SoftRM: Self-Organized Fault-Tolerant Resource Management for Failure Detection and Recovery in NoC Based Many-Cores

VASILEIOS TSOUTSOURAS, DIMOSTHENIS MASOUIROS, SOTIRIOS XYDIS, and
DIMITRIOS SOUDRIS, National Technical University of Athens, Greece

Many-core systems are envisioned to leverage the ever-increasing demand for more powerful computing systems. To provide the necessary computing power, the number of Processing Elements integrated on-chip increases and NoC based infrastructures are adopted to address the interconnection scalability. The advent of these new architectures surfaces the need for more sophisticated, distributed resource management paradigms, which in addition to the extreme integration scaling, make the new systems more prone to errors manifested both at hardware and software. In this work, we highlight the need for Run-Time Resource management to be enhanced with fault tolerance features and propose SoftRM, a resource management framework which can dynamically adapt to permanent failures in a self-organized, workload-aware manner. Self-organization allows the resource management agents to recover from a failure in a coordinated way by electing a new agent to replace the failed one, while workload awareness optimizes this choice according to the status of each core. We evaluate the proposed framework on Intel Single-chip Cloud Computer (SCC), a NoC based many-core system and customize it to achieve minimum interference on the resource allocation process. We showcase that its workload-aware features manage to utilize free resources in more than 90% of the conducted experiments. Comparison with relevant state-of-the-art fault tolerant frameworks shows decrease of up to 67% in the imposed overhead on application execution.

CCS Concepts: • **General and reference** → **Cross-computing tools and techniques**; • **Computer systems organization** → **Multicore architectures**; • **Networks** → *Network on chip*; • **Computing methodologies** → *Self-organization*;

Additional Key Words and Phrases: Many-core, Network-on-Chip, Distributed Run-Time Resource Management, Fault Tolerance

1 INTRODUCTION AND MOTIVATION

Since the beginning of this decade, many-core systems have been envisioned as a prominent solution to support the ever increasing demand for increased computational power as well as to facilitate the influx of computing and monitoring devices in all aspects of human life. While a few years ago such systems were merely the product of pioneering thoughts [1, 33], we are now introduced with the reality of the first implementations of kilo-core systems on chip [3, 5, 30]. As projected, the high number of processors incorporated in these SoCs, mandates the need of Network-on-Chip (NoC) communication and message passing infrastructure to facilitate efficient inter-core communication.

This new system architecture imposes new requirements regarding resource supervision and management, which in turn has spawned extensive and on-going research [35, 36]. While the majority of works have targeted the optimization of metrics like system throughput or energy consumption, the fact is that in order to successfully incorporate kilo-core systems to every-day user experience, the dependability of their operation becomes of critical importance. Projection studies indicate that due to the extreme transistor scaling in nano-scale many-core systems, the probability of operation variability and sub-system failures is highly elevated [17, 34]. This reliability deficiencies are attributed to inherent aging and wear-out mechanisms of the chip such as Negative Bias Temperature Instability (NBTI), Hot Carrier Injection (HCI) and Time Dependent Dielectric Breakdown (TDDB) [25]. Variability issues and faults can also occur and aggravate dynamically due to ambient or workload related thermal fluctuations as well as drops in the supply voltage of an integrated circuit [31].

The distributed nature of the targeted systems on both Processing Elements (PEs) and Resource Management imposes extra design requirements and increased complexity to provide fault tolerance guarantees in an online and timely manner. Therefore, it has been identified that in order to effectively mitigate variability issues in kilo-core SoCs, it is mandatory to intervene and leverage techniques for increased dependability in all layers of system design ranging from hardware [39] to high level application development [18, 31].

Fault and lifetime aware application mapping [14, 26, 40] is a first level of dependability extension. Proceeding to system level, fault tolerance is either achieved via centralized decision making [6, 18, 32] or via hierarchical designs which rely on spare PEs provisioning [9, 13, 21, 38]. The latter succeed at concurrent and reliable mapping of many applications but neglect to examine the possibility of failures on the higher level parts of the resource management hierarchy. Consequently, in this work we aim at bridging this design gap by proposing a self-organized, fault-tolerant run-time resource management framework, which integrates mechanisms able to detect and recover faults in every level of its design. We focus on the emerging Distributed Run-Time Resource Management (DRTRM) paradigm to address the scalability requirements and increased workload variability of modern many-core platforms [2, 22, 36].

Motivation: We examine a NoC based many-core system (Figure 1) governed by a DRTRM framework. The PEs in red indicate a number of dedicated cores, responsible for the correct operation of DRTRM. The rest of them are either idle (grey) or assigned to workload execution of an application (black). The system is at a stable state and all PEs and software stacks are in fine condition.

At a point t_0 in time the DRTRM SW stack of the bottom left core fails. The failure is permanent and can be attributed either to a HW fault or a non-recoverable software error. However, DRTRM cannot operate appropriately without a replacement to the failed core due the fact that its correct operation is vital to resource allocation decision making process. Since the system is fully distributed, there is no central point to determine the replacement core and communicate

Fig. 1. Many-core system snapshots.

this decision. As a consequence, there is the need of a self-organizing process to take place amongst the cores.

This process should offer a unique outcome and guarantee that this outcome will be received and respected by all cores (as shown in Figure 1 (t_2 - stable)). A simple replacement tactic i.e. “The first core that detects the failed one will act as its replacement”, is bound to fail due to the distributed nature of the system. There is no guarantee that only one core will detect the failure and volunteer to be the replacement (see ‘!’ at t_1 in Figure 1). In this case, more than one cores will have the same responsibilities which will create collisions and lead to a new unstable system state (t_2 - unstable).

In summary, the focus of this work is to provide a solution and design alternative to the following question. *Given a many-core NoC based system managed in a hierarchical and distributed manner through assignment of distinct roles on different PEs (agents), is there a way to guarantee that a manifested, unrecoverable fault in any agent will be dynamically mitigated in a coordinated way, propagated to all and respected by all healthy PEs on the system?*

Considering the above, our **novel contributions** are:

- i. We introduce SoftRM, a fault tolerant distributed resource management framework which is able to dynamically react to failures of PEs, in a self-organized way.
- ii. We provide guarantees of our recovery strategy based on a consensus agreement algorithm enhanced with workload awareness to achieve an effective replacement policy.
- iii. We incorporate a failure detection algorithm, which takes advantage of the resource allocation related communication of PEs in SoftRM in order to significantly reduce the inflicted communication overhead of detection.
- iv. We implement and evaluate SoftRM on a NoC based many-core system in order to capture the correlation of its design parameters to the efficiency of resource allocation and fault

recovery actions. Comparison against state-of-the-art fault tolerance techniques highlights the advantages of self-optimization against spare core provisioning.

SoftRM is designed on a purely distributed concept, implementing a hierarchy of different roles of PEs to achieve resource allocation and negotiation between applications. Self-organization is preferred over PEs provisioning in order to fully utilize system resources and avoid design-time limits on the number of failures that can be tolerated. The hierarchical design of SoftRM allows its high level fault tolerant features to be cooperatively combined with other techniques for increased dependability, such as hardened PEs [19] or customized fault tolerant schemes per application [20]. In addition, we show that SoftRM mitigates all manifested errors by effectively re-assigning the workload of failed cores to idle resource in over 90% of the examined scenarios, while its overhead on application execution is up to 67% less compared to other state-of-the-art fault tolerance techniques based on spare cores provisioning.

The rest of the paper is organised as follows. Section 2 provides an in-depth discussion on related research. Section 3 introduces PAXOS, the utilized algorithm for consensus agreement. The design of SoftRM is presented in Section 4 by describing its key points, i.e. the employed system model (Section 4.1), its Resource Allocation scheme (Section 4.2) and its Fault Tolerance Infrastructure (Section 4.3). The experimental evaluation of SoftRM is presented in Section 5, while Section 6 concludes the paper.

2 RELATED WORK

A considerable amount of work is focused on extending the lifetime reliability (i.e. minimizing core failures) of many-core systems by appropriately mapping tasks using metrics such as Mean Time to Failure (MTTF). A runtime task mapping design, which extends lifetime of the system by using a wear-based heuristic is presented in [15]. However, this work does not examine component failures and assumes that such failures are detected by the operating system or another hardware resource. Authors in [11] present a task-mapping technique to maximize MTTF of an MPSoC considering the ageing of NoC-links. However, in case of a permanent core failure the system collapses and has to be restarted in order to work properly.

The works in [26, 40], utilize Branch and Bound based algorithms to perform application mapping on reconfigurable many-core NoCs, where the configuration of PEs functions and interconnections is adjusted dynamically. Mapping decisions are centralized and the optimization objectives take into account reliability, energy and performance. The authors assume that spare cores can substitute any faulty PEs and thus focus on tolerating transient errors of the communication links between them. Application mapping is also targeted in [14], where a centralized feedback loop approach is employed to achieve maximum system performance under dark silicon thermal constraints and meet the reliability requirements of the applications under ageing and thermal effects.

Task mapping is combined with Dynamic Voltage and Frequency scaling in [12] to increase the reliability of the system, while transient faults in hardware are tolerated via selective task replication. A genetic algorithm is used to perform design space exploration and determine the task to PE binding, as well as its VFS values. Triple or Double Modular Redundancy (TMR or DMR) is extended in [29] to provide prolonged fault free system lifetime. In case of failures, instead of task remapping to sustain the number of redundant tasks, the authors propose to preserve the mapping and adjust the functionality of the included voter task. N Modular Redundancy (NMR) is the main fault tolerance mechanism in [28], targetting commercial many-core SoCs such as [3]. Transient faults in PEs are encountered by comparing the output of the N-times replicated task and repairing any instance that exhibits output divergence. The required voter task is also doubled

and the two voters constantly repair one another in a round robin way. Authors in [6] propose a self-adaptive engine for fault management in multi-core systems. Failure detection is achieved by task replication and comparison of the different outputs. Application activity is coordinated by a centralized core called *fabric controller*, which executes the fault management layer and is considered hardened by-design.

Authors in [18] and [32] opt for increased system dependability by examining the combination of reliability techniques in different layers of many-core systems. Regarding resource allocation, in [18] the run-time system layer performs reliability-driven core allocation and task mapping. Similarly in [32], the resource manager utilizes redundant multi-threading for task execution resilience accompanied by a variation-aware task mapping. Both approaches depend on centralized resource allocation and no recovery mechanism is presented in case of resource manager failure.

In [9, 21] run-time fault-aware resource management is addressed using three types of cores, i.e. *regular*, *spare* and *manager*. Managers overview the system and monitor the status of other cores, while regular and spare cores execute incoming applications. In case of a permanent or transient failure of a regular core, spare cores are used to replace it and workload is migrated technique from the faulty to the spare core. The difference between the two works is that [9] employs static spare core provisioning while [21] dynamic, i.e. a subset of the cores provided to the application are provisioned for fault tolerance. However, a manager failure is not examined at all and in such a case the fault-aware nature of the proposed schemes disappears. Our work bridges this gap by focusing on the failure of similar managerial agents and provides a design alternative to both static and dynamic provisioning of spare cores. In [16] fault mitigation in networks consisting of nodes of re-configurable PEs (ReCoNodes) is proposed by leveraging both HW/SW techniques. An individual OS overviews each ReCoNode and lightweight “shadow” tasks replicate applications to provide error mitigation. The proposed system, provides an efficient intra-application fault tolerance infrastructure under the assumption that the administrative part (OS) is fault free, where our focus lies.

A System-level and Hierarchical Fault-Aware (SHiFA) approach is presented in [13] to provide run-time tolerance of manifested faults both in PEs and links of the NoC. To achieve efficient fault tolerant mappings and resilient application execution, SHiFA requires two kinds of distributed agents, i.e. system mobile master (MM) and application managers (AM) in its hierarchy. Despite the fact that this hierarchy is critical for the correct system operation, the authors provide no mechanism for recovery actions in case of one or more agent failures. Likewise, in [38] resource management is based on a hierarchical scheme, where a Global Manager (GMP) supervises the system and Local Managers (LMP) supervise application execution. Authors employ both hardware and software mechanisms to tolerate faults both in PEs and links of the NoC. Specifically, in the software stack a fault is mitigated in a cooperative manner between an LMP and GMP. Nevertheless, there is no provision for a dynamic scheme which will ensure system stability in case that one of these dedicated PEs fail. A holistic infrastructure for fault tolerant resource management is also presented in [4]. A System Health Monitoring Unit has a global overview of the status of the links and PEs in a many-core NoC based system and collects and classifies information about manifested errors. This information is propagated to the Mapper-Scheduler unit, which produces appropriate run-time mappings of the applications.

In conclusion, reliability aware mapping techniques concentrate their effort on prolonging the lifetime of a system often neglecting to provide fault tolerance. Even if faults are addressed, the existing solutions mainly rely on utilizing redundancy techniques. On the other hand, our approach belongs to the category of fault tolerant frameworks which aim at mitigating all manifested errors, while providing the infrastructure for application deployment and execution. In this case, reliable application mapping can still be used as a key component of resource management infrastructure.

Fig. 2. Prepare and Accept phases of Paxos Protocol with two Proposers P1 and P2 with proposal numbers n_1 and n_2 ($n_2 > n_1$).

We highly differentiate in comparison to state-of-art [9, 13, 21, 38], by providing fault tolerance without spare core provisioning, thus constantly exploiting the full potential of our target many-core system as well as providing fault tolerant guarantees for all involved agents of the system, in all levels of the resource management hierarchy.

3 PAXOS CONSENSUS PROTOCOL

The core component of the self-organized aspect of the SoftRM is a consensus protocol. It ensures that in a situation where different values are proposed by different processes, only one of them is chosen. Paxos, proposed by Lamport [24], is an algorithm for consensus achievement in a network of unreliable processors. The safety requirements for consensus achievement are (i) only a value that has been proposed may be chosen (*non-triviality*), (ii) only a single value is chosen (*safety*) and (iii) a process never learns that a value has been chosen unless it has actually been (*liveness*). Processes can have any of three different roles in Paxos; *proposers*, who propose values to be chosen, *acceptors*, who accept or reject the proposed values and *learners*, who will eventually learn the chosen value once it has been decided. In a general scenario, a single process may have multiple roles simultaneously. We summarize below the three phases of the complete flow of the protocol.

Prepare Phase: Each proposer picks a unique proposal number n greater than any n_{max} previously sent by any proposer and sends a *prepare* request with number n to all acceptors. On the acceptor side, if a *prepare* request is ever received with number n greater than that of any *prepare* request to which he has already responded ($n > n_{max}$), then he responds with the highest numbered proposal that he has accepted (if any). Additionally, he makes a promise not to accept any more proposals numbered less than n ($n_{max} \leftarrow n$). A request is rejected if the acceptor receives a *prepare* request with n lower than the highest n_{max} proposal number ever received.

Accept Phase: After a proposer has received a response to his *prepare* requests from the majority of acceptors, he sends an *accept* request to those acceptors with a value v . This value is either the highest numbered proposal among the responses received in Prepare phase, or any value if the responses reported no other proposals. On the acceptor side, if an accept message is received with a proposal number n , he accepts the proposal by replying with an *accepted* message, unless a higher proposal number n_{max} has been received in the *Prepare* phase.

Learn Phase: Similarly to the Accept phase, once a proposer has received *accepted* messages from the majority of acceptors, he realizes that his proposed value has been accepted and broadcasts a *learn*(v) message to all learners.

Figure 2 illustrates an example of the Paxos Protocol with two proposers P1, P2 (black nodes) and three acceptors A1, A2, A3 (grey nodes). At time t_0 ($t_0 < t_1 < \dots < t_5$) both P1 and P2 send prepare messages to all acceptors with proposal numbers n_1 and n_2 (grey and black arrows respectively),

with $n_1 < n_2$. At times t_1 and t_4 acceptors A3 and A1 receive the $prepare(n_1)$ from P1 whereas at t_3 acceptor A2 receives the $prepare(n_2)$ message from P2 and they promise **not to accept any other proposals with a lower proposal number**. Since this is the first time the acceptors receive a $prepare(n)$ message, they automatically reply with an accept message. Acceptors A3 and A1 receive the $prepare(n_2)$ from P2 at t_2 and t_5 respectively. They reply with an accept message since $n_2 > n_1$. However, when A2 receives the $prepare(n_1)$ from P1 this message is rejected (red arrow), due to the promise to P2 not to accept any messages with proposal number lower than n_2 . In the Learn phase, v_2 proposed by P2 has been accepted by the majority of acceptors, so P2 broadcasts this value to all learners. In this work, we propose a version of the Paxos algorithm which takes into account the workload status of different processors when a consensus agreement effort is made.

4 SOFTRM: A FAULT TOLERANT DRTRM

4.1 System Model

Target HW model: We target homogeneous¹ many-core systems with N Processing Elements (PEs) deployed on the same SoC. Inter-core communication is achieved in a synchronous message passing way via a Network-on-Chip. Each PE on the system is uniquely identified by an identifier $i \in \mathcal{I}$, where $\mathcal{I} = \{0, 1, \dots, N - 1\}$ is a static, finite set of N identifiers known by all cores. PEs may exhibit failures but we assume that there is always a working communication link between any two functional PEs of the system.

Target Application model: We assume a number of K parallel applications which can dynamically adjust to their allocated PEs. Each application is characterized by a unique type T_a , a workload W and a function $speedup(C_{num})$ which estimates the gained speedup if the application is executed in C_{num} PEs where $1 \leq C_{num} \leq \max(C_{num})$. $\max(C_{num})$ is the highest number of worker cores that an application can run on without speedup saturation. Each application is managed by a designated PE which henceforth will be referred to as **Manager core**. Its responsibility is to orchestrate workload allocation to the available C_{num} worker PEs of the application. In addition, it is involved in resource negotiation rounds with other Managers in order to optimize the total speedup of the running applications on the many-core platform.

Error Model: Exhibited errors in many-core systems are broadly categorized as permanent, intermittent and transient. Permanent faults result in sub-component malfunction which is not restored during run-time of the system. On the contrary, intermittent faults regard errors which occur repeatedly, interchanged with periods when the system is fault free, while transient faults are temporary.

Without loss of generality, we demonstrate the properties of our proposed design focusing on permanent faults. We focus on faults manifested on PEs and do not examine faults on the links of the target NoC based system. We model the probability of permanent fault on a PE of the system using a Weibull distribution [23] with Probability Density Function (PDF) defined as:

$$f(t) = \frac{\beta t^{\beta-1}}{\eta^\beta} e^{-(t/\eta)^\beta}, t \geq 0 \quad (1)$$

where η and β are the scale and shape parameters respectively. According to Equation (1), the reliability of the system is defined as:

$$R(t) = \int_t^\infty f(t) dt = e^{-(t/\eta)^\beta} \quad (2)$$

¹Heterogeneous, multi-ISA systems can also be supported by the proposed design, but this highly complicates the implementation of task migration.

Fig. 3. Failure rate at different periods of chip's lifetime.

Additionally, the failure rate of an individual component is:

$$\lambda(t) = \frac{\beta}{\eta} (t/\eta)^{\beta-1} \quad (3)$$

The correlation between the parameters of the PDF is presented in Equation (4), associated with Mean Time To Failure (MTTF) and Γ function. Assuming an MTTF of some years it follows that:

$$\eta = \frac{MTTF}{\Gamma(1 + \frac{1}{\beta})} \quad (4)$$

To further enhance the utilized error model, we refrain from using a constant error rate but adjust the parameters of the error PDF according to the simulated lifetime of the target chip. In more detail, a chip exhibits different error rates in different periods of its lifetime. Without loss of generality, this variability has been modelled as a “bathtub” curve [39], as shown in Figure 3. In its infant period, there is a very high but decreasing failure rate, until a plateau of minimum, constant failure rate is reached at its grace period. After a prolonged period of execution, ageing and wear-out effects are accumulated and aggravated leading to its breakdown period, where there is an ever increasing probability of error manifestation.

Last but not least, we must notice that the fault tolerant capabilities of the proposed framework are not bound by the employed system error characteristics. As presented in Section 4.3.2, our framework incorporates failure detection mechanisms and thus the only limitation of our design is that the manifested errors must affect the high level functionality of a PE. Therefore, intermittent and transient faults can also be mitigated as long as they are manifested long enough to be identified by our detection process. In the opposite case, they are silently masked and might affect only the outcome of the executed application. To mitigate such silent errors, the designer can take advantage of our proposed hierarchical design, in order to employ fault resilient deployment of the application, e.g. redundant execution of its tasks [28, 29].

4.2 Resource Allocation

SoftRM was designed on concepts introduced by state-of-the-art Distributed Run-Time Resource Management frameworks as in [2, 22]. First, a number of PEs need to serve as the backbone of SoftRM. These cores are referred to as **Controller cores** and serve as the means of keeping track of the status of the system as well as providing this information to the rest of the PEs. This role is crucial, since there is no central point of system monitoring. Controllers are dedicated to their task

Fig. 4. Overview of SoftRM at some point of execution.

and are not involved in application management or workload execution. We name *Cluster* the area of the system monitored by each Controller. These areas are not overlapping and their number and topology is parametric and can be defined at every system initialization but cannot change at run-time.

Application management is the task of **Manager cores**. The relation between a Manager core and an application is one to one, meaning that one Manager can handle only one application and the management of an application cannot be divided to several Managers. This enhances the ability of a Manager core to adopt any workload allocation scheme, without any resource sharing between applications. A PE that has been assigned to an application is identified as a **Worker core**. We define $w_i^j = 1$ if core i is a worker of core j and 0 otherwise. The design choice of having both Controllers and Managers is in order to decouple system monitoring and application management. All cores that have not being appointed any of the above roles, are considered **Idle cores**.

Regarding inter-core communication, all PEs maintain their own *neighbourhood* set Π_i with the ids of all cores inside their Cluster. In addition, each Manager core j maintains a *workers* set $W_j = \{i \mid w_i^j = 1\}$ which consists of all its Worker cores. Finally, each Controller with id i maintains a set with all the Managers that possess cores inside its Cluster. This is called the *Distributed Directory Service* (DDS), where $DDS_i = \{j \mid p \in \Pi_i \wedge w_p^j = 1\}$.

Figure 4 shows an abstract overview of the platform at some point of execution. In this example, there are two Controller cores (0 and 9) and two Manager cores (6 and 10). The Cluster of Controller 0 consists of cores 0-7 whereas the Cluster of Controller 9 consists of cores 8-15. In addition, Workers 5, 7 and 14 belong to Manager 6 whereas Worker 11 to Manager 10. The respective information is also included in the DDS sets of the two Controller cores.

The resource allocation objective of SoftRM is to maximize the average speedup of applications and thus minimize the average application execution latency. For a new application, an initial mapping is performed with the use of Controller cores and then gradually the new Manager core searches for new resources for its application. This search involves looking up for free resources or asking for resource migration from other Manager cores with elevated resource utilization. In this way, resource allocation is cooperative i.e. applications exchange resources in order for the

system to reach a state of maximized average application speedup. The system reaches a stable state when all applications have maximized their speedup or there are no free resources and occupied resources have been fairly distributed to running applications.

At any time t the overall state of framework $\mathcal{F}$ is represented as a tuple $\mathcal{F} = \langle S_{t+1}, A, S_t \rangle$, where $S_t = [s_0^t \ s_1^t \ \dots \ s_n^t]$ is a vector of size N with the current states of all cores. A is a set containing all the applications a_i currently executed on the system and S_{t+1} are the next possible states of all cores, which depend on both S_t and A . We define $s_i^t = 3$ for an Idle core, $s_i^t = 2$ for a Worker, $s_i^t = 1$ for a Manager and $s_i^t = 0$ for a Controller.

4.3 Fault Tolerance Infrastructure

4.3.1 Workload-Aware Paxos Algorithm. When examining the case of a core failure at run-time, Paxos algorithm can be employed in a straightforward way in order to designate a replacement core. Having identified the failure, the rest of the PEs can volunteer to take its place by proposing themselves as the replacement core (i.e. the proposed value is their id). This process while effective, inherently lacks of the ability to capture the state of the proposer PEs. For example, it is much more efficient to replace the failed PE with an Idle one instead of a Worker core, which is occupied with execution of application workload.

Inspired by this inefficiency, we introduce the *willingness factor* wf_j to capture the suitability of a PE with id j to act as a replacement core under its current workload characteristics as dictated by its role on the system. The willingness factor is defined as:

$$wf_j = s_j^t + c \sum_{k \in I} w_j^k \sum_{p \in I \cup J} w_p^k \quad (5)$$

where s_j^t is the current state of core with id j as described in 4.2 and $\sum_{k \in I} w_j^k \sum_{p \in I \cup J} w_p^k$ is the number of co-workers in an application. The constant c is calculated based on the number of maximum workers of an application, so that $c \sum_{k \in I} w_j^k \sum_{p \in I \cup J} w_p^k < 1$. In our case, the value of c was set equal to 0.1. The formulation of wf favours idle cores (higher s_j^t), to express higher willingness to act as replacement cores. Additionally, in the case of a worker, increased number of co-workers leads to higher willingness in order to prevent resource starvation in applications with few cores.

Once a core with id i detects that a Controller or Manager has failed, it acts as a proposer and sends a *prepare*(n_i) message, where n_i is unique and greater than any n_{max} previously sent by any proposer (Π_i is defined in Section 4.2).

$$\forall j \in \Pi_i \rightarrow \text{send} : \text{prepare}(n_i) \quad (\text{phase } 1^a)$$

Since each core has a unique identifier i , determining n_i can be achieved by picking the smallest sequence number n_i greater than any previously sent (n_{max}) such that $n_i \bmod n_{max} = i$. Initially, the value of n_{max} is -1. Note that n_{max} and modulo are used for two different purposes: n_{max} guarantees that the accepted proposal numbers increase monotonically, whereas modulo guarantees that all proposal numbers are unique.

Subsequently, when a core with id j receives a *prepare*(n_i) message (Section 3), if $n_i > n_{max}$ where n_{max} is the highest numbered proposal received then it replies with its highest accepted proposal (v_{acc}) or -1 if no value has been accepted. The reply is paired with its willingness factor (wf_j). N_{max} becomes n_i and a promise is made not to accept any proposals numbered less than n_{max} .

$$\begin{aligned} \forall \text{prepare}(n_i) \rightarrow \text{send} : \text{response}(u_{acc}, wf_j), \\ n_i \geq n_{max} \end{aligned} \quad (\text{phase } 1^b)$$

When the proposer receives an accept from a majority² of acceptors, it sends an $accept(n_i, v_i)$ where v_i is the value of the highest accepted proposal replied in prepare phase. If no such value has been replied (i.e. all replies were negative), then it proposes the core with the highest willingness factor as the replacement core:

$$v_i = \begin{cases} v_{acc}, & \text{if any} \\ \operatorname{argmax}_j(wf_j), & \text{otherwise} \end{cases} \quad (6)$$

In case that the maximum value is proposed by more than one core, the choice is made in a First Come First Served manner. Afterwards, an $accept(n_i, v_i)$ message is sent to all acceptors.

$$\forall j \in \Pi_i \rightarrow \text{send} : accept(n_i, v_i) \quad (\text{phase } 2^a)$$

On the contrary, if a value v_{acc} is replied, it is a previously chosen value by another proposer based on willingness factors, since no other value had been accepted when the choice was made. This value has already been accepted by the majority of acceptors and is replied so that the *safety* requirement of Paxos is ensured. As mentioned in Section 3, when an acceptor receives an $accept(n_i, v_i)$ message it replies with *accepted* iff it has not seen a higher proposal number before.

$$\begin{aligned} \forall accept(n_i) \rightarrow \text{send} : accepted(), \\ n_i \geq n_{max} \end{aligned} \quad (\text{phase } 2^b)$$

Eventually, when the proposer receives *accepted* messages from a majority of acceptors, it realises that its proposed value v_i has been accepted and broadcasts the accepted value (i.e. the id of the replacement core) to the platform.

$$\forall j \in \mathcal{I} \rightarrow \text{send} : learn(v_i) \quad (\text{Phase } 3)$$

It should be noted, that the willingness factor designation builds on top of PAXOS algorithm without interfering with its functionality. More precisely, the requirements of uniqueness and monotonicity of the proposal number (Section 4.1) are not affected and only the proposed value v is customized according to the suitability of one PE to substitute a faulty one. Thus, our proposed technique does not affect the correctness of the employed consensus algorithm.

It must be highlighted that the described infrastructure is proposed in order to mitigate faults of the administrative tasks of a hierarchical resource management scheme, such as the presented Controllers and Managers. Regarding worker nodes executing the parallel workload, a failed worker is similar to a shrink operation, while the existence of a Manager per application allows the adoption of intra-application error resilient resource management techniques such as Double or Triple Modular Redundancy [28, 29], which can be seamlessly integrated in SoftRM.

4.3.2 Failure Detection. Failure detectors proposed in [8] are responsible for detection of node failures or crashes in distributed systems. A failure detector $\mathfrak{D}$ is classified by its *completeness*, meaning the suspicion of faulty processes and *accuracy*, meaning the suspicion of non-faulty processes. A perfect failure detector $\mathcal{P}$ satisfies both *strong completeness* and *strong accuracy*. Every faulty process is eventually permanently suspected by every non-faulty and no process is suspected by anybody before it has actually crashed. On a synchronous system with a known upper bound communication delay Δ , the simplest Perfect Failure Detector (PFD) [7] can be implemented as follows (HB stands for Heart Beat):

²In our case, the majority is any set of cores $S \subseteq \Pi_i$ such that $|S| > \frac{|\Pi_i|}{2}$, where $|S|$ denotes the cardinality of set S .

Table 1. Messages/sec for $\mathcal{P}$ and $\mathcal{P}_t$ for Different Workloads

Detector Workload	Perfect Failure Detector $\mathcal{P}$ [7]	Proposed tweaked Perfect Failure Detector $\mathcal{P}_t$
Light	346.62	122.61
Medium	356.96	115.46
Heavy	388.33	114.84

- i. Broadcast $\langle HB_REQUEST \rangle$ message.
- ii. Wait for the upper bound delay Δ .
- iii. If node j did not reply with a $\langle HB_REPLY \rangle$ message within Δ , then detect j as faulty.

Although this approach is tolerable by general purpose systems, it comes with limited scalability and excessive message traffic since $[N - 1]^2$ messages are sent every Δ seconds, where N is the number of PEs. We propose a tweaked version of PFD (tPFD), tailored for on-chip communication summarized as follows:

Set a timer to expire in Δ seconds. If cores i and j from the same Cluster ($\Pi_i = \Pi_j$) have exchanged any message until timer expiration then both are considered alive. If not, a heart beat request is sent from i to j and vice versa and timer is reset to Δ seconds. If one of i or j has not replied until timer expiration, then it is considered a failed core.

The algorithm takes advantage of the frequent inter-core communication for resource negotiation and application management on SoftRM to establish that a core is alive. A $\langle HB_REQUEST \rangle$ is sent from one core to another only if they have not exchanged any messages during a Δ interval. If a $\langle HB_REPLY \rangle$ has not been received within Δ seconds, then the recipient j core is declared as faulty. Liveness check signals are exchanged only inside a Cluster to provide a scalable detector.

To quantify the inflicted overhead on SoftRM by the presented detectors, we examine the average exchanged messages per second for failure detection in scenarios of 16 incoming applications. Scenarios differ in intensity of average application workload W (See Sections 4.1, 5.1). Table 1, summarizes the results which indicate that the proposed tPFD imposes more than half the overhead in exchanged messages compared to the original. Interestingly, as application workload increases less messages are sent by the tweaked detector, since inter-core communication is more frequent as applications are active for more time. The trade-off of the proposed detector is that it requires at most 2Δ latency to detect a faulty PE. This behaviour is acceptable for the examined applications since they are not bound by time-critical recovery characteristics.

4.3.3 Recovery. The recovery phase involves all the necessary actions for the system to be restored to a stable state. It takes place after all PEs have received the *learn* signal, which communicates the id of the replacement core as outcome of the execution of Paxos algorithm. All sets presented in Section 4.2 are updated to reflect the new state of the system, omitting the failed core j . In similar manner, the new Controller builds its *Cluster* and *DDS* sets using information gathered by the rest of the system and according to the Cluster region topology of the failed Controller, as this administrative region cannot change dynamically.

In case of a Manager core failure, the replacement Manager signals all its Worker cores to inquire about the portion of workload that has been executed and thus make an assessment about the remaining workload of the application. In addition, by inspection of the logged checkpoints of the application all the portions of the workload under execution when the failure occurred are marked for re-execution to ensure a correct outcome. Then, workload is redistributed to Worker cores and application execution is restarted.

Fig. 5. Example of workload aware Paxos in SoftRM.

In case of a Worker core failure, application execution is paused and workload is re-distributed to the rest of healthy workers. Erroneous results are not propagated to healthy workers due to the blocking, MPI-like send/receive style of data exchange between workers. If a Worker core has been designated as the replacement core, it concludes its workload execution until the next checkpoint and then proceeds to its new duties. In parallel, its former Manager informs its Controller the worker loss and re-distributes the remaining workload to the rest of its workers.

SoftRM run-time example: Figure 5 illustrates an instance of the operation of SoftRM. Figure 5(a) presents a point in time where Controller 9 has failed. Core 14 identified the failure and triggered workload-aware Paxos algorithm. The willingness factors of the various PEs have been annotated on Figure 5(a) (all idle cores have equal willingness factor). To designate the id of the replacement Controller core, an instance of the workload-aware Paxos is executed inside Cluster 2. An idle core has been chosen as replacement since it had the highest willingness factor i.e. the highest suitability to act as a new Controller. Then, the necessary recovery actions are performed (Figure 5(b)) and Manager cores with workers inside Cluster 2 subscribe to the DDS set of the new Controller.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

SoftRM was developed and deployed on Intel Single-chip Cloud Computer (SCC) NoC based platform [27]. The SCC chip consists of 24 dual-IA-core tiles connected by a 2D-grid on-die network, where each tile contains two P54C cores. Inter-core communication is achieved in a message passing way which accurately fits our targeted HW model (Section 4.1).

An evaluation scenario includes a number of applications requiring to be executed on the system. The applications involve stencil computations, i.e. matrix multiplication. We consider the execution kernel over 4096×4096 data matrix as the baseline workload, i.e. $W = 1$. We model workload variability by considering matrices scaled up W times. Each application is characterised by its workload W and the distribution of workload values for 128 incoming applications is presented in Figure 6, for three different scenarios of workload intensity (Low, Medium, High). The values in X axis correspond to the range of possible workload values per application. In addition, the outcome of an examined scenario is affected by the arrival rate of applications on the system. In this experimental evaluation, this rate is generated randomly according to a Poisson distribution

Fig. 6. Workload distribution of examined scenarios.

Fig. 7. Recovery efficiency vs Message traffic overhead.

with λ equal to 48, as in [37]. Each of the examined scenarios explores a different design parameter of SoftRM and is considered complete when all input applications have performed their execution life-cycle.

5.2 Evaluation

Our first evaluation in Figure 7, shows the correlation between the Δ interval of tPFD (Section 4.3.2) and the required time for SoftRM to recover from a Controller core failure, presented in mean value and standard deviation for a configuration of 4 Clusters. The moment of core failure is designated according to Equation (3) of the presented error model (Section 4.1). In all presented experiments MTTF is set at 6.56 years as in [10]. For this experiment, β is equal to 1 and all measurements have been repeated 10 times, to filter out the randomness introduced by the parallel execution of framework instances on the target HW. We observe that as Δ increases, both the average value and its deviation rise in absolute numbers. However, in relative numbers, in all cases the deviation is up to 15% of the mean value. The bars present the overhead induced on SoftRM for the detection of failed cores. Increased frequency of liveness check signals results in high toll on exchanged messages. Considering this trade-off, Δ was set to 1000 msec for the rest of the experiments.

The next experiment aims at analyzing the correlation between the granularity of self-organisation and Resource Allocation (RA) effectiveness. The evaluated design parameter is the Cluster size, since the cores inside it need to self-organize to elect a replacement core. The bars in Figure 8 correspond to the overhead in percentage of the Fault Tolerance (FT) infrastructure on the rate of exchanged messages for RA per second by SoftRM. Smaller Cluster size leads to overhead reduction, since a Paxos instance is executed amongst a reduced number of PEs. However, smaller size equals more Clusters, more Controller cores and thus reduced number of available PEs for workload execution. This is reflected in the total execution latency of each scenario (triangles),

Fig. 8. FT infrastructure overhead vs RA effectiveness.

Table 2. Frequency Distribution of Replacement Core's State

Protocol \ Previous State	Fixed IDs	Basic Paxos	Workload-aware Paxos
Idle	16.67%	33.33%	91.67%
Worker	58.33%	58.33%	8.33%
Manager	25%	8.34%	0%

where a significant increase is observed for small Cluster sizes due to the prolonged execution latency of applications because of the limited resources assigned to them. For a Cluster of size 6, application slow down is so severe that the overhead of FT infrastructure is sky-rocketed.

In Table 2, an evaluation of the workload-aware Paxos protocol is performed against two other strategies for replacement core designation. More precisely, in Basic Paxos, each proposer volunteers to be the replacement core in step [Phase 2^a](#) (Section 4.3). In Fixed IDs strategy, the replacement core is the one with $\text{id}(\Pi_j)$ that resides in the same Cluster as the failed core j . The evaluation quantifies what was the state of the core before being chosen as the replacement one. The proposed workload-aware Paxos protocol manages to highly outperform its rival implementations and achieves to designate an idle core as the replacement with frequency of over 91%. This highly reduces the recovery overhead, compared to when a Worker or Manager core is chosen since it introduces no interference to the executed applications. Figure 9 illustrates the average distribution of core roles the moment that the various strategies engaged in determining a replacement core. Different examined scenarios involved different numbers of Cluster areas on the system and therefore distributions are grouped according to this variable.

To quantify the efficiency of SoftRM versus state-of-the-art techniques at tolerating multiple errors we present in Figure 10 the overview of system behaviour in a scenario of 64 incoming applications. Errors have been generated according to the utilized error model, taking into account all phases of the chip lifetime (Section 4.1) and affecting any PE of the system. Comparison is performed against two techniques based on spare cores provisioning, either dynamically at runtime [21] or statically at design time [9]. Spare cores are distributed among all clusters (e.g. in a configuration with 4 clusters, 16 spare cores would equal to 4 spare cores per cluster).

The X-axis of Figure 10 represents time, showing the arrival of applications and the time points in which errors occurred (thunder symbols). The different fault tolerance schemes are compared in terms of the number of completed applications within a specific time frame. SoftRM facilitates the

Fig. 9. Distribution of core states during the execution of different strategies for failed core replacement.

Fig. 10. Overview of the evolution of the system according to different employed fault tolerant techniques.

completion of a significantly higher number of applications compared to the rest of the techniques, while managing to mitigate all errors. This happens because no core is excluded from workload execution unless it is mandatory for fault recovery and at the same time the overhead of self-optimization is considerably low. The dynamic spare core provisioning scheme [21] as well as the static of 24 cores also manage to mitigate all errors but they impose a heavy toll on application execution, since a high number of PEs is inactive until an error occurs. Most importantly, all other static spare core allocation configurations fail at mitigating all errors, a fact that we consider a breaking point in the functionality of the system. The less the number of pre-allocated spare cores, the faster this breaking point is reached.

To further quantify the reduced overhead of SoftRM, Figure 11 summarizes the system throughput expressed as successfully completed applications per minute, achieved by SoftRM versus the spare cores provisioning techniques, taking also into account different Cluster configurations. In all cases, our approach results in higher throughput, since no cores are excluded from application execution. As expected the highest overhead, reaching 67%, is observed when 24 cores (half the available ones) are provisioned for fault tolerance. We should note that this high number does not offer higher error resilience as equal number of errors can be mitigated by SoftRM. Dynamic core provisioning leads to overhead of up to 25%, however our implementation of this approach was

Fig. 11. Comparison between SoftRM, static and dynamic spare core allocation for fault free execution.

Fig. 12. SoftRM evaluation for varying β of error model.

conservative in the sense that only one spare cores is dynamically provisioned per application. This implies that only one error can be tolerated per application and in case of more, its execution fails.

In the last conducted experiment, presented in Figure 12, we quantify the behavior of SoftRM under variable error manifestation time series. By adjusting the value of β parameter of the utilized error model, we are able to customize the error manifestation rate on the system. For fair comparison purposes, we maintain the number of errors constant and equal in all scenarios. The influence of errors is quantified as the percentage of execution penalty on the input applications, using the fault free scenario as a baseline. Results confirm that high execution penalty is observed when more errors are manifested earlier in the lifetime of a scenario (lower β). This is because a high number of error corresponds to less available PEs for workload execution and the faster these PEs are damaged, the higher is the number of application that are affected thus leading to higher execution latency penalty.

6 CONCLUSIONS

In this paper we introduced SoftRM, a self-organizing and fault tolerant Distributed Run-Time Resource Management framework for NoC-based many-core systems. Its main aspects were presented i.e. fault detection, replacement core determination and recovery actions for the framework to be restored to a stable state. The replacement policy is based on Paxos consensus reaching algorithm, enhanced with workload-aware features for effective use of free resources on the system. To showcase the effectiveness of SoftRM it was implemented and evaluated on Intel Single-chip

Cloud Computer (SCC) NoC based many-core system. Impact analysis of its design parameters enabled its fine-tuning to achieve minimum interference on the resource allocation process. Results showed that its workload-aware features manage to efficiently utilize free resources in more than 90% of the conducted experiments.

REFERENCES

- [1] Nilmini Abeyratne, Reetuparna Das, Qingkun Li, Korey Sewell, Bharan Giridhar, Ronald G. Dreslinski, David Blaauw, and Trevor Mudge. 2013. Scaling towards kilo-core processors with asymmetric high-radix topologies. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 496–507.
- [2] Iraklis Anagnostopoulos, Vasileios Tsoutsouras, Alexandros Bartzas, and Dimitrios Soudris. 2013. Distributed runtime resource management for malleable applications on many-core platforms. In *Proceedings of the 50th Annual Design Automation Conference*. ACM.
- [3] Tatsumi Aoyama, Ken-Ichi Ishikawa, Yasuyuki Kimura, Hideo Matsufuru, Atsushi Sato, Tomohiro Suzuki, and Sunao Torii. 2016. First application of lattice QCD to pezy-SC processor. *Procedia Computer Science* 80 (2016), 1418–1427.
- [4] Siavoosh Payandeh Azad, Behrad Niazmand, Jaan Raik, Gert Jervan, and Thomas Hollstein. 2016. Holistic approach for fault-tolerant network-on-chip based many-core systems. *arXiv preprint arXiv:1601.07089* (2016).
- [5] Brent Bohnenstiehl, Aaron Stillmaker, Jon Pimentel, Timothy Andreas, Bin Liu, Anh Tran, Emmanuel Adeagbo, and Bevan Baas. 2016. A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*.
- [6] Cristiana Bolchini, Matteo Carminati, and Antonio Miele. 2013. Self-adaptive fault tolerance in multi-/many-core systems. *Journal of Electronic Testing* 29, 2 (2013), 159–175.
- [7] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media.
- [8] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43, 2 (1996), 225–267. DOI : <https://doi.org/10.1145/226643.226647>
- [9] Chen-Ling Chou and Radu Marculescu. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In *2011 Design, Automation & Test in Europe*.
- [10] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2013. Communication and migration energy aware design space exploration for multicore systems with intermittent faults. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 1631–1636.
- [11] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2013. Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 689–694.
- [12] Anup Das, Akash Kumar, Bharadwaj Veeravalli, Cristiana Bolchini, and Antonio Miele. 2014. Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 1–6.
- [13] Mohammad Fattah, Maurizio Palesi, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. 2014. Shifa: System-level hierarchy in run-time fault-aware management of many-core systems. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 1–6.
- [14] Mohammad-Hashem Haghbayan, Antonio Miele, Amir M Rahmani, Pasi Liljeberg, and Hannu Tenhunen. 2016. A lifetime-aware runtime mapping approach for many-core systems in the dark silicon era. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 854–857.
- [15] Adam S. Hartman and Donald E. Thomas. 2012. Lifetime improvement through runtime wear-based task mapping. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'12)*. ACM, New York, NY, USA, 10. DOI : <https://doi.org/10.1145/2380445.2380455>
- [16] Christian Haubelt, Dirk Koch, Felix Reimann, Thilo Streichert, and Jürgen Teich. 2010. ReCoNets design methodology for embedded systems consisting of small networks of reconfigurable nodes and connections. In *Dynamically Reconfigurable Systems*. Springer, 223–243.
- [17] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. 2013. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proc. of the 50th Annual Design Automation Conference*. ACM.
- [18] Jörg Henkel, Lars Bauer, Hongyan Zhang, Semeen Rehman, and Muhammad Shafique. 2014. Multi-layer dependability: From microarchitecture to application level. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
- [19] Viacheslav Izosimov, Ilia Polian, Paul Pop, Petru Eles, and Zebo Peng. 2009. Analysis and optimization of fault-tolerant embedded systems with hardened processors. In *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE.

- [20] Paris Christos Kanellakis and Alex Allister Shvartsman. 2013. *Fault-tolerant Parallel Computation*. Vol. 401. Springer Science & Business Media.
- [21] Fatemeh Khalili and Hamid R Zarandi. 2013. A reliability-aware multi-application mapping technique in networks-on-chip. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE.
- [22] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. 2011. DistRM: Distributed resource management for on-chip many-core systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*.
- [23] Israel Koren and C. Mani Krishna. 2010. *Fault-tolerant Systems*. Morgan Kaufmann.
- [24] Leslie Lamport and others. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.
- [25] Xiaojun Li, Jin Qin, and Joseph B. Bernstein. 2008. Compact modeling of MOSFET wearout mechanisms for circuit-reliability simulation. *IEEE Transactions on Device and Materials Reliability* 8, 1 (2008), 98–121.
- [26] Leibo Liu, Chen Wu, Chenchen Deng, Shouyi Yin, Qinghua Wu, Jie Han, and Shaojun Wei. 2015. A flexible energy-and reliability-aware application mapping for NoC-based reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2015).
- [27] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, and others. 2010. The 48-core SCC processor: The programmer’s view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 1–11.
- [28] Peter Munk, Mohammad Shadi Alhakeem, Raphael Lisicki, Helge Parzyjegl, Jan Richling, and Hans-Ulrich HeiB. 2015. Toward a fault-tolerance framework for COTS many-core systems. In *Dependable Computing Conference (EDCC), 2015 Eleventh European*. IEEE.
- [29] Badrun Nahar and Brett H. Meyer. 2015. RotR: Rotational redundant task mapping for fail-operational MPSoCs. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), 2015 IEEE International Symposium on*. IEEE, 21–28.
- [30] Andreas Olofsson. 2016. Epiphany-v: A 1024 processor 64-bit risc system-on-chip. *arXiv preprint arXiv:1610.01832* (2016).
- [31] Abbas Rahimi, Luca Benini, and Rajesh K. Gupta. 2016. Variability mitigation in nanometer CMOS integrated systems: A survey of techniques from circuits to software. *Proc. IEEE* 104, 7 (2016), 1410–1448.
- [32] Muhammad Shafique, Philip Axer, Christoph Borchert, Jian-Jia Chen, Kuan-Hsun Chen, Björn Döbel, Rolf Ernst, Hermann Härtig, Andreas Heinig, Rüdiger Kapitza, and others. 2015. Multi-layer software reliability for unreliable hardware. *it-Information Technology* 57, 3 (2015), 170–180.
- [33] Muhammad Shafique and Jorg Henkel. 2013. Agent-based distributed power management for Kilo-core processors: Special Session: Keeping Kilo-core chips cool: New directions and emerging solutions. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*. IEEE, 153–160.
- [34] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 389–398.
- [35] Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak. 2017. A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi/many-core systems. *ACM Comput. Surv.* (2017).
- [36] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2013. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 1.
- [37] Vasileios Tsoutsouras, Sotirios Xydis, and Dimitrios Soudris. 2015. Job-arrival aware distributed run-time resource management on intel SCC manycore platform. In *Embedded and Ubiquitous Computing (EUC), 2015 IEEE 13th International Conference on*. IEEE, 17–24.
- [38] Eduardo Wachter, Vinicius Fochi, Francisco Barreto, Alexandre Amory, and Fernando Moraes. 2016. A hierarchical and distributed fault tolerant proposal for NoC-based MPSoCs. *IEEE Transactions on Emerging Topics in Computing* (2016).
- [39] Sebastian Werner, Javier Navaridas, and Mikel Luján. 2016. A survey on design approaches to circumvent permanent faults in Networks-on-Chip. *Comput. Surveys* (2016).
- [40] Chen Wu, Chenchen Deng, Leibo Liu, Jie Han, Jiqiang Chen, Shouyi Yin, and Shaojun Wei. 2015. An efficient application mapping approach for the co-optimization of reliability, energy, and performance in reconfigurable NoC architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 8 (2015), 1264–1277.