniques addressing specific types of systems or application domains. The proposed approach is closer to the one followed in [Siewiorek and Swarz 1982], having a similar motivation: organizing the categories by the type of techniques that are applied allows the universality of techniques to be manifested. This gives more opportunities to the designer to identify a technique, that is potentially fitting for his problem, and customize it according to the specific features of a given system and application. To this end, the presentation of pros and cons for the classes can more actively assist. For example, consider that the designer needs a generally applicable solution for a medical application, in which a strong requirement for reliability is posed. Based on the property of high error protection and general applicability, he would have to select more costly techniques, like *TMR* or checkpointing/rollback. It is important to note that, in general, detection costs should also be taken into account when considering a mitigation scheme. In many cases, detection is strongly correlated with the mitigation and the combined outcome will decide the cost. Other organizations are also possible. A different way would be to organize the classification around specific modules (e.g. schemes for processors). However, this way suppresses the re-usability of techniques. For example, *TMR* or isolation capability can work for many types of modules beyond processors. Another organization could be around specific error effects (e.g. present solutions for transient errors vs. aging). But there, also, reusability is possible. For instance, in an example placed under leaf **H.2**, the authors in [Shin et al. 2008] propose a proactive scheme to periodically activate and deactivate modules (cache arrays) in order to suspend and/or recover the wear-out that is caused due to NBTI and the deviation of cell stability due to process variation (using one spare array). Potentially a similar scheme could be applied for errors due to supply noise (but then in a demand-driven way rather than periodically), to allow the effects of the noise go away.

Of course, since the work in [Siewiorek and Swarz 1982] a lot of progress has taken place in the domain, as already discussed, which has made the number of types of techniques become significantly larger: parallel processing, software stack evolution, sensing and monitoring evolution to name a few notable examples. Without claiming an exhaustive presentation of the domain, an effort to keep a balance between breadth and depth has been made, illustrated by multiple examples, in order to give the reader a comprehensive view. Also, hardware and software solutions have been discussed using a similar reasoning, to allow interrelations to become more visible and facilitate experts from different disciplines to come closer to each other. Moreover, the complementarity of the leaves allows hybrid combinations to be better understood. Finally, for the interested user, this paper can act complementary to previous survey works. They can become part but also enrich a more global framework in which older, more recent but also potential future resilience approaches can be classified.

8. CONCLUSION

In this paper, techniques that increase resilience and mitigate functional reliability errors were classified in a novel way. This was achieved through a framework with complementary splits, in which primitive mitigation concepts are defined. That allows every type of technique to be classified, by combining the appropriate components. The framework has been accompanied by a wide variety of sources from the published literature. In this way, insight can be provided to the designers and researchers about the nature of existing schemes, since every node has some unique properties. But also the development of efficient solutions in the future is facilitated, since the desired properties of a new technique, required to satisfy a certain need, can be more easily identified when they are presented in a structured way.

REFERENCES

Sarah Abdallah, Ali Chehab, Imad H. Elhajj, and Ayman Kayssi. 2012. Stochastic hardware architectures: A survey. In *Energy Aware Computing, 2012 International Conference on*. 1–6.

Rishi Agarwal, Pranav Garg, and Josep Torrellas. 2011. *Rebound: scalable checkpointing for coherent shared memory*. Vol. 39. ACM.

Nidhi Aggarwal. 2008. *Achieving high availability with commodity hardware and software*. ProQuest.

Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P Jouppi, and James E Smith. 2007. Configurable isolation: building high availability systems with commodity multi-core processors. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 470–481.

Rana Ejaz Ahmed, Robert C Frazier, and Peter N Marinos. 1990. Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*. IEEE, 82–88.

Robert Aitken, Görschwin Fey, Zbigniew T Kalbarczyk, Frank Reichenbach, and Matteo Sonza Reorda. 2013. Reliability analysis reloaded: how will we survive?. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 358–367.

Yiannis Andreopoulos. 2013. Error tolerant multimedia stream processing: There's plenty of room at the top (of the system stack). *Multimedia, IEEE Transactions on* 15, 2 (2013), 291–303.

T.M. Austin. 1999. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*. 196 –207.

Algirdas Avizienis. 1985. The N-version approach to fault-tolerant software. *IEEE Transactions on software engineering* 12 (1985), 1491–1501.

Algirdas Avizienis, George C Gilley, Francis P Mathur, David A Rennels, John A Rohr, and David K Rubin. 1971. The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. *Computers, IEEE Transactions on* 100, 11 (1971), 1312–1321.

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.* 1, 1 (Jan. 2004), 11–33.

Rajanikanth Batchu, Yoginder S Dandass, Anthony Skjellum, and Murali Beddhu. 2004. MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing* 7, 4 (2004), 303–315.

A.A. Bertossi, L.V. Mancini, and F. Rossini. 1999. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *Parallel and Distributed Systems, IEEE Transactions on* 10, 9 (sep 1999), 934 –945.

Douglas M Blough, Fadi J Kurdahi, and Seong Y Ohm. 1997. Optimal algorithms for recovery point insertion in recoverable microarchitectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 16, 9 (1997), 945–955.

Andrea Bondavalli, Silvano Chiaradonna, Felicita Di Giandomenico, and Fabrizio Grandoni. 2000. Threshold-based mechanisms to discriminate transient from intermittent faults. *Computers, IEEE Transactions on* 49, 3 (2000), 230–245.

S. Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro* 25, 6 (Nov 2005), 10–16.

Fred A Bower, Daniel J Sorin, and Sule Ozev. 2005. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 197–208.

Christian Brehm, Matthias May, Christina Gimmler, and Norbert Wehn. 2012. A case study on error resilient architectures for wireless communication. In *Proceedings of the 25th international conference on Architecture of Computing Systems (ARCS'12)*. 13–24.

Thomas C Bressoud and Fred B Schneider. 1996. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (1996), 80–107.

Nicholas P Carter, Helia Naeimi, and Donald S Gardner. 2010. Design techniques for cross-layer resilience. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 1023–1028.

Sayantan Chakravorty, Celso L Mendes, and Laxmikant V Kalé. 2006. Proactive fault tolerance in MPI applications via task migration. In *High Performance Computing-HiPC 2006*. Springer, 485–496.

Vishal Chandra. 2014. Monitoring reliability in embedded processors-A multi-layer view. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 1–6.

K Mani Chandy and Chittoor V Ramamoorthy. 1972. Rollback and recovery strategies for computer programs. *Computers, IEEE Transactions on* 100, 6 (1972), 546–556.

Mengly Chean and Jose AB Fortes. 1990. A taxonomy of reconfiguration techniques for fault-tolerant processor arrays. *Computer* 23, 1 (1990), 55–69.

Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. 2015. Deterministic Replay: A Survey. *ACM Comput. Surv.* 48, 2, Article 17 (Sept. 2015), 47 pages.

Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. 2012. Ersa: Error resilient system architecture for probabilistic applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 31, 4 (2012), 546–558.

Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: an architectural framework for software recovery of hardware faults. In *Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10)*. ACM, New York, NY, USA, 497–508.

André DeHon, Heather M Quinn, and Nicholas P Carter. 2010. Vision for cross-layer optimization to address the dual challenges of energy and reliability. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 1017–1022.

M. M. Dickinson, J. B. Jackson, and G. C. Randa. 1964. Saturn V launch vehicle digital computer and data adapter. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part I (AFIPS '64 (Fall, part I))*. ACM, New York, NY, USA, 501–516.

Björn Döbel, Hermann Härtig, and Michael Engel. 2012. Operating system support for redundant multithreading. In *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 83–92.

Pradeep Dubey. 2005. Recognition, mining and synthesis moves computers to the era of tera. (2005).

Jason Duell. 2005. The design and implementation of berkeley lab's linux checkpoint/restart. *Lawrence Berkeley National Laboratory* (2005).

Nikil Dutt, Puneet Gupta, Alex Nicolau, Abbas BanaiyanMofrad, Mark Gottscho, and Majid Shoushtari. 2014. Multi-layer memory resiliency. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 1–6.

Ifeanyi Egwutuoha, David Levy, Bran Selic, and Shiping Chen. 2013. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *Journal of Supercomputing* 65, 3 (2013).

Elmootazbellah Nabil Elnozahy. 1994. Manetho: fault tolerance in distributed systems using rollback-recovery and process replication. (1994).

Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)* 34, 3 (2002), 375–408.

David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 78.

Sunondo Ghosh, R. Melhem, and D. Mosse. 1994. Fault-tolerant scheduling on a hard real-time multiprocessor system. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*. 775–782.

Sunondo Ghosh, Rami Melhem, Daniel Mossé, and Joydeep Sen Sarma. 1998. Fault-tolerant rate-monotonic scheduling. *Real-Time Systems* 15, 2 (1998), 149–181.

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 *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*. 1–6.

Dennis Gnad, Muhammad Shafique, Florian Kriebel, Semeen Rehman, Duo Sun, and Jörg Henkel. 2015. Hayat: Harnessing Dark Silicon and Variability for Aging Deceleration and Balancing. In *Proceedings of the 52Nd Annual Design Automation Conference (DAC '15)*. ACM, NY, USA, Article 180, 6 pages.

Mohamed Gomaa, Chad Scarbrough, TN Vijaykumar, and Irith Pomeranz. 2003. Transient-fault recovery for chip multiprocessors. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 98–109.

Meeta S Gupta, Jude A Rivers, Pradip Bose, Gu-Yeon Wei, and David Brooks. 2009. Tribeca: design for PVT variations with local recovery and fine-grained adaptation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 435–446.

S. Gupta, Shuguang Feng, A. Ansari, J. Blome, and S. Mahlke. 2008. The StageNet fabric for constructing resilient multicore systems. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*. 141–151.

Richard W Hamming. 1950. Error detecting and error correcting codes. *Bell System technical journal* 29, 2 (1950), 147–160.

Haibo He, Sheng Chen, Kang Li, and Xin Xu. 2011. Incremental learning from stream data. *Neural Networks, IEEE Transactions on* 22, 12 (2011), 1901–1914.

Wu He and Li Da Xu. 2014. Integration of distributed enterprise applications: a survey. *Industrial Informatics, IEEE Transactions on* 10, 1 (2014), 35–42.

Rajamohana Hegde and Naresh R Shanbhag. 2001. Soft digital signal processing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 9, 6 (2001), 813–823.

Erik Hendriks. 2002. VMADump. (2002).

Jörg Henkel, Lars Bauer, Hongyan Zhang, Semeen Rehman, and Muhammad Shafique. 2014. Multi-Layer Dependability: From Microarchitecture to Application Level. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*. ACM, New York, NY, USA, Article 47, 6 pages.

John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

Amr Hussien, Muhammed S Khairy, Amin Khajeh, Kiarash Amiri, Ahmed M Eltawil, and Fadi J Kurdahi. 2010. A combined channel and hardware noise resilient Viterbi decoder. In *Signals, Systems and Computers (ASILOMAR)*. IEEE, 395–399.

Amr Hussien, Muhammad S Khairy, Amin Khajeh, Ahmed M Eltawil, and Fadi J Kurdahi. 2011. A class of low power error compensation iterative decoders. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*. IEEE, 1–6.

IEEE_Std. 1990. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990* (Dec 1990), 1–84. DOI:http://dx.doi.org/10.1109/IEEESTD.1990.101064

Intel. 2011. *Intel® Xeon® Processor E7 Family: Reliability, Availability, and Serviceability*[10]. Technical Report. Data Center Group – Intel Corporation.

Casey M Jeffery and Renato JO Figueiredo. 2012. A flexible approach to improving system reliability with virtual lockstep. *Dependable and Secure Computing, IEEE Transactions on* 9, 1 (2012), 2–15.

Doug Jewett. 1991. Integrity S2: A fault-tolerant Unix platform. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*. IEEE, 512–519.

---

[10]http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-e7-family-ras-server-paper.pdf

Eric Karl, David Blaauw, Dennis Sylvester, and Trevor Mudge. 2006. Reliability Modeling and Management in Dynamic Microprocessor-based Systems. In *Proceedings of the 43rd Annual Design Automation Conference (DAC '06)*. ACM, New York, NY, USA, 1057–1060.

Amin Khajeh, Minyoung Kim, Nikil Dutt, Ahmed M Eltawil, and Fadi J Kurdahi. 2012. Error-Aware Algorithm/Architecture Coexploration for Video Over Wireless Applications. *ACM Transactions on Embedded Computing Systems (TECS)* 11, 1 (2012), 15.

Florian Kriebel, Semeen Rehman, Duo Sun, Muhammad Shafique, and Jörg Henkel. 2014. ASER: Adaptive Soft Error Resilience for Reliability-Heterogeneous Processors in the Dark Silicon Era. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*. ACM, NY, USA, Article 12, 6 pages.

C Mani Krishna and Kang G Shin. 1986. On scheduling tasks with a quick recovery from failure. *Computers, IEEE Transactions on* 100, 5 (1986), 448–455.

K. J. Kuhn, M. D. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. T. Ma, A. Maheshwari, and S. Mudanai. 2011. Process Technology Variation. *IEEE Trans. on Electron Devices* (2011), 2197–2208.

Chung-Chi Jim Li, Elliot M Stewart, and W Kent Fuchs. 1994. Compiler-assisted full checkpointing. *Software: Practice and Experience* 24, 10 (1994), 871–886.

Tuo Li, Muhammad Shafique, Semeen Rehman, Jude Angelo Ambrose, Jörg Henkel, and Sri Parameswaran. 2013a. DHASER: Dynamic Heterogeneous Adaptation for Soft-error Resiliency in ASIP-based Multicore Systems. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD '13)*. IEEE Press, Piscataway, NJ, USA, 646–653.

Tuo Li, Muhammad Shafique, Semeen Rehman, Swarnalatha Radhakrishnan, Roshan Ragel, Jude Angelo Ambrose, Jörg Henkel, and Sri Parameswaran. 2013b. CSER: HW/SW configurable soft-error resiliency for application specific instruction-set processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 707–712.

Frank Liberato, Rami Melhem, and Daniel Mossé. 2000. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *Computers, IEEE Transactions on* 49, 9 (2000), 906–914.

Jane WS Liu, Wei-Kuan Shih, Kwei-Jay Lin, Riccardo Bettati, and Jen-Yao Chung. 1994. Imprecise computations. *Proc. IEEE* 82, 1 (1994), 83–94.

Klaus Lochmann and Andreas Goeb. 2011. A unifying model for software quality. In *Proceedings of the 8th international workshop on Software quality (WoSQ '11)*. ACM, New York, NY, USA, 3–10.

Matthias May, Matthias Alles, and Norbert Wehn. 2008. A case study in reliability-aware design: a resilient LDPC code decoder. In *Proceedings of the DATE conference*. ACM, 456–461.

J. W. McPherson. 2006. Reliability Challenges for 45Nm and Beyond. In *Proceedings of the 43rd Annual Design Automation Conference (DAC '06)*. ACM, New York, NY, USA, 176–181.

A. Meixner and D.J. Sorin. 2008. Detouring: Translating software to circumvent hard faults in simple cores. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. 80–89.

Jim Mitchell, Daniel Henderson, George Ahrens, and Julissa Villareal. 2009. *IBM Power Platform Reliability, Availability and Serviceability (RAS)*[11] Technical Report POW03003.doc. International Business Machines (IBM) Corporation.

Sparsh Mittal and Jeffrey Vetter. 2015. A Survey of Techniques for Modeling and Improving Reliability of Computing Systems. *IEEE Transactions on Parallel & Distributed Systems* (2015).

Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. 2010. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 335–338.

A. Orailoglu and R. Karri. 1994. Coactive scheduling and checkpoint determination during high level synthesis of self-recovering microarchitectures. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 2, 3 (Sept 1994), 304–311.

Krishna Palem and Avinash Lingamneni. 2012. What to do about the end of Moore's law, probably!. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 924–929.

Mihir Pandya and Miroslaw Malek. 1998. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *Computers, IEEE Transactions on* 47, 10 (1998), 1102–1112.

Aashish Pant, Puneet Gupta, and Mihaela Van Der Schaar. 2012. AppAdapt: Opportunistic application adaptation in presence of hardware variation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 20, 11 (2012), 1986–1996.

Matthias Pflanz and Heinrich Theodor Vierhaus. 2001. Online check and recovery techniques for dependable embedded processors. *IEEE Micro* 5 (2001), 24–40.

---

[11] https://www-304.ibm.com/webapp/set2/sas/f/lopdiags/info/Power6RASOverview.pdf.

James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1994. *Libckpt: Transparent checkpointing under unix*. Computer Science Department.

Stefan Poledna. 1996. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, Norwell, MA, USA.

Stefan Poledna. 2007. "System Aspects of Dependable Systems ", Lecture Notes on Dependable Computer Systems. (2007).

Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. 2009. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09)*. 93 –104.

Milos Prvulovic, Zheng Zhang, and Josep Torrellas. 2002. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 111–122.

Abbas Rahimi, Andrea Marongiu, Paolo Burgio, Rajesh K Gupta, and Luca Benini. 2013. Variation-tolerant OpenMP tasking on tightly-coupled processor clusters. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. IEEE, 541–546.

Balkrishna Ramkumar and Volker Strumpen. 1997. Portable checkpointing for heterogeneous architectures. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*. IEEE, 58–67.

B. Randell, P. Lee, and P. C. Treleaven. 1978. Reliability Issues in Computing System Design. *ACM Comput. Surv.* 10, 2 (June 1978), 123–165.

Joydeep Ray, James C Hoe, and Babak Falsafi. 2001. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 214–224.

Vijay Janapa Reddi, David Z Pan, Sani R Nassif, and Keith A Bowman. 2012. Robust and resilient designs from the bottom-up: Technology, CAD, circuit, and system issues. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE, 7–16.

Semeen Rehman, Florian Kriebel, Duo Sun, Muhammad Shafique, and Jörg Henkel. 2014. dTune: Leveraging Reliable Code Generation for Adaptive Dependability Tuning Under Process Variation and Aging-Induced Effects. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*. ACM, New York, NY, USA, Article 84, 6 pages.

Semeen Rehman, Muhammad Shafique, Pau Vilimelis Aceituno, Florian Kriebel, Jian-Jia Chen, and Jörg Henkel. 2013. Leveraging variable function resilience for selective software reliability on unreliable hardware. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '13)*. EDA Consortium, San Jose, CA, USA, 1759–1764.

Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. 2011. Reliable software for unreliable hardware: embedded code generation aiming at reliability. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 237–246.

Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. 2012. Raise: Reliability-aware instruction scheduling for unreliable hardware. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE, 671–676.

George A Reis, Jonathan Chang, and David I August. 2007. Automatic instruction-level software-only recovery. *IEEE micro* 1 (2007), 36–47.

Jude A Rivers, Meeta S Gupta, Jeonghee Shin, Prabhakar N Kudva, and Pradip Bose. 2011. Error tolerance in server class processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 30, 7 (2011), 945–959.

Dimitrios Rodopoulos, Georgia Psychou, Mohamed M. Sabry, Francky Catthoor, Antonis Papanikolaou, Dimitrios Soudris, Tobias G. Noll, and David Atienza. 2015. Classification Framework for Analysis and Modeling of Physically Induced Reliability Violations. *ACM Comput. Surv.* 47, 3, Article 38 (Feb. 2015), 33 pages.

Bogdan F Romanescu and Daniel J Sorin. 2008. Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 43–51.

Tajana Simunic Rosing, Kresimir Mihic, and Giovanni De Micheli. 2007. Power and reliability management of SoCs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 15, 4 (2007), 391–403.

E. Rotenberg. 1999. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*. 84 –91.

Goutam Kumar Saha. 2006. Software based fault tolerance: a survey. *Ubiquity* 2006, July, Article 1 (July 2006), 1 pages.

Adrian Sampson, James Bornholt, and Luis Ceze. 2015. Hardware-Software Co-Design: Not Just a Cliché. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Jose Carlos Sancho, Fabrizio Petrini, Kei Davis, Roberto Gioiosa, and Song Jiang. 2005. Current practice and a direction forward in checkpoint/restart implementations for fault tolerance. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 8–pp.

Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. 2014. The EDA challenges in the dark silicon era. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 1–6.

Jeonghee Shin, Victor Zyuban, Pradip Bose, and Timothy M Pinkston. 2008. A proactive wearout recovery approach for exploiting microarchitectural redundancy to extend cache SRAM lifetime. In *ACM SIGARCH Computer Architecture News*, Vol. 36. IEEE Computer Society, 353–362.

Philip P Shirvani, Nirmal R Saxena, and Edward J McCluskey. 2000. Software-implemented EDAC protection against SEUs. *Reliability, IEEE Transactions on* 49, 3 (2000), 273–284.

D.P. Siewiorek. 1990. Fault tolerance in commercial computers. *Computer* 23, 7 (July 1990), 26–37.

D. Siewiorek and R. Swarz. 1982. *The Theory and Practice of Reliable System Design*. Digital Press.

Joseph Slember and Priya Narasimhan. 2006. Living with nondeterminism in replicated middleware applications. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*. Springer-Verlag New York, Inc., 81–100.

J Hamilton Slye and Elmootazbellah Nabil Elnozahy. 1996. Supporting nondeterministic execution in fault-tolerant systems. In *Fault Tolerant Computing, Proceedings of Annual Symposium on IEEE*. 250–259.

D.J. Sorin, M.M.K. Martin, M.D. Hill, and D.A. Wood. 2002. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. 123–134.

Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. *Scheduling sporadic and aperiodic events in a hard real-time system*. Technical Report. DTIC Document.

Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. 2005. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA '05)*. IEEE Computer Society, Washington, DC, USA, 520–531.

Hongbin Sun, Pengju Ren, Nanning Zheng, Tong Zhang, and Tao Li. 2011. Architecting high-performance energy-efficient soft error resilient cache under 3D integration technology. *Microprocessors and Microsystems* 35, 4 (2011), 371–381.

Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. 2000. Slipstream processors: improving both performance and fault tolerance. In *ACM SIGARCH Computer Architecture News*, Vol. 28. 257–268.

Sumant Tambe. 2010. *Model-driven fault-tolerance provisioning for component-based distributed real-time embedded systems*. Ph.D. Dissertation. Vanderbilt University.

Olivier Temam. 2012. A defect-tolerant accelerator for emerging high-performance applications. *ACM SIGARCH Computer Architecture News* 40, 3 (2012), 356–367.

James E Tomayko. 1986. Lessons learned in creating spacecraft computer systems: Implications for using Ada (R) for the space station. (1986).

Dean M Tullsen, Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, and Rebecca L Stamm. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ACM SIGARCH Computer Architecture News*, Vol. 24. ACM, 191–202.

Shyamsundar Venkataraman, Rui Santos, Akash Kumar, and Jasper Kuijsten. 2015. Hardware task migration module for improved fault tolerance and predictability. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. IEEE, 197–202.

John Von Neumann. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies* 34 (1956), 43–98.

Nicholas J Wang and Sanjay J Patel. 2006. ReStore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on* 3, 3 (2006), 188–201.

Kun-Lung Wu and W Kent Fuchs. 1990. Recoverable distributed shared virtual memory. *Computers, IEEE Transactions on* 39, 4 (1990), 460–469.

Kun-Lung Wu, W Kent Fuchs, and Janak H Patel. 1990. Error recovery in shared memory multiprocessors using private caches. *Parallel and Distributed Systems, IEEE Transactions on* 1, 2 (1990), 231–240.

Jun Yan and Wei Zhang. 2005. Compiler-guided register reliability improvement against soft errors. In *Proceedings of the 5th ACM international conference on Embedded software*. ACM, 203–209.