J Electron Test (2013) 29:159–175
DOI 10.1007/s10836-013-5367-y

Self-Adaptive Fault Tolerance in Multi-/Many-Core Systems

Cristiana Bolchini · Matteo Carminati · Antonio Miele

Received: 2 October 2012 / Accepted: 7 March 2013 / Published online: 17 April 2013

Abstract This paper presents a novel approach to the design of multi-/many-core systems with an adaptive level of reliability. The approach defines a layer at the operating system level that achieves fault detection/tolerance/diagnosis properties by means of thread replication and re-execution mechanisms. The layer applies the most convenient hardening mechanism to achieve the desired trade-off between reliability and performance by adapting at run-time to the changes of the working scenario. The proposed strategy has been applied in a set of experimental sessions considering a real-world parallel application, to evaluate its benefits on the final system with respect to various strategies selected at design time.

Keywords Tunable fault tolerance · Adaptive systems · Multi-/many-core architectures

1 Introduction

The trend of building new complex systems by integrating low-cost, inherently unreliable Commercial Off-The-Shelf (COTS) components is one of today's challenges in the design, analysis and development of systems exhibiting a certain degree of dependability [11]. In fact, the last decade has seen the complexity of electronic systems growing faster and faster, thanks to the decrease of the components' size and cost. However, the use of low-cost execution resources to achieve high performance makes modern electronic devices more and more unreliable, because of the increasing susceptibility of such components to faults. In fact, the number of hard faults, as well as soft ones, is growing due mainly to the shrinking of components themselves, to the variations in the manufacturing process and to the exposition of devices to radiations and noise fluctuations. Radiations, in particular, are the cause of transient faults and have also permanent effects [7]; they are particularly frequent in space, but are becoming an issue also at ground level [18], thus requiring some degree of reliability also for devices used in everyday appliances, and not only in typical safety-critical environments [27]. Therefore, when adopting the COTS-based design approach for the realization of modern and pervasive electronic systems, *reliability* has become one of the main optimization goals, together with *performance*. Nevertheless, in non-critical environments, reliability must be leveraged in order not to introduce too high costs, associated with not so stringent requirements; moreover, in many situations, the need for reliability may change during the activity depending on the specific working scenario. For these reasons, we claim that there is a need for a new way to dynamically "tune" fault management properties based on the working scenario, thus finding a satisfying trade-off between benefits and costs at run-time.

The work presented in this paper aims at introducing a novel approach in the design of complex systems with *self-adaptive reliability*, here targeted to multi-/many-core systems, and acting at the thread scheduling level. The adoption of fault management strategies at such abstraction level has been preliminary investigated in [5]. In the scenario

Responsible Editor: D. Gizopoulos

C. Bolchini · M. Carminati · A. Miele (✉)
Dipartimento di Elettronica, Informatica e Bioingegneria,
Politecnico di Milano,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
e-mail: miele@elet.polimi.it

C. Bolchini
e-mail: bolchini@elet.polimi.it

M. Carminati
e-mail: mcarminati@elet.polimi.it

of a multi-/many-core architecture, the authors proposed a reliable dynamic scheduler, describing a unique hardening technique, but focusing their attention on a broader spectrum of issues, including diagnosis, components aging and more. In this work, we propose a fault management layer at the operating system level, implementing a strategy for dynamically adapting the reliability level. The layer lies on top a revised version of the previously presented scheduler [5], and introduces new elements to define the overall *adaptable* reliable system. The performance/reliability trade-off is evaluated at run-time and the most convenient fault management technique, according to globally defined metrics, is chosen, while not considering, at the moment, other issues.

Even if in this paper we focus on the multi-/many-core scenario, the approach we propose is a general one. More precisely, the methodology we envision is suited for application scenarios where the dependability requirements need to be enforced only in specific working conditions (e.g., when an emergency situation arises) and/or for limited time-windows. In these cases, the system can adapt its behavior and expose robust properties with possibly limited performance, or the other way around. This is mainly due to the fact that performance and reliability are usually conflicting goals. To balance this trade-off, the methodology can act both on the hardware-level (e.g., FPGA-based systems, on which dynamic partial reconfiguration can be exploited to vary the required reliability of a hardware component) and on the software-level (as the multi-core scenario proposed in this paper), based on *i)* how hardening techniques can be applied and *ii)* how the tuning of the hardening is managed. Common stages for the methodology's application in different scenarios can be identified, as it will be pointed out in the following sections.

The rest of the paper is organized as follows. A description of the related work is presented in Section 2, while Section 3 investigates the models for the architecture, the application and the fault at the basis of the proposed methodology. Section 4 presents the proposed Fault Management Layer, and, then, the details of the adaptation engine and the Observe-Decide-Act control loop the engine is based on are discussed in Section 5. The results of the performed experiments are listed and commented in Section 6. Finally, Section 7 wraps up the authors' conclusions.

2 Related Work

Self-adaptation is a term that is becoming more and more popular in these recent years, and in particular, *self-adaptiveness* is the property at the top of a complex self-* taxonomy proposed and defined in [12, 20]. To let a system benefit from these properties, a new system design paradigm must be introduced: the *autonomic control loop* [9]. Practical implementations of this control loop are the *self-adaptation control loop* [20], the *MAPE-K* loop, and the *ODA* loop [12], the last one used as a reference for our work due to its clearness and simplicity.

The autonomic control loop paradigm has been implemented in many research fields with promising results; in particular, various approaches, as Angstrom Project [24], Barrelfish [4], and K42 by IBM [3], focus their attention on enhancing operating systems with self-* capabilities through the introduction of an autonomic control loop. Their goal is to improve the system performance in terms of throughput or execution time and, sometimes, balance performance with power consumption, by applying self-adaptation to different aspects of the system (e.g., process scheduling or memory access management).

When considering reliability issues, several approaches for multi-core systems have been proposed in the last years. A comprehensive survey discussing, among the others, solutions based on redundant execution to achieve fault detection/tolerance, is presented in [8]. Most of these works (e.g., [2, 14, 17]) present approaches based on application-level replication, re-execution or checkpointing. However, they adopt a single hardening technique, thus representing a rigid solution which is not able to adapt to the possible changes in the environment or in the system. Finally, they usually consider a single threaded application, an assumption that does not fit today data-intensive applications. Moreover, a part of them (e.g., [2]) requires a custom architectural support to manage the replicated threads and the checking/voting activity.

Few approaches present a limited adaptability to the evolving scenario. For instance, in [2, 14], two different strategies for dynamically coupling processors are presented, aimed at re-configuring the system architecture after the identification of a permanently faulty core. The approach proposed in [10] features a thread remapping mechanism for recovering from permanent faults; however the recovery schemes are defined at design time, thus offering a limited number of recoveries. An on-line approach for task remapping has been proposed in [16], to deal with the occurrence of permanent processor failures. The authors propose a run-time manager in charge of making decisions concerning the adaptation of the system to changing resource availability. In particular, by means of online task remapping heuristics, it identifies the most convenient resource on which to remap the task. Although the proposal exposes dynamic adaptability characteristics, they are exploited for the decision of the best remapping candidate, with respect to a static pre-computed solution, whereas there is no dynamic tuning of the achieved reliability.

The approach presented in [26] offers the possibility to switch on and off the application duplication strategy depending on the specified requirement, that may be

performance or reliability. An interesting adaptive approach is proposed in [6]: it is based on a set of simple rules for changing the checkpoint schema at run-time according to the architecture's health and current configuration aiming at optimizing the performance. Finally, in our previous work [5], we proposed an adaptive approach for multi-/many-core systems executing parallel applications. The approach applies a single fault tolerance strategy and tries to perform a load balancing of the threads on the available processing units to optimize the performance while fulfilling reliability requirements. Moreover, the approach is able to adapt in case a unit is switched off after the detection of a permanent failure.

In most of the existing run-time adaptable approaches for achieving a dependable and high-performance system, the overall goal is to achieve dependability while minimizing performance degradation, by adapting the execution of the application or the hardening to a changing platforms, affected by faults. Therefore, most efforts are devoted to determine at run-time, which new *system configuration* can provide the best performance, with a modified (partially faulty) architecture. No solution has yet tried to address the issue of dynamically adapting and leveraging the achieved level of dependability, according to the evolving modified architecture, as faults occur, in order to still balance the dependability/performance trade-off.

3 Models Definition

In this section we introduce the models adopted in the proposed approach; in particular, we will characterize the multi-/many-core architecture, the applications, and the possible faults affecting the system.

3.1 Architecture Model

The reference architecture adopted in this work is a multi-/many-core computing fabric, highly modular and configurable, devoted to the acceleration of parallel multi-threaded data-intensive applications. The architecture, shown in Fig. 1, is organized in basic tiles, each one consisting of a multiprocessor system, interconnected to each other and to the I/O interface by means of a global communication channel, generally a Network-on-Chip (NoC). The platform activity is coordinated by a fabric controller that dispatches the applications to the various tiles.

The basic tile, shown in Fig. 2, is composed by a set of processing cores interconnected through a local communication channel, such as a NoC or a crossbar, to a shared memory. The tile contains also a controller core, provided with a (minimal) operating system: it functions as an interface with the rest of the platform and performs the boot of

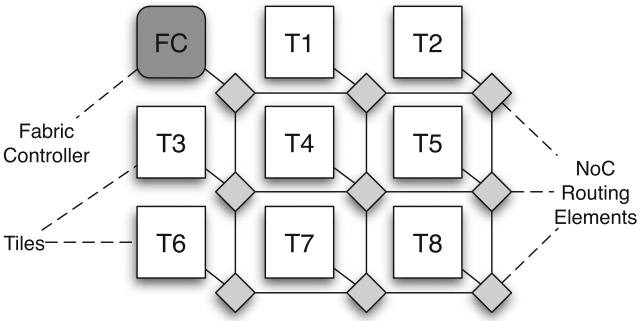


Fig. 1 Many-core architecture

the issued applications. The controller manages thread synchronization, by executing the thread primitives (e.g., thread create or join) and scheduling, by using the classical First-In-First-Out (FIFO) policy. Finally, threads communication is implemented by means of the shared memory.

Such a model is implemented by many real examples, such as the ST/CEA P2012 platform [21] or the Teraflux one [22]. Nevertheless, the presented approach is designed not to be limited by a specific architectural instance, but to be as general as possible. Moreover, the work will be presented considering a single-tile architecture, while leaving the extension of the approach to a multi-tile platform as a future work.

3.2 Application Model

In data-intensive computing environments it is common to consider parallel multi-threaded applications as sample applications. The programming model commonly adopted for such applications is the **fork-join** model, used for instance by OpenMP [23]. Within this programming model, a program is composed by a master thread that may fork to create child threads; when it does, the master is blocked by means of a barrier mechanism waiting for the termination of all its children and only when this occurs, it resumes its execution. In turn, each child thread can fork to create

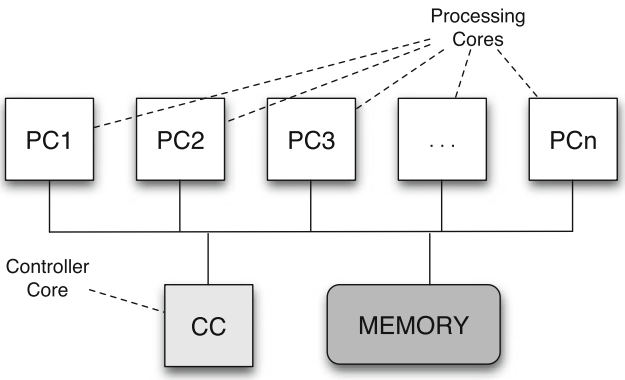


Fig. 2 Basic tile architecture

other threads and will be blocked until their termination. Such an application can be represented by means of a direct acyclic graph, as the example shown in Fig. 3. Each node in this **task-graph** represents the execution of a segment of the program code (a thread or a part of it). More precisely, the fork-join graph is composed by four different types of nodes (we will refer to them also as **tasks**), characterized on the basis of the topology of the graph and the thread primitive called in the associated code segment. They are discussed in the next paragraphs.

- **Elaboration node**. This is the simplest type of node (E nodes in Fig. 3), representing a single thread that executes some computation without creating any child thread. This thread may be created by a fork primitive.
- **Fork node**. As its name suggests, this node is terminated by a fork primitive requesting the creation of a certain number of new threads. In the example task-graph in Fig. 3 it is represented by the F node, having a single incoming arrow and multiple output arrows.
- **Join node**. Dually, this node represents the resume of a thread after the termination of its child threads; node J in Fig. 3 is an example. Multiple arrows are expected to enter this node, while only one exits it.
- **Join_fork node**. It is the last possible type of node, referred to as FJ node, and is bounded by the barrier waiting children termination and the fork primitive to create a new set of children. In this case, multiple arrows both enter and exit the node, as shown in the example in Fig. 4.

For the sake of completeness, it is worth noting that a fork node always generates child tasks belonging to different threads, while the join node is part of the same thread the fork node belongs to. Hence, in our example, F node and J node belong both to the same master thread, which generates the two child threads containing the E nodes.

In addition to this classification, we can also annotate each node in the task-graph with the amount of produced data for the final result. In fact, as the example in Fig. 3 shows, in the considered applications, data processing is

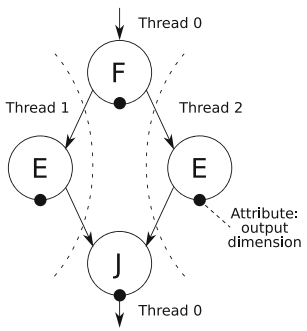


Fig. 3 A simple application represented by a task-graph

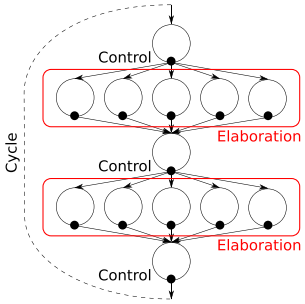


Fig. 4 Elaboration sections are the ones surrounded by rectangles, the remainder are control sections

split into several parallel elaboration tasks, while the other nodes have a synchronization and control functionality. Moreover, an application may be composed by a sequence of stages, divided by a join_fork node, as shown in Fig. 4. In this case, the first stage produces intermediate results taken in input by the second stage.

Finally, applications usually running on many-core architectures perform stream processing, such as audio or video elaborations. This means that the overall algorithm represented by the task-graph is continuously repeated in a cyclic fashion in order to process each received data chunk, shown by the dashed edge labeled cycle in Fig. 4.

3.3 Fault Model

Given the abstraction level adopted in the characterization of the architecture and applications, the considered fault model tries to capture the effect of physical faults in a processing core executing the application's threads. The fault model is referred to as **processor failure** and it may be caused by transient, permanent or intermittent faults. In particular, when a fault affects one processing core, the tasks executed on such core will exhibit an incorrect behavior by either

i) producing an erroneous result, or
ii) crashing, or
iii) leading to an infinite execution loop.

Moreover we assume the processor failure to be **single**, that means that, at each time instant, only one failure can affect the architecture (independently from the number of physical faults causing it), and a subsequent failure will occur only after an amount of time that allows the detection of the previous one. This is a commonly adopted assumption, that does not impose particular restrictions. In fact, the cardinality of the physical faults is not limited, but rather the area of their impact is restricted to a single tile of the architecture. In case of a transient fault, the task being executed on the affected processing core will be the only task to exhibit an incorrect behavior. In the case of a permanent

or intermittent fault, potentially more than one task will exhibit an incorrect behavior, that are all the tasks running on the processing unit where the fault is activated.

Fault effects, if not detected and mitigated, can propagate to the subsequent tasks. In particular, with reference to the above-identified scenarios, in case i), the erroneous result of a task can be propagated to the subsequent ones (see Fig. 5), while in the other two–namely ii) and iii)–, the blocked or crashed task will block the execution of the rest of the application (see Fig. 6). In fact, if a fork task is not completed, the child thread will not be created; while if any other task is not completed, the subsequent join task will not be released by the barrier.

For the fault model to hold, we assume the architecture is provided with some additional features and mechanisms, usually adopted or investigated in other solutions (similarly to [5, 25]): (a) the control unit is fault tolerant by design, (b) threads data are organized in separate segments of the memory and no thread can access the segment of any other thread and, finally, (c) watchdogs are used for killing tasks in infinite execution loop.

4 Fault Management Layer

The adaptive mechanisms that we envision are implemented by a fault management layer (FM Layer) introduced on top of the architecture and the operating system, as shown in Fig. 7. The main goal of this layer is to harden the execution of the issued applications and, at the same time, to balance reliability and performance according to a set of conditions that may evolve in time. This new layer borrows some concepts from the one presented in [5], as far as hardening strategies are concerned, although we here generalize the approach in this respect, while focusing the attention of the new self-adaptive features.

The architecture of the fault management layer is now introduced, while the adaptive engine will be discussed in the next section. Do note that in this paper we focus on the proposal of the adaptive fault tolerant strategy from a functional point of view. Therefore, we present some details on the internals of the fault tolerant layer, but we will not delve into its implementation-specific aspects. Since these aspects are strictly related to the specific target architecture and operating system, we leave them as a future work related to the presentation of a particular architectural platform.

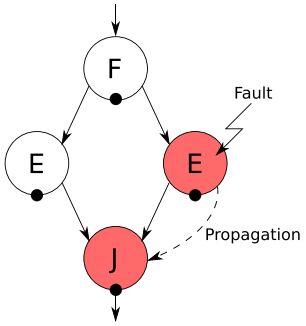


Fig. 5 Erroneous result due to a fault propagates to the subsequent join task

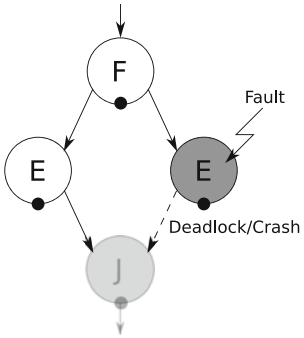


Fig. 6 The crashed thread blocks the execution of the subsequent join task

4.1 Fault Management Mechanisms

The fault management layer introduces reliability properties (fault detection or fault tolerance) in the application execution, by means of mechanisms based on replication and re-execution of the overall applications or some of its threads. These mechanisms are also used for performing diagnosis activities in order to identify suspect cases of permanently damaged processing cores. In particular, the following techniques are supported at present.

- **Duplication with Comparison (DWC).** As shown in Fig. 8, the technique guarantees the fault detection property by creating a replica of the application and by comparing the outputs. In particular, a checker task is issued at the end of each node of the application's task-graph to identify discrepancies in the (intermediate) results.
- **Triplication (TMR).** This technique creates two replicas of the original application, so to have three results

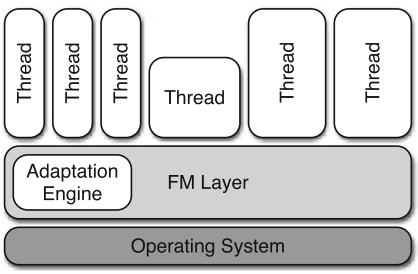


Fig. 7 System structure overview: the FM layer acts between the applications and the OS

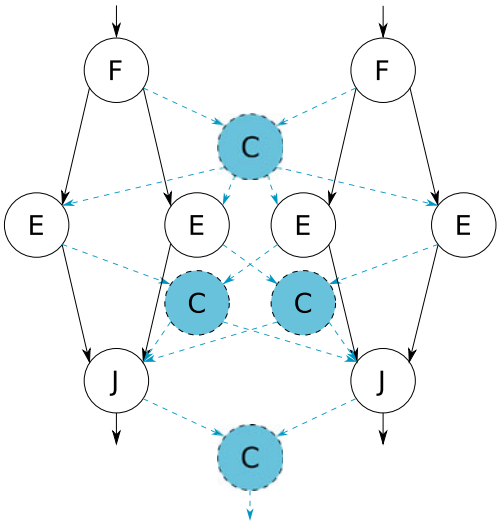


Fig. 8 Duplication with Comparison (DWC) technique applied to all the tasks of the sample application of Fig. 3

to be voted by specific 2-of-3 majority voter tasks that mitigate the possible occurred faults (Fig. 9). Besides the fault tolerance property, the technique is also able to achieve fault diagnosis features, by identifying the core producing the erroneous mismatching value.

– **Duplication with Comparison and Re-execution (DWCR).** Similarly to the DWC technique, the original application is duplicated to have the possibility to detect possible errors by comparing two results by means of a checker task. If an error is detected, a third replica of the task is created and executed: in this way, a voter task is able to identify the correct result. Figure 10 shows an example. This technique provides the fault tolerance property, and may provide fault diagnosis features, i.e., it is able to identify the core that caused the faulty result. This technique is characterized by a limited overhead for achieving the fault tolerance property, because the third replica is used and scheduled only after a problem has been detected. As a result the technique exhibits a time-redundancy style of mitigation causing a delay in computing the correct results, when compared against the previous techniques. In fact, the technique aims at limiting the overheads in the most common case of fault-free execution, while incurring in an additional timing overhead when a problem occurs.

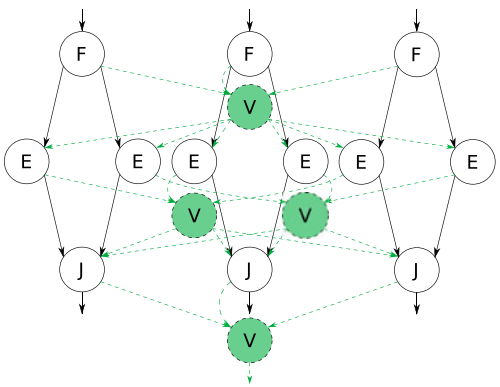


Fig. 9 Triplication (TMR) technique applied to all the tasks of the sample application of Fig. 3

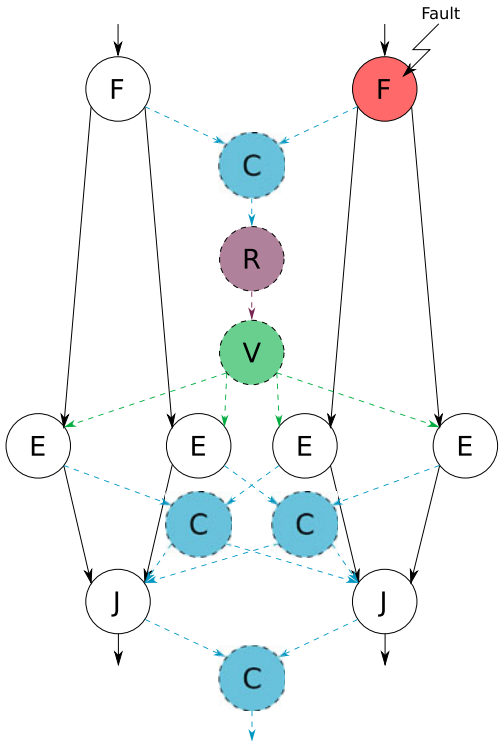


Fig. 10 Duplication with Comparison and Re-execution (DWCR) technique applied to all the tasks of the sample application of Fig. 3

Since we need to guarantee the correctness of the voters and checkers results, these tasks must be executed on the controller core, which is the only unit hardened by-design. Moreover, their duration is variable and, in particular, it depends on the amount of data they are testing (as discussed in Section 3.2, these data are annotated on the various tasks in the application model). Finally, when a join node has only a synchronization functionality and does not generate any output data, the voter will check only the correct termination of its replicas.

Do note that the architecture uses a communication model based on a shared memory. Therefore, there is no actual transmission to the control core of the data to be compared/voted, that may cause a communication congestion. At the end of a task, the processing unit will signal the termination and, during the execution of the voter/checker task, the controller core will access the data in the memory region where the task wrote the result.

4.2 Layer Internals

In order to perform the additional hardening activities for enabling self-adaptive reliability, similarly to [13], the layer wraps a set of system-calls to the underlying operating system which are: `thread_create`, `thread_join` and `thread_exit`, devoted to thread start, termination and synchronization, respectively, and `_start` and `_exit`, for the start and termination of the main application thread.

Let us explain the basic mechanism through a simple example. Consider a single-threaded application requiring fault tolerance properties, on which TMR is applied; when it is started, the layer intercepts the `_start` call and creates two additional thread replicas. In turn, the tile scheduler will execute the replicas, and, after their termination identified by trapping the `_exit` call, the layer creates and executes a task that votes the obtained results. To implement the exemplified behavior, the layer collects the information of the threads; in particular, it keeps track of the corresponding replicas of the same thread, to issue voting/checking tasks upon their termination. The FIFO scheduler uses a number of queues equal to the number of generated replicas and adopts a round-robin policy for selecting the current queue from which pops the task to be executed; in this way it guarantees a balancing in the execution of the various replicas. Moreover, to avoid that the use of permanently failed units for executing more than a replica mines the effectiveness of the hardening techniques, the scheduler maps corresponding replicas on different cores.

The selection of the fault management mechanism to be applied is the most critical activity, because each strategy provides specific reliability properties and presents a different overhead on the performance due to the number of generated replicas. Thus, an adaptation engine has been implemented for performing this decision, and is presented in the following section.

5 Adaptation Engine

The innovative aspect of the proposed approach is the adaptation engine implemented in the fault management layer, able to autonomously act on some parameters, called **knobs**, for modifying the behavior of the system according to the evolving environment. More precisely, these knobs act on the fault management mechanisms and the scheduler behavior. The dynamism of the environment consists in changes in the kind of issued applications, the application requirements (performance vs. reliability), health status of the processing core within the architecture, and the system workload.

To achieve such adaptability, the system needs to monitor itself and its context, discern significant changes, determine how to react, and execute decisions. Such a behavior has been implemented in the system by means of the so called Observe–Decide–Act (ODA) control loop [12], shown in Fig. 11. The ODA loop is divided into three simple and sharply distinct stages. The *observe* phase consists in sensing the status of the system and, in particular, collecting execution data for computing a set of performance- and reliability-related metrics. Then, the *decide* phase is performed by taking into account the measured metrics and a high-level goal specified as a requirement on the application (performance vs. reliability). The knowledge of the goal guides the adaptation engine in making a suitable decision on how to execute the applications. Finally, once the decision has been taken, it is put into practice in the *act* phase through the actuators, which modify the system's knobs in order to alter its behavior. The behavior of the system is sensed again in the *observe* phase and the control loop is restarted. The implementation of the three phases is discussed in details in the following subsections.

5.1 Observe

A crucial step in designing the observe phase is the definition of the quantities to be sensed, that are the **metrics**. They provide a description of the system's current status as much complete and concise as possible to be useful to make decisions in the subsequent phase. The following metrics have been identified in order to observe the two main aspects of interest in the considered multi-/many-core scenario: the performance, since such architectures are used as intensive data processing applications, and the reliability, due to the increasing failure trend.

For estimating the performance, the classical metrics measuring the execution time are adopted: in particular, the **average execution time** and the **throughput**. In an application modeled through a task-graph (as the example in Fig. 3) the execution time of a single application cycle is measured by evaluating the elapsed time between the beginning of the

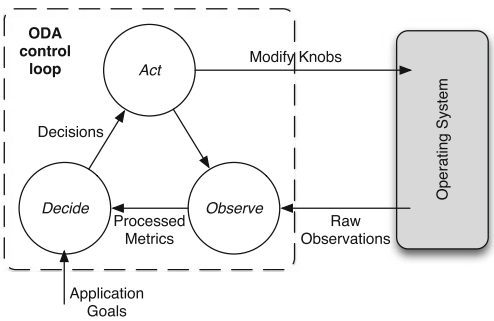


Fig. 11 The Observe–Decide–Act control loop

execution of the first fork node and the end of the execution of the last join node:

$$exec\_time_{overall} = t_{end} - t_{start} \quad (1)$$

Then, the average execution time is computed on a window of the last n executed cycles.

An alternative metric is the throughput, that is defined as the average quantity of data processed by the system in a time unit. To evaluate this metric, the adaptive engine considers a window of the last n executed cycles of the application and computes the ratio between the amount of produced data and the overall execution time for that window:

$$throughput = \frac{n \cdot d_{out}}{t_{end_n} - t_{start_1}} \quad (2)$$

where d_{out} is the amount of data produced during a single cycle and the cycles are numbered from 1 to n.

When referring to the reliability, the biggest issue of the considered data-intensive computing applications is related to the amount of errors they may experience. Thus, the defined reliability-oriented metric measured by the adaptive environment is the **detected error ratio**. In particular, the metric is defined as the percentage of erroneous data on the results detected during the execution by the checker/voter tasks introduced by the fault management techniques. Moreover, it is computed on a window of the last n cycles of the application execution. When considering the application model, we can estimate the error ratio by quantifying the portions of the results produced by faulty tasks. Do note that in quantifying the error ratio, fault propagation and task crashes have to be taken into account too.

Not all the hardening strategies are able to mitigate the detected errors: in fact, DWC is not able to do it and in some scenarios other techniques may potentially fail, as discussed in the next section. When an error is not mitigated, it will be observed on the final output of the application. Therefore, we also defined the **not-mitigated error ratio** which computes this percentage of errors.

5.2 Act

The second step, the act phase, is the one where the system directly acts on a set of parameters, called **knobs**, according to the choices taken in the decision phase (presented in the next section). In our scenario we can identify several knobs: the selection of the fault management mechanism to be applied, the selective application of the mechanism, the *granularity* at which it is applied, the possibility to activate or deactivate a processing unit or changing its frequency, and so on. In this paper we focus on a subset of them presented in the following; all the other knobs are left for a future investigation.

The first considered knob is **mechanism selection**, that is the choice of the fault management mechanism to be applied on the running application. In Section 4.1 we have presented a set of techniques; as discussed each technique has specific properties (fault detection, fault tolerance and fault diagnosis) and reduces the performance. In particular the performance for the two mechanisms offering fault tolerance is listed in Table 1. TMR can achieve a greater throughput when there is a high number of available processing cores and the system is experiencing a high error ratio: this is due to the fact that resources are better exploited when running the three replicas in parallel; DWCR would not be able to exploit these resources since the third replica would be executed strictly after the first two and a voter will be also executed in addition to the checker. On the other hand, DWCR proved to achieve a greater throughput in a scenario with a low error ratio and a reduced number of resources since the third replica is optionally executed. In the other two cases the performance of the two techniques depends on the specific scenario and cannot be classified a-priori. Experimental evidences of these considerations can be found in Section 6.

The second knob is the **granularity** the mechanism is applied at. The granularity represents the possibility to perform a varying placement strategy of the voter/checker tasks introduced by each fault management mechanism, thus offering another way to tune the performance/reliability trade-off. In this paper we consider only two different granularity levels, even if other intermediate ones could be added:

- **Coarser**: checker/voter tasks (according to the considered mechanism) are added only after the execution of an independent part of the program: after join and fork_join tasks.
- **Finer**: checker/voter tasks are placed after each task in the task-graph.

The concept of granularity can be, in theory, applied to all the hardening techniques. Figure 12 shows an example applying the TMR at the two different levels of granularity on the sample application of Fig 3. It is worth noting that in the coarser level only the final results are checked/voted,

Table 1 Qualitative performance comparison between TMR and DCR techniques

Detected Error Ratio	#Resources	
	low	high
low	TMR < DWCR	TMR ∼ DWCR
high	TMR ∼ DWCR	TMR > DWCR

The comparison refers to the same level of granularity

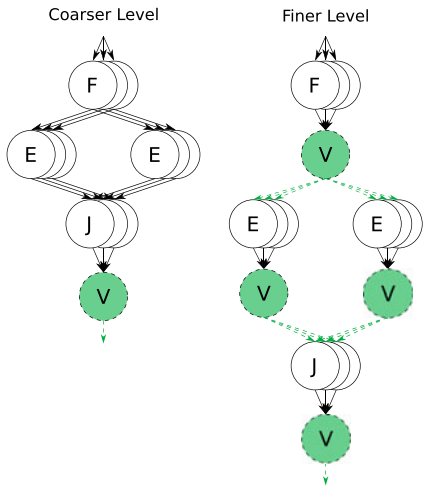


Fig. 12 Application's task–graph hardened at the two different levels: the colored dashed tasks represent the voter nodes added to make fault mitigation possible

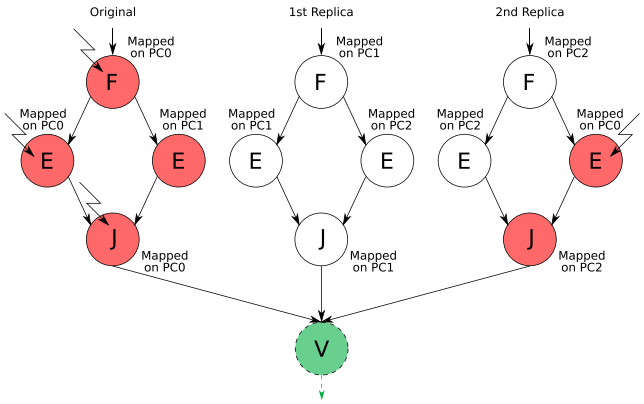


Fig. 13 If processing core PC0 is permanently faulty, the presented mapping of the tasks on three processing cores, PC0, PC1 and PC2, causes TMR to fail

while the possible intermediate results are discarded. Moreover, when the join task has only a synchronization role and has no data as final output, the voter will test the output data of the child elaboration threads.

From a reliability point of view, some combinations of granularity level and fault management mechanism do not offer any advantage. This is the case of the DWC that offers the same features for both configurations; however, as discussed later, the performance of the two configurations depends on the specific working scenario and therefore it is not possible to choose the best configuration a-priori. At the opposite, DWCR and TMR are strategies for which the two levels offer different reliability properties.

When the coarser granularity level is selected, it is not possible to guarantee diagnosis features since error propagation effects would be introduced and the source of the error would not be identifiable. Moreover, in case of permanent faults, mechanisms offering fault tolerant properties cannot guarantee a 100 % coverage. In fact, the mapping constraint of the sibling replicas discussed in Section 4.2 allows a single permanent processor failure to invalidate tasks belonging to different group of replicas before they are voted, as shown in Fig. 13. In order to guarantee a 100 % coverage, the processing cores should be partitioned in a number of groups equal to the number of generated replicas, i.e. three. However, since in general the number of processing units is not multiple of three, this partitioning would imply lower performance. Nevertheless, even if not providing the full error coverage, these mechanisms still provide 100 % fault detection coverage.

The placement of voter and checker tasks after each node, allows DWCR and TMR to achieve a 100 % fault tolerance coverage; moreover, this placement offers also diagnosis capabilities. Table 2 summarizes the reliability aspects presented above.

The granularity level has also an impact on performance. Usually, the selection of the finer granularity level incurs in a higher overhead on the execution latency with respect to the coarser level, because a larger number of checker/voter tasks is introduced. However, in some specific situations the opposite behavior may occur, as we will show in the experimental results (Section 6). More precisely, when the architecture is composed by a small number of processing units and the application generates a large number of parallel threads, the finer granularity level offers the possibility to better schedule on the controller core a larger number of checker/voter tasks comparing the results of the elaboration threads, since the other processing units are overloaded with the execution of the tasks' replicas. Due to this reason, DWC applied at the finer granularity level has not been discarded even if it does not offer any advantage from the reliability point of view. Finally, do note that the considerations drawn in Table 1 hold at each level of granularity and, more precisely, these relations are exacerbated as the level of granularity becomes coarser. In particular, if the system experiences a high detected error ratio, DWCR is highly disadvantageous with respect to TMR since it would require a considerable number of tasks to be re-executed.

The last knob is the **resource activation/deactivation**. As discussed in the description of the fault management techniques, during the execution, the adaptation engine may diagnose a suspected damaged processing core. In this case, the engine can deactivate the processing core in order to further analyze it by means of specific diagnosis tasks. Later, if the result of the accurate analysis is negative, the unit can be reactivated and used again for executing the application.

As a final remark, do note that knobs can be adjusted only between two different iterations of the application execution for allowing the context switch. This is particularly

Table 2 Qualitative evaluation of the described techniques with reference to reliability aspects

Granularity	#Replicas	100 % FD	100 % FT	Diagnosis
Duplication with Comparison (DWC)				
Coarser	2	Yes	No	No
Finer	2	Yes	No	No
Duplication with Comparison and re-Execution (DWCR)				
Coarser	2/3	Yes	No	No
Finer	2/3	Yes	Yes	Yes
Triplication (TMR)				
Coarser	3	Yes	No	No
Finer	3	Yes	Yes	Yes

true for the fault management mechanisms since it is quite complex, even not unfeasible, to adapt internal data and synchronizations related to replicas and checker/voter tasks from a mechanism to another one.

5.3 Decide

The central step in the adaptation engine's control loop evaluates the system's current status on the basis of the sensed metrics acquired in the observe step, and makes a decision on the actions to be taken through the knobs to reach the desired goal, specified as input when the application has been issued.

As discussed in the previous sections, the two conflicting aspects on which the proposed system focuses are performance and reliability; moreover, each one of the considered hardening mechanisms together with the granularity level of its application implies a specific impact on each one of the two aspects: some mechanisms obtain a reduced performance overhead but limited reliability properties, while other ones achieve a high reliability but incurring in a large performance overhead. Thus, the aim of the system is to decide the most suitable mechanism to be adopted in each situation to achieve a good trade-off between the two aspects. In particular, the proposed adaptation engine allows one to specify the **goal** for each issued application, by specifying which of the two aspects has to be considered as **constraints**; consequently, the system will try to adapt its behavior to optimize the remaining dimension. The adaptation is even more necessary if we consider that the system conditions can evolve: in particular, different applications with different goals may be issued, and the architecture may change due to the deactivation of some faulty units.

The goal is specified as a threshold on the value of the metrics representing the aspect to be controlled (not-mitigated/detected error ratio for the reliability, or average execution time/throughput for the performance). Do note that the adaptation engine will try to fulfill the specified constraint, however, there is no guarantee that it will be satisfied *completely*; in fact, the various decisions are made at run-time, and therefore only at that time it will be possible to know whether the goal can be reached.

The implemented decision mechanism is a simple rule–based system. The specified goal is used by the system, first, to identify the constrained variable and the free one, and then to set a threshold used to take a decision. In particular, the idea is to start the execution by choosing, among the hardening mechanisms, the one having, in theory, the best expected result for the unconstrained dimension and to analyze the effects on the constraint variable. The value specified as a constraint on the application is compared with the current value of the metrics: if the constraint is met, the mechanism currently in use does not need to be changed; otherwise, a more appropriate technique is selected. This first threshold is strict and a mechanism change occurs every time the threshold is reached. Other thresholds can be determined at run-time by estimating the effects of the available mechanisms on the current status of the system.

We discuss here the adaptation engine's design, when setting the reliability as the constrained dimension. The goal is specified together with the input task-graph as a not-mitigated error ratio R_1 not to be reached. The adaptation engine modeled with a hierarchical FSM is shown in Fig. 14. The initial macro-state, labeled *Fault Detection*, forces the system to apply the DWC mechanism, that is the one that has the smallest impact on performance and allows to perfectly keep track of the value of the current detected error ratio. As discussed, DWC will only detect the errors without mitigating them. In particular, the first step is a tuning phase that aims at evaluating the performance achieved by DWC when applied at finer and coarser granularity levels: each one of the two levels is tested for one cycle. Then, the engine evolves in a second state where DWC is applied at the granularity level that obtained better performance.

If the not-mitigated error ratio threshold R_1 is reached, then a mechanism enforcing the required fault coverage needs to be applied. This will have the effect of reducing the error ratio, while penalizing performance.

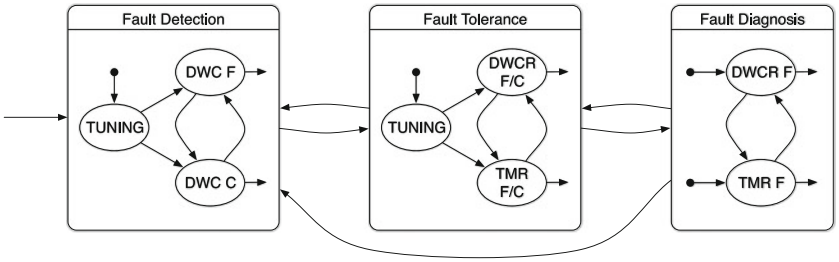


Fig. 14 A FSM representation of the decision process in the case reliability is the constrained dimension

Therefore, the adaptation engine evolves in the *Fault Tolerance* macro-state. In this macro-state, the adaptation engine first performs a tuning phase for profiling the two hardening mechanisms. In particular, for each mechanism both the granularity levels are evaluated, and the less convenient one is discarded; then, the engine selects the mechanism offering higher performance and consequently evolves in the related sub-state. Do note that the performance of DWCR depends on the number of required re-executions while the TMR one is almost constant. For this reason, the adaptation stage will switch among the two mechanisms (at the granularity level identified during the tuning phase) according to the performance monitored during the execution.

If the detected error ratio does not decrease in a given executions' window, it means that some permanent failures may affect the architecture. Therefore, to perform fault diagnosis, the adaptation engine evolves to the *fault diagnosis* macro-stage, which behaves similarly to the previous one but exploiting only the finer granularity level. Note that the tuning step is not necessary since the techniques have been already tested in the previous macro-state. During the diagnosis activity, based on the analysis of the fault detection on each processing core, if a unit is determined to be faulty, it is switched off, and then the adaptation engine evolves to the DWC state.

Finally, if the detected error ratio decreases, the adaptation engine evolves from a macro-state to the previous one.

The FSM derived for the case of setting reliability as the constrained dimension can be straightforwardly adapted to the case where performance is the constrained dimension, or the case where a set of batch applications is issued. In particular, the same thresholding mechanism will be used, while the specific actions/techniques applied in each state will be different. Moreover, the described decision phase is clearly extensible with new mechanisms, automatically taken into account during the tuning state.

6 Experimental Results

In this section we present the experimental sessions carried out to demonstrate the effectiveness of an adaptation engine. First, the experimental set-up is illustrated and, later, the results of the experiments are discussed in two different subsections.

6.1 Experimental Set-up

We adopted and enhanced the simulator previously defined for the experimental sessions presented in [5]. The original tool consists of a SystemC transaction level model [1] of the discussed architecture able to simulate the execution of applications modeled in terms of fork-join graphs. Thus, we implemented in the SystemC model all the considered fault management mechanisms, the support for the granularity levels and the discussed adaptation engine.

We considered a set of commonly used parallel applications in the performed experimental sessions. In particular, we present here the results obtained on a specific application aligned with the application scenario we refer to, that is an **edge detector**, an image processing algorithm. We characterized it with a static analysis on the source code for building the task-graphs annotated with the amount of data processed by each task. Moreover, we used an instruction set simulator running a sequential version of the application for estimating the execution times of the various tasks, as in [15]; such execution times contains also bus and memory access latencies. Indeed, as noted in [15], when working at this level of abstraction, interactions among tasks (e.g., resources conflicts) are not of interest since they are later considered in more accurate lower-level simulations. In particular we used the ReSP simulation environment [19] for simulating an ARM7 unit working at $333MHz$ connected to a memory through a bus, both with a transmission latency equal to $10ns/word$. Finally, we also estimated the execution times of the voters and checkers for the different amount of data to be compared. The resulting task-graph is shown in Fig. 15, while a subset of the execution times for the voters and checkers are presented in Table 3.

We generated fault lists according to an exponential probability distribution over the experiment duration, since it is well suited for describing transient faults. They were created by varying the failure rate of the distribution in order to stimulate the system in different ways, thus causing mutating environmental conditions. The generated lists obey the single processor failure model assumption. Note

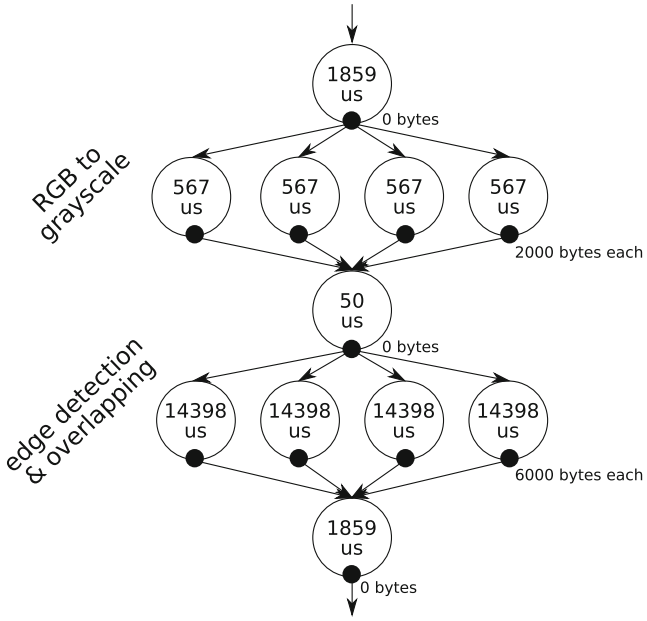


Fig. 15 Task-graph for the edge detector application

that, the considered fault probabilities have been created with a high frequency w.r.t. fault occurrence in the real world; the aim is to perform a sort of *accelerated* experiment to better highlight the capability of the engine to adapt to the environment stimuli. Therefore, we expect the system to behave in the same way in the real world scenario, although on a longer time window. It is worth noting that the accelerated experiment requires the size of the window for the metric computation to be shrunk to make the engine more reactive to the higher external stimuli.

6.2 First Experimental Session

The goal of the first experimental session is to show how each single fault management mechanism behaves differently according to the specific scenario, offering a different performance/reliability trade-off. Moreover, the adaptation engine is able to select, during the execution, the most suitable fault management mechanism at each time, outperforming the single statically-selected strategies. We considered four different architectures composed of 6, 8, 10 and 12 processing cores, respectively, and we generated three fault lists $F1$, $F2$ and $F3$ with a failure rate (i.e., probability of a single processor failure) equal to 0.0005, 0.001 and 0.003 every $200\mu s$.

Table 3 Execution times for voting and checking various amount of data

# Data (byte)	Checker	Voter
2000 byte	$423\mu s$	$920\mu s$
6000 byte	$1237\mu s$	$2550\mu s$

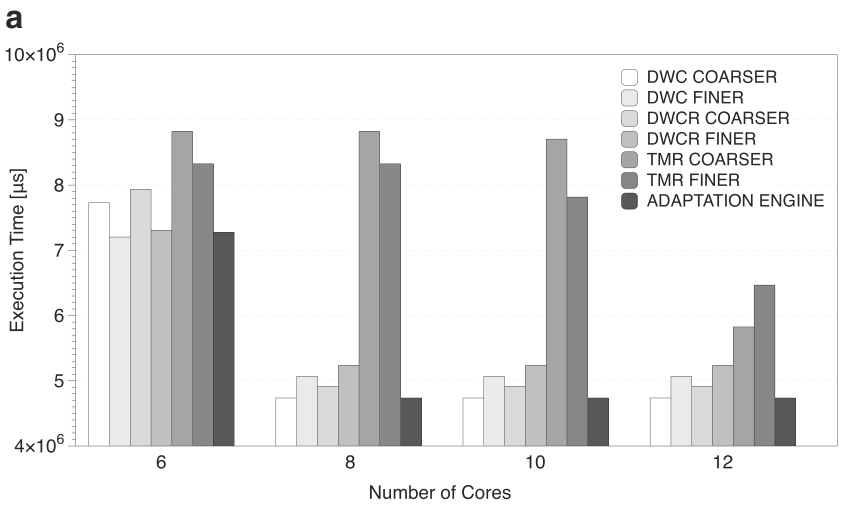
In this experimental session we ran the application hardened with all the available mechanisms, selected at design-time, and with the proposed adaptive engine. Moreover, we tested all the strategies on each pair <architecture, fault list>. The comparison of the various approaches was performed by executing the application for a specific number of cycles (equal to 200), thus mimicking the elaboration of a fixed amount of data. Moreover, for the adaptation engine, we considered a threshold for the not-mitigated error of the 5 % and a window for the metrics computations of 10 cycles.

The results of the experimental session are presented in Fig. 16, where the overall execution times are reported. Each chart presents the results obtained for each fault distribution when adopting various mechanisms on the considered architecture. The average of not-mitigated error ratios are not reported since they confirmed the expected values: 0 % for all fault tolerance strategies (since no permanent fault was injected), less than 5 % for the adaptation engine, while DWC is not able to mitigate any error.

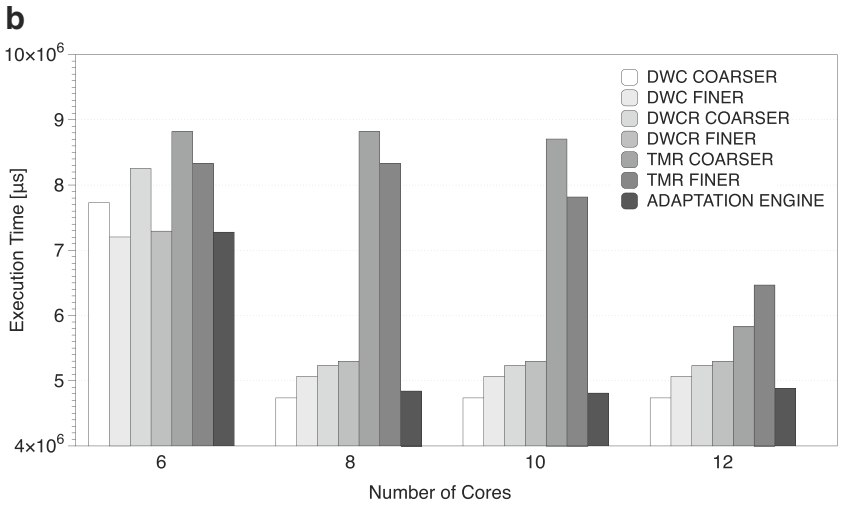
From an accurate analysis of the proposed bar charts we can notice that, as expected, there is not a mechanism that is always prevailing on the other ones, and each of them achieves better performance in a specific scenario. Moreover, we can deduce the following empirical rules. In general, even if coarser granularity level introduces a smaller number of voter/checker tasks, the finer one is preferable when the number of available processing cores is small, as already motivated in Section 5.2. In particular, this can be observed from the results reported in Fig. 16 for the architectures with six cores. A detailed investigation of the scheduling Gantt charts allowed us to highlight the following situation: when the number of cores is limited, voter tasks analyzing intermediate results used in DWC/F do not overload the control core thus achieving an overall better performance than DWC/C, that postpones all results' checking at the end of the application's replicas execution. This phenomenon is exploited by the adaptive engine that thus achieves better performance.

Moreover for DWC and TMR, the break-even point is represented by the value obtained by multiplying the maximum number of parallel threads generated by the application by the number of replicas required by the mechanism (for instance, for the edge detector, 4×3 for TMR and 4×2 for DWC); on the other hand, it is not possible to identify an exact break-even point for DWCR even if the trend is similar. The second consideration is that the performance of the two granularity levels for DWCR depends also on the fault frequency: in fact, when the frequency is high the finer level is more convenient since it allows to re-execute only a reduced portion of the application. A final consideration is that TMR seems not to be convenient in any scenario; however, as shown in the following experiments,

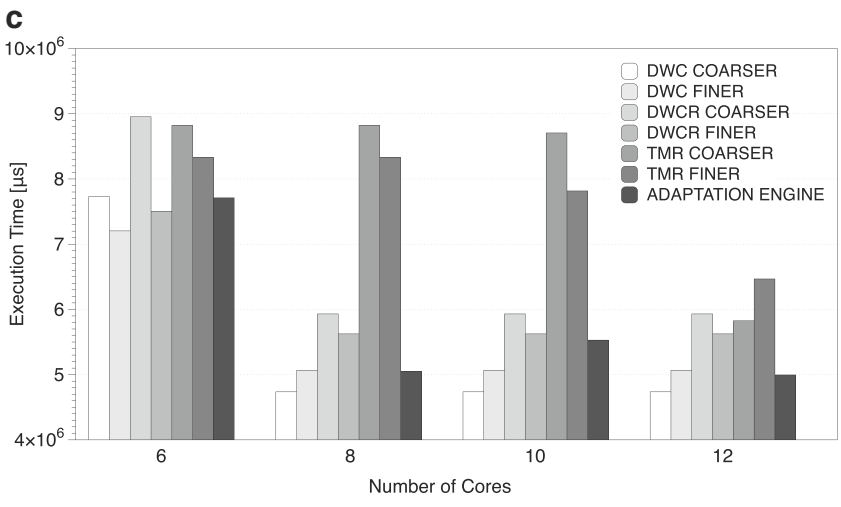
Fig. 16 Overall execution times for the edge detector on the various architectures under the effects by each fault list



Fault list F1.



Fault list F2.



Fault list F3.

it will outperform the other mechanisms when the error frequency is even higher, i.e., when a permanent fault affects the architecture.

When considering the adaptation engine (last bar in each chart), we can notice that it outperforms all the statically selected approaches (DWC should not be considered since it is not able to mitigate the errors thus producing corrupted results). As expected, the engine is able to select the best mechanism in each instant of time, thus maximizing the performance while meeting the specified threshold on the not-mitigated error ratio. The performance improvement on the best of static approaches in each scenario spans from 1 % to 13 %. Only in one scenario the engine exhibits worse performance than the DWCR applied at finer granularity (i.e. on the architecture with 6 cores stimulated by the fault list $F3$); indeed, the limited number of processing units in the considered architecture causes a high latency to execute a single cycle, and, consequently, the engine evolves too slowly to select the appropriate mechanism to mitigate the high frequency of faults.

Even if the scenarios presented so far proved the necessity of a dynamic mechanism selection, they are not the situations in which the proposed engine shows its effectiveness. In fact, the considered fault lists have been generated by means of a constant failure rate and no permanent fault has been injected. Therefore, in a second experimental session, we aimed at showing the adaptation capability of the engine by considering a fault list generated with a variable failure rate and a permanent fault corrupting a single processing core.

6.3 Second Experimental Session

In the second experimental session, we executed the edge detector in a scenario with evolving environmental conditions in terms of fault distribution. For about the first

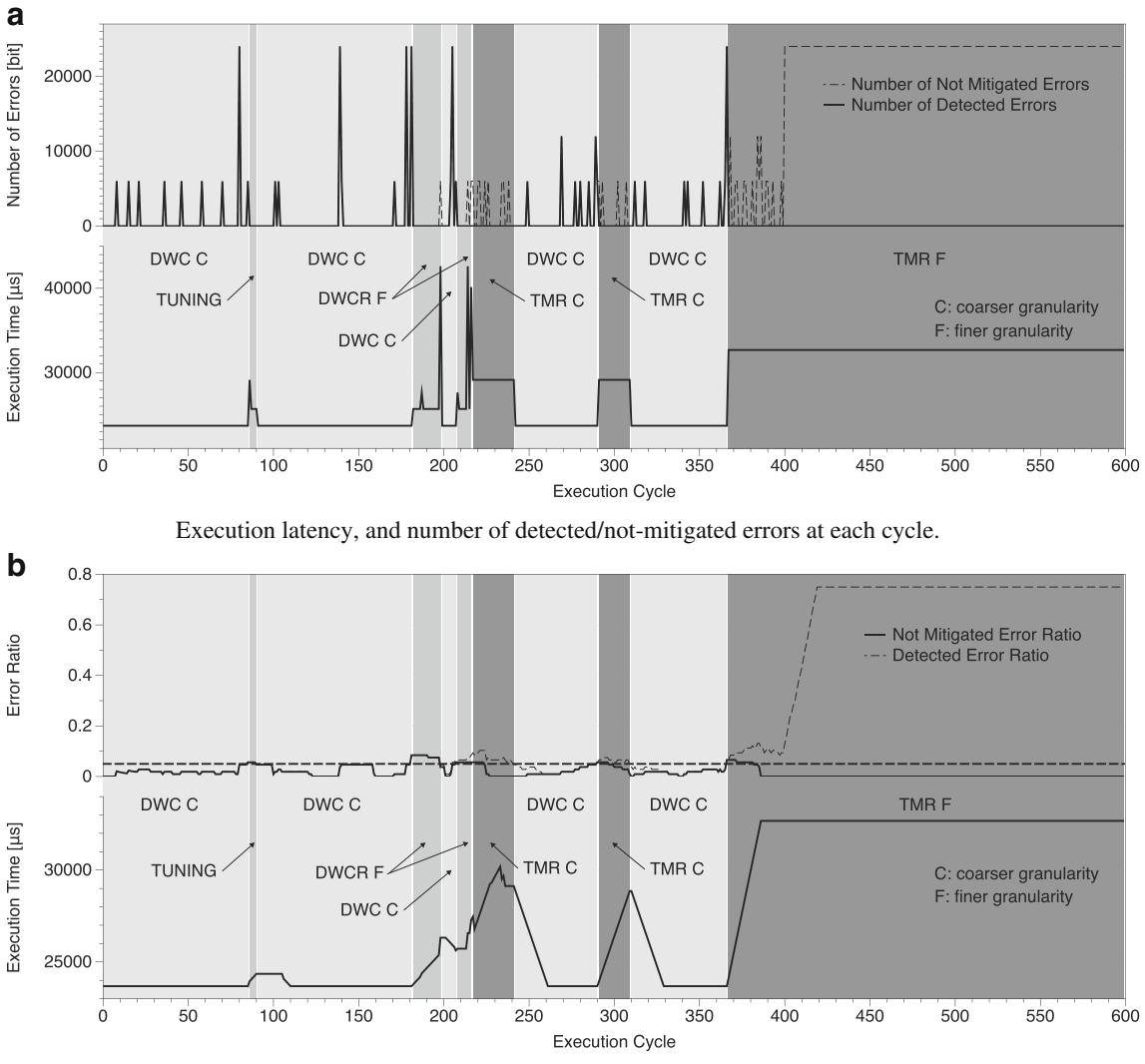


Fig. 17 Metrics computed over the overall experiment execution

4s the system experiences a failure rate equal to 0.001 every $200\mu s$; then, from time 4s to 10s this probability is increased to 0.003 every $200\mu s$: in this way the behavior of the system when a varying failure rate occurs is tested. To make the scenario more complete and complex, at time $10s$ a permanent failure is injected in one of the architecture cores. The edge detector is run for 600 cycles on an architecture made up of 12 cores.

The first plot (Fig. 17a), aims at showing which is the system's behavior perceived from the outside. The top part of the graph shows the number of detected and not-mitigated errors experienced at each cycle by the system. This allows to figure out the role played by the fault mitigation techniques in reducing the number of visible faults. Finally, the gray vertical areas describe the used hardening approach in each time interval.

The second plot (Fig. 17b) shows the internal status of the engine, and in particular presents the values of the metrics on which the adaptation engine makes decisions on the mechanism to select at each execution cycle. On the top graph, the average window values for the detected and not-mitigated error ratios are plot (for this experiment a window value of 20 cycles was chosen), while the dashed horizontal line at value 0.05 represents the reliability constraint specified by the user. In the bottom plot the average window value for the performance is shown. Therefore, the engine selects a specific mechanism according to the current values of the metrics; the selected mechanism is reported by means of gray areas of different intensity.

By looking at all the graphs, it is possible to see how the proposed approach actually allows to adapt the behavior of the considered system to cope with the mutating fault distribution. Indeed, at the beginning of the execution, when the failure rate is relatively low, the engine decides not to mitigate any fault, but only to detect them. When the number of detected errors becomes too high, the engine performs a tuning phase, analyzing the performance of the techniques providing fault tolerance. Thus, in this first phase, DWCR at the finer level is selected to be executed alternatively to DWC at the coarser level. When the failure probability increases, DWCR performance is no more satisfying and TMR at the coarser level becomes the chosen fault tolerant mechanism, to be executed in alternative to DWC. Finally, from about the 370^{th} execution cycle, the engine identifies an even higher error ratio suspecting the presence of a permanent fault and definitively switches to TMR applied at the finer level to perform the fault diagnosis; if the response is positive, the engine will switch-off the damaged unit and will apply again DWC.

As a final note, Fig. 18 compares the throughput achieved by the presented approach with all the other statically selected mechanisms also with respect to the plain execution; on each bar the relative percentage value with respect to the plain execution is reported. As discussed above, DWC applied at both levels obtains best performance but it violates the reliability requirement on the error ratio. Then, when considering the other strategies, the engine obtains an improvement of only the 5 % with respect to the TMR at coarser level; however, this mechanism is not able, in the second part of the experiment, to perform fault diagnosis. Thus, a fair comparison can be performed only with TMR and DWCR applied at finer level since they offer all the discussed reliability properties; in such a scenario, the adaptation engine obtains an improvement of 18 % and 23 % respectively, thus demonstrating its effectiveness. As a final note, the throughput of the plain execution is not considerably higher than hardened ones

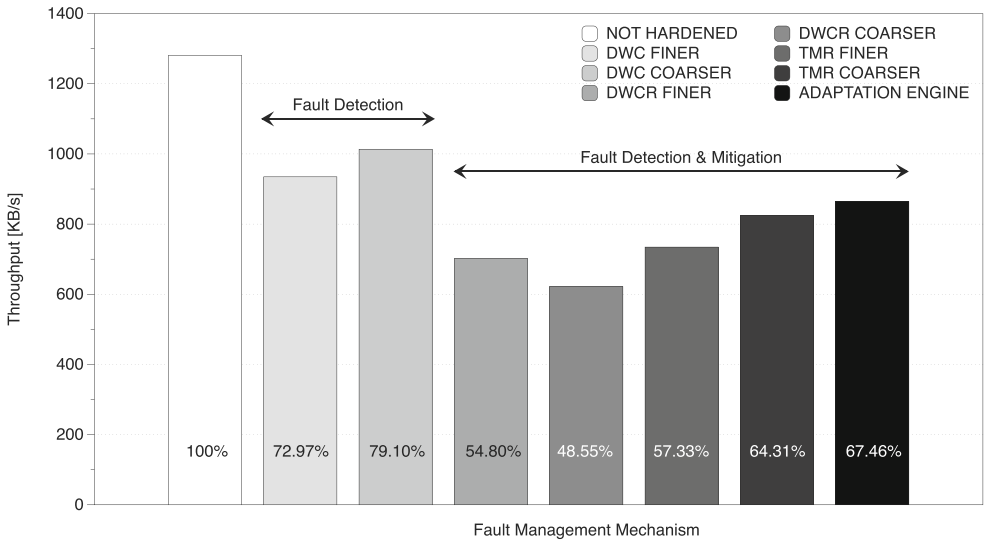


Fig. 18 Throughput for the edge detector on the architecture with 12 processing cores, stimulated with a fault list presenting a variable failure rate and a permanent fault

as it may expect. This is due to the fact that the number of threads generated by each fork is decided at design-time (equal to 4) and therefore it is not possible to fully exploit the architecture capabilities; a possible future development of the approach may consider the dynamic decision of the number of threads as also supported in OpenMP [23].

7 Conclusion

This paper introduces an adaptation engine for fault management in multi-core architectures, able to monitor and adapt itself in order to pursue the desired performance/reliability trade-off. The engine is implemented in a fault management layer working on top of the operating system, in charge of providing reliability properties when desired. The proposed solution dynamically selects and applies fault detection/tolerance mechanisms to mitigate the effects of faults while optimizing performance. Experimental results obtained in the image processing scenario show that the provided self-adaptability feature allows to achieve better performance while fulfilling the reliability requirements with respect to traditional, static solutions.

Ongoing work is taking into account the re-design of the FM layer for the fabric controller to consider the dispatching of the applications on the multi-tile architecture also including the communication issues of the adopted NoC.

Acknowledgment This work is partially supported by EU-ARTEMIS SMECY project, grant no. 100230.

References

1. Accelera Systems Initiative: http://www.accellera.org. Accessed 27 Mar 2013
2. Aggarwal N, Ranganathan P, Jouppi NP, Smith JE (2007) Configurable isolation: building high availability systems with commodity multi-core processors. In: Proceeding international symposium on computer architecture, pp 470–481
3. Auslander M, Dasilva D, Edelsohn D, Krieger O, Ostrowski M, Rosenburg B, Wisniewski RW, Xenidis J (2002) K42 overview. Tech. rep., IBM T. J. Watson Research Center
4. Baumann A, Barham P, Dagand PE, Harris T, Isaacs R, Peter S, Roscoe T, Schüpbach A, Singhania A (2009) The multikernel: a new OS architecture for scalable multicore systems. In: Proceeding ACM symposium on operating systems principles (SOSP), pp 29–44, New York
5. Bolchini C, Miele A, Sciuto D (2012) An adaptive approach for online fault management in many-core architectures. In: Proceeding conference on design, automation and test in Europe (DATE), pp 1429–1432
6. Chen Z, Yang M, Francia G, Dongarra J (2007) Self adaptive application level fault tolerance for parallel and distributed computing. In: Proceeding international parallel and distributed processing symposium (IPDPS), pp 1–8
7. ECSS: Methods for the calculation of radiation received and its effects andapolicyfordesignmargins. Tech. Rep. ECSS-E-ST-10-12C European Cooperation for Space Standardization (2008)
8. Gizopoulos D, Psarakis M, Adve S, Ramachandran P, Hari S, Sorin D, Meixner A, Biswas A, Vera X (2011) Architectures for online error detection and recovery in multicore processors. In: Proceeding conference on design, automation and test in europe (DATE), pp 533–538
9. Horn P (2001) Autonomic Computing: IBM's Perspective on the State of Information Technology
10. Huang J, Blech J, Raabe A, Buckl C, Knoll A (2011) Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In: Proceeding international conference Hw/Sw codesign and system synthesis, pp 247–256
11. International Technology Roadmap for Semiconductors–Emerging Research Devices Section (2010) http://public.itrs.net/. Accessed 27 Mar 2013
12. Kephart JO, Chess DM (2003) The vision of autonomic computing. IEEE Comput 36:41–50
13. Kouadri A, Heron O, Montagne R (2011) A lightweight API for an adaptive software fault tolerance using POSIX-thread replication. In: Proceeding international conference on architecture of computing systems (ARCS), pp 16–19
14. LaFrieda C, Ipek E, Martinez JF, Manohar R (2007) Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In: Proceeding conference dependable systems and networks (DSN), pp 317–326
15. Lattuada M, Pilato C, Tumeo A, Ferrandi F (2009) Performance modeling of parallel applications on MPSoCs. In: Proceeding 11th international conference on system-on-chip (SoC), pp 64–67
16. Meloni P, Tuveri G, Raffo L, Cannella E, Stefanov T, Derin O, Fiorin L, Sami M (2012) System adaptivity and fault-tolerance in NoC-based MPSoCs: the MADNESS project approach. In: Proceeding EUROMICRO conference digital system design (DSD), pp 517–524
17. Mukherjee S, Kontz M, Reinhardt S (2002) Detailed design and evaluation of redundant multi-threading alternatives. In: Proc Intl Symp Comput Architecture. 99–110
18. Normand E (1996) Single event upset at ground level. IEEE Trans Nuclear Sci 43(6):2742–2750
19. Politecnico di Milano: ReSP web site. http://code.google.com/p/resp-sim/. Accessed 27 Mar 2013
20. Salehie M, Tahvildari L (2009) Self-adaptive software: Landscape and research challenges. ACM Trans Autonomous and Adaptive Systems 4:14:1–14:42
21. STMicroelectronics and CEA (2010) Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. In: Research workshop on STMicroelectronics Platform 2012
22. Teraflux (2011) Definition of ISA extensions, custom devices and external COTSon API extensions. In: Teraflux: Exploiting dataflow parallelism in Tera-device computing
23. The OpenMP API specification for parallel programming (2011). http://openmp.org/wp/. Accessed 27 Mar 2013
24. Various Authors (2011) The MIT Angstrom Project: Universal Technologies for Exascale Computing. http://projects.csail.mit.edu/angstrom/. Accessed 27 Mar 2013
25. Weis S, Garbade A, Wolf J, Fechner B, Mendelson A, Giorgi R, Ungerer T (2011) A fault detection and recovery architecture for a teradevice dataflow system. In: Workshop on data-flow execution models for extreme scale computing (DFM), pp 38–44
26. Wells PM, Chakraborty K, Sohi GS (2009) Mixed-mode multicore reliability. In: Proceeding international conference architectural support for programming languages and operating systems, pp 169–180

27. Wirthlin M, Johnson E, Rollins N, Caffrey M, Graham P (2003) The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets. In: Proceeding symposium field-programmable custom computing machines (FCCM), pp 133–142

Cristiana Bolchini received the degree in Electronic Engineering and the PhD in Automation and Computing Engineering from Politecnico di Milano, where she is an Associate Professor at the Dipartimento di Elettronica, Informazione e Bioingegneria.
Her research interests cover the areas of digital system design with a specific focus on reliability properties, hardware/software co-design of dependable systems and reconfigurable systems. She has authored several papers in this area. In the last years, she has also contributed to innovative research on context-aware data design, tailoring and management.
Dr. Bolchini has been an associate editor of the IEEE Transactions on Computers, she is an IEEE Senior Member and a HiPEAC Member. Dr. Bolchini is or has been member of different Technical Program Committee of conferences and symposia in the area of test and fault tolerance for digital systems.

Matteo Carminati received a master degree in Computer Engineering from Politecnico di Milano in 2011, and a MSc in Computer Science from the University of Illinois at Chicago in 2012.
He is currently a PhD student in Information Technology at Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria. His research interests include design methodologies for adaptive systems and for dependable embedded systems.

Antonio Miele holds a master degree in Computer Engineering from Politecnico di Milano and the MSc in Computer Science from the University of Illinois at Chicago. In 2010 he received a PhD degree in Information Technology from Politecnico di Milano, where he is currently a Research Assistant at the Dipartimento di Elettronica, Informazione e Bioingegneria.
His main research interests include the methodologies for the design and the analysis of dependable embedded systems, in particular focusing on hardware and software hardening techniques, design space exploration w.r.t. reliability and failure analysis and characterization.