Execution Trace–Driven Energy-Reliability Optimization for Multimedia MPSoCs

ANUP DAS, AMIT KUMAR SINGH, and AKASH KUMAR, National University of Singapore

Multiprocessor systems-on-chip (MPSoCs) are becoming a popular design choice in current and future technology nodes to accommodate the heterogeneous computing demand of a multitude of applications enabled on these platform. Streaming multimedia and other communication-centric applications constitute a significant fraction of the application space of these devices. The mapping of an application on an MPSoC is an NP-hard problem. This has attracted researchers to solve this problem both as stand-alone (best-effort) and in conjunction with other optimization objectives, such as energy and reliability. Most existing studies on energy-reliability joint optimization are static—that is, design time based. These techniques fail to capture runtime variability such as resource unavailability and dynamism associated with application behaviors, which are typical of multimedia applications. The few studies that consider dynamic mapping of applications do not consider throughput degradation, which directly impacts user satisfaction. This article proposes a runtime technique to analyze the execution trace of an application modeled as Synchronous Data Flow Graphs (SDFGs) to determine its mapping on a multiprocessor system with heterogeneous processing units for different fault scenarios. Further, communication energy is minimized for each of these mappings while satisfying the throughput constraint. Experiments conducted with synthetic and real SDFGs demonstrate that the proposed technique achieves significant improvement with respect to the state-of-the-art approaches in terms of throughput and storage overhead with less than 20% energy overhead.

Categories and Subject Descriptors: B.8.1 [**Performance and Reliability**]: Reliability, Testing and Fault-Tolerance; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design (CAD); D.4.7 [**Organization and Design**]: Real-Time Systems and Embedded Systems

General Terms: Design, Algorithms, Performance, Reliability

Additional Key Words and Phrases: Task mapping and scheduling, fault tolerance, energy consumption, multimedia applications, Synchronous Data Flow Graphs

ACM Reference Format:
Anup Das, Amit Kumar Singh, and Akash 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

1. INTRODUCTION

To accommodate the ever-increasing demand of applications and to address scalability, multiple processing cores are integrated using an interconnect network such as bus or

This work was supported in part by Singapore Ministry of Education Academic Research Fund Tier 1 with grant number R-263-000-655-133.
Authors' addresses: A. Das is a research fellow at the School of Electronics and Computer Science, University of Southampton, United Kingdom; email: a.k.das@soton.ac.uk; A. K. Singh is a research fellow at the Department of Computer Science, University of York, United Kingdom; email: amit.singh@york.ac.uk; A. Kumar is an assistant professor at the Department of Electrical and Computer Engineering, National University of Singapore, Singapore; email: akash@nus.edu.sg.

DOI: http://dx.doi.org/10.1145/2665071

networks-on-chip (NoCs) to form multiprocessor systems-on-chip (MPSoCs). Due to the power penalty associated with homogeneous architectures, modern MPSoCs integrate heterogeneous processing elements such as general-purpose processors (GPPs), field programmable gate arrays (FPGAs), and digital signal processors (DSPs). Examples of heterogeneous MPSoCs are OMAP from Texas Instruments, NEXPERIA from Philips, and NOMADIK from STMicroelectronics. Streaming multimedia applications such as the H.264 decoder and MP3 encoder constitute a large fraction of the embedded application space for these MPSoCs [Wolf 2005]. These applications are characterized by large data exchange among different tasks. Data communication agnostic mapping of these applications on an MPSoC can lead to a significant energy consumption on the communication infrastructure, such as NoCs, constituting as high as nearly 60% of the overall application energy consumption [Hu and Marculescu 2004]. This has motivated researchers to investigate communication-aware application mapping on MPSoCs [Singh et al. 2010].

Shrinking transistor geometries and aggressive voltage scaling are negatively impacting the dependability of the processing elements (e.g., cores) and the communication backbone of an MPSoC [Borkar et al. 2004]. One of the design objectives in deep submicron technologies is to provide support for tolerating multiple faults (transient, intermittent, and permanent) without sacrificing solution quality (measured as throughput for streaming applications) and respecting the given energy budget. One of the traditional techniques for fault tolerance is redundancy (hardware and/or software) [Koren and Krishna 2007]. This involves using spare processing elements to assume responsibility when faults occur (hardware redundancy) or executing the same task multiple times on same or different cores (software redundancy). However, stringent area and energy budgets are prohibiting the use of hardware redundancy in modern systems. System-level fault-tolerance techniques such as task mapping and scheduling are gaining popularity among the research community. Both offline [Das et al. 2012; Lee et al. 2010] and online [Derin et al. 2011; Chou and Marculescu 2011] analysis techniques have been studied over the years to determine task mappings for all processor fault scenarios. The offline analysis techniques offer limited flexibility to mapping readjustment after faults and a limited scope for new application support. The existing online techniques in this domain do not consider throughput degradation, which is an important consideration for streaming multimedia applications to deliver a given quality of service to end users.

1.1. Scope of This Work

This article attempts to solve the following problem: given a heterogeneous MPSoC architecture and a set of multimedia and other communication-centric embedded applications, how to assign and order the tasks of every application at runtime on the component cores such that the communication energy consumption is minimized while guaranteeing the satisfaction of the performance requirement (e.g., throughput) of the application under all possible core fault scenarios. The scope of this article is limited to permanent and intermittent faults of cores. It assumes a given MPSoC architecture (floorplan), and therefore the selection of cores (number and/or types) for the architecture and their placement (coordinates) are not addressed. Furthermore, the proposed technique deals with resource management after fault occurrence and is orthogonal to any fault-detection techniques (e.g., Reinhardt and Mukherjee [2000]). To the best of our knowledge, this is the first work on communication energy minimization considering throughput degradation for runtime fault tolerance of heterogeneous MPSoC platforms.

1.2. Contributions

This article proposes a trace-based runtime remapping of applications on a multiprocessor system to minimize communication energy while satisfying the throughput requirement for all core fault scenarios. The following are the key contributions:

—Execution trace-based dynamic reconfiguration of application mapping for different fault scenarios
—Communication energy and storage overhead minimization on a given MPSoC
—Consideration of throughput degradation for communication-centric multimedia applications
—Fault tolerance for heterogeneous MPSoC architecture
—Implementation of the runtime algorithm on a real embedded system.

The proposed technique analyzes the execution trace of an application modeled as Synchronous Data Flow Graphs (SDFGs) [Lee and Messerschmitt 1987]. Based on such analysis, a runtime manager (RTM) determines the most suitable task remapping that minimizes communication energy while satisfying the application throughput requirement. Experiments conducted with synthetic and real-life application SDFGs demonstrate that the proposed technique achieves significant improvement both in terms of throughput and storage overhead with less than 20% energy overhead from a static mapping.

In our recent work [Das et al. 2013b], some of these contributions are addressed. Specifically, an execution trace–based application remapping is proposed in Das et al. [2013b] for homogeneous multiprocessor systems to minimize communication energy while satisfying the throughput constraint for all processor fault scenarios. This article extends our earlier work by considering heterogeneous multiprocessor systems and the embedded implementation of the overall approach.

1.3. Article Organization

The rest of this article is organized as follows. The background on fault tolerance and a motivating example are presented in Section 2. An overview of related works is presented in Section 3. This is followed by an introduction to SDFGs and the problem formulation in Section 4. The execution trace–driven design methodology is presented in Section 5. Experimental results are presented in Section 6. Finally, Section 7 concludes the work with a discussion on future improvements.

2. BACKGROUND AND MOTIVATION

2.1. Background on Fault Tolerance

With shrinking feature size, dependability is becoming a major concern even for NoCs. It is predicted that a 20% to 30% fault rate is expected in future MPSoCs [Furber 2006]. These faults can be classified into three categories: transient, intermittent, and permanent. Transient faults are point failures and occur due to alpha or neutron emissions. Transient faults are measured in flit error rate. Prior work in transient fault tolerance reveals a flit rate between 10^{-9} and 10^{-12} for NoC [Murali et al. 2005]. Permanent faults, as the name itself indicates, are damages to the circuit caused by phenomena such as electromigration, dielectric breakdowns, and broken wires. These faults are caused during manufacturing or during the product lifetime due to component wearout. Behavior of a system under permanent faults is time invariant. Intermittent faults present separate challenges for system designers due to the unpredictability associated with their occurrence and their fault duration. Naively suspending the system operation for the intermittent fault duration can degrade system performance. Unlike transient faults due to single-event upsets, task re-execution cannot guarantee

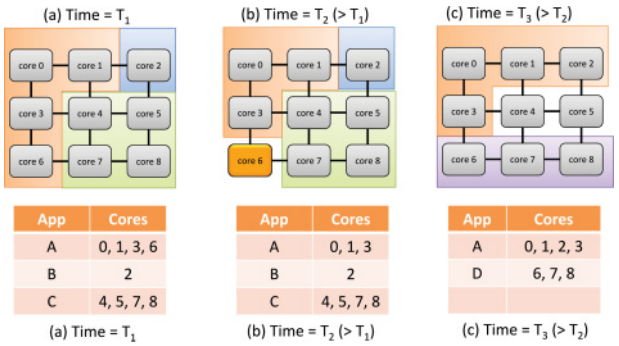

Fig. 1. Limitations of offline task mapping techniques.

elimination of the intermittent faults, as they usually persist for a few cycles, if not a few seconds or more. There are major factors contributing to intermittent faults—wear-outs, manufacturing defects, and PVT variations. Wearouts such as negative bias temperature instability (NBTI), hot carrier injection (HCI), and time-dependent dielectric breakdown are susceptible to PVT variations at deep submicron nodes (45nm and beyond) [McPherson 2006]. Studies have shown than microprocessors typically experience intermittent faults frequently after the first occurrence until the time this intermittent fault converts to permanent failure. Another major cause for intermittent faults is manufacturing defects. Although deterministic defects are detected in the testing phase, nondeterministic faults manifest as intermittent faults.

2.2. Limitations of the Offline Task Mapping Techniques

The existing design-time (offline) task mapping techniques suffer from two limitations: mapping readjustment and new application support. The existing design-time–based fault-tolerance techniques fail to readjust the task mappings on fault detection; Figure 1 shows a scenario at time $T = T_1$ with three applications (denoted as A, B, and C in the figure, respectively) active simultaneously on a multiprocessor system with nine cores. The cores utilized by each of these applications are reported in the corresponding table. These mappings are determined at design time by analyzing the throughput and communication energy. At a later instant of time, say at $T = T_2$, *core 6* becomes faulty (intermittent fault due to temperature increase). The mapping corresponding to application A needs to be altered. The design-time mapping for A corresponding to three cores is fetched and applied. This is shown in Figure 1(b). Eventually, at time instant $T = T_3$, *core 6* becomes operational and a fourth application D (of higher priority) is loaded in the system. This application requires *core 4*, *7*, and *8*. The other two applications B and C complete their execution. Thus, at time instant $T = T_3$, the application A has the choice of utilizing *core 2* or *core 4*. However, none of these two mappings for application A (mapping 0-1-2-3 or 0-1-3-4) has been analyzed at design time in terms of communication energy or throughput. Although the mapping 0-1-2-3 is selected for application A, this mapping provides no guarantee on energy or throughput. A point to note here is that for systems where analyzed task mappings from design time are only allowed, application A will continue to execute on three cores (0-1-3) with an acceptable throughput degradation, even though there are idle cores available in the system.

Another limitation of the design-time approaches is that the applications need to be known before the actual hardware implementation. This poses a restriction on the support of new applications post device manufacturing.

3. RELATED WORKS

Task mapping and scheduling have received significant attention among researchers starting from classical optimization metrics (e.g., performance and power) up to recent ones (e.g., reliability). The details of performance and power-driven mapping techniques are beyond the scope of this work. Interested readers can refer to Singh et al. [2013]. Some of the key studies on communication energy and reliability-aware task mapping are presented here.

The existing communication energy-aware fault-tolerant techniques can be classified into two categories: static and dynamic. Static task mapping techniques compute task mapping decisions at design time for different fault scenarios. As faults occur, these mappings are looked up at runtime to carry out task migration. A fixed order Band and Band reconfiguration technique is studied in Yang and Orailoglu [2007]. Cores of target architecture are partitioned into two bands. When one or more cores fail, tasks on these core(s) are migrated to other functional core(s) determined by the band in which these tasks belong. The core partitioning strategy is fixed at design time and is independent of the application throughput requirement. Consequently, throughput is not guaranteed by this technique. A re-execution slot–based reconfiguration mechanism is 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, such as throughput degradation. At runtime, tasks on a faulty core migrate to their re-execution slot on a different core. However, a limitation of this technique is that the schedule length can become unbounded for high fault-tolerant systems. A task remapping technique based on offline computation and virtual mapping is proposed in Lee et al. [2010]. Here, task mapping is performed in two steps: determining the highest throughput mapping followed by generation of a virtual mapping to minimize the cost of task migration to achieve this highest throughput mapping. A limitation of this technique is that the migration overhead significantly increases, as this is not considered in the initial optimization process. Moreover, throughput-constrained streaming applications do not benefit from a throughput higher than required and can increase buffer requirements at output. Das and Kumar [2012] propose minimizing migration overhead while satisfying the throughput requirement for different fault scenarios. However, energy is not considered in this technique. The technique in Das et al. 2012] jointly minimizes communication energy, migration overhead, and throughput degradation for streaming multimedia applications modeled using SDFGs. These existing offline task mapping techniques suffer from the following limitations. First, these techniques fail to capture the dynamism observed at runtime due to unavailability of one or more component cores. Such unavailability can be attributed to resource blockage due to execution of multiple simultaneous applications or running maintenance jobs. The static mappings fail to adapt to such a changing environment, leading to suboptimal results both in terms of throughput and energy consumption. Second, the existing static techniques determine task mapping for every application enabled on the multiprocessor platform for every fault scenario. These mappings are stored in a table to be looked up as, and when, faults occur. The size of the look-up table grows exponentially with the number of applications and the level of fault tolerance. This is crucial, especially for multimedia MPSoCs where increased storage space leads to an increase in area and cost of the system. Third, the existing approaches can only support applications analyzed at design time, and therefore throughput for new applications is not guaranteed. This imposes limitations on the support of new applications, as it is difficult to foresee all future applications a priori. Fourth, the existing static techniques also suffer from runtime overhead of table look-up to fetch a mapping of an application for a fault scenario. The size of the mapping table needs to be small to avoid the requirement of large memory space. In other words, the number of faults

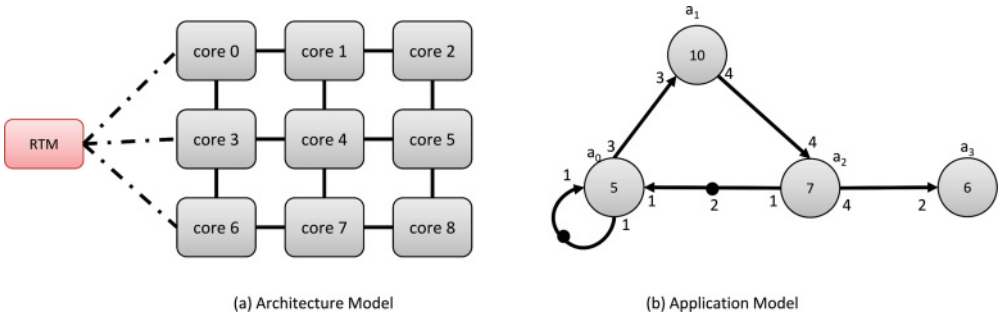

Fig. 2. Architecture and application model.

and/or applications that can be practically supported for an MPSoC is limited. Last, scheduling is not considered in these techniques, and therefore these techniques suffer from large schedule construction overhead at runtime or schedule storage overhead from design time.

Dynamic approaches monitor system status and decide to migrate tasks at runtime. A fault-aware resource management is proposed in Chou and Marculescu [2011] to deal with the occurrence of transient, intermittent, and permanent faults. The remapping decisions are taken at runtime to minimize data communication while avoiding the faulty and other stressed processors. An integer linear programming(ILP)-based task remapping technique is proposed in Derin et al. [2011] to remap tasks while minimizing communication energy. A common limitation of these approaches is that throughput is not guaranteed. Furthermore, these techniques are based on directed acyclic graphs and therefore require significant modification (if at all applicable) for multimedia applications with cyclic task dependency and pipelined scheduling.

4. PROBLEM FORMULATION

4.1. Architecture and Application Model

The MPSoC platform consists of N_p processing cores interconnected using a mesh-based topology.[1] This is shown in Figure 2(a). Such an architecture is modeled as a directed graph $\mathcal{G}_{arc} = (\mathcal{V}_{arc}, \mathcal{E}_{arc})$, where $\mathcal{V}_{arc}$ is the set of nodes $\boldsymbol{c}_j$ representing the processing cores and $\mathcal{E}_{arc}$ is the set of edges $\boldsymbol{e}_{ij}$ representing communication channel between $\boldsymbol{c}_i$ and $\boldsymbol{c}_j$. Here, $N_p = |\mathcal{V}_{arc}|$. Each processing core $\boldsymbol{c}_j$ is associated with a heterogeneity type h_j. For an architecture with three different processing elements (e.g., DSP, GPP type 1, and GPP type 2), $0 \leq h_j \leq 2$, where $h_j = 0$ implies DSP, $h_j = 1$ implies GPP type 1, and $h_j = 2$ implies GPP type 2. The platform activities (e.g., dispatching an application, updating of resource utilization table, dynamic task remapping) are coordinated using a special node—the RTM—and is linked to the interconnect network. For simplicity, only some of the links from the RTM are shown in the figure. It is to be noted that this work is based on the assumption that the RTM is fault free. This is justified by implementing the RTM on a radiation-hardened and reliable processor. In the future, the relaxation of this restriction will be addressed. Although a specific architecture, such as the one defined here, is assumed for this work, the proposed technique can trivially be applied to other architectures.

The NoC model adopted for this work is based on the principle of spatial division multiplexing (SDM) [Leroy et al. 2008] with cores interconnected in a mesh-based architecture. SDM-based NoCs have been proposed as an alternative to time division

[1]Although a mesh-based network is assumed for simplicity, the proposed approach can be applied directly to any other NoC architectures, such as Torus and Tree.

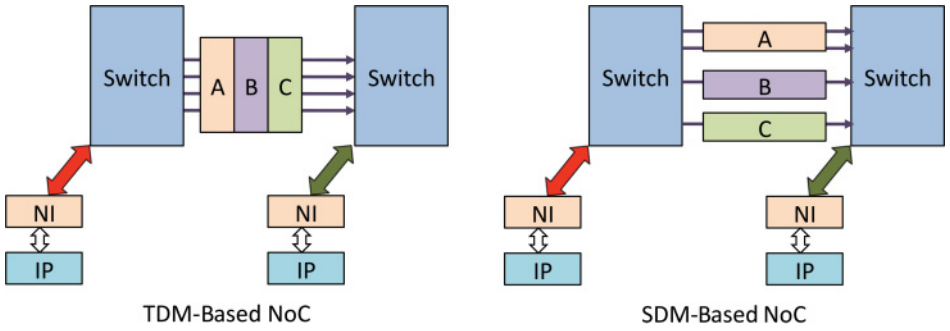

Fig. 3. Difference between TDM- and SDM-based NoCs.

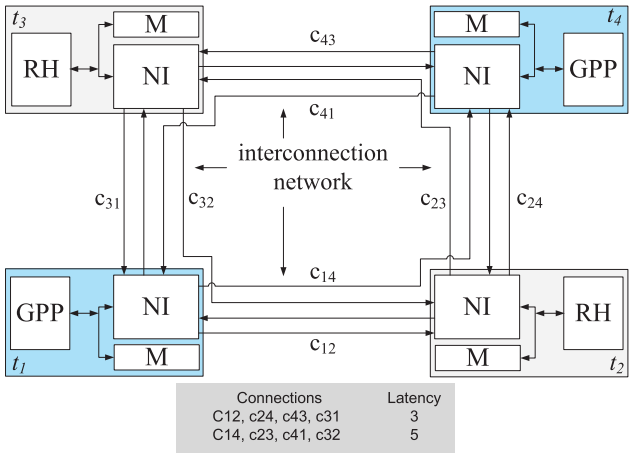

Fig. 4. Mesh-based architecture.

multiplexing (TDM)-based NoCs. In SDM-based NoCs, a subset of available wires is dedicated between a pair of cores to form a connection. Each connection thus has exclusive use of the wires assigned to it. Data from cores is serialized at the transmitter network interface and sent over the wire. At the receiving end, data is de-serialized at the receiver network interface. Figure 3 shows the difference between an SDM-based NoC and a TDM-based NoC. For a TDM-based NoC, three different connections—A, B, and C—are time multiplexed over the wires. For an SDM-based NoC, data for all connections is transmitted simultaneously over multiple wires. The number of wires dedicated for a connection is bandwidth dependent. In Figure 3, two of the wires are assigned to connection A and one each to connection B and C. SDM networks do not require any switching at the routers once the connections are configured; they essentially are circuit switched once the data leaves the network interface. Thus, packets no longer need to be buffered in the switches, saving area and power.

To ensure guaranteed latency between cores at runtime, analysis is performed assuming full connectivity between the cores. In other words, every edge $\boldsymbol{e}_{ij}$ is assumed to be comprised of $N_p \times (N_p - 1)$ physical wires. End-to-end connections providing fixed latencies between cores are used to connect the cores. This is shown in Figure 4. The latencies depend upon the hop distance between the cores, as shown at the bottom of Figure 4 . The application edges can get mapped onto the connections between cores. Each edge occupies one connection between the cores at its full bandwidth, and the occupied connection always serves only the assigned edge. This ensures that the latency

between cores remains constant. Although the worst-case connectivity is assumed, a total number of physical wires equal to the sum of the input and output connections of an actor is sufficient in the average case. This improves the wire utilization significantly. However, routability is not guaranteed. To address this routability, a partially reconfigurable (PR) cross-bar router for SDM-based NoC is adopted. Further details are omitted because of space limitations. Interested readers can refer to Hoo and Kumar [2012] for a detailed treatment of PR SDM-based NoC.

Dataflow graphs (DFGs) are a natural paradigm for modeling modern DSP applications and for designing concurrent and pipelined multimedia applications implemented on an MPSoC [Kavi et al. 1986]. Among the various dataflow models, this article focuses on SDFGs proposed in Lee and Messerschmitt [1987]. The nodes of an SDFG are called *actors*; they represent functions that are computed by reading *tokens* (data items) from their input ports and writing the results of the computation as tokens on the output ports. Figure 2(b) shows an example of an SDFG. There are four actors in this graph. In the example, $\boldsymbol{a}_1$ has an input rate of 3 and output rate of 4. An actor is called *ready* when it has sufficient input tokens on all of its input edges and sufficient buffer space on all of its output channels; an actor can only fire when it is ready. The edges may also contain *initialtokens*, indicated by bullets on the edges, as seen on the edge from actor $\boldsymbol{a}_2$ to $\boldsymbol{a}_0$ in Figure 2(b). The following definitions are stated. For a detailed treatment and proofs, interested readers are urged to refer to Ghamarian et al. [2006].

Definition 1 (*SDFG*). An SDFG is a directed graph $\mathcal{G}_{app} = (\mathcal{A}, \mathcal{C}, \mathcal{T}_c)$ consisting of a finite set $\mathcal{A}$ of actors, a finite set $\mathcal{C} \subseteq Ports^2$ of channels, and the throughput constraint $\mathcal{T}_c$. Let $N_{app} = |\mathcal{A}|$.

Definition 2 (*Actor*). An actor $\boldsymbol{a_i}$ is a tuple $(\mathcal{N}_i, \Gamma_i)$, where $\mathcal{N}_i$ is the set of execution cycles of $\boldsymbol{a_i}$ and Γ_i is the set of tokens produced. The execution cycle $\mathcal{N}_i$ is the set $\{n_{il}\}$, representing the CPU cycles needed to execute actor $\boldsymbol{a_i}$ on core type l. For homogeneous systems, the execution cycles of an actor on all cores are the same. Γ_i is the set $\{\gamma_{ij} | \forall j\}$, where γ_{ij} represent the tokens communicated from actor $\boldsymbol{a_i}$ to actor $\boldsymbol{a}_j$.

Definition 3 (*Repetition Vector*). Repetition Vector RV of an SDFG $\mathcal{G}_{app} = (\mathcal{A}, \mathcal{C}, \mathcal{T}_c)$ is defined as the vector specifying the number of times actors in $\mathcal{A}$ are executed in one iteration of SDFG $\mathcal{G}_{app}$. For example, in Figure 2(b), $RV[\boldsymbol{a}_0\ \boldsymbol{a}_1\ \boldsymbol{a}_2\ \boldsymbol{a}_3] = [1\ 1\ 1\ 2]$.

The repetition vector is also referred to as the periodic admissible sequential schedule (PASS) [Lee and Messerschmitt 1987].

Definition 4 (*Application Period*). Application Period *Per(A)* is defined as the time that SDFG $G_{app} = (A, C)$ takes to complete one iteration on average.

4.2. Communication Energy Model

In Hu and Marculescu [2004], bit energy (E_{bit}) is defined as the energy consumed in transmitting one data bit through an NoC router and link:

$$E_{bit} = E_{S_{bit}} + E_{L_{bit}}, \tag{1}$$

where $E_{S_{bit}}$ and $E_{L_{bit}}$ are the energy consumed in the switch and the link, respectively. The energy per bit consumed in transferring data between cores $\boldsymbol{c}_p$ and $\boldsymbol{c}_q$, situated $n_{hops}(p, q)$ away, is given by Equation (2) according to Hu and Marculescu [2004]:

$$E_{bit}(p, q) = \begin{cases} n_{hops}(p, q) * E_{S_{bit}} + (n_{hops}(p, q) - 1) * E_{L_{bit}} & \text{if } p \neq q \\ 0 & \text{otherwise} \end{cases}. \tag{2}$$

ALGORITHM 1: Application mapping

Require: G_{app}, G_{arc}, throughput constraint $\mathcal{T}_c$
Ensure: Mappings for all fault scenarios
1: /* Design-time analysis */
2: Determine initial mapping $\mathcal{M}_{N_{arc}}$
3: /* Runtime analysis */
4: **while** true **do**
5: Execute one iteration of the application using $\mathcal{M}_{N_{arc}}$.
6: **if** core faulty **then**
7: Determine the faulty core $\boldsymbol{c}_j$.
8: Determine mapping of $\mathcal{G}_{app}$ on $\mathcal{V}_{arc} \setminus \boldsymbol{c}_j$.
9: **end if**
10: **end while**

The total communication energy is given by Equation (3):

$$E_{comm} = \sum_{\forall \boldsymbol{a}_i, \boldsymbol{a}_j \in A} d_{ij} * E_{bit}(\Phi(i), \Phi(j)), \tag{3}$$

where $\Phi(i)$ and $\Phi(j)$ are the cores where actors $\boldsymbol{a}_i$ and $\boldsymbol{a}_j$ are mapped, respectively, and d_{ij} is the data communicated from actor $\boldsymbol{a}_i$ to $\boldsymbol{a}_j$ and is given by

$$d_{ij} = RV[\boldsymbol{a}_i] * \gamma_{ij} * sze, \tag{4}$$

where *sze* is the size of a token in bits.

4.3. Problem Representation

The mapping of $\mathcal{G}_{app}$ on $\mathcal{G}_{arc}$ with n cores is denoted as $\mathcal{M}_n$. The pseudocode of the optimization problem is shown in Algorithm 1. The algorithm consists of two sections: starting mapping generation at design time and fault-aware task remapping at runtime. Exploring all possible task to core mapping to minimize energy consumption while satisfying throughput constraint is an NP-hard problem. The design-space exploration time grows exponentially with an increase in the number of actors and/or cores. For example, executing 14 actors on 14 cores, a total of 190,899,321 mapping options need to be evaluated exhaustively to determine the best mapping with one faulty core. The complete evaluation takes 2 days assuming 1ms for simulating (evaluating) one mapping. This evaluation is to be performed every time a core becomes faulty.[2] Even pruning the exploration space with existing analysis strategies does not lead to acceptable evaluation time for complex applications.

To cater for a fault scenario at runtime, performing the exploration by employing simulative evaluations may lead to missed timing deadlines. Therefore, analytical estimations need to be employed to get fast results. However, the accuracy of the estimations with respect to the simulations needs to be validated. In contrast to existing approaches, the proposed approach performs faster exploration for a runtime fault scenario by analyzing the execution traces of actors and edges of the application executing on a fixed number of cores. Further, the proposed approach jointly optimizes for the throughput and energy consumption, unlike most of the existing approaches.

5. PROPOSED DYNAMIC FAULT-TOLERANT RECONFIGURATION METHODOLOGY

Figure 5 presents an overview of the proposed fault-tolerant task remapping methodology. The flow starts with a mapping (referred to in the figure as *Current Mapping*)

[2]Although a large exploration time can be tolerated for permanent faults that are infrequent, the overhead is large for frequently occurring intermittent faults leading to a large performance degradation.

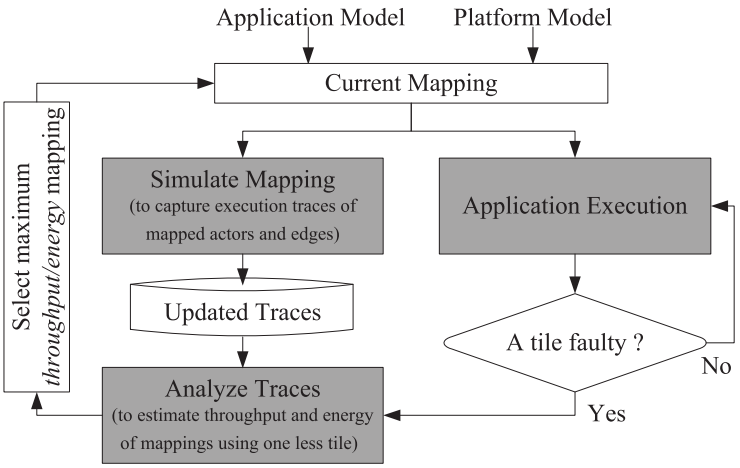

Fig. 5. Proposed dynamic fault-tolerant reconfiguration approach.

of the application $\mathcal{G}_{app}$ enabled on the platform $\mathcal{G}_{arc}$. At the start of the application execution, the current mapping uses the maximum number of cores, which is given by the minimum value of the number of cores available in the system and the number of actors in the application. For a large architecture, where the number of cores is greater than the number of actors, this current mapping uses one actor per core. The mapping reduces the stress on the core and improves the lifetime reliability [Das et al. 2013a]. However, the proposed technique can trivially be used along with any other heuristics [Singh et al. 2010; Ost et al. 2013] to generate this starting mapping.

The mapping process is handled by the RTM (Figure 2(a)). One of the platform processors is used as manager processor that manages the job of mapping, scheduling, platform resource control, and configuration control. The resources' status is updated at runtime when an actor is loaded in the platform or a faulty core gets detected to provide the RTM with accurate knowledge of resource occupancy, which is required for making better mapping decisions based on available resources at runtime.

In the subsequent steps of the technique, the application is executed on the platform using current mapping, and in parallel, the execution traces of actors and edges are captured using the *Simulate Mapping* block. The simulation process also computes the throughput of the mapping. The captured execution traces are stored in a database by deleting the earlier traces (*Update Traces*). Thus, the trace storage space is reused, minimizing the storage overhead significantly. Updated traces are analyzed to estimate throughput and energy consumption of the mappings using one less core in case a tile becomes faulty during the application execution. Out of all evaluated mappings, the one with highest throughput/energy ratio is selected as the current mapping to execute the application toward achieving the fault tolerance. Selection of such a mapping optimizes the throughput and energy consumption jointly.

In the event when a core recovers from faults (as in the case when a core is affected by intermittent faults), the same strategy is applied—that is, the updated traces are analyzed to select the configuration with the highest throughput/energy ratio while using one extra core. The current mapping is updated accordingly.

The parallel simulation of the current mapping along with the application execution prepares the updated traces faster, which is essential in case a fault occurs in the early stage of the application execution. The simulation process also guarantees accurate execution traces, which facilitates accurate analytical estimations (*Analyze Traces*).

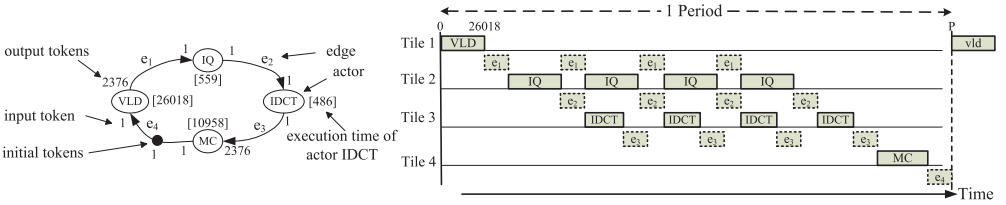

Fig. 6. Execution trace of actors/edges of the H.263 decoder for one periodic execution.

The process is repeated as more cores become faulty. To handle multiple simultaneous faults (recovery), the same process is repeated by considering faults in (recovery of) one core at a time until traces considering all faulty (recovered) cores are explored.

5.1. Fault Tolerance for Concurrent Applications

The focus of this work is on runtime energy optimization for different fault scenarios. However, for the sake of completeness, an overview is provided on the use of the proposed approach for the resource management and fault tolerance of concurrent applications in a use case. Formally, a use case is defined as a collection of multiple applications that are active simultaneously on an MPSoC. Several techniques have been proposed in the literature for the runtime resource management of use cases [Bellasi et al. 2012; Ykman-Couvreur et al. 2006; Das et al. 2013a]. To provide fault tolerance for all applications of a use case, the first step is to determine the number of processors allocated to each of them. This can be determined by analyzing the energy consumption of each application or using the mean time to failure (MTTF)-aware resource distribution technique of Das et al. [2013a], which reduces the wearout of the processors. The trace-based dynamic fault-tolerance methodology is then performed for every application of the use case. Detailed consideration of use cases is left to future work.

5.2. Simulation Strategy

The simulation of the current mapping involves two steps: throughput computation and execution trace capturing.

5.2.1. Throughput Computation. The throughput for a mapping is computed by taking the resource allocations of actors/edges on the platform into account. To compute the throughput, first, a static-order schedule for each core is constructed, which orders the execution of bound actors. Thereafter, all binding and scheduling decisions are modeled in a graph referred to as binding-aware SDFG. Finally, the throughput is computed by self-timed state-space exploration of the binding-aware SDFG [Ghamarian et al. 2006]. Toward this, states visited during self-timed execution are examined and stored until a recurrent state is found. The throughput is computed from the periodic part of the state space.

5.2.2. Execution Trace Capturing. The execution traces of actors and edges are captured based on their execution pattern for a given mapping during one periodic execution. For example, Figure 6 shows the execution pattern of actors and edges of the H.263 decoder when each actor and edge is mapped on a different core and connection between cores, respectively. First, actor *VLD* fires (executes) as it has sufficient input tokens on its incoming edge e_4. Then, it generates 2,376 tokens to be transferred through e_1 to process them one by one by *IQ*. The transfer of tokens through edges and their processing by different actors follows the shown trace. For easier realization, the shown trace considers four tokens in places of 2,376, and thus actors *VLD*, *IQ*, *IDCT*, and *MC* fire 1, 4, 4, and 1 times during one period, respectively. The execution traces for each

ALGORITHM 2: Analysis strategy

Require: Execution trace for the current *mapping* μ using m cores
Ensure: Mappings with throughput and energy consumption using $(m-1)$ cores
1: Initialize the mapping set M —that is, $M = \{\}$
2: Select m cores containing actor(s)
3: **for** each unique pair of selected cores **do**
4: **if** actor(s) on the selected cores can be combined on the same core **then**
5: Move actor(s) from one core to another to generate a *new mapping* η using $(m-1)$ cores
6: Estimate *throughput* and E_{comm} of η
7: Add η with its *throughput* and E_{comm} to set M
8: **end if**
9: **end for**

actor and edge is captured as the start and end time of their active executions (firings) in the whole period. For example, four active executions of actor *IQ* gets captured with different start and end times.

5.3. Analysis Strategy

The analysis strategy to perform the analytical estimation is presented in Algorithm 2. The strategy takes the updated execution traces of actors/edges for the current mapping μ using m cores as input and estimates throughput and energy consumption of mappings using $(m-1)$ cores with one faulty core. The strategy first selects m cores containing actor(s). Then, for each unique pair of selected cores, actors of one core are moved to another to generate a new mapping η that uses $(m-1)$ cores, provided that the actors on the selected cores can be combined on the same core. Depending upon the fault occurring in a particular core type, the movement is performed by taking one lesser core of the faulty core type. The actors can be combined on the same core if they can be supported by the core—that is, the actors have implementations for the core type. For each new generated mapping, its throughput (1/period) and energy consumption are estimated, and the mapping with its throughput and energy values is added to mapping set M.

The period (P) of the mapping using $(m-1)$ cores (P_η) is estimated by utilizing the period of the current mapping using m tiles (P_μ) by Equation (5), where $inc_{\mu,\eta}$ and $dec_{\mu,\eta}$ are the increase and decrease in the period of the mapping μ when the new mapping η is generated by moving actors from one core to another in μ.

$$P_\eta = P_\mu + inc_{\mu,\eta} + dec_{\mu,\eta} \tag{5}$$

The period increases when parallel executing actors (e.g., *IQ* and *IDCT* in Figure 6) mapped on selected pair of cores in mapping μ are forced to execute sequentially by mapping the actors on the same tile in mapping η. The period decreases when execution of the edge(s) between the selected pair of cores is not in parallel with other actors and edges (e.g., execution of edge e_1 in Figure 6). The $inc_{\mu,\eta}$ is calculated by assuming sequential execution of the actors mapped on the selected pair of cores. The nonparallel executions of the actors (with executions of other actors/edges) contribute to $inc_{\mu,\eta}$. The $dec_{\mu,\eta}$ is calculated by considering execution traces of edge(s) mapped between the selected pair of cores. The nonparallel executions of the edge(s) (with executions of other actors/edges) contribute to $dec_{\mu,\eta}$.

In Algorithm 2, a total of m-choose-2 (${}^mC_2$) unique pairs are found for the selected m cores. Each unique pair provides a mapping that uses $(m-1)$ cores. Out of all of the mappings M using $(m-1)$ cores, the proposed flow (Figure 5) selects the mapping with maximum throughput/energy to jointly optimize for through and energy consumption toward achieving the fault tolerance.

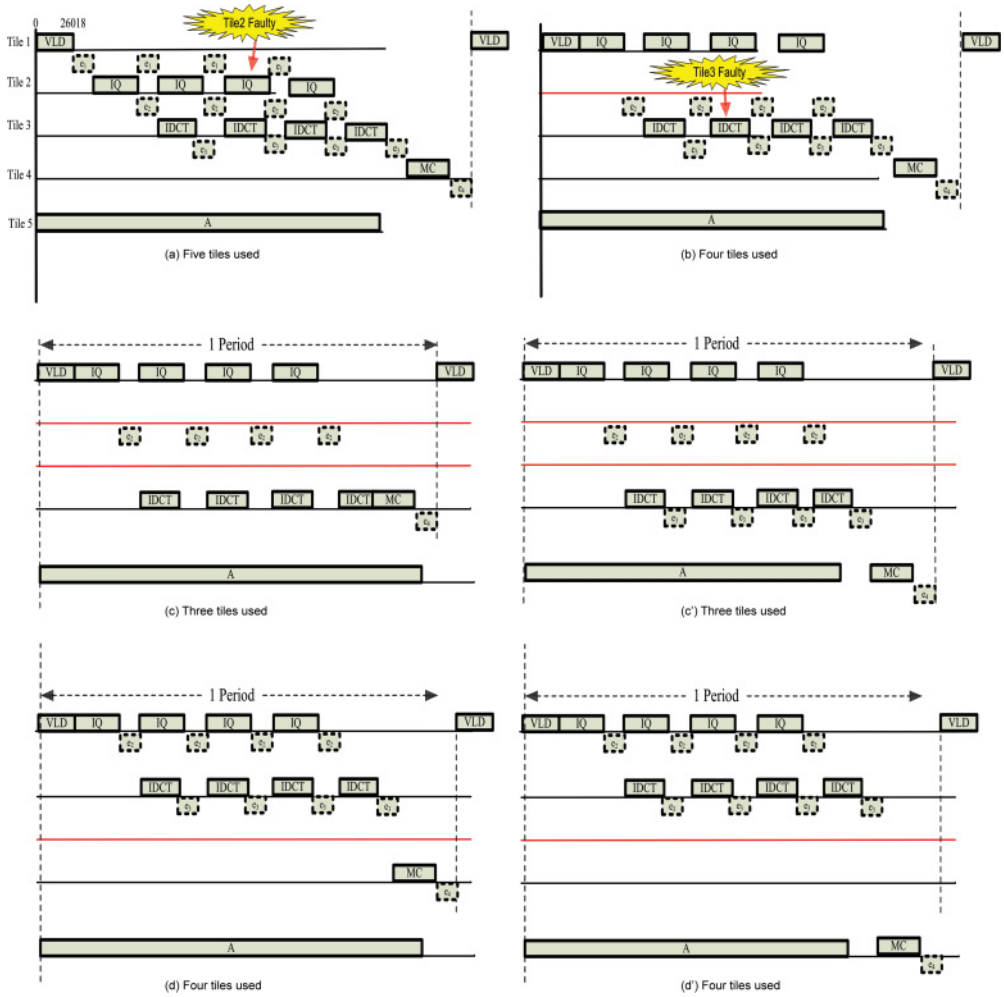

Fig. 7. Dynamic fault tolerance demonstration.

5.4. Example Demonstration

Figure 7 demonstrates the proposed dynamic fault-tolerant flow for the H.263 decoder of Figure 6 executing on four cores periodically. Figure 7(a) shows that a fault has occurred on core 2 during a particular periodic execution. To achieve fault tolerance, the proposed strategy finds the best mapping (having maximum throughput/energy) using three cores. In the best mapping, the actor from the faulty core (core 2) is moved to some other core. The platform is then reconfigured with the best mapping to start the application execution, as shown in Figure 7(b). At a later instant of time, a fault occurs on core 3, as shown in Figure 7(b). The proposed strategy moves the actor from a faulty core to a nonfaulty one to get the best mapping using two cores. Two cases can be considered in this case. If tile 5 is busy executing another application, then *IDCT* is shown to be executed along with *MC* in Figure 7(c). If, however, tile 5 has a free slot to accommodate another actor, the best throughput/energy mapping is to execute *MC* on tile 5 as shown in Figure 7(c′). At a later stage, when tile 4 becomes available again, the

same process is repeated and the best mapping for the corresponding cases is shown in Figure 7(d) and (d′).

6. EXPERIMENTAL RESULTS

Experiments are conducted on a quad-core Intel Xeon 2.4GHz server running Linux with 10 synthetic and 10 real-life applications modeled as SDFGs. The synthetic applications are generated from the SDF^3 tool [Stuijk et al. 2006], with the number of actors ranging from 4 to 12. The real applications are derived from the benchmarks provided in the tool. These applications are executed on an MPSoC with six tiles arranged in 2×3 mesh architecture. All algorithms developed in the article are coded in C++ and used in conjunction with the SDF^3 tool for throughput computation. For embedded implementation of our runtime algorithm, we used dual-core ARM cortex-A9 processor of Zynq board [Santarini 2011]. One of the ARM processors operating at a frequency of 533MHz has been used to execute the same C++ description of the runtime algorithm.

6.1. Throughput Comparison for Different Fault Scenarios

As established in Section 3, there are no runtime fault-tolerant techniques available in the literature that consider throughput degradation. Therefore, the proposed trace-based dynamic reconfiguration technique is compared to the existing design-time–based throughput-aware fault-tolerant techniques [Das et al. 2012; Hu and Marculescu 2004; Lee et al. 2010]. The throughput obtained using the proposed technique is compared to the throughput obtained using the communication energy-aware static fault-tolerant technique of Das et al. 2012] (referred to as *TConCEMin*) and the communication energy-aware dynamic technique of Hu and Marculescu [2004] (referred to as *DCEMin*). It is important to note that the technique of Hu and Marculescu [2004] does not incorporate fault tolerance in its native form. However, we have modified the technique to compare to the proposed approach. Specifically, the technique of Hu and Marculescu [2004] is applied to determine the minimum communication energy for a different core count. Further, to determine how far the proposed approach is from the highest throughput obtained for different fault scenarios, the results are compared to the throughput maximization technique of Lee et al. [2010] (referred to as *TMax*). Another important point is that the design-time–based techniques obtain better results, as any sophisticated algorithms can be implemented in these techniques. However, for the runtime approaches, the algorithm needs to be simple to satisfy the real-time requirements. The objective to compare to the design-time approaches is to determine how close the results of the runtime algorithms are to that obtained from the design-time–based sophisticated heuristics.

Figures 8(a) and 8(b) plot the throughput of *TConCEMin*, *DCEMin*, and the proposed technique normalized with respect to the throughput obtained using the *TMax* technique for five real applications for single and double faults, respectively. *Single fault* refers to a fault in one PE, and a fault in two PEs is referred to as a *double fault*. There are a few trends to follow from these figures. First of all, the throughput of *DCEMin* is the least among all techniques. This is expected, as *DCEMin* does not consider throughput degradation. The throughput constraint (80% of the highest throughput) is violated for all applications except the MP3 Decoder, signifying the inapplicability of this technique for throughput-constrained multimedia applications. Second, the throughput of the proposed fault-tolerant technique is better than the energy-aware fault-tolerant technique—that is, *TConCEMin* [Das et al. 2012]. This is expected because the proposed technique remaps the tasks from the faulty tile to other working tile(s) such that communication energy is minimized with the least degradation of throughput, facilitating for least actor migration. The static fault-tolerant technique of *TConCEMin*, on the

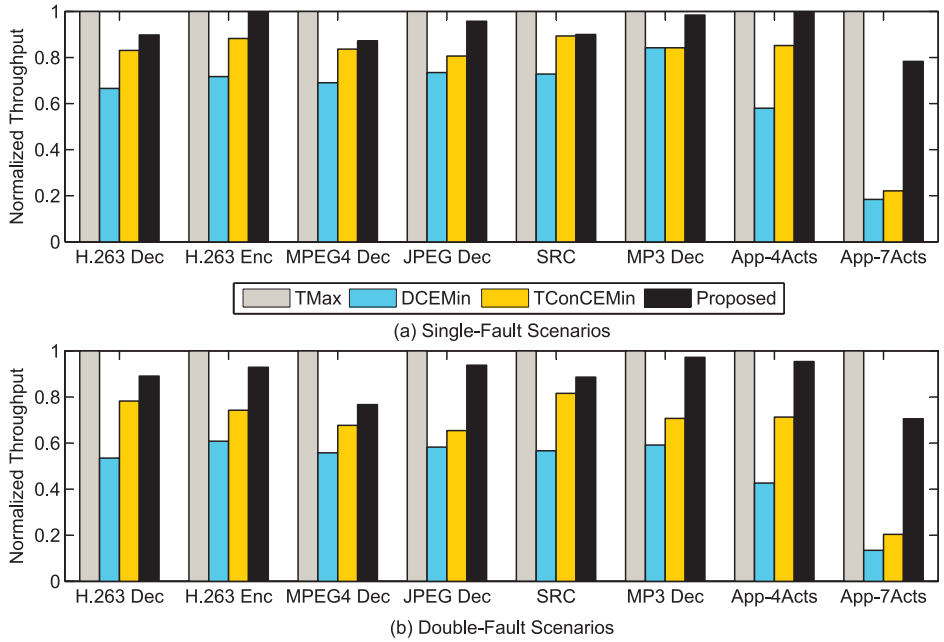

Fig. 8. Throughput performance of the proposed technique.

other hand, determines one mapping that results in minimum communication energy satisfying the throughput requirement even at the expense of more actor migrations. On average for all single-fault scenarios, the proposed technique achieves 70% and 42% better throughput than the *DCEMin* and *TConCEMin* techniques, respectively. For double-fault scenarios, these numbers are 110% and 52%, respectively. Finally, the proposed technique is only within 8% and 12% of the highest throughput technique of *TMax* for single- and double-fault scenarios, respectively.

6.2. Energy Comparison for Different Fault Scenarios

Figure 9 plots the energy consumption of the same four techniques for the same set of applications. The energy values are average for all single and double faults and are normalized with respect to the energy obtained using the *TMax* technique. As can be seen from the figure, the energy of *DCEMin* is the least, as communication energy is explicitly minimized in this technique without considering the throughput degradation. Second, the communication energy of *TConCEMin* is better than the proposed technique. This is because *TConCEMin* uses ILP to solve the minimum communication energy problem and is guaranteed to determine the optimum solution. The proposed technique, on the other hand, uses a heuristic to solve the same. On average for single and double faults, the proposed technique results in 18% and 12% lower energy than the static technique of *TMax*. The energy of the proposed heuristic is within 20% of the minimum energy of *TConCEMin*.

The important conclusion to make from these results is that the proposed technique maximizes throughput (on average, 70% and 110% for single and double faults, respectively) compared to the existing dynamic techniques with less than 20% degradation of energy as compared to existing static throughput-aware fault-tolerant techniques.

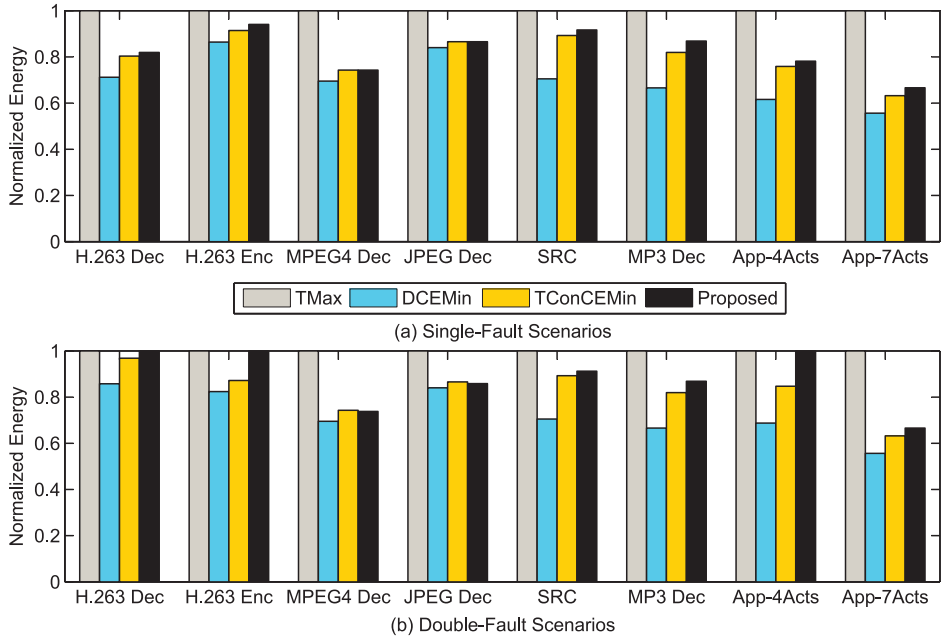

Fig. 9. Energy performance of the proposed technique.

Table I. Storage Requirement (KB) with Increasing Tiles for a Three-Fault-Tolerant System

Tiles	*TConCEMin* [Das et al. 2012]	Proposed
4	4.6	1
8	32.8	1.2
12	92.4	1.3
16	187.5	1.4
20	320.7	1.6
24	494.3	1.8
28	709.8	1.9
32	968.7	2.0

6.3. Storage Overhead Performance

Table I shows the storage requirement of the proposed dynamic technique compared to the static technique of Das et al. 2012] for different numbers of tiles. The number of actors for the application is the same as the number of tiles. Synthetic applications with different numbers of actors are used for different numbers of tiles. As can be seen from the table, and also expected, the static technique requires significant storage overhead. This is because the static technique evaluates and stores the application mapping and scheduling for all processor fault scenarios. On the other hand, the proposed dynamic approach derives the schedule for a fault scenario from a master execution trace as, and when, faults occur. Thus, the storage overhead associated with the dynamic technique is that required to store the master execution trace. This is shown in the third column of the table. Clearly, the proposed technique achieves significant savings in storage, which is crucial for multimedia applications where increased storage space leads to higher area and cost.

Table II. Overhead (in Milliseconds) for a Single-Fault Scenario for Different Applications

Application	*TConCEMin* [Das et al. 2012]	Proposed	
		Ex. Time	Migration Overhead
H.263 decoder	2055.20	0.69	1.6
H.263 encoder	2124.61	15.74	0.6
Sample rate converter	17922.48	0.55	0.01

6.4. Algorithm Overhead

Table II shows the execution time of different approaches to find mappings in the case of a single-fault scenario for different multimedia applications. The shown times for our proposed approach are for the stand-alone implementation on a dual-core ARM cortex-A9 processor present in the Zynq board [Santarini 2011]. Such embedded implementation provides more accurate timing information. The initial mapping for each application has been assumed to use the same number of tiles as the number of actors in the application. Therefore, for each application, the approaches need to evaluate mappings using one less number of tiles. Both approaches evaluate the same number of mappings for the single-fault scenario. The number of mappings is n-choose-2 (${}^{n}C_2$), where n is the number of actors in the application. However, the static technique of Das et al. 2012] employs simulative evaluations, and the proposed approach employs analytical estimations. As can be seen from the table, and also expected, the static technique needs a large time compared to the proposed technique (column 2 vs. column 3). The difference in execution times is observed due to simulative and analytical evaluations. In case of other fault scenarios, such as double fault, similar results are obtained. Thus, the proposed technique reduces the execution time significantly, which is essential to make rapid decisions at runtime.

Finally, column 4 of Table II reports the migration overhead of the proposed approach, which is measured as the time for reconfiguring the tasks for the new mapping. This time is measured as the time required to migrate the state space of the actor from the core in the current configuration to the new core, corresponding to the mapping following a fault detection or fault recovery. The migration overhead also includes the hop distance through which the state space is migrated in the NoC. Furthermore, the migration overhead is incurred once when the configuration changes, and the energy lost can be recovered in the subsequent iterations.

7. CONCLUSIONS AND FUTURE WORK

This article proposes an execution trace–based runtime technique to minimize the communication energy and throughput degradation of applications for different processor fault scenarios. Experiments conducted with applications modeled as SDFGs clearly indicate that the proposed technique provides significant throughput improvement (on average, 70% and 110% for single and double faults, respectively) with respect to the existing dynamic technique with less than 20% deviation in communication energy obtained with an ILP-based technique. Task computation energy minimization is left as future work.

REFERENCES

P. Bellasi, G. Massari, and W. Fornaciari. 2012. A RTRM proposal for multi/many-core platforms and reconfigurable applications. In *Proceedings of the International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC'12)*. 1–8. DOI:http://dx.doi.org/10.1109/ReCoSoC.2012.6322885

S. Borkar, T. Karnik, and V. De. 2004. Design and reliability challenges in nanometer technologies. In *Proceedings of the Design Automation Conference (DAC'04)*.

C.-L. Chou and R. Marculescu. 2011. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'11)*. 1 –6.

A. Das and A. Kumar. 2012. Fault-aware task re-mapping for throughput constrained multimedia applications on NoC-based MPSoCs. In *Proceedings of the IEEE International Symposium on Rapid System Prototyping (RSP'12)*.

A. Das, A. Kumar, and B. Veeravalli. 2012. Energy-aware communication and remapping of tasks for reliable multimedia multiprocessor systems. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS'12)*.

Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2013a. 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 (DATE'13)*. 1631–1636. DOI:http://dx.doi.org/10.7873/DATE.2013.331

A. Das, A. Kumar Singh, and A. Kumar. 2013b. Energy-aware dynamic reconfiguration of communication-centric applications for reliable MPSoCs. In *Proceedings of the International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC'13)*. 1–7.

O. Derin, D. Kabakci, and L. Fiorin. 2011. Online task remapping strategies for fault-tolerant network-on-chip multiprocessors. In *Proceedings of the IEEE/ACM Symposium on Networks on Chip (NoCS'11)*.

S. Furber. 2006. Living with failure: Lessons from nature? In *Proceedings of the IEEE European Test Symposium (ETS'06)*.

A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T, Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. 2006. Throughput analysis of synchronous data flow graphs. In *Proceedings of the IEEE Conference on Application of Concurrency to System Design (ACSD'06)*. 25–36.

C. H. Hoo and A. Kumar. 2012. An area-efficient partially reconfigurable crossbar switch with low reconfiguration delay. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'12)*. DOI:http://dx.doi.org/10.1109/FPL.2012.6339136

J. Hu and R. Marculescu. 2004. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'04)*.

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 Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'11)*. ACM, New York, NY.

K. M. Kavi, B. P. Buckles, and U. N. Bhat. 1986. A formal definition of data flow graph models. *IEEE Transactions on Computers* 35, 11, 940–948.

I. Koren and C. M. Krishna. 2007. *Fault-Tolerant Systems*. Morgan Kaufmann.

C. Lee, H. Kim, H. Park, S. Kim, H. Oh, and S. Ha. 2010. A task remapping technique for reliable multicore embedded systems. In *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'10)*. ACM, New York, NY.

E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proceedings of the IEEE* 75, 9, 1235–1245.

A. Leroy, D. Milojevic, D. Verkest, F. Robert, and F. Catthoor. 2008. Concepts and implementation of spatial division multiplexing for guaranteed throughput in networks-on-chip. *IEEE Transactions on Computers* 57, 9, 1182–1195. DOI:http://dx.doi.org/10.1109/TC.2008.82

J. W. McPherson. 2006. Reliability challenges for 45nm and beyond. In *Proceedings of the Design Automation Conference (DAC'06)*.

S. Murali, T. Theocharides, N. Vijaykrishnan, M. J. Irwin, L. Benini, and G. De Micheli. 2005. Analysis of error recovery schemes for networks on chips. *IEEE Design and Test of Computers* 22, 5, 434–442.

L. Ost, M. Mandelli, G. M. Almeida, L. Moller, L. S. Indrusiak, G. Sassatelli, P. Benoit, M. Glesner, M. Robert, and F. Moraes. 2013. Power-aware dynamic mapping heuristics for NoC-based MPSoCs using a unified model-based approach. *ACM Transactions on Embedded Computing Systems* 12, 3, Article No. 75.

S. K. Reinhardt and S. S. Mukherjee. 2000. Transient fault detection via simultaneous multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, 25–36. DOI:http://dx.doi.org/10.1145/339647.339652

M. Santarini. 2011. Zynq-7000 EPP Sets Stage for New Era of Innovations. Available at http://www.design-reuse.com/articles/26686/xilinx-zynq-7000-arm-cortex-a9-mpcore.html.

A. Kumar Singh, M. Shafique, A. Kumar, and J. Henkel. 2013. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proceedings of the Design Automation Conference (DAC'13)*.

A. Kumar Singh, T. Srikanthan, A. Kumar, and W. Jigang. 2010. Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms. *Elsevier Journal of Systems Architecture* 56, 7, 242–255.

S. Stuijk, M. C. W. Geilen, and T. Basten. 2006. SDF^3: SDF for free. In *Proceedings of the IEEE Conference on Application of Concurrency to System Design (ACSD'06)*.

W. Wolf. 2005. Multimedia applications of multiprocessor systems-on-chips. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'05)*.

C. Yang and A. Orailoglu. 2007. Predictable execution adaptivity through embedding dynamic reconfigurability into static MPSoC schedules. In *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*. ACM, New York, NY.

C. Ykman-Couvreur, V. Nollet, F. Catthoor, and H. Corporaal. 2006. Fast multi-dimension multi-choice knapsack heuristic for MP-SoC run-time management. In *Proceedings of the International Symposium on System-on-Chip*. 1–4. DOI:http://dx.doi.org/10.1109/ISSOC.2006.321966

Received October 2013; revised July 2014; accepted August 2014