Fault-Tolerant Dynamic Task Mapping and Scheduling for Network-on-Chip-Based Multicore Platform

NAVONIL CHATTERJEE, SURAJ PAUL, and SANTANU CHATTOPADHYAY,
Indian Institute of Technology Kharagpur

In Network-on-Chip (NoC)-based multicore systems, task allocation and scheduling are known to be important problems, as they affect the performance of applications in terms of energy consumption and timing. Advancement of deep submicron technology has made it possible to scale the transistor feature size to the nanometer range, which has enabled multiple processing elements to be integrated onto a single chip. On the flipside, it has made the integrated entities on the chip more susceptible to different faults. Although a significant amount of work has been done in the domain of fault-tolerant mapping and scheduling, existing algorithms either precompute reconfigured mapping solutions at design time while anticipating fault(s) scenarios or adopt a hybrid approach wherein a part of the fault mitigation strategy relies on the design-time solution. The complexity of the problem rises further for real-time dynamic systems where new applications can arrive in the multicore platform at any time instant. For real-time systems, the validity of computation depends both on the correctness of results and on temporal constraint satisfaction. This article presents an improved fault-tolerant dynamic solution to the integrated problem of application mapping and scheduling for NoC-based multicore platforms. The developed algorithm provides a unified mapping and scheduling method for real-time systems focusing on meeting application deadlines and minimizing communication energy. A predictive model has been used to determine the failure-prone cores in the system for which a fault-tolerant resource allocation with task redundancy has been performed. By selectively using a task replication policy, the reliability of the application, executing on a given NoC platform, is improved. A detailed evaluation of the performance of the proposed algorithm has been conducted for both real and synthetic applications. When compared with other fault-tolerant algorithms reported in the literature, performance of the proposed algorithm shows an average reduction of 56.95% in task re-execution time overhead and an average improvement of 31% in communication energy. Further, for time-constrained tasks, deadline satisfaction has also been achieved for most of the test cases by the developed algorithm, whereas the techniques reported in the literature failed to meet deadline in about 45% test cases.

CCS Concepts: • **Networks** → **Network on chip**; • **Hardware** → *Fault tolerance*; • **Software and its engineering** → *Scheduling*; Embedded software; Real-time schedulability;

Additional Key Words and Phrases: Network-on-chip, dynamic mapping and scheduling, task replication, communication energy, deadline, fault tolerance

ACM Reference Format:
Navonil Chatterjee, Suraj Paul, and Santanu Chattopadhyay. 2017. Fault-tolerant dynamic task mapping and scheduling for network-on-chip-based multicore platform. ACM Trans. Embed. Comput. Syst. 16, 4, Article 108 (May 2017), 24 pages.
DOI: http://dx.doi.org/10.1145/3055512

Authors' address: N. Chatterjee (corresponding author), S. Paul, and S. Chattopadhyay, Department of Electronics and Electrical Communication Engineering, Indian Institute of Technology Kharagpur, West Bengal, India, 721302; emails: navonil@iitkgp.ac.in, Suraj.Paul@alumnimail.iitkgp.ac.in, santanu@ece.iitkgp.ernet.in.

1. INTRODUCTION

Advancements in semiconductor technology have made it possible to integrate a large number of Intellectual Property (IP) cores, Memory Units, and Processing Elements (PEs) onto a single chip, resulting in a System-on-Chip (SoC) design. Major challenges faced by SoC designers include intercore communication, synchronization, parallelism between the tasks, and so forth. Traditional bus-based SoC designs often exhibit limited performance, higher latency, and increased power consumption. To overcome these drawbacks of a bus-based SoC, Network-on-Chip (NoC) has been proposed as a viable solution [Benini and De Micheli 2002]. In the NoC paradigm, intercore communication is provided by a set of routers and point-to-point communication links between them. A core is attached to a router through a Network Interface (NI) module, which interfaces the computation and communication units and converts the data from the IP cores into packetized form. Routers route the packet from source node to destination node depending on the underlying network topology and routing strategy [Kundu and Chattopadhyay 2014]. With rapidly shrinking transistor geometries and rigorous voltage scaling, the gain in performance in terms of power, packing density, and area consumption has been offset by the dependability of on-chip communication elements and processing entities [Borkar et al. 2004]. This is mainly attributed to vulnerability of fabricated NoC components to transient, intermittent, and permanent faults [Constantinescu 2003]. Transient faults occur when there is one or more bit-errors in the transmitted packet, which may be the effect of crosstalk [Kohler et al. 2010] or impact of alpha and neutron particle strikes [Chou and Marculescu 2011]. Intermittent faults occur in burst, repeat themselves from time to time, and tend to occur in the same location [Constantinescu 2002]. Transient and intermittent faults can often be handled at the software level. Permanent faults occur due to device wearout caused by Negative Bias Temperature Instability (NBTI), electromagnetic interference, and oxide breakdown [Fick et al. 2009]. If a permanent fault occurs during the operation of the chip, we need to have additional resources available to replace the faulty components. The redundancy introduced in the design may help to overcome such a situation but at the cost of additional components. This leads to consumption of extra energy, which may even exceed the energy budget for a given application. Therefore, one of the design goals in case of permanent faults is to provide support for tolerating multiple faults without sacrificing solution quality and honoring the energy budget of the application. This motivates us to limit our research to permanent faults only and to evolve a software-redundancy-based strategy to overcome such fault scenarios, where the IP cores are likely to be impaired by faults, which may be during their field operation. The present work is confined to failure of processing elements, while the network communication elements (i.e., routers and links) are assumed fault free.

In this work, we present a unified mapping and scheduling strategy for dynamic resource allocation in Fault-Tolerant Dynamic Systems (FTDSs). The goal is to achieve resource-efficient fault-resilient application execution in many-core systems under core-to-core reliability variations, where no a priori knowledge of the complete structure of applications exists. Such variation in reliability can be caused by aging [Das et al. 2013], device parameter variation, and thermal variations [Hajimiri et al. 2011]. Important issues that need to be addressed while solving the problem of dynamic task mapping and scheduling for FTDSs are *energy consumption* and *timeliness of task completion*, along with imparting fault tolerance for reliable application execution.

Computation energy is mostly governed by the type of application (e.g., multimedia, networking, etc.), while communication energy is largely determined by the mapping solution. Another important issue that needs to be given due attention is the timing characteristics of the applications, which affect the quality of the solution. An application is characterized by execution times and deadlines associated with individual tasks in it. To improve the quality of the solution and maintain the utility of the results of computations (depending on the execution time), task deadlines must be honored for many practical real-time applications. To achieve higher reliability and deadline performance on multiple cores, in this article, we have introduced a reliability-driven task mapping and scheduling strategy that involves determination of *task allocation modes* (i.e., task allocation with or without redundancy) and *task allocation decisions* (i.e., mapping the (redundant) tasks on many-cores with heterogeneous reliability characteristics). The overall contributions of this article are as follows:

—Proposes method to determine allocation mode of tasks for tolerating single and multiple permanent core failures
—Proposes a communication-energy- and deadline-aware dynamic mapping and scheduling method for real-time applications
—Presents algorithm for incorporating software-redundancy-based fault tolerance during resource allocation of concurrent applications

The rest of this article is organized as follows. In Section 2, we present the literature review. Section 3 describes the NoC-based system model. Some preliminary definitions and the problem statement are given in Section 4. The fault-aware dynamic resource allocation policy that helps to mitigate the problem of core failures at runtime is presented in Section 5. A detailed description of the experimental setup and associated results are given in Section 6. In Section 7, the conclusion and future work are presented.

2. LITERATURE REVIEW

2.1. Hardware Redundancy-Based Fault-Tolerant Techniques

In this section, we present some selected works on fault-tolerant approaches for the NoC platform. Some works [Shivakumar et al. 2012; Schuchman and Vijaykumar 2005] propose using microarchitecture-level redundancy to improve system reliability and increase processor lifetime. Shivakumar et al. [2012] utilized the inherent redundancy present in modern microprocessors to improve yield and enhance graceful degradation of the system performance in case of failure. It considers granularity from the processor subcomponent level to the whole processor for proposing fail-in-place capabilities within a single chip. Schuchman and Vijaykumar [2005] proposed a defect-tolerant microarchitecture, namely, *Rescue*. Core-level redundancy has been shown to be more beneficial compared to microarchitectural-level redundancy for homogeneous multiprocessor platforms.

For many-core systems, as a single core is small and inexpensive compared to the entire chip, core-level redundancy is considered to be an efficient solution for yield improvement [Zhang et al. 2008, 2009]. Here, an n core processor has been used with m redundant cores. Zhang et al. [2009] proposed a core-level redundancy scheme that effectively tolerates defects in homogeneous multicore systems and increases yield. Defective cores change the topology, and the programmer is faced with the challenge to optimize parallel applications. To address this problem, Zhang et al. [2009] have provided a unified topology that is isomorphic, regardless of the underlying physical topologies. In Wang et al. [2013], shift operations—row bi-shift and column shift—have been used for redistributing fault-free PEs of a processor array in reconfiguration. This reduces congestion and latency of the reconfigured topology by reducing long links. The increase in area, throughput, and delay due to the use of spare cores has been addressed in Ren et al. [2015]. Using the maximum flow algorithm, it optimizes the use of spare cores with improvement in the repair rate of faulty PEs, having polynomial-time complexity.

An analytical model for yield and cost estimation for spare core-based multicore systems has been provided in Shamshiri et al. [2008] and Shamshiri and Cheng [2011]. The yield model for multicore systems takes into account the number of cores, number of spares, core yield, manufacturing test quality, in-field test quality, and burn-in process. Based on this model, the authors have experimentally investigated the effects of these factors on the total cost of the chip. It is shown that the overall cost can be significantly reduced by adding a few dynamically reconfigurable spares. A fault-aware method to manage resources in NoC-based multiprocessors, called FARM (Fault-Aware Resource Management), has been proposed in Chou and Marculescu [2011]. The placement of spare cores is fixed for all applications for migrating the tasks on occurrence of faults. However, as spare core allocation is not adaptively set based on the structure of incoming applications, FARM imposes high communication energy and performance overhead after failure recovery. Khalili and Zarandi [2013] proposed a method to adaptively determine spare cores for each application depending on the criticality of its tasks. It assigns spare cores near to the processing cores for critical tasks. The proposed mapping technique reduces overall communication energy and performance overhead [Chou and Marculescu 2011] by reducing the distance between the spare core and the failed core.

2.2. Software-Redundancy-Based Fault-Tolerant Techniques

A fixed-order Block and Band reconfiguration technique has been studied in Yang and Orailoglu [2007]. The initial schedule is statically partitioned into multiple bands, and regular reassignment capability is embedded into it. A group transfer of a set of dependent tasks to a new PE is performed upon execution-driven resource variation. This migration to other functional core(s) is determined by the band in which the tasks belong.

Das et al. [2015] and Das and Kumar [2012] present an execution trace-based dynamic reconfiguration of application mapping for different fault scenarios. The proposed technique performs a design-time analysis on the execution trace of an application, modeled as synchronous data flow graphs. The process captures the optimal points with respect to the throughput and migration overhead for multiple fault scenarios. The results are updated in the database. Out of all captured points, the one with the highest throughput/energy ratio is selected as the final mapping and executed at runtime for tolerating core faults. In Lee et al. [2010], the technique presented analyzes all possible processor failure scenarios at compile time and stores resultant task remapping information to use it for tolerating processor failures at runtime. A re-execution slot-based reconfiguration mechanism has been studied in Huang et al. [2011]. Normal and re-execution slots of a task are scheduled at design time using an evolutionary algorithm to minimize certain parameters, like throughput degradation. At runtime, tasks on a faulty core migrate to their re-execution slot on a different core. However, schedule length can become unbounded for high-fault-tolerance systems. Moreover, analysis is based on acyclic graphs and therefore cannot be applied to streaming applications with cyclic task dependencies. The work reported in Bolanos et al. [2013] also extends the mapping and scheduling solution by performing dynamic remapping of tasks executing on processors detected as faulty at runtime. It calculates mapping solutions for some defined failure scenarios and stores in the database. At runtime, upon detection of a fault, the precomputed mapping solution is evoked from the database. Das et al. [2014] discuss an offline task remapping technique for a heterogeneous multiprocessor system for all processor fault scenarios that minimizes the communication energy and task migration overhead. This is a two-step technique, where the first step includes the initial mapping phase (communication-energy-driven design space exploration), and the second step includes fault tolerance (communication-energy- and migration-overhead-aware task mapping). The later step is used to generate fault-tolerant mappings that are stored in memory for use at runtime. When one or more PEs fail, the best mapping in terms of communication energy and migration overhead is fetched and applied. Arnold and Fettweis [2011] proposed a fault-aware dynamic task scheduling approach that detects and isolates erroneous PEs and ensures error-free execution of applications. The interconnects and *CoreManager* are considered fault free. In case of permanent PE faults, the processor is considered to be nonfunctional, while the task is re-executed on a different PE.

From the previous discussion, it can be noted that some of the existing strategies compute the reconfigured mapping necessary to counteract the PE failures and store in the memory. This requires complete application task graph information and assumes that the application to be executed is fully known beforehand. However, with an increase in the degree of fault tolerance, the time taken to explore the remappings for all possible sequences of fault occurrence becomes a determining factor in fault-resilient operation of the system. Other strategies adopt an approach that brings down the overhead of exhaustively searching, by storing all remapping scenarios for a given degree of fault tolerance. It is affected by exploiting the fact that the probability of multiple PE failures occurring simultaneously is very low. Thus, at each stage, task reconfiguration for only a single PE failure scenario is computed and stored in memory to be used for fault tolerance. Methods using task migration to reliable PEs need checkpointing to store the task state and transfer it to different PEs.

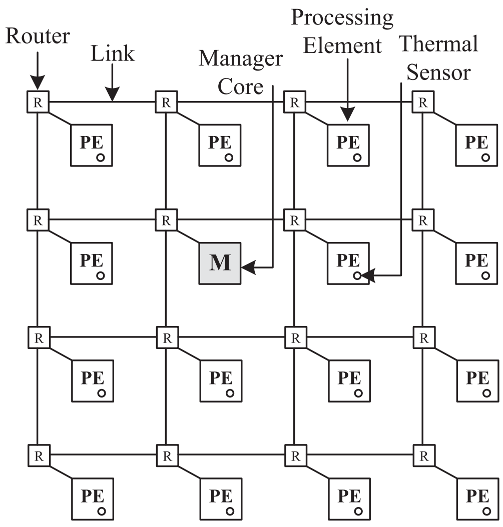
This article proposes an improved software-based technique that eliminates the need for any checkpointing. No additional memory is required to store the PE fault-tolerant mapping. Also, the energy spent on reconfiguring the tasks to reliable PEs is reduced by the proposed technique. In addition, unlike hardware redundancy methods, there is no additional energy and area consumption.

3. SYSTEM MODEL

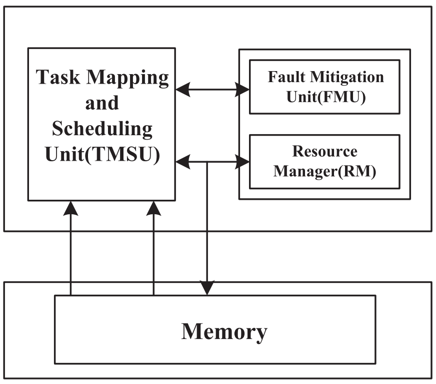
3.1. Hardware Architecture Model

We have considered a two-dimensional (2D) mesh-based NoC topology consisting of heterogenous PEs, as shown in Figure 1(a). A special-purpose PE runs the Real-Time Operating System (RTOS), while general-purpose PEs are for executing tasks. The method proposed in this work can be extended to any other topological configuration of the PEs. The *Manager Core*, shown in Figure 1(b), is implemented as a software module and makes decisions using runtime mapping/scheduling algorithm in the Task Mapping and Scheduling Unit (TMSU) to allocate the tasks of incoming applications to appropriate resources. In addition, the Manager Core also monitors the status of every PE to determine its health/availability. The resource occupancy status of individual general-purpose PEs is updated in the Resource Manager (RM) at runtime to provide the Manager Core with the latest information about the resource availability. Thus, the Manager Core performs dynamic resource allocation by scheduling the tasks in a ready list and suitably mapping them on available cores. For fault tolerance, the Fault Mitigation Unit (FMU) in the Manager Core performs task duplication and allocates the recovery tasks on an alternative PE. The selection of the alternative PE and task allocation methodology has been detailed in Section 5.

A PE has three parameters (PE_i, PE_{cap}, PE_{tasks}). PE_i is the PE identifier that provides a unique identification for a PE. Each PE has a fixed amount of memory. The number of tasks a PE can support is determined by the size of available memory, similar to Maqsood et al. [2015]. In the given NoC system, PEs can support a maximum of PE_{cap} number of tasks. The set PE_{tasks} denotes the set of tasks allocated to the PE. A PE can execute only one task at a time even though multiple tasks might be allocated to it. In the proposed work, we have considered a PE to exist in two states: (1) idle and (2) busy.



(a) A 4 x 4 NoC platform


Task Mapping and Scheduling Unit(TMSU)
Fault Mitigation Unit(FMU)
Resource Manager(RM)
Memory

(b) Manager Core (M)

Fig. 1. A Network-on-Chip (NoC) architecture with multiple cores.

If no task is assigned to the PE or it has completed the assigned tasks, then the state of the PE is defined as idle. On the other hand, if the PE is processing any allotted task, it is said to be in busy state. Additionally, an assigned task may wait for its input data packets to arrive on the PE before it is finally taken up for execution.

3.2. Communication Infrastructure Model

The communication infrastructure consists of a router-based on-chip network that uses wormhole packet switching. The routers considered in this work have no virtual channel; that is, one logical channel is associated with each physical I/O port. The FIFO depth of each input channel is considered as 8, with a data width of 32 bits. The arbiter supports multiflit packets and uses round-robin scheduling. A packet has a maximum size of 64 flits, each flit having a width of 32 bits. The packet composition is as follows: (1) header flit, (2) body flit, and (3) tail flit. The bits 0 and 1 of the flit denote the flit type: Head (01), Body (10), or Tail (11). Invalid flits are represented by 00. Next, for the source address, bits 2 to 9 are allotted, while for the destination address, bits 10 to 17 are reserved. Here, source and destination addresses have been assumed to be 8-bit, supporting a maximum of 256 PEs. The rest of the bits are used for data payload. Deterministic XY routing has been used for routing packets from the source node to the destination node through an on-chip communication network. It is a dimension order routing where each packet is first routed along the x-axis till it reaches the destination column. Then, the packets are traversed along the y-axis to reach the destination node. The XY routing algorithm uses minimum path for packet communication and ensures deadlock- and livelock-free routing at the same time.

3.3. Fault Model

As noted in Figure 1, the NoC system contains several components, each of which may be prone to faults. Failure may occur on one or more routers, links, and PEs. However, this work considers the faulty behaviors of PEs only. A processor may become permanently faulty due to aging or wear and tear of its elements over time. In such cases, no new tasks are mapped onto it, which is ensured by the Manager Core. The

Manager Core and the on-chip communication entities have been assumed to be fault free, as in Chou and Marculescu [2011] and Khalili and Zarandi [2013]. The work of Chatterjee et al. [2014] has presented schemes to ensure fault tolerance of the NoC infrastructure. Although we have considered a single Manager Core, having its multiple copies when permitted by system architecture can further impart fault tolerance for the platfrom. Since the NoC infrastructure has been assumed to be fault free, even in the case of failed PEs, the router to which it is associated functions normally. Therefore, in our case, the mesh architecture remains connected and hence supports the XY routing scheme for packet traversal. We assume that suitable fault detection mechanisms [Liberato et al. 2000] are already existing on the given platform and the time consumed for detection of faults to be negligible.

3.4. Temperature Sensors

Thermal sensors are integrated in a chip for runtime temperature measurement. Such sensors could be found in Intel Xeon Series [Intel 2008] processors with one embedded sensor per core and IBM POWER6 [Floyd et al. 2007] processors with 25 thermal sensors. The measurement inaccuracy of these devices is rectified by sensor calibration techniques as in Yao et al. [2011]. In this work, we have concentrated on the problem of fault-tolerant task scheduling and mapping assisted by the temperature readings of the thermal sensors embedded in the PEs. The thermal sensors are placed across the chip with one sensor per PE, as shown in Figure 1. These sensors send the individual core temperatures to the Manager Core at some periodic interval of time. The temperature of all the cores is stored in an array, *coretemp*. The Manager Core uses the reliability model presented in Equation (9) to generate the reliability of individual processing cores depending on the temperature values stored in *coretemp*. Based on the reliability values of individual PEs, the task allocation mode for individual tasks is determined at runtime.

4. PROBLEM FORMULATION

In this section, we will first introduce necessary definitions and notations required to formally state the dynamic mapping and scheduling problem for fault-tolerant resource allocation.

Application Task Graph: An application is represented as a directed acyclic graph $G = (T, E)$, where T is the set of nodes representing the tasks of the application and E is the set of directed edges showing the dependency and communication between the tasks of the application. A task $t_i \in T$ is represented by four parameters (id_i, ex_i, dl_i, sl_i), where id_i is the task identifier, ex_i is the execution time of t_i, and dl_i is the completion deadline. sl_i indicates the slack time of the task, as explained next.

1. The *execution time* of a task is the time taken by the task to complete its computation while running uninterrupted on a PE.
2. The *completion deadline* is the time by which a task must be completed.
3. The *slack time* is the margin between the time at which a task would complete if it started now and its deadline. This indicates the size of the available scheduling window. It is expressed as

$$sl_i = deadline - current\ time - task\ execution\ time. \tag{1}$$

This is a dynamic attribute of the task as it depends on the runtime situation.

An edge $e_{ij} \in E$ represents the communication between the tasks t_i and t_j. The weight of edge e_{ij}, denoted by w_{ij}, represents the communication volume from t_i to t_j.

Topology Graph: The NoC topology graph is a directed graph $N = (P, L)$ with each core $PE_i \in P$ representing a PE in the topology. Each PE has an associated router by which it is attached to the NoC. The directed edge $l_{ij} \in L$ represents a direct communication link between the router for processors PE_i and PE_j. The weight of the edge l_{ij}, denoted by bw_{ij}, indicates the bandwidth available across the edge l_{ij}.

Task Mapping: A *processor allocation* of the task graph G on a finite set of processors P is the processor allocation function *map:* $T \longmapsto P$ such that if a given task $t_i \in T$ is assigned to $PE_j \in P$, $map(t_i) = PE_j$ determines the spatial allocation of tasks.

In dynamic task mapping, processor allocation is invoked when an allocated task requests to communicate with a task that has not yet been mapped/scheduled. Temporal assignment is the determination of starting time of execution of each task. It assumes that allocation of tasks to processors has been completed before.

Task Scheduling: A *schedule* of a task graph G on a finite set P of processors is the function pair $(start, map)$, such that

—$start : T \longmapsto \mathbb{Q}_+$ is the function giving the start time of a task in G,
—$map : T \longmapsto P$ is the processor allocation function,

where $start(t_i)$ represents the time at which task t_i begins execution on an identified PE given by $map(t_i)$ [Sinnen 2007].

Thus, the two functions *start* and *map* describe temporal and spatial assignment of tasks, represented by the nodes of the task graph, to the processor set P of a target system. If the time at which a scheduled task is completed is given by *finish*(), then

$$finish(t_i) = start(t_i) + ex_i. \tag{2}$$

Communication Time: *Communication time* $ct(e_{ij})$ of an edge e_{ij} is the time the communication volume (represented by w_{ij}) takes from the origin PE to completely arrive at its destination PE. In other words, $ct(e_{ij})$ is the time taken for sending the data between PEs executing tasks t_i and t_j. Due to the property of the system model considered, no two tasks can execute on the same PE at the same time. Also, the dependency between the tasks of an application G must be met. The following conditions express this:

Condition 1.1: For any two tasks, t_i and $t_j \in T$,

$$map(t_i) = map(t_j) \Rightarrow start(t_i) \geq finish(t_j) \text{ or } start(t_j) \geq finish(t_i). \tag{3}$$

Condition 1.2: For every edge e_{ij},

$$start(t_j) \geq finish(t_i) + ct(e_{ij}). \tag{4}$$

Earliest Available Time: For a schedule $\mathcal{S}$ of a task graph G on P, the earliest time at which a $PE_k \in P$ will be available is given by function EAT defined as

$$EAT(PE_k) = \max_{\forall t_i \in T \mid PE_k = map(t_i)} finish(t_i). \tag{5}$$

When all the leaf tasks of G finish, the schedule $\mathcal{S}$ terminates and the resources used are released, which may then be allocated to a new application.

Task Allocation Modes: A task t_i has an *allocation mode* denoted by $\psi_i = \{EA, FAR\}$, where EA represents the communication-energy-aware mode in which task t_i is assigned to a core in a given NoC platform such that the communication energy of the application is reduced. FAR denotes failure-aware redundancy mode, which is activated for task t_i to tolerate core failures. For a given degree of fault tolerance f, we allocate f redundant copies of a task. For the task set T with N tasks, we use an array $\psi = (\psi_1, \psi_2, \ldots, \psi_N)$ to indicate the allocation decision taken for all tasks in the given application. Each element of the array belongs to subset T_{EA} or T_{FAR}, $T_{EA} \cap T_{FAR} = \emptyset$, where T_{EA} and T_{FAR} denote the set of tasks allocated in the EA and FAR modes, respectively. It may be noted that for each task marked to be implemented in failure-aware mode, there are duplicate copies of the task. Thus, defining $T_{mod} = T_{EA} \cap T_{FAR}$, we have $|T_{mod}| \geq |T|$ depending on the degree of fault tolerance.

Communication Energy: Energy is consumed when data packets are transferred from a source PE to a destination PE. The Manhattan distance between the source and destination PEs is used to obtain the number of physical links N_l and the number of routers N_r, traversed for communication between two dependent tasks. Let E_r be the energy consumption in pJ in a router for 1-bit transmission and E_l represent the energy consumption in pJ in a link to transmit 1 bit. Communication energy, E_{comm}, for allocated applications on a given NoC platform is estimated as

$$\begin{aligned} E_{comm} &= \sum_{\forall G_i} \sum_{\forall e_{ij} \in G_i} (E_r * N_r + E_l * N_l) * w_{ij} \\ N_r &= HC(map(t_i), map(t_j)) + 1 \\ N_l &= HC(map(t_i), map(t_j)), \end{aligned} \tag{6}$$

where HC represents the hop count and w_{ij} is the communication volume between tasks t_i and t_j in megabits.

4.1. Problem Statement

Given the following as inputs:

(1) A set $\mathbb{G}$ of arrived applications, each of which is represented by an application task graph $G = (T, E)$
(2) A target NoC platform given by $N = (P, L)$
(3) Timing information of the tasks of the applications
—Execution time, $ex_i > 0$
—Completion deadline, $dl_i \geq ex_i$

Determine a dynamic resource allocation for all tasks of the arrived application to find

—allocation array, ψ,
—$map : T_{mod} \longmapsto P \mid PE_{tasks} \leq PE_{cap}$, and
—$start(t_i)$ on the identified PE,

such that while tolerating f core failures:

(1) $finish(t_i) \leq dl_i$, and
(2) E_{comm} of application is reduced.

Migration energy for fault-tolerant design adds substantially to the energy consumption for any given application. As faults occur, tasks from faulty core(s) need to be migrated to other functional core(s). Low migration overhead is possible by remapping only the tasks from faulty core(s) and keeping all other task mappings unchanged. On the other hand, for real-time applications, satisfying the deadline demands remapping techniques with low execution overhead for fault recovery mechanisms. Methods such as Das and Kumar [2012] and Das et al. [2013] use the offline one-fault look-ahead policy to get a mapping, but it also imposes a penalty on deadline performance of the application. For real-time applications, we address the aforementioned challenges by taking advantage of a selective task replication policy for failure recovery.

5. THE PROPOSED APPROACH

The first stage of our algorithm honors the task deadline and minimizes the communication energy consumption of the application. Next, the algorithm addresses processor failures using a proactive policy where tasks allotted to susceptible PEs are replicated to minimize the effect of faulty processors on the executing application. The replicated tasks are mapped onto available PEs with higher reliability. In order to determine the candidate tasks for applying redundancy, a reliability-driven approach is adopted as outlined in the following subsection.

5.1. Reliability-Driven Selective Task-Redundancy

Before detailing the fault-tolerant resource allocation policy, we first determine the potential tasks for applying the fault-tolerant strategy. This is accomplished as follows.

In the mesh-based NoC topology, we have used thermal sensors to detect the temperature of individual cores. The recorded temperature values are then communicated to the Manager Core. Using the temperature value, the reliability of the corresponding PE, described in Section 5.3, is estimated and stored in the manager PE. Depending on the reliability value, the PEs are categorized into reliable and unreliable PEs as follows:

$$type(PE) = \begin{cases} reliable, if\ reliability\ of\ PE\ >\ critical\ reliability \\ unreliable, otherwise. \end{cases} \tag{7}$$

To determine the value of critical reliability for PE categorization, the temperature threshold is selected as in Ahmed et al. [2014]. Although unreliable PEs are available for runtime task allocation, they are more likely to undergo failure during task execution. Thus, the task running on such a PE should have a backup that may be used if the corresponding PE fails while the application is running. This strategy enables tasks of applications to execute uninterrupted despite core failures.

To improve the system reliability, we have proposed a software-redundancy-based task replication technique. This method generates a cloned task that inherits all the processing requirements such as execution time, data dependency on other tasks, and completion deadline from the original task. The cloned/recovery tasks obtained by replication are available in the system in two forms and are executed depending on the time of their activation: active replication and passive replication [Eles et al. 2008; Ahn et al. 1997]. Active replication is a space redundancy technique, where the duplicated tasks are executed along with their original counterparts, irrespective of the occurrence of faults. On the other hand, passive replication involves activation of the backup task only if the fault occurs at the PE executing the original task. It assumes that the system is equipped with a fault detection mechanism that updates the failure status of the individual processors to the Manager Core. If the task assigned on a failed PE has a duplicate copy on a different PE, at the instance of failure, the duplicate task is activated.

5.2. Fault-Aware Dynamic Task Allocation (FA-DRA)

Algorithm 1 presents the proposed scheme. In order to allocate resources to an application, we initialize the *Ready_List* with the mature tasks, ready to be executed. Mature tasks are tasks with no predecessors or whose predecessors have already finished execution. The function $find\_unoccupied\_PE\_list$ keeps a record of the set of processors currently free. Based on the availability of free processors and the Task Selection Function $TSF()$, tasks in the *Ready_List* are scheduled in a particular sequence. As the time taken to run the algorithm adds to the fault-aware core allocation time, we consider the timing characteristics of tasks to choose $TSF()$:

1. Minimum Processing Time First (Min_exc): $TSF(t_i) = Min(ex_i)\ \forall\ t_i \in Ready\_List$
2. Maximum Processing Time First (Max_exc): $TSF(t_i) = Max(ex_i)\ \forall\ t_i \in Ready\_List$
3. Least Deadline First (Min_dl): $TSF(t_i) = Min(dl_i)\ \forall\ t_i \in Ready\_List$

The *Minimum Processing Time First* scheme selects the task with the least execution time, while the *Maximum Processing Time First* does task selection based on the maximum execution time for mapping and scheduling. The *Least Deadline First* heuristic chooses the task having the earliest deadline among the mature tasks to allocate next.

ALGORITHM 1: ***Fault-Aware Dynamic Resource Allocation***

```
Input: $G = (T, E)$; $N = (P, L)$;
Output: fault-aware mapping and scheduling $\forall\ t_i \in T$
1  $Ready\_List \leftarrow$ mature tasks of application;
2  $ProcAvl = find\_unoccupied\_PE\_list()$;
3  $ProcBusy = \emptyset$;
4  $PE_{rel} = \emptyset$;
5  \\$PE_{unrel}$ is the set of PEs with low reliability
6  $PE_{unrel} = get\_unreliable\_PE()$;
7  while $Ready\_List != \emptyset$ do
8      for $t_i \in Ready\_List$ do
9          $Sel\_task = TSF(t_i)$;
10     if $ProcAvl != \emptyset$ then
11         $Taget\_Parent$ = most communicating parent of $Sel\_task$;
12         $Best\_Free\_PE$ = processor assigned to $Target\_Parent$;
13         if $Best\_Free\_PE \in ProcAvl$ AND can support $Sel\_task$ then
14             $PE_{sel} = Best\_Free\_PE$;
15             update $ProcAvl$ and $ProcBusy$;
16         else
17             $ProcBusy = find\_occupied\_PE\_list(PE_k)$ with EAT < Slack time of $Sel\_task$;
18             if $ProcBusy != \emptyset$ then
19                 $Best\_Busy\_PE$=select a PE $\in ProcBusy$ nearest to $Target\_Parent$;
20                 $PE_{sel} = Best\_Busy\_PE$;
21             else
22                 $Target\_Parent$ = next most communicating parent;
23                 go to line 12;
24     \\Identify a PE with reliability higher than $PE_{sel}$
25     if $PE\_sel \in PE_{unrel}$ then
26         $Sel\_task^*$ = replica of $Sel\_task$;
27         for $PE_j \in P$ do
28             if $PE_j \in ProcAvl$ AND $PE_j \notin PE_{unrel}$ then
29                 $PE_{rel} = PE_{rel} \cup PE_j$;
30         $PE_{recovery}$= minimum distance $PE \in PE_{rel}$ from $PE_{sel}$
31     allocate $Sel\_task$ to $PE_{sel}$ and $Sel\_task^*$ to $PE_{recovery}$;
32     assign start-time of $Sel\_task$ on $PE_{sel}$ and $Sel\_task^*$ on $PE_{recovery}$;
33     update EAT of $PE_{sel}$ and $PE_{recovery}$;
34     delete $Sel\_task$ from $Ready\_List$;
```

A task can have dependency on multiple predecessors. In case of multiple parents, it is preferable to map the task to the processor of the highest communicating parent, provided that processor is available. However, if such a PE is unavailable, the algorithm does not assign the task to an available PE closest to its parent, even if it can support the task. This is in contrast with the as-soon-as-possible paradigm of resource allocation. The primary idea is that if the task has enough slack time, a PE advantageous in terms of communication energy consumption can be chosen. The algorithm then uses $find\_occupied\_PE\_list()$ to choose a target PE for execution of the selected task depending on the position and Earliest Available Time (EAT) of the busy PEs. Such a PE can be the one on which the parent was executed, or in its neighborhood.

The currently executing task on such a PE should finish within the slack time margin of the selected task. Assigning such a PE helps to lower the communication energy while satisfying its deadline constraint. In case this condition is not met, the selected task is assigned to an available PE, which is closest to the PE where the parent task was executing.

Next, we describe the fault-aware dynamic resource allocation scheme. As seen in Algorithm 1, PEs are selected depending on the task deadline and communication energy consumption. The selected PEs are then checked for susceptibility to failure using their corresponding reliability values. If the selected PE is unreliable, the algorithm clones the original task and executes the fault-tolerant scheme presented in lines 25 to 30 of Algorithm 1. It tries to map the recovery task to an available PE in close vicinity of the original task to reduce the communication overhead. The chosen PE (PE_{rel}) should be a PE not present in the set of unreliable PEs, PE_{unrel}. Such a PE is referred to as reliable PE (PE_{rel}). Next, the algorithm allocates the cloned task on $PE_{recovery}$, a reliable PE chosen closest to PE_{sel}. Once the PE for the selected task and its replica (PE_{sel} and $PE_{recovery}$, respectively) are decided, the start times of execution of the tasks on the identified PEs are assigned depending on the communication delay with its predecessors. After completing the resource allocation for the chosen task, the task is deleted from the *Ready_List*. Finally, the set of occupied and free PEs is updated. This process is continued till all the tasks of applications are mapped and scheduled (with/without redundancy).

5.3. Reliability Analysis

The proposed task recovery mechanism replicates tasks executing on a PE that is likely to fail and resumes execution in the form of its replica on a different PE. In order to decide the tasks to be replicated, a model to determine the reliability of the processing core is required. The failure rate of each PE present in the network graph is estimated by using the model presented in Chou and Marculescu [2011]. It associates the failure rate of PE with its temperature and is given as

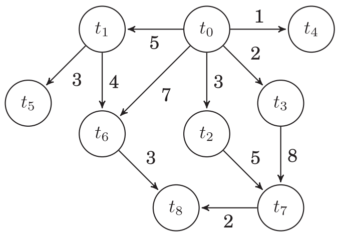
$$\lambda_p = Ae^{\frac{-E}{KT}}, \tag{8}$$

where E is the activation energy (0.9eV), K is the Boltzmann constant (8.617×10^{-5} eV K^{-1}), and T is the absolute temperature given in Kelvin. A is constant, and its value is selected such that the failure rate per cycle for each core operating at useful life is 10^{-11}, under normal core temperature, that is, 55°C [Chou and Marculescu 2011]. The previous equation is used for reliability analysis of the PEs in the NoC system. After the PEs have gone through the prenatal mortality period, their reliability is formulated [Chang et al. 2011] as

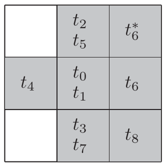
$$R(t) = e^{-\lambda_p t}. \tag{9}$$

Based on the value of the reliability of individual PEs, a subset of them are marked as PE_{unrel}. These are the set of PEs that are more likely to fail as their reliability values are low. For a mesh-based NoC system, composed of PEs, routers, and interconnecting links, system reliability can be modeled using a series configuration where failure of any component results in the failure of the entire system. As router and link reliability computation are beyond the purview of this article, they are not taken into account while formulating reliability for NoC. Schemes described in Chatterjee et al. [2014] may be applied to address such cases. Thus, the expression for reliability is defined as

$$R_{PE\_total}(t) = \prod_{j=1}^{|P|} R_P^j(t). \tag{10}$$



(a) Application G_1



(b) FA-DRA

$Task\ Id$	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
$Exe.\ Time(msec)$	2	2	4	6	6	2	4	1	3
$Deadline(msec)$	2.5	5	7	8	9	9	11	11	14

(c) The timing characteristics of the given application G_1

Fig. 2. Task mapping for application task graph G_1 using FA-DRA.

Here, $R_P^j(t)$ is the reliability of j^{th} PE at time t. $R_{PE\_total}(t)$ corresponds to the reliability of all PEs taken together. Incorporating the aforementioned reliability model in fault-tolerant application mapping, the number of PEs executing the tasks of an application should be taken into consideration. Therefore, we obtain the application reliability for a given NoC platform as

$$R_{app}(t) = \prod_{\forall i | PE_i \in PE_{map}} R_P^i(t), \qquad (11)$$
$$where, PE_{map} = \{PE_i | PE_i = map(t_i), \forall t_i \in T\ of\ G(T, E)\}.$$

While allocating resources, a task may be mapped onto an available PE_{unrel}. Since such PEs are less reliable (i.e., they are more susceptible to faults), a fault-tolerant policy is necessary. Toward this end, cloned copies of the original tasks are assigned to PEs with higher reliability. Otherwise, original tasks are considered for mapping onto PEs without any recovery copies. Therefore, the series model of application reliability is transformed into the series-parallel model by incorporating the proposed fault-tolerant approach. As a result, the new application reliability becomes

$$R_{app}{}'(t) = \prod_{\forall i | PE_i \in PE_{map}} R_P^i(t) + \sum_{\substack{\forall j | PE_j \in PE_{unrel} \\ \forall k | PE_k \notin PE_{unrel}}} (1 - R_P^j(t)) R_P^k(t), \qquad (12)$$

where PE_{unrel} represents the set of processors whose reliability is low.

$$\Delta R_{app}(t) = R_{app}{}'(t) - R_{app}(t) \qquad (13)$$

The increase in application reliability with the fault-tolerant policy of task replication is shown in Equation (13).

5.4. Working of the Proposed Algorithm

In this section, the working of the proposed algorithm has been illustrated using application G_1 on a 3×3 NoC platform, as shown in Figure 2. The details about the tasks and other characteristics, such as execution time and deadline, have been mentioned in the figure. We have represented the NoC platform using a grid structure, where each grid represents a PE. The PEs are numbered using a row major order, considering the left-hand top corner grid as the starting PE, that is, PE_0. The shaded grids represent the selected PEs, and the labels inside them indicate the tasks assigned to that PE. For simplicity, we have considered that at most two tasks can be mapped per PE. In the given example, PE_5 is assumed to be an unreliable PE. The *TSF*() function uses the *Minimum Processing Time First* scheme for task selection from *Ready_List*.

Let the root task t_0 be mapped onto the starting PE (PE_4). After the task t_0 completes, the children tasks of t_0 (i.e., t_1, t_2, t_3, and t_4) are ready for execution. Applying the task selection function, t_1 is the next candidate selected for mapping. As PE_4 is the processor where the parent task was executed and is available, it is selected for mapping of t_1. Next, t_2 is selected by *TSF*(). As the parent processor, PE_4 is occupied, and the neighbourhood of PE_4 is searched. Four possible positions at one-hop distance are available, all of which are equally suitable. Let t_2 be allocated to PE_1. The algorithm uses a similar procedure for mapping the rest of the tasks t_3 and t_4. When t_1 completes, t_5 and t_6 are ready for execution. As t_5 has a lower execution time compared to t_6, the *TSF*() selects t_5 and allocates it to PE_1.

Since more than two tasks cannot be mapped onto a single PE, PE_1 and PE_4 are not available for allocating t_6. Next, the algorithm explores the occupied neighboring PEs at one-hop distance and checks for the condition mentioned in step 17 of the algorithm. Since both PE_3 and PE_7 are busy and their earliest available time exceeds the slack time of t_6, task t_6 is assigned to PE_5. As PE_5 is an unreliable PE, it is likely to fail while executing t_6. Therefore, a replica of t_6 (i.e., t_6^*) is created and assigned to PE_2 with higher reliability and is a close neighbor of the PE_5 where the original task is allotted. This policy is carried out by lines 25 to 30, in Algorithm 1. After completion of PE allocation for the duplicate task, the algorithm is executed until all remaining tasks in the *Ready_List* are mapped and scheduled. Next, task t_7 is ready to be mapped. Algorithm 1 intelligently allocates it to the neighborhood of the most communicating parent, that is, t_3 (mapped to PE_7) instead of t_2 (mapped to PE_1), which aids in reducing communication energy. Task t_8 is mapped to PE_8, which is in the vicinity of processor PE_5 and PE_7 where its parents tasks t_6 and t_7 are mapped, respectively—thus, the final mapping obtained by the proposed approach as shown in Figure 2(b). It is observed that the number of tasks allocated to an unreliable PE is less compared to the reliable ones. This reduces the overhead of replication of additional tasks if assigned to the unreliable processors.

5.4.1. Response of FA-DRA to PE Failures. In this section, we describe the behavior of the FA-DRA algorithm on the occurrence of faults in PEs. The system on which the application is mapped may consist of unreliable PEs, that is, PEs with low reliability. Let us consider the situation given in Figure 2. Using the FA-DRA algorithm, task t_6 is assigned to an unreliable PE, PE_5. As PE_5 is prone to failure, Algorithm 1 makes a replica of t_6, that is, t_6^*, for safety. Using the replication policy mentioned in Algorithm 1, the cloned copy t_6^* is mapped onto PE_2. This cloned task helps to overcome the unavailability of the original PE and reduce its impact on the execution of the application.

We explain a fault scenario in which PE_5, where t_6 was originally assigned, becomes faulty at a time instant, say, $t = 9$. Two strategies can be adopted for re-executing t_6^* on the allocated processor PE_2. Figure 3(a) shows the active replication strategy where the replica is scheduled in a parallel manner for execution independent of occurrence of faults. Under such a scheme, the result of computation of t_6^* is available to compensate for the loss of data of the original task t_6 due to failure of PE_5. Here, we observe that t_6 meets its deadline even though PE_5 failed. Figure 3(b) shows a passive replication strategy where the replica t_6^* is executed only after failure of the core. t_6 starts its execution at $t = 6$, but owing to the failure of its assigned processor at $t = 9$, it

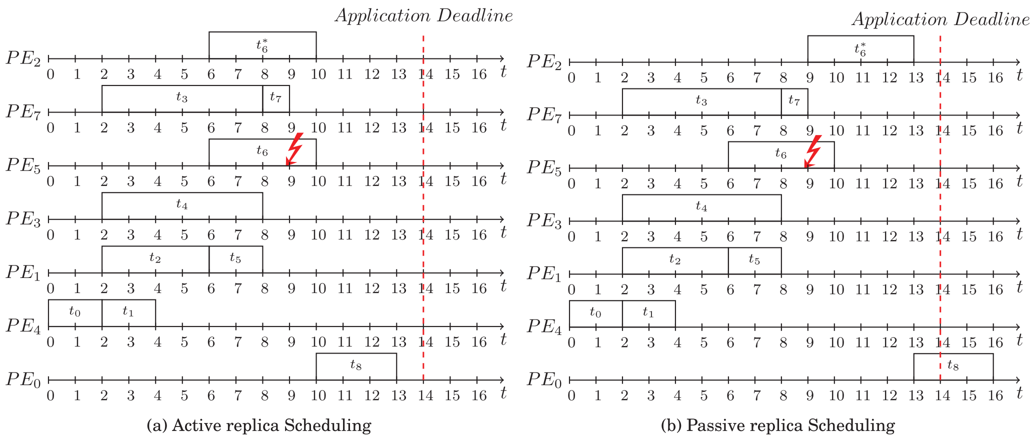


Fig. 3. Different fault-tolerant scheduling polices.

remains incomplete. In response to this core failure, task t_6^* starts execution on PE_2. Subsequently, task t_6^* completes at time $t = 13$, violating its deadline. The finish time of the application extends by three time units compared to that of Figure 3(a).

6. PERFORMANCE EVALUATION

6.1. Test Setup

We have developed a C++-based discrete event-driven simulator for dynamic mapping and scheduling of applications onto an NoC-based MPSoC system. The input to the simulator is the list of applications, the network architecture of the platform, strategies for mapping and scheduling, the routing algorithm, and the fault tolerance policy. The applications to be mapped are maintained as various input files. Next, we have allowed the user to select a target platform size with the choice of degree of multitasking of the constituent PEs (i.e., number of tasks each PE can support). The routing module of the simulator determines the route as per the selected routing method, by which the communication of packets between the tasks takes place. Faults are injected randomly in the PEs and depend on the maximum number of faults that the system can withstand without failure. The selective task duplication policy as mentioned in Section 5 has been used for fault tolerance. The simulator is triggered as per the occurrence of an event. An event is identified as the arrival of any application or departure of a completed application from the target platform or failure of a functional PE. Whenever one such event occurs, the allocation algorithm is executed. We have assumed that the centralized manager core that controls the dynamic allocation and fault mitigation is fault free and is always functional during the course of simulation. Depending on the choice, the corresponding modules for recording the selected performance metrics of interest are enabled for various test cases. In our case, the primary objective consists of honoring the task deadlines and dynamically minimizing the communication energy consumption of all running applications while imparting fault tolerance to the multicore platform.

For each task of application, its execution time requirement has been assigned randomly. For a task t_i, the corresponding deadline is allotted as $dl_i = k + ex_i$, where ex_i is the task execution time and k is a simulation parameter with positive value. Depending on the value of k, the urgency of scheduling for a given task is estimated. The smaller the value of k, the lesser is the slack time available for scheduling the task. On the other hand, a large value of k indicates higher slack time availability. This is

Table I. Test Categories

Category	Description
Urgent	$sl_i \leq 0.3 \times ex_i$
Moderate	$0.3 \times ex_i < sl_i \leq 0.6 \times ex_i$
Relaxed	$sl_i > 0.6 \times ex_i$

Table II. Simulation Parameters Used in ORION 2.0

Parameter Name	Value
Technology	90nm
Transistor Type	NVT
V_{dd}	1.0V
Router Frequency	500MHz
Flit Width	32
Number of Virtual Channels	2
Input Buffer Size	16
Number of Pipeline Stages	4
Wire Layer Type	Global
Wire Width and Spacing	DWIDTH_DSPACE

shown in Table I. In case of the Urgent category, the task to be scheduled has a time window that is only 30% more than the execution time. Similarly, for the Moderate and Relaxed categories, the corresponding time windows are shown in Table I. Simulations have been conducted on 1,000 test cases each composed of 100 different test scenarios. These scenarios are randomly generated and consist of combinations of the applications of the aforementioned test categories with varying grades of scheduling difficulty. In each of these scenarios, random PE faults are injected and distributed across the NoC platform. The test cases are investigated, taking into account both single and multiple PE failures along with the order of their occurrence. The success of a test case is determined by the number of applications successfully scheduled (i.e., satisfied their deadline in the presence of PE failures). For brevity, 15 representative test results have been reported.

We have conducted experiments for a 2D mesh-based NoC platform with link energy ($E_{link} = 3.125 \times 10^{-13}\ J/bit$) and router energy ($E_{router} = 5.24 \times 10^{-12}\ J/bit$) derived from the ORION 2.0 power model [Kahng et al. 2009]. Table II shows the parameter settings for ORION. For simplicity, we have used a generic five-port router for energy evaluation. The model used to determine the communication energy and communication time is based on the Manhattan distance between the source and destination node. In case the NoC is heavily loaded (i.e., the traffic is high), the waiting time in the buffers would rise. A sophisticated delay model would be required to estimate the communication time and energy spent in packet communication between the PEs. As the proposed algorithm deals with fault tolerance, such cases fall out of the scope of this work. However, the congestion-aware delay estimation model given in Chao et al. [2016] and Carvalho and Moraes [2008] could be used to estimate and mitigate the network congestion problem.

Simulation has been carried out on an Intel i5 processor running at 3.0GHz frequency. A target platform of an 8×8 NoC with one Manager Core at the center of the platform has been considered for experiments. We have used XY routing for transmission of data (in flits) between parent and child tasks (original and duplicate) mapped on different PEs. The data packets from the original and recovery task copies are distinguished at the destination PE using the source address. The proposed algorithms have been tested on both real benchmarks and synthetic applications. Real benchmarks

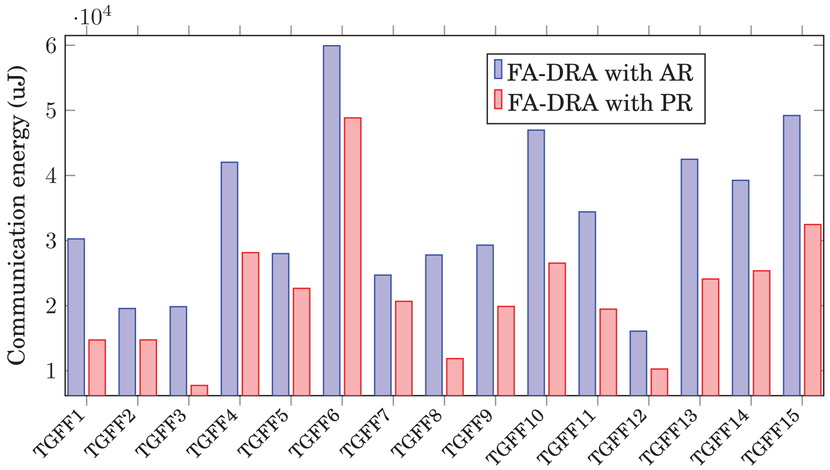


Fig. 4. Effect of task replication on communication energy of the mapped applications.

used in simulation are 263 decoder Mp3 decoder, 263 encoder Mp3 decoder, MPEG, MWD, and Mp3 encoder Mp3 decoder. Synthetic applications with different topologies and computation-communication behavior were generated using TGFF [Dick et al. 1998]. Each such application has a number of tasks varying from five to 64. While representing the results, the nomenclature used for synthetic applications are given as TGFFX, where "X" represents the application number. Each task is analyzed based on its computation-communication behavior. The computation time for each task depends on the PE on which it has been mapped. As we have considered a homogeneous platform, PEs have identical clock frequency, and the computation energy consumed by a task is similar for all PEs in the given NoC platform. In our experiments, the computation time requirement of the tasks has been uniformly distributed between 5 and 300 clock cycles.

6.2. Evaluation of Fault-Aware Dynamic Resource Allocation Algorithm

In this subsection, we present the performance of the proposed algorithm to mitigate the effect of unreliable PEs on application execution. In the simulation exercises, we have assumed the unreliable PEs to be 10% of total PEs supported by the given NoC platform. We also show the overhead incurred by the fault-tolerant policy on the deadline and energy consumption of the running applications.

By choosing tasks with closer deadlines, the tasks can be completed within their deadline. However, when tasks with a high volume of data communication demand resources, they are more likely to get suitable free PEs, provided the already mapped tasks on such PEs complete their execution. Therefore, giving preference to smaller time-consuming tasks, when allocating resources to tasks in *Ready_List*, improves the communication energy consumption. This helps to reduce the hop counts between the communicating PEs, as the highly communicating tasks are mapped to PEs close to one another. Thus, the Min_exc function has been used in the proposed algorithm for fault-tolerant dynamic mapping and scheduling.

6.2.1. Communication Energy. At first, we analyze the behavior of different variations of task redundancy—active and passive replication—used in the FA-DRA algorithm for task recovery. In Figure 4, it can be observed that Active Replication (AR) consumes more communication energy than Passive Replication (PR). On average, the active replication strategy resulted in 35% more communication energy of the mapped

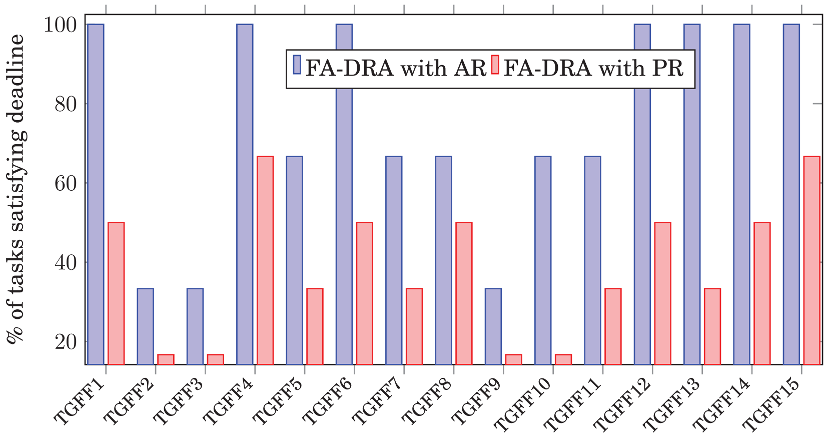


Fig. 5. Task deadline satisfaction for different task replication techniques in FA-DRA.

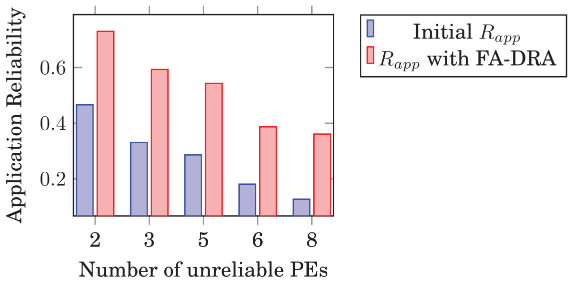


Fig. 6. Change in application reliability with faults.

application compared to that of passive replication. This is due to the fact that the recovery tasks under the active replication strategy are always operating in the background. This increases the use of network routers and links of the NoC platform along with the computation energy of the cores on which the replicated tasks are placed. In contrast, cloned tasks under the passive replication scenario being activated only in case of occurrence of faults gives a reduced increase in communication energy.

6.2.2. Deadline Performance. The deadline performance of the FA-DRA algorithm under active and passive replication policies is shown in Figure 5. We observe that the number of tasks finishing execution within the deadline is, on average, 45.73% more in case of active replication compared to that of passive replication. This is attributed to the time of failure of a particular PE, which is not known a priori. If a task t_i executing on PE_j fails at some time instant t and t happens to be very close to the finish time of the task, it has a small time margin left to satisfy its deadline when it is activated on PE fault detection. As a result, tasks having stringent deadlines fail to complete within the given time. On the contrary, recovery tasks under active replication are scheduled in parallel with the original task, which lowers the occurrence of deadline misses.

6.2.3. Effect on Application Reliability. The application reliability for a given application on an NoC platform is given by Equation (11). From Figure 6, we observe that as the number of unreliable PEs increases, the initial reliability of execution of the application drops. The proposed algorithm successfully alleviates the drop in reliability by incorporating recovery copies for vulnerable tasks, as depicted in Figure 6. The tasks

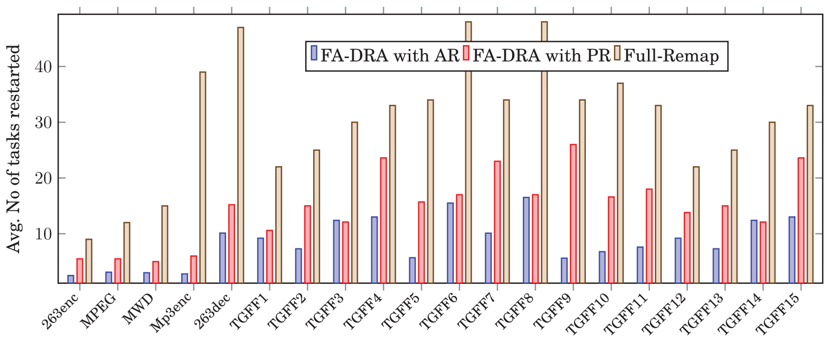


Fig. 7. Average number of tasks restarted.

executing on the unreliable PEs are duplicated and allotted to PEs having high reliability values. This enhances the reliability by exploiting the parallelism between the duplicated tasks.

Thus, it is seen from the analysis that the fault-tolerant policy of the proposed FA-DRA algorithm can potentially reduce the communication energy of the reconfigured application and also satisfies the deadline constraint of the given task set. Moreover, if the objective of the user is to reduce communication energy, the Manager Core implements passive replication of the tasks placed on unreliable PEs. In case of deadline-constrained application, active replication is implemented. As in dynamic scenarios the application characteristics are not known a priori, the fault-tolerant policy for dynamic resource allocation is decided by the user. Also, FA-DRA results in improvement in application reliability.

6.3. Comparison with Existing Works

Next, we compare the quality of the solution achieved by the algorithm proposed in this work with the solution given by similar algorithms [Das et al. 2015, 2013; Chou and Marculescu 2011; Khalili and Zarandi 2013] reported in literature. Results on both real and synthetic benchmarks have been shown to demonstrate the effectiveness of the solution obtained in this work. The number of faulty PEs varies between two and six and the results are averaged over all PE fault cases.

6.3.1. Number of Tasks Restarted. We have compared the impact of the FA-DRA algorithm and the Full Re-map [Das et al. 2015, 2013] policy, henceforth called FR, on the re-execution of tasks when responding to the failed cores. Figure 7 represents the average number of tasks restarted when FR and FA-DRA policies are used for each of the application task graphs. We observe that the FR policy results in a higher number of tasks needing to be restarted as compared to the policy used in FA-DRA. This is because, in case of FR policy, all tasks of the given application are re-executed, including those being executed by reliable cores, after reconfiguring the application to new mapping. On the other hand, task recovery in FA-DRA reduces such restarting overhead by using cloned tasks. The requirement of restarting the tasks occurs both in active and passive replication strategies when the tasks running on the failed cores have no replica. This happens when the failed cores are different from the one predicted by the reliability model in Section 6. Particularly, for passive replication, tasks re-execute in the form of its recovery copy when an unreliable PE executing it fails. On average, the task re-execution overhead in the passive replication strategy is 43.53% more compared to that of the active replication strategy. But both of these techniques

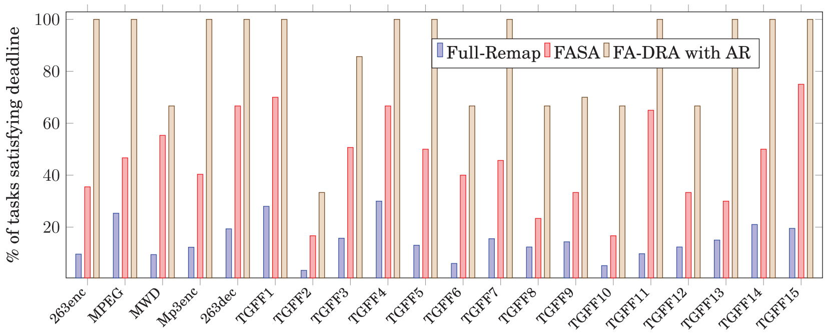


Fig. 8. Comparison of deadline performance.

when implemented in the FA-DRA algorithm give a 56.95% average reduction in task re-execution as compared to the FR policy.

The results of deadline performance are shown in Figure 8. The number of tasks restarted affects the deadline satisfaction of the application. In the FR policy [Das et al. 2015, 2013], remapping the tasks of applications to reliable PEs during failure of currently assigned PEs incurs time overhead. This is attributed to transferring the task state space and resuming its execution on the different PE. This is reflected from the results obtained in Section 6.2.1. The Failure-Aware Spare Allocation (FASA) [Khalili and Zarandi 2013] technique resumes the execution of task(s) affected by PE failure at the allocated spare core. This lowers the re-execution overhead of other tasks for the given application. Consequently, the deadline performance improves as compared to the FR policy. On the other hand, the proposed scheme when implemented with the AR policy further lowers the number of tasks restarted (depicted in Figure 7). Active replicas being scheduled in parallel with the original tasks ensures that the tasks complete execution within their deadline even in the event of PE failure. Experimental results show that, on average, a 47% improvement in task deadline satisfaction is achieved with respect to the FASA [Khalili and Zarandi 2013] scheme. When compared to the FR policy [Das et al. 2015, 2013], the proposed method achieves the best-case improvement in deadline of about 82%.

6.3.2. Energy Consumption. A processor is considered to be faulty if it cannot be used to execute tasks anymore.

Tasks on failed processors are migrated to an available processor for re-execution. Thus, Reconfiguration Energy is defined as the cost of migrating a task t_i from the failed processor PE_{fail} to recovery processor $PE_{recover}$. This is represented in Equation (14) [Maqsood et al. 2015]:

$$E_{reconfig}(t_i) = size(t_i) \times \left[\left(HC\left(PE^{t_i}_{fail}, PE^{t_i}_{recover}\right)\right) \times E_l + \left(HC\left(PE^{t_i}_{fail}, PE^{t_i}_{recover}\right) + 1\right) \times E_r\right]. \tag{14}$$

Here, $size(t_i)$ represents the size of task t_i, which includes both code and data. Therefore, the total energy spent in reconfiguring an executing application is given by

$$E^{total}_{reconfig} = \sum_{\forall i | t_i \in T} E_{reconfig}(t_i). \tag{15}$$

As depicted in Equation (15), the number of tasks restarted by the aforementioned policies affect the total reconfiguration energy. Figure 9 depicts the reconfiguration energy overhead normalized with respect to the FR strategy. We observe that the proposed FA-DRA algorithm results in 55.1% reduced reconfiguration energy as compared to the FR policy. This is due to the fact that the FR policy recomputes the allocation for all

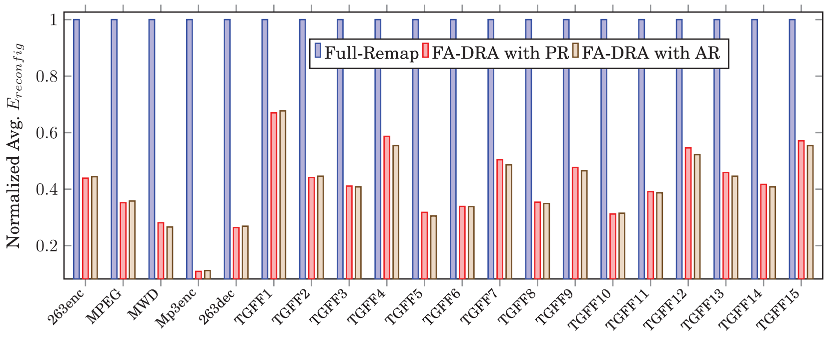


Fig. 9. Reconfiguration energy spent in fault-tolerant policies.

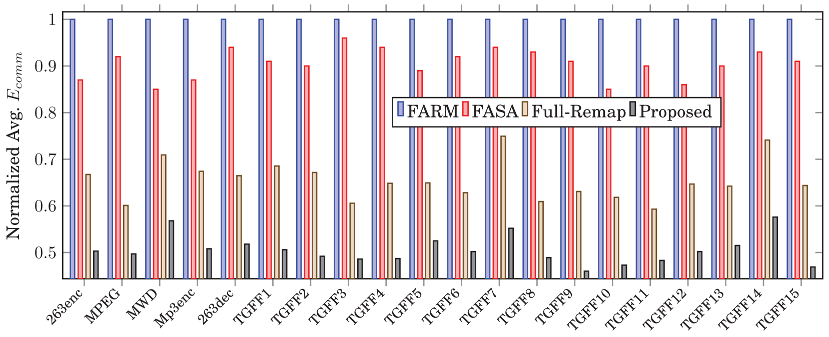


Fig. 10. Communication energy comparison with different fault-tolerant policies.

tasks of the running application upon PE failures. As a result, more energy is spent on migrating the tasks to newly allocated PEs. On the other hand, the FA-DRA policy, due to the use of task redundancy, avoids such overheads by executing the task replicas to recover the affected task(s) in the event of PE failures. However, when tasks executing on a failed PE do not have any replica, the application is reconfigured by reallocating the tasks. Therefore, the requirement of reconfiguring the entire application is reduced, which in turn saves the reconfiguration energy.

Figure 10 shows a comparison of the final value of average communication energy consumption after reconfiguring the application. The values are normalized with respect to the FARM [Chou and Marculescu 2011] technique. It is observed that FASA [Khalili and Zarandi 2013] results in, on average, 9.4% lesser energy consumption. This is because FASA determines the placement of spare cores dynamically for each application. This is in contrast to the FARM technique, which allocates a fixed number of spare core(s) at a predefined location on the NoC platform for all applications. As FASA considers the application characteristics while allocating the spare cores, it results in improved performance, compared to FARM. When the strategy of Das et al. [2015] is used with the initial mapping given in Das and Kumar [2012], the resultant mapping of tasks shows a further reduction in communication energy by 42% as compared to FASA. This is due to improved mapping of communicating tasks on the same or nearby PEs. The proposed FA-DRA technique further improves the communication energy by 31% on average by exploiting the timing information of the tasks while allocating them to PEs. It allocates heavily communicating tasks closer in terms of hop count. Due to this, the data packets need to traverse fewer intermediate routers and links, giving additional savings in communication energy compared to the aforementioned policies.

Thus, the observations show that for designing an energy-efficient fault-tolerant multiprocessor system, both the resource allocation strategy and fault-recovery mechanism affect the postfailure system performance.

7. CONCLUSIONS AND FUTURE WORKS

In this work, we have presented an improved fault-tolerant dynamic resource allocation policy. The proposed algorithm presents a unified mapping and scheduling method for real-time systems focusing on the application deadline and communication energy while mitigating the effect of failure-prone PEs on application execution. By selectively using a task replication policy, the reliability of the application executing on the given NoC platform is improved.

We conducted a detailed evaluation of the performance of the algorithm on both real and synthetic benchmark applications. On the basis of simulation results, it can be concluded that the proposed algorithm exhibits better performance in terms of deadline satisfaction, task re-execution overhead, and communication energy when compared with other fault-tolerant algorithms to perform processor fault-aware task assignment. Results show that using task slack time while placing the task replicas gives notable improvements in communication energy, alleviating the effects of faults on the executing application. In addition, the proposed algorithm is able to honor the deadline for most of the test cases. For fault-tolerant dynamic systems, where satisfaction of the energy constraint is necessary, the proposed algorithm is attractive to do online resource allocation. The future work can be broadly classified in three dimensions: (1) placement and fault tolerance of manager core, (2) traffic management depending on the workload, and (3) extension of present work considering a heterogeneous multicore platform.

REFERENCES

R. Ahmed, P. Ramanathan, and K. K. Saluja. 2014. Necessary and sufficient conditions for thermal schedulability of periodic real-time tasks. In *Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems (ECRTS'14)*. 243–252. DOI:http://dx.doi.org/10.1109/ECRTS.2014.15

K. Ahn, J. Kim, and S. Hong. 1997. Fault-tolerant real-time scheduling using passive replicas. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems, 1997*. 98–103. DOI:http://dx.doi.org/10.1109/PRFTS.1997.640132

O. Arnold and G. Fettweis. 2011. Resilient dynamic task scheduling for unreliable heterogeneous MPSoCs. *2011 Semiconductor Conference Dresden,* Dresden, 1–4. DOI:10.1109/SCD.2011.6068747

L. Benini and G. De Micheli. 2002. Networks on chips: A new SoC paradigm. *Computer* 35, 1 (Jan. 2002), 70–78. DOI:http://dx.doi.org/10.1109/2.976921

F. Bolanos, F. Rivera, J. E. Aedo, and N. Bagherzadeh. 2013. From UML specifications to mapping and scheduling of tasks into a NoC, with reliability considerations. *J. Syst. Archit.* 59, 7 (Aug. 2013), 429–440.

S. Borkar, T. Karnik, and V. De. 2004. Design and reliability challenges in nanometer technologies. In *Proceedings of the 41st Annual Design Automation Conference (DAC'04)*. ACM, New York, NY, 75–75. DOI:http://dx.doi.org/10.1145/996566.996588

E. Carvalho and F. Moraes. 2008. Congestion-aware task mapping in heterogeneous MPSoCs. In *Proceedings of the International Symposium on System-on-Chip, 2008 (SOC'08)*. 1–4. DOI:http://dx.doi.org/10.1109/ISSOC.2008.4694878

Y. C. Chang, C. T. Chiu, S. Y. Lin, and C. K. Liu. 2011. On the design and analysis of fault tolerant NoC architecture using spare routers. In *Proceedings of the 2011 16th Asia and South Pacific Design Automation Conference (ASP-DAC'11)*. 431–436. DOI:http://dx.doi.org/10.1109/ASPDAC.2011.5722228

H.-L. Chao, S.-Y. Tung, and P.-A. Hsiung. 2016. Dynamic task mapping with congestion speculation for reconfigurable network-on-chip. *ACM Trans. Reconfig. Technol. Syst.* 10, 1, Article 3 (Sept. 2016), 25 pages. DOI:http://dx.doi.org/10.1145/2892633

N. Chatterjee, N. Prasad, and S. Chattapadhyay. 2014. A spare link based reliable network-on-chip design. In *Proceedings of the 18th International Symposium on VLSI Design and Test*. 1–6. DOI:http://dx.doi.org/10.1109/ISVDAT.2014.6881036

C. L. Chou and R. Marculescu. 2011. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'11)*. 1–6. DOI:http://dx.doi.org/10.1109/DATE.2011.5763113

C. Constantinescu. 2002. Impact of deep submicron technology on dependability of VLSI circuits. In *Proceedings of the International Conference on Dependable Systems and Networks, 2002 (DSN'02)*. 205–209. DOI:http://dx.doi.org/10.1109/DSN.2002.1028901

C. Constantinescu. 2003. Trends and challenges in VLSI circuit reliability. *IEEE Micro* 23, 4 (July 2003), 14–19. DOI:http://dx.doi.org/10.1109/MM.2003.1225959

A. Das and A. Kumar. 2012. Fault-aware task re-mapping for throughput constrained multimedia applications on NoC-based MPSoCs. In *Proceedings of the 2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP'12)*. 149–155. DOI:http://dx.doi.org/10.1109/RSP.2012.6380704

A. Das, A. Kumar, and B. Veeravalli. 2013. Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'13)*. 689–694. DOI:http://dx.doi.org/10.7873/DATE.2013.149

A. Das, A. Kumar, and B. Veeravalli. 2014. Communication and migration energy aware task mapping for reliable multiprocessor systems. *Future Generation Comput. Syst.* 30 (2014), 216–228. DOI:http://dx.doi.org/10.1016/j.future.2013.06.016 Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, {ICPADS} 2012 Selected Papers.

A. Das, A. K. Singh, and A. Kumar. 2013. Energy-aware dynamic reconfiguration of communication-centric applications for reliable MPSoCs. In *Proceedings of the 2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC'13)*. 1–7. DOI:http://dx.doi.org/10.1109/ReCoSoC.2013.6581540

A. Das, A. Kumar Singh, and A. Kumar. 2015. Execution trace–driven energy-reliability optimization for multimedia MPSoCs. *ACM Trans. Reconfig. Technol. Syst.* 8, 3, Article 18 (May 2015), 19 pages. DOI:http://dx.doi.org/10.1145/2665071

R. P. Dick, D. L. Rhodes, and W. Wolf. 1998. TGFF: Task graphs for free. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign*. IEEE Computer Society, 97–101.

P. Eles, V. Izosimov, P. Pop, and Z. Peng. 2008. Synthesis of fault-tolerant embedded systems. In *Proceedings of the Design, Automation and Test in Europe, 2008 (DATE'08)*. 1117–1122. DOI:http://dx.doi.org/10.1109/DATE.2008.4484825

D. Fick, A. DeOrio, G. Chen, V. Bertacco, D. Sylvester, and D. Blaauw. 2009. A highly resilient routing algorithm for fault-tolerant NoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09)*. European Design and Automation Association, Belgium, 21–26. http://dl.acm.org/citation.cfm?id=1874620.1874628

M. S. Floyd, S. Ghiasi, T. W. Keller, K. Rajamani, F. L. Rawson, J. C. Rubio, and M. S. Ware. 2007. System power management support in the IBM POWER6 microprocessor. *IBM J. Res. Devel.* 51, 6 (Nov. 2007), 733–746. DOI:http://dx.doi.org/10.1147/rd.516.0733

H. Hajimiri, S. Paul, A. Ghosh, S. Bhunia, and P. Mishra. 2011. Reliability improvement in multicore architectures through computing in embedded memory. In *Proceedings of the 2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS'11)*. 1–4. DOI:http://dx.doi.org/10.1109/MWSCAS.2011.6026672

J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll. 2011. Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In *Proceedings of the 2011 Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'11)*. 247–256. DOI:http://dx.doi.org/10.1145/2039370.2039409

Intel. 2008. *Quad-Core Intel Xeon Processor 5400 Series*. Retrieved from http://www.intel.com/Assets/enUS/PDF/datasheet/318589.pdf.

A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. 2009. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition, 2009 (DATE'09)*. 423–428. DOI:http://dx.doi.org/10.1109/DATE.2009.5090700

F. Khalili and H. R. Zarandi. 2013. A fault-tolerant core mapping technique in networks-on-chip. *IET Comput. Digital Techniques* 7, 6 (Nov. 2013), 238–245. DOI:http://dx.doi.org/10.1049/iet-cdt.2013.0032

A. Kohler, G. Schley, and M. Radetzki. 2010. Fault tolerant network on chip switching with graceful performance degradation. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 29, 6 (June 2010), 883–896. DOI:http://dx.doi.org/10.1109/TCAD.2010.2048399

S. Kundu and S. Chattopadhyay. 2014. *Network-on-Chip: The Next Generation of System-on-Chip Integration*. CRC Press.

C. Lee, H. Kim, H. W. Park, S. Kim, H. Oh, and S. Ha. 2010. A task remapping technique for reliable multi-core embedded systems. In *Proceedings of the 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'10)*. 307–316.

F. Liberato, R. Melhem, and D. Mosse. 2000. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Trans. Comput.* 49, 9 (Sept. 2000), 906–914. DOI:http://dx.doi.org/10.1109/12.869322

T. Maqsood, S. Ali, S. U. R. Malik, and S. A. Madani. 2015. Dynamic task mapping for network-on-chip based systems. *J. Syst. Architecture* 61, 7 (2015), 293–306.

Y. Ren, L. Liu, S. Yin, J. Han, and S. Wei. 2015. Efficient fault-tolerant topology reconfiguration using a maximum flow algorithm. *ACM Trans. Reconfig. Technol. Syst.* 8, 3, Article 19 (May 2015), 24 pages. DOI:http://dx.doi.org/10.1145/2700417

E. Schuchman and T. N. Vijaykumar. 2005. Rescue: A microarchitecture for testability and defect tolerance. In *Proceedings of the 32nd International Symposium on Computer Architecture, 2005 (ISCA'05)*. 160–171. DOI:http://dx.doi.org/10.1109/ISCA.2005.44

S. Shamshiri and K. T. Cheng. 2011. Modeling yield, cost, and quality of a spare-enhanced multicore chip. *IEEE Trans. Comput.* 60, 9 (Sept. 2011), 1246–1259. DOI:http://dx.doi.org/10.1109/TC.2011.32

S. Shamshiri, P. Lisherness, S. J. Pan, and K. T. Cheng. 2008. A cost analysis framework for multi-core systems with spares. In *Proceedings of the IEEE International Test Conference, 2008 (ITC'08)*. 1–8. DOI:http://dx.doi.org/10.1109/TEST.2008.4700562

P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. 2012. Exploiting microarchitectural redundancy for defect tolerance. In *2012 IEEE 30th International Conference on Computer Design (ICCD'12)*. 35–42. DOI:http://dx.doi.org/10.1109/ICCD.2012.6378613

O. Sinnen. 2007. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience.

C. Wang, J. Wu, G. Jiang, and J. Sun. 2013. An efficient topology reconfiguration algorithm for NOC based multiprocessor arrays. In *Proceedings of the 2013 IEEE 10th International Conference on High Performance Computing and Communications, 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC EUC'13)*. 873–880. DOI:http://dx.doi.org/10.1109/HPCC.and.EUC.2013.125

C. Yang and A. Orailoglu. 2007. Predictable execution adaptivity through embedding dynamic reconfigurability into static MPSoC schedules. In *Proceedings of the 2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*. 15–20.

C. Yao, K. K. Saluja, and P. Ramanathan. 2011. Calibrating on-chip thermal sensors in integrated circuits: A design-for-calibration approach. *J. Electronic Testing* 27, 6 (2011), 711–721. DOI:http://dx.doi.org/10.1007/s10836-011-5253-4

L. Zhang, Y. Han, Q. Xu, and X. Li. 2008. Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology. In *Proceedings of the Design, Automation and Test in Europe, 2008 (DATE'08)*. 891–896. DOI:http://dx.doi.org/10.1109/DATE.2008.4484787

L. Zhang, Y. Han, Q. Xu, X. W. Li, and H. Li. 2009. On topology reconfiguration for defect-tolerant noc-based homogeneous manycore systems. *IEEE Trans. Very Large Scale Integration (VLSI) Syst.* 17, 9 (Sept. 2009), 1173–1186. DOI:http://dx.doi.org/10.1109/TVLSI.2008.2002108

Received July 2016; revised February 2017; accepted February 2017