A

Energy-Aware Task Mapping and Scheduling for Reliable Embedded Computing Systems

ANUP DAS, National University of Singapore
AKASH KUMAR, National University of Singapore
BHARADWAJ VEERAVALLI, National University of Singapore

Task mapping and scheduling is critical in minimizing energy consumption while satisfying the performance requirement of applications enabled on heterogeneous multiprocessor systems. An area of growing concern for modern multiprocessor systems is the increase in the failure probability of one or more component processors. This is especially critical for applications where performance degradation (throughput, for example) directly impacts the quality of service requirement. This paper proposes a design-time (offline) multi-criterion optimization technique for application mapping on embedded multiprocessor systems to minimize energy consumption for all processor fault-scenarios. A scheduling technique is then proposed based on self-timed execution to minimize the schedule storage and construction overhead at run-time. Experiments conducted with synthetic and real applications from streaming and non-streaming domain on heterogeneous MPSoCs demonstrate that the proposed technique minimizes energy consumption by 22% and design space exploration time by 100x while satisfying the throughput requirement for all processor fault-scenarios. For scalable throughput applications, the proposed technique achieves 30% better throughput per unit energy as compared to the existing techniques. Additionally, the self-timed execution based scheduling technique minimizes schedule construction time by 95% and storage overhead by 92%.

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

1. INTRODUCTION

As the performance demands of embedded applications (multimedia in particular) are growing, multiple processing elements (PEs) are integrated on the same chip to form multiprocessor systems-on-chip (MPSoCs) with networks-on-chip (NoCs) as the communication backbone [Wolf et al. 2008]. Homogeneous architectures are associated with high area and power requirements [Kumar et al. 2004]. Modern MPSoC designs are resorting to heterogeneous PEs such as General Purpose Processors (GPPs), Digital Signal Processors (DSPs), Reconfigurable Areas (RAs) and Application Specific Integrated Circuits (ASICs) [Popovici et al. 2008]. Examples of heterogeneous MPSoCs are OMAP from TI [Cumming 2003], NOMADIK from ST [Artieri et al. 2003] and NEXPERIA from Philips [De Oliveira and Van Antwerpen 2003]. Most of these PEs support a wide range of voltages and frequencies, which are often exploited to meet performance and to perform dynamic voltage and frequency scaling (DVFS) to mini-

Author's addresses: {A. Das, A. Kumar, B. Veeravalli}, Department of Electrical and Computer Engineering, National University of Singapore, 21 Lower Kent Ridge Road, Singapore 119077; e-mail: akdas@nus.edu.sg;

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

mize energy consumption [Rakhmatov and Vrudhula 2003; Irani et al. 2003; Quan and Hu 2007; Kim et al. 2008; Schranzhofer et al. 2010].

Another area of growing concern for MPSoCs is the need for fault-tolerance. Shrinking transistor geometries, aggressive voltage scaling and higher operating frequencies have negatively impacted MPSoC dependability by increasing the chances of failures (transient, intermittent and permanent) [Constantinescu 2003]. Although, transient faults are more frequent than permanent faults, recovery from the latter category is crucial for an MPSoC (or any component) to continue its operation albeit some acceptable performance degradation. This research focuses on permanent faults in MPSoCs.

Permanent faults are traditionally tackled using hardware redundancy [Koren and Krishna 2007]. Due to stringent area and power budgets, software techniques such as task-migration are gaining popularity among the research community [Yang and Orailoglu 2007; Lee et al. 2010; Huang et al. 2011; Das and Kumar 2012]. Most MPSoCs consist of processing cores interconnected with NoCs in a mesh-based architecture[1]. When one or more cores fail, tasks on these cores need to be migrated to other functional core(s) to continue correct operation. The new location (core) is pivotal in determining the energy consumption associated with communication among its dependent tasks. Additionally, as new tasks (those from a faulty core) are mapped to a core, the voltage and frequency may need to be raised to meet the throughput requirement. These concerns have motivated researchers in recent years towards joint optimization of fault-tolerance and energy [Wei et al. 2011; Zhu 2011; Das et al. 2012].

1.1. Scope of this work

This paper attempts to solve the following problem. Given a heterogeneous MPSoC architecture and a set of multimedia and other high performance embedded applications, how to assign and order the tasks of every application on the component cores such that the total energy consumption (computation and communication) is minimized while guaranteeing to satisfy the performance requirement (throughput for example) of the application under all possible core fault-scenarios. The scope of this paper is limited to permanent 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 (co-ordinates) are not addressed. To the best of our knowledge, this is the first work considering the joint optimization of throughput, computation energy and communication energy for *reactive* fault-tolerance (defined in a later section) of heterogeneous MPSoC platforms.

1.2. Key contributions

Following are the key contributions of this paper.

— Fault-aware task mapping technique to minimize the computation and communication energy while satisfying the application throughput requirement.
— A scheduling technique to minimize the run-time schedule construction and schedule storage overhead.
— A heuristic to minimize the design space exploration time.
— Floorplan-aware task remapping for heterogeneous MPSoC

1.3. Paper organization

The rest of this paper is organized as follows. A brief overview of the related work is provided in Section 2. Introduction to Synchronous Data Flow Graphs (SDFGs) is provided in Section 3 and the problem formulation in Section 4. The design methodology is discussed in Section 5 and the proposed scheduling technique in Section 6. Experimen-

[1]While a mesh-based topology is assumed for the target MPSoC, the research is orthogonal to any other topology such as torus and tree.

tal setup and results are discussed next in Section 7. Lastly, conclusions are presented in Section 8 along with discussions on possible extensions.

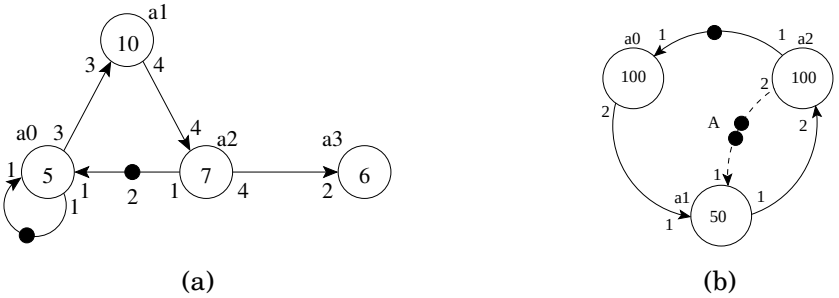
2. RELATED WORKS

Task mapping and scheduling for energy optimization has received significant attention over the past years for extending the battery-life of embedded MPSoCs. A communication energy-aware task mapping heuristic is studied in [Singh et al. 2010; Mandelli et al. 2011]. However, task computation energy is not considered. Dynamic power aware application mapping technique is proposed in [Goh et al. 2009; Schranzhofer et al. 2010]. Floorplanning and inter-task communication are not addressed. The slack budgeting technique of [Hu and Marculescu 2004] distributes execution time slack of a task among other tasks, to reduce their frequency of operation. However, throughput degradation is not accounted in this technique. A common limitation of these energy optimization techniques is that they do not address mappings for different fault-scenarios making them unsuitable for joint optimization of fault-tolerance and energy.

Task mapping and scheduling have also shown significant potential for fault-tolerance in deep sub-micron technologies. For permanent faults two research directions are prominent – *proactive* fault-tolerance i.e. preventing (or delaying) the occurrence of failures [Huang and Xu 2010; Chou and Marculescu 2011; Das et al. 2013] and *reactive* fault-tolerance i.e. develop techniques dealing with core failures once they have occurred [Yang and Orailoglu 2007; Lee et al. 2010; Huang et al. 2011; Das and Kumar 2012; Derin et al. 2011].

Reactive fault-tolerance techniques can be classified into run-time and design-time based. Run-time approaches monitor system-status and decide on task migration at run-time to minimize migration overhead [Al Faruque et al. 2008; Derin et al. 2011] or balance processor load [Zhang et al. 2010]. However, throughput is not always guaranteed in these techniques. Moreover, migration algorithms need to be simple to minimize computation. Design-time based task mapping techniques compute task mapping decisions statically for different fault-scenarios [Yang and Orailoglu 2007; Huang et al. 2011; Lee et al. 2010; Das and Kumar 2012]. As faults occur, these mappings are looked up at run-time to carry out task-migration. An advantage of these techniques is that any sophisticated algorithm can be used at design-time despite the associated mapping storage overhead. This research focus on design-time analysis for resource management after fault(s) have occurred.

A fixed order Band and Band reconfiguration technique is studied in [Yang and Orailoglu 2007]. Cores of the 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 evolutionary algorithm to minimize certain parameters like throughput degradation. At run-time, 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. 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 the 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 even increase the buffer requirements at output. The



(a) (b)

Fig. 1: SDFG model

technique in [Das and Kumar 2012] jointly minimizes the migration overhead and throughput degradation for streaming multimedia applications modeled using Synchronous Data Flow Graphs (SDFGs). A limitation of this technique is that scheduling is not considered and therefore suffers from huge schedule construction overhead at run-time or schedule storage overhead from design-time. A common limitation of these fault-tolerant techniques is the non-consideration of energy minimization.

Recently, there are some works which jointly optimizes energy and fault-tolerance. An ILP based approach is presented in [Wei et al. 2011]. Energy optimization is performed under execution time constraint which incorporates fault-tolerance overhead using check-pointing based recovery model. This technique is not suitable for permanent failures as it does not address the actual task migration under different faulty-scenarios. Moreover, throughput, communication energy and migration overhead are not addressed in this work. A lifetime-reliability aware scheduling technique is developed in [Huang and Xu 2010] to minimize energy consumption. Although the useful life of a device is maximized, the paper does not address migration overhead, resource management and throughput degradation. In a recent work [Das et al. 2012], the authors addressed these objectives while minimizing the task communication energy. However, computation energy is not minimized and so are scheduling and mapping storage overhead.

3. INTRODUCTION TO SYNCHRONOUS DATA FLOW GRAPHS

Synchronous Data Flow Graphs (SDFGs, see [Lee and Messerschmitt 1987]) are often used for modeling modern DSP applications [Sriram and Bhattacharyya 2000] and for designing concurrent multimedia applications implemented on a multi-processor system-on-chip. Both pipelined streaming and cyclic dependencies between tasks can be easily modeled in SDFGs. Tasks are modeled by the vertices of an SDFG, which are called *actors*. SDFGs allow one to analyze a system in terms of throughput and other performance properties, e.g. latency, buffer requirements [Stuijk et al. 2006a].

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. The number of tokens produced or consumed in one execution of an actor is called port $rate$, and remains constant. The rates are visualized as port annotations. Actor execution is also called $firing$, and requires a fixed amount of time, denoted with a number in the actors. The edges in the graph, called $channels$, represent dependencies among different actors.

Figure 1 (a) shows an example of an SDF Graph. 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 its input edges and sufficient buffer space on all its output channels; an actor can only fire when it is ready. The edges may also contain $initial\ tokens$, indicated by bullets on the edges, as seen on the edge from actor

$\boldsymbol{a}_2$ to $\boldsymbol{a}_0$ in Figure 1 (a). A set $Ports$ of ports is assumed and with each port $p \in Ports$, a finite rate $Rate(p) \in \mathbb{N} \setminus \{0\}$ is associated. Formally an SDFG is defined as follows.

DEFINITION 1. (ACTOR) *An actor $\mathbf{a}_i$ is a tuple (I_i, O_i, N_i, μ_i) consisting of a set I_i ($\subseteq Ports$) of input ports, a set O_i ($\subseteq Ports$) of output ports with $I_i \cap O_i = \emptyset$, N_i is the set of execution cycles of $\mathbf{a}_i$ and μ_i is its state space (program and data memory). The execution cycle N_i is a set $\{n_{il} \mid 1 \leq l \leq h\}$, representing the CPU cycles needed to execute actor $\mathbf{a}_i$ on core type l. For homogeneous systems, h = 1 and therefore execution cycles of an actor on all cores are the same.*

DEFINITION 2. (SDFG) *An SDFG is a directed graph $G_{app} = (A, C)$ consisting of a finite set A of actors and a finite set $C \subseteq Ports^2$ of channels. The source of channel $ch_i^j \in C$ is an output port of actor $\mathbf{a}_i$, the destination is an input port of actor $\mathbf{a}_j$. All ports of all actors are connected to precisely one channel, and all channels are connected to ports of some actor. The source and the destination port of channel ch_i^j are denoted by $SrcP(ch_i^j)$ and $DstP(ch_i^j)$ respectively. The channels connected to the input and output ports of an actor $\mathbf{a}_i$ are denoted by $InC(\mathbf{a}_i)$ and $OutC(\mathbf{a}_i)$ respectively.*

Before an actor $\boldsymbol{a}_i$ starts its firing, it requires $Rate(q_i)$ tokens from all $(p, q_i) \in InC(\boldsymbol{a}_i)$. When the actor completes execution, it produces $Rate(p_i)$ tokens on every $(p_i, q) \in OutC(\boldsymbol{a}_i)$. One of the most interesting properties of SDFGs relevant to this paper is throughput. Throughput is defined as the inverse of the long term period, i.e. the average time needed for one iteration of the application. (An iteration is defined as the minimum non-zero execution such that the original state of the graph is obtained.) This is the performance parameter used in this paper. The following properties of an SDF graph are defined.

DEFINITION 3. (REPETITION VECTOR) *Repetition Vector* Rpt *of an SDFG $G_{app} = (A, C)$ is defined as the vector specifying the number of times actors in A are executed for one iteration of SDFG G_{app}.*

For example, in Figure 1 (a), $Rpt[\boldsymbol{a}_0\ \boldsymbol{a}_1\ \boldsymbol{a}_2\ \boldsymbol{a}_3]$ = [1 1 1 2].

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

The amount of data communicated from actor $\boldsymbol{a}_i$ to actor $\boldsymbol{a}_j$ is given by

$$d_{ij} = Rpt[\boldsymbol{a}_i] * Rate(SrcP(ch_i^j)) * sze \tag{1}$$

where sze is the size of a token in bits. The total amount of data communicated between actors $\boldsymbol{a}_i$ and $\boldsymbol{a}_j$ is $d_{ij} + d_{ji}$.

The period of an SDFG can be computed by analyzing the maximum cycle mean (MCM) of an equivalent homogeneous SDFG (HSDFG). The period thus computed gives the minimum period possible with infinite hardware resources e.g. buffer space. If worst-case execution time estimates of each actor are used, the performance at run-time is guaranteed to meet the analyzed throughput. For multiple applications with soft real-time constraint, an iterative approach similar to [Kumar et al. 2010] can be adopted to analyze and estimate throughput.

SDFGs allow buffer-sizes to be modeled as a back-edge with initial tokens. In such cases, the number of tokens on that edge indicates the buffer-size available. When an actor writes data on a channel, the available size reduces; when the receiving actor consumes this data, the available buffer increases. Figure 1 (b) shows such an example, where the buffer size of the channel from actor $\boldsymbol{a}_1$ to $\boldsymbol{a}_2$ is shown as two. Before $\boldsymbol{a}_1$ can be executed, it has to check if enough buffer space is available. This is modeled by requiring tokens from the back-edge to be consumed. Since it produces one token per

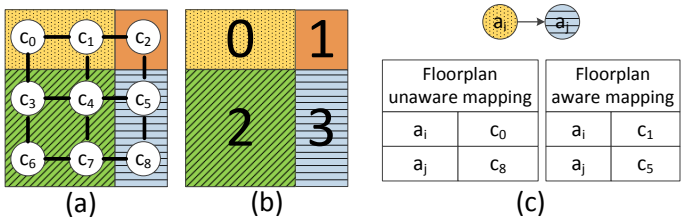


Fig. 2: Conceptual architecture model

firing, one token from the back-edge is consumed, indicating reservation of one buffer space on the output edge. On the consumption side, when $\boldsymbol{a}_2$ is executed, it frees two buffer spaces, indicated by a release of two tokens on the back-edge. In the model, the output buffer space is claimed at the start of execution, and the input token space is released only at the end of firing. This ensures atomic execution of the actor.

Self-timed strategy is widely used for scheduling SDFGs on multiprocessor systems. In this technique, the exact firing of an actor on a core is determined at design-time using worst-case actor execution-time. The timing information is then discarded retaining the assignment and ordering of the actors on each core. At run-time, actors are fired in the same order as determined from design-time. Thus, unlike fully-static schedules, a self-timed schedule is robust in capturing the dynamism in actor execution time. In this respect the following lemmas are stated. For proof, readers are urged to refer [Ghamarian et al. 2006].

LEMMA 1. *For a consistent and strongly connected SDFG, the self-timed execution consists of a transient phase followed by a periodic (steady-state) phase.*

LEMMA 2. *For a consistent and strongly connected SDFG, the throughput of an actor is given by the average firing of the actor per unit time in the periodic phase of the self-timed execution.*

This paper focuses on streaming applications represented as SDFGs. However, the techniques proposed are generic and applicable to both SDFGs and DAGs. Sections requiring special treatment for either of them are appropriately highlighted.

4. PROBLEM FORMULATION

4.1. Architecture model

The architecture assumed for the target platform is shown in Figure 2 (a) with the processing cores interconnected in a mesh-based topology. Figure 2 (b) shows the corresponding floorplan where different zones represent heterogeneity with the cores within each zone being homogeneous. In all existing *reactive* fault-tolerant studies, floorplanning of the cores is ignored i.e. heterogeneous cores are considered without their actual coordinates. This can impact application communication energy as shown in Figure 2 (c). Here, actors $\boldsymbol{a}_i$ and $\boldsymbol{a}_j$ requires core type 0 and 1 respectively. Floorplan-unaware and floorplan-aware mapping examples are provided in two tables as shown in the figure. Clearly, floorplan-unaware mapping can lead to higher data communication energy (data communicated over 4 hops between $\boldsymbol{c}_0$ and $\boldsymbol{c}_8$ as compared to 2 hops between $\boldsymbol{c}_1$ and $\boldsymbol{c}_5$ in floorplan-aware mapping).

An architecture is represented as a graph $G_{arc} = (V_{arc}, E_{arc})$, where V_{arc} is the set of nodes representing cores of the architecture and E_{arc} is the set of edges representing communication channels among the cores. Each core $\boldsymbol{c}_j \in V_{arc}$ is a tuple $\langle h_j, F_j \rangle$, where h_j represents the heterogeneity type of the core and F_j is the set $\{\omega_{jk}\}$ of frequencies supported on the core.

4.2. Mapping representation

For ease of representation, the following notations are defined.

n_{arc}	number of cores of the architecture i.e. $n_{arc} = \|V_{arc}\|$
n_{app}	number of actors of the given application i.e. $n_{app} = \|A\|$
M_n	mapping of G_{app} on G_{arc} with n cores where $n \leq n_{arc}$
$\Phi(i)$	core on which actor $\boldsymbol{a}_i$ is mapped in mapping M_n
$\Omega(i)$	frequency assigned to actor $\boldsymbol{a}_i$
$\Psi(j)$	set of actors mapped to core $\boldsymbol{c}_j$
s_f	Fault scenario with f faulty cores $= \langle \boldsymbol{c}_{i_1}, \boldsymbol{c}_{i_2}, \cdots, \boldsymbol{c}_{i_f} \rangle$

The mapping M_n is a $2 \times n_{app}$ matrix as shown below.

$$M_n = \begin{pmatrix} \Phi(1) & \Phi(2) & \cdots & \Phi(n_{app}) \\ \Omega(1) & \Omega(2) & \cdots & \Omega(n_{app}) \end{pmatrix}$$

The core assignment for the actors of the mapping M_n are indexed by $M_n.\Phi[1 : n_{app}]$. Here n is the number of cores used for the mapping and is equal to the number of unique elements in the set $\{M_n.\Phi(1), M_n.\Phi(2), \cdots, M_n.\Phi(n_{app})\}$. The frequency assignment for the actors are indexed as $M_n.\Omega[1 : n_{app}]$.

An ID is assigned to each mapping M_n as calculated in Equation 2.

$$mID(M_n) = \sum_{j=1}^{n_{app}} M_n.\Phi(j) * (n_{arc})^j \tag{2}$$

Clearly, every mapping can be uniquely represented using this linearization technique. For the ease of problem formulation a variable x_{ijk} is defined as follows

$$x_{ijk} = \begin{cases} 1 & \text{if actor } \boldsymbol{a}_i \text{ is mapped on core } \boldsymbol{c}_j \text{ at frequency } \omega_k \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

4.3. Computation energy modeling of an application

The total computation power of an application is given as the sum of the dynamic and the leakage power. The focus of this research is on reduction of dynamic power and hence is orthogonal to any leakage power reduction techniques. The dynamic power of a circuit is given by Equation 4 where β is the activity factor, ω is the frequency of operation, C_{eff} is the effective load capacitance and V_{dd} is the supply voltage. The frequency of operation is related to the supply voltage according to the α-Sakurai law as given in equation 6, where K is a constant, α is a process-dependent parameter that models velocity saturation and V_t is the CMOS threshold voltage.

$$P = \beta * \omega * C_{eff} * V_{dd}^2 \tag{4}$$

$$\omega = K * \frac{(V_{dd} - V_t)^\alpha}{V_{dd}} \tag{5}$$

As established in [Meijer and de Gyvez 2008], for 65nm low power CMOS, the frequency scales linearly with supply voltage. Equation 5 can be rewritten as

$$\omega \propto V_{dd} \tag{6}$$

The computation energy of an SDFG is given by $E_{comp} = E_{comp}^{tr} + N_{iter} * E_{comp}^{ss}$ where E_{comp}^{tr} is the actor computation energy in the transient phase of the schedule, E_{comp}^{ss} is the actor computation energy per iteration of the steady state phase and N_{iter} is the number of iterations of the steady state phase. Usually, the number of steady state iterations (i.e. N_{iter}) is a large number (can be regarded as periodic decoding of every frame for a video application) and hence for all practical purposes, the computation

energy of the steady state phase dominates over that in the transient phase. Similar reasoning applies for actor communication energy. Throughout the rest of this paper, computation (or communication) energy implies computation (or communication) energy of the steady state phase per iteration.

Denoting t_{ijk} as the execution time of the actor $\boldsymbol{a}_i$ on core $\boldsymbol{c}_j$ operating at frequency ω_k, the dynamic energy consumption is given by Equation 7 where $Rpt[\boldsymbol{a}_i]$ is the number of firings of actor $\boldsymbol{a}_i$ per steady state iteration of the SDFG.

$$e_{ijk} = P * t_{ijk} * Rpt[\boldsymbol{a}_i] = \beta * \omega_k * C_{eff} * V_{dd}^2 * t_{ijk} * Rpt[\boldsymbol{a}_i] \tag{7}$$

The execution time of an actor can be expressed in terms of its execution cycles i.e. $t_{ijk} = \frac{n_{ij}}{\omega_k}$. Substituting this in Equation 7 and using Equations 4 and 6 yields

$$e_{ijk} = \alpha * C_{eff} * V_{dd}^2 * n_{ij} * Rpt[\boldsymbol{a}_i] \approx K' * \omega_k^2 * n_{ij} * Rpt[\boldsymbol{a}_i] \tag{8}$$

where K' is a constant. The computation energy of the application is given by[2]

$$E_{comp} = \sum_i \sum_j \sum_k e_{ijk} * x_{ijk} \tag{9}$$

4.4. Communication energy modeling of applications

Communication energy modeling for NoC-based MPSoCs has received significant attention in recent years. In [Ye et al. 2003], bit energy (E_{bit}) is defined as the energy consumed in transmitting one bit of data through the routers and links of a NoC.

$$E_{bit} = E_{S_{bit}} + E_{L_{bit}} \tag{10}$$

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 11 according to [Hu and Marculescu 2004] where $n_{hops}(p,q)$ is the number of routers between cores $\boldsymbol{c}_p$ and $\boldsymbol{c}_q$.

$$E_{bit}(p,q) = \begin{cases} n_{hops}(p,q) * E_{S_{bit}} + (n_{hops}(p,q) - 1) * E_{L_{bit}} & \text{if } p \neq q \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

The communication energy (per iteration) is therefore given by Equation 12 where $\Phi(i)$ and $\Phi(j)$ are the cores where actors $\boldsymbol{a}_i$ and $\boldsymbol{a}_j$ are mapped respectively.

$$E_{comm} = \sum_{\forall \boldsymbol{a}_i, \boldsymbol{a}_j \in A} d_{ij} * E_{bit}(\Phi(i), \Phi(j)) \tag{12}$$

4.5. Migration overhead modeling of application

Migration overhead associated with moving from one mapping to another is governed by two quantities – the state space of the actors(s) participating in the migration process and the distance (hops) through which the state space is migrated[3]. It is assumed that a given multiprocessor system consists of one or more *task migration modules (TMMs)* which can access the memory of a core without interfering its operation. For these systems, state space of an actor (mapped to a faulty core) can be recovered and hence migrated to some other core where the actor is mapped post fault occurrence. For multiprocessor systems without *TMM(s)*, task migration involves migrating the state space of an actor from the main memory to the new core where it is mapped.

[2]The proposed technique deals with scheduling-based energy minimization of an application and is orthogonal to circuit level energy minimization techniques for the NoC (e.g. at switch fabric or network interface) or the processing cores (e.g. clock gating/power gating).

[3]The state space of an actor consists of the the data memory and the pre-compiled object code for the h different core types.

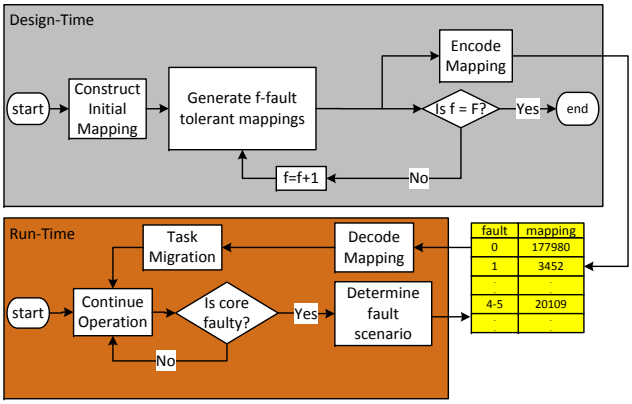


Fig. 3: Design Methodology

To better couple with the computation and the communication energy, the migration overhead is represented as energy and is termed as *migration energy*. Let $\boldsymbol{a}_i$ be an actor mapped on core $\boldsymbol{c}_j$. Denoting $\boldsymbol{c}_k$ as the core on which the actor $\boldsymbol{a}_i$ is migrated after core $\boldsymbol{c}_j$ becomes faulty, the migration energy is calculated according to Equation 13.

$$MigEnergy(j \rightarrow k) = \sum_{\forall \boldsymbol{a}_i \in \Psi(j)} \mu_i * E_{bit}(j,k) \tag{13}$$

The migration energy constitutes a very small fraction of the overall energy consumption as established in Section 7.5. Unless otherwise stated, the migration energy is ignored for most of the experiments.

5. DESIGN METHODOLOGY

The fault-tolerant task mapping methodology consists of two phases – analysis of applications at design-time and execution at run-time. The focus of this research is on the design-time analysis; however, for the sake of completeness, a brief overview is provided on how to use the design-time analysis results at run-time.

The fault-tolerant task mapping methodology is outlined in Figure 3. For every fault-scenario with f faulty cores, an optimal mapping is generated which satisfies the throughput requirement and results in minimum energy overhead. These mappings are encoded by the *Encode Mapping* block (according to Equation 2) and stored in memory. At run-time, an application is executed until faults occur. On detection of a fault[4], the corresponding fault-scenario is identified and the encoded mapping is fetched from the memory. This mapping is then decoded by the *Decode Mapping* block and forwarded to the *Task Migration* block where actual migration is performed[5].

The rest of this section is organized as follows. In Subsection 5.1 the fault-tolerant mapping generation technique is highlighted. An essential component of this is the minimum energy mapping generation which is described in Subsection 5.2. Finally in Subsection 5.3, a technique is proposed to select an initial mapping which minimizes application computation and communication energy.

[4]Our research is orthogonal to any fault-detection mechanism

[5]It is to be noted that, mappings and schedules determined at design-time for different fault-scenarios satisfy an application throughput requirement. By enforcing these mappings and schedules at run-time post fault occurrences, throughput is guaranteed for the application under all processor fault-scenarios.

Algorithm 1 Generate fault-tolerant mappings

Input: Initial mapping $M_{n_{arc}}$, G_{app}, G_{arc}, throughput constraint C, fault-tolerance level F
Output: Minimum energy mappings for all fault scenarios with $f = 1$ to F faults

1: **for** $f = 1$ to F **do**
2: $S^f = genFaultScenarios(f)$
3: **for** $s_f \in S^f$ **do**
4: $s_f = (\boldsymbol{c}_{i_1}, \boldsymbol{c}_{i_2}, \cdots, \boldsymbol{c}_{i_{f-1}}, \boldsymbol{c}_{i_f})$ *//represent fault-scenario*
5: $s_{f-1} = (\boldsymbol{c}_{i_1}, \boldsymbol{c}_{i_2}, \cdots, \boldsymbol{c}_{i_{f-1}})$ *//generate reduced fault-scenario*
6: $M_{f-1} = HashMap[s_{f-1}].getMap()$ *//fetch mapping for reduced fault-scenario*
7: $M_f = genMinEnergyMap(M_{f-1}, G_{app}, G_{arc}, C, \boldsymbol{c}_{i_f}, s_f)$ *//gen. min. energy map*
8: $HashMap[s_f].setMap(M_f)$ *//store mapping for the fault-scenario*
9: **end for**
10: **end for**

5.1. Fault-tolerant mapping generation

Fault-tolerant mappings are generated using Algorithm 1. There are F stages of the algorithm, where F is a user-defined parameter denoting the maximum number of faults to be tolerated in the device. At every stage f ($1 \leq f \leq F$), mappings are generated, one for each fault-scenario with f faulty cores.

The first step at every stage of the algorithm is the generation of a set (S^f) of fault-scenarios (line 2). The cardinality of this set (denoting the number of fault-scenarios) is ${}^{n_{arc}}P_f$, where n_{arc} is the initial number of cores in G_{arc}. An example set with 2 out of 3 cores as faulty ($f = 2, n_{arc} = 3$) is the set $S^f = \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 0,2\rangle, \langle 2,0\rangle, \langle 1,2\rangle, \langle 2,1\rangle\}$[6]. For every scenario of the set S^f, the last core ($\boldsymbol{c}_{i_f}$) of the tuple $\langle \boldsymbol{c}_{i_1}, \boldsymbol{c}_{i_2}, \cdots, \boldsymbol{c}_{i_f}\rangle$ is considered as the current faulty core and a lower order tuple is generated by omitting $\boldsymbol{c}_{i_f}$ (line 5). This gives fault-scenario s_{f-1} with $f-1$ faulty cores for which the optimal mapping is already computed (and stored in $HashMap$) in the previous stage (i.e. at stage $f-1$). As an example, the fault-scenario $\langle 3,1,5\rangle$ implies that faults occurred first on core $\boldsymbol{c}_3$ followed by on core $\boldsymbol{c}_1$ and finally on core $\boldsymbol{c}_5$. Thus, to reach this fault-scenario, the system need to encounter fault-scenario $\langle 3,1\rangle$ first. Mapping for $\langle 3,1\rangle$ is therefore considered as the starting mapping for $\langle 3,1,5\rangle$ with core $\boldsymbol{c}_5$ as current failing core. Similarly, mapping for $\langle 3\rangle$ is the starting mapping for scenario $\langle 3,1\rangle$ with core $\boldsymbol{c}_1$ failing next. A point to note here is that, the scenario $\langle 3\rangle$ is a single fault scenario and to reach this, the starting mapping is the no fault initial mapping $M_{n_{arc}}$.

An important aspect of Algorithm 1 is the generation of the minimum energy mapping $genMinEnergyMap()$. This routine takes a starting mapping (M_n), the current faulty core (j) and the fault-scenario (s_f) and generates a new mapping M_{n-1} with core $\boldsymbol{c}_j$ as faulty. This new mapping satisfies the throughput constraint and gives minimum energy (computation and communication). Details of this routine are provided in the next subsection. Once an optimal mapping is determined (line 7), the algorithm stores it in the $HashMap$ for the particular fault-scenario (line 8). This is repeated for every scenario of the set S^f.

5.2. Generate minimum energy mapping

Mapping and scheduling of applications on a multiprocessor platform is an NP hard problem [Gary and Johnson 1979]. A heuristic is proposed to simplify this process. This is shown as a pseudo-code in Algorithm 2. The algorithm has two sections – remapping the mandatory actors (lines 2-3) and search for the minimum energy mapping (lines 5-17). The mandatory mappings are generated by remapping only the tasks on the faulty core ($\boldsymbol{c}_\vartheta$). This is done in a brute force manner by selecting $|\Psi(\vartheta)|$ cores from the set of operating cores $V_{arc} \setminus s_f$ to remap all $\boldsymbol{a}_i \in \Psi(\vartheta)$. The number of such mappings is equal

[6]A fault-scenario (0,1) implies fault occurring first at core $\boldsymbol{c}_0$ and then at core $\boldsymbol{c}_1$. Thus, fault-scenario (0,1) is different from fault-scenario (1,0) implying a permutation in the fault-scenario computation.

Algorithm 2 $GenMinEnergyMap()$: Energy aware mapping

Input: Mapping M_n, graphs G_{app} and G_{arc}, throughput constraint C, fault ID ϑ and the fault-scenario s_f
Output: New mapping M_{n-1}
1: //***mandatory section: move actors from faulty core to other operating cores***
2: Γ_ϑ = Set of mappings generated from M_n by remapping all $\boldsymbol{a}_i \in \Psi(\vartheta)$ to some $\boldsymbol{c}_j \in V_{arc} \setminus s_f$
3: $sort(\Gamma_\vartheta)$ in ascending order of communication energy
4: //***performance section: remap for minimum energy satisfying throughput***
5: $numIter = 0$; $M_{best} = M_t = \Gamma_\vartheta[numIter]$; $E_{best} = calcEnergy(M_t)$
6: **while** $numIter \leq maxMap$ **do**
7: $[\boldsymbol{a}_i\ \boldsymbol{c}_j\ \omega_k] = RemapActor(M_t, G_{app}, G_{arc}, C, s_f)$
8: **if** $\boldsymbol{a}_i \neq \emptyset$ **then**
9: $M_t.\Phi(\boldsymbol{a}_i) = \boldsymbol{c}_j$; $M_t.\Omega(\boldsymbol{a}_i) = \omega_k$
10: **else**
11: $E = calcEnergy(M_t)$
12: **if** $E < E_{best}$ **then**
13: $M_{best} = M_t$; $E_{best} = E$
14: **end if**
15: $numIter++$; $M_t = \Gamma_\vartheta[numIter]$
16: **end if**
17: **end while**
18: Return $M_{n-1} = M_{best}$

Algorithm 3 $RemapActor()$: Remap actors to minimize energy

Input: Mapping M_i, graphs G_{app} and G_{arc}, throughput constraint C, fault-scenario s_f
Output: Determine an actor to be remapped, the corresponding core and frequency
1: $E_i := calcEnergy(M_i)$; $T_i = SDF^3_M(M_i)$; $G_{best} = 0$; $\boldsymbol{a}_{best} = \emptyset$; $\boldsymbol{c}_{best} = \emptyset$; $\omega_{best} = \emptyset$
2: **for all** $\boldsymbol{a}_i \in V_{app}$ **do**
3: **for all** $\boldsymbol{c}_j \in V_{arc} \setminus s_f$ **do**
4: **for all** $\omega_k \in F_j$ **do**
5: $M = M_i$; $M.\phi(\boldsymbol{a}_i) = \boldsymbol{c}_j$; $M.\Omega(\boldsymbol{a}_i) = \omega_k$; $T = SDF^3_M(M)$; $E = calcEnergy(M)$
6: **if** $((T \geq C)$ && $(E < E_i))$ **then**
7: $G = \frac{E_i - E}{T - T_i}$
8: **if** $G > G_{best}$ **then** $G_{best} = G$; $\boldsymbol{a}_{best} = \boldsymbol{a}_i$; $\boldsymbol{c}_{best} = \boldsymbol{c}_j$; $\omega_{best} = \omega_k$
9: **end if**
10: **end for**
11: **end for**
12: **end for**
13: return $[\boldsymbol{a}_{best}\ \boldsymbol{c}_{best}\ \omega_{best}]$

to the number of ways of choosing a sample of $|\Psi(\vartheta)|$ balls with replacement from a set of $|V_{arc} \setminus s_f|$ balls. This is equal to $|V_{arc} \setminus s_f|^{|\Psi(\vartheta)|}$. These mappings are pruned according to standard speed-up techniques (such as processor load [Jiashu et al. 2012]). These mappings are stored in an array Γ_ϑ and is sorted in terms of communication energy (line 3). The $maxMap$ best mappings are selected and used in the next stage. This number ($maxMap$) is equal to the number of iterations of the performance section and determines the termination (and hence the execution time) of the algorithm. It is to be noted that the communication energy based sorting provides better result (less energy) than migration overhead or throughput-based mappings.

The performance section of the algorithm remaps one or more actors selectively to determine the minimum energy. At each iteration, the starting mapping is one of the mapping of the set Γ_ϑ. The $RemapActor()$ routine selects an actor to be remapped satisfying the throughput requirement. If the return set is non-empty (implying actors can be remapped without violating the throughput constraint), the actor is remapped to a core at a frequency as determined by the $RemapActor()$ routine (line 9). The process is continued as long as no actors can be found to be remapped without violating the throughput. When this happens (line 10), the total energy of the mapping is calcu-

Algorithm 4 Generate initial mapping

Input: G_{app} and G_{arc}
Output: Minimum energy initial mapping $M_{n_{arc}}$
1: $M_t = SDF^3(G_{app}, G_{arc})$;
2: **while** true **do**
3: $[\boldsymbol{a}_i\ \boldsymbol{c}_j\ \omega_k] = RemapActor(M_t, G_{app}, G_{arc}, C, \emptyset)$
4: **if** $\boldsymbol{a}_i \neq \emptyset$ **then**
5: $M_t.\Phi(\boldsymbol{a}_i) = \boldsymbol{c}_j;\ M_t.\Omega(\boldsymbol{a}_i) = \omega_k$
6: **else**
7: break
8: **end if**
9: **end while**
10: Return $M_{n_{arc}} = M_t$

lated using the $calcEnergy()$ routine which incorporates the two energy components[7] – computation (Equation 9) and communication (Equation 12). If this is less than the minimum energy (E_{best}) obtained so far, the best values are updated (line 13).

Algorithm 3 provides the pseudo-code for the $RemapActor()$ subroutine which uses a gradient function to evaluate each actor to core assignment. The total energy and the throughput is evaluated by moving every actor on every core at every frequency supported (line 5). The SDF^3_M is the SDF^3 engine of [Stuijk et al. 2006b] modified to compute the schedule and throughput from a given mapping[8]. If the throughput for this move is greater than the throughput constraint and the energy is lower than the energy of the initial mapping (M_i), the gradient is computed (line 7). If the gradient is higher than the best gradient obtained so far, the best values are updated (line 8). The best actor, core and frequency values are returned.

5.3. Generate initial mapping

In the existing *reactive* fault-tolerant techniques, the starting mapping is determined by searching the design space exhaustively. The computation time grows exponentially with the number of actors and cores. The problem becomes computationally infeasible beyond a certain number of cores and actors. The situation becomes even worse for heterogeneous architecture where the infeasibility point settles in at a much lower value of actors and cores. Algorithm 4 provides the pseudo-code for the initial mapping generation procedure of the proposed methodology. The initial mapping (M_t at line 1) is obtained by any deterministic task mapping and scheduling algorithm e.g. $HEFT$ of [Topcuoglu et al. 2002] for DAGs or the unmodified SDF^3 engine for SDFGs. The $RemapActor()$ routine selects one actor to be remapped to a core at a frequency such that energy is minimized with least degradation of throughput. This follows the same principle as the performance section of Algorithm 2 with the all working cores i.e. setting $s_f = \emptyset$.

6. SCHEDULING

An important aspect of any application graph (cyclic and acyclic) is the scheduling of actors on cores. There are different scheduling schemes proposed both for DAGs and SDFGs [Kwok and Ahmad 1999; Sriram and Bhattacharyya 2000; Davis and Burns 2011; Damavandpeyma et al. 2012]. None of the existing fault-tolerant techniques address scheduling. If the run-time schedule is different from that used for analysis at design-time, the throughput obtained will be significantly different than what is guaranteed at design-time. There are therefore two approaches to solve the problem.

[7]The migration energy component (Equation 13) is ignored as justified in Section 7.5.
[8]For DAGs, multiple iterations are usually executed sequentially (in a non-overlapped fashion). For these graphs $CPTO$ routine of [Goh et al. 2009] can be used to compute the performance measured as *makespan*.

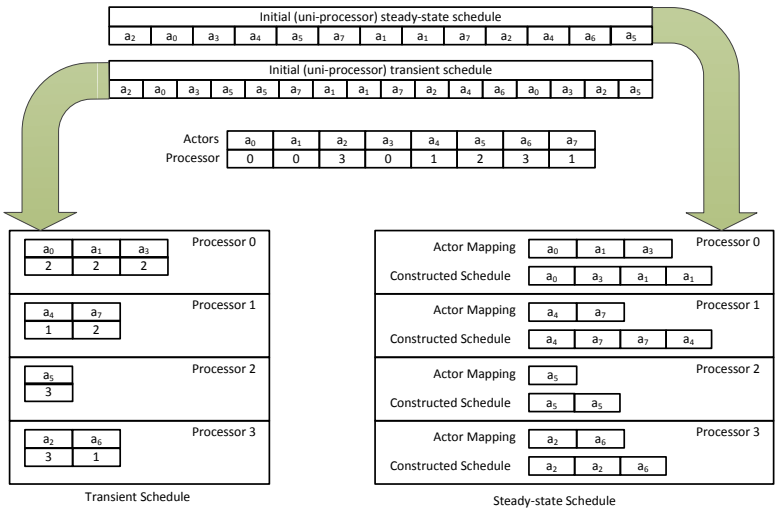


Fig. 4: Schedule construction from an initial schedule and actor allocation

— store the actor mapping and scheduling for all fault-scenarios and for all applications from design-time (*storage-based*)
— constructs the schedule at run-time based on the mappings stored from the design-time (*construction-based*)

The former is associated with high storage overhead and the latter with longer execution time. Both storage and execution time overhead are crucial for streaming applications. A self-timed execution based scheduling is proposed to solve the two problems.

Based on the basic properties of self-timed scheduling, it can be proven that if the schedule of actors on a uni-processor system is used to derive the schedules for a multiprocessor system maintaining the actor firing order, the resultant multiprocessor schedule will be free of deadlocks [Blazewicz 1976]. However, throughput obtained using this technique can be lower than the maximum throughput of a multiprocessor schedule constructed independently. Thus, as long as this throughput deviation is bounded, the schedule for any processor can be easily constructed from the mapping of actors to this processor and a given uni-processor schedule.

Figure 4 shows the operation of the proposed scheduling technique. The actor-processor mapping indicates that actors $\boldsymbol{a}_0$, $\boldsymbol{a}_1$ and $\boldsymbol{a}_3$ are mapped to processor 0. The initial steady-state schedule indicates that there are two instances of $\boldsymbol{a}_1$ and one each for actors $\boldsymbol{a}_0$ and $\boldsymbol{a}_3$ respectively. The steady-state order of actor firing on processor 0 is determined from this initial schedule by retaining only the mapped actors. In a similar way the steady-state schedules are constructed for all other processors. The transient part of the schedules are constructed from the given initial uni-processor transient schedule by retaining the mapped actors. However, the only difference of the transient phase schedule construction with the steady-state phase is that for the transient phase, the number of actors firing is important and not the exact order. This is indicated by a number against each actor for each processor as shown in the figure.

During the steady-state operation, every processor maintains counts of the number of remaining steady-state firings for the actors mapped to the core. These numbers are updated when an actor completes its execution. When a fault occurs, the mapped actors on the faulty core are moved to new location(s) (cores) along with the remaining firing count. On such cores, which have at least one incoming migrated actor, all actors are

Algorithm 5 Schedule generation

Input: $G_{app},\ N_s,\ \Delta$
Output: Schedule for all fault-scenarios

forall $f \in [1 \cdots F]$ **do** $S^f = genFaultScenarios(f)$
$maxIter = |S^f|;\ sDB = constructUniSchedule(G_{app}, N_s);\ sDB_t = sDB$
while $S^f \neq \emptyset$ **do**
  //compute rank of each initial schedule
  for all schedule $l_i \in sDB$ **do**
    Initialize $count = 0$
    for all $s_f \in S^f$ **do**
      $M_t = HashMap[s_f].getMap();\ P = SDF^3_{MS}(M_t, l_i)$
      if $P \geq C$ **then** $count++$
    end for
    $l_i.rank = count$
  end for
  $l_{min} = getHighestRankSchedule(sDB)$
  for all $s_f \in S^f$ **do**
    $M_t = HashMap[s_f].getMap();\ P = SDF^3_{MS}(M_t, l_i)$
    if $P \geq C$ **then**
      $Sche[s_f] = l_{min};\ S^f.eliminate(s_f)$
    end if
  end for
  $numIter++$
  //avoid stuck in loop
  if $numIter > maxIter$ **then** $numIter = 0;\ C = C - \Delta$
end while

allowed to execute in a self-timed manner to finish the remaining firing counts of the current pending iteration (similar to initial transient phase). From the subsequent iteration onwards, the steady-state order can be enforced for the moved actors. This will prevent the application from going into deadlock when a fault occurs. In determining the actor counting in the steady-state iterations, schedule minimization is disabled. As an example, in Figure 4, the steady state schedule constructed for processor 2 consists of two executions of actor $\boldsymbol{a}_5$ as opposed to one in the otherwise minimized schedule.

Algorithm 5 provides the pseudo-code for the modified self-timed execution technique for generating the steady-state schedule. The first step towards this is the construction of uni-processor schedules. A *list scheduling* technique is adopted for this purpose along with several algorithms for tie-breaking e.g. ETF (earliest task first), DLS (dynamic level scheduling) etc. These algorithms are implemented in the $constructUniSchedule()$ routine. The number of uni-processor schedules constructed using this routine is a user-defined parameter N_s. These schedules are stored in a database in memory (sDB). The list of fault-scenarios possible with F faults are also listed in the set S^f. Using each of the uni-processor schedules as the initial schedule, throughput is computed for the given application for all fault-scenario mappings. The SDF^3_{MS} computes the throughput of a mapping using a given uni-processor schedule.

For each uni-processor schedule from sDB, a count (termed as $rank$) is determined. The value indicates the number of fault-scenarios for which the throughput constraint is satisfied with this as the initial schedule. The schedule with the highest rank is selected and assigned as the initial schedule for the successful fault-scenarios. A fault-scenario is termed successful with respect to a schedule if the throughput constraint is satisfied with the given schedule. The successful candidates and the selected schedule are discarded from the list of fault-scenarios (S^f) and schedule database (sDB) respectively and stored in a 2-Dimensional database $FSche$. The process is repeated as long as the set S^f is non-empty.

The limited set of uni-processor schedules does not guarantee throughput satisfiability for all fault-scenarios. If such a fault-scenario exists, S^f is never $\emptyset$ causing the

algorithm to be stuck in a loop. To avoid such situations, a check is performed (line 22) to limit the number of iterations. The maximum number of iterations is upper bounded by the number of fault-scenarios. Every time the iteration count reaches this value, the throughput constraint is decremented by a small quantity (Δ). The granularity of this is based on the execution time and solution quality trade-off.

7. RESULTS

Experiments are conducted on synthetic and real application graphs on Intel Xeon 2.4 GHz server running Linux. Fifty synthetic applications are generated with the number of actors in each application selected randomly from the range 8 to 100. Additionally, fifteen real applications are considered with seven from streaming and the remaining eight from non-streaming domain. The streaming applications are obtained from the benchmarks provided in the SDF^3 tool [Stuijk et al. 2006b]. These are *H.263 Encoder*, *H.263 Decoder*, *H.264 Encoder*, *MP3 Decoder*, *MPEG4 Decoder*, *JPEG Decoder* and *Sample Rate Converter*. The non-streaming application graphs considered are *FFT*, *Romberg Integration* and *VOPD* from [Bertozzi et al. 2005] and one application each from *automotive*, *consumer*, *networking*, *telecom* and *office automation* benchmark suite [Dick 2013]. These applications are executed on MPSoC architectures consisting of 4 to 25 cores arranged in a mesh-based topology. A heterogeneity of 3 ($h = 3$) is assumed for the cores i.e. each core can be of one of the three different types. Four frequency levels are assumed for each core. Although, these parameters are assumed for simplicity, the algorithms can be trivially applied to any heterogeneity levels with any supported frequencies. The bit energy (E_{bit}) for modeling communication energy of an application is calculated using expressions provided in [Ye et al. 2003] for packet-based NoC with Batcher-Banyan switch fabric using 65nm technology parameters from [Zhao and Cao 2007].

All algorithms developed in this work are coded in *C++*. Since this is the first work on *reactive* fault-tolerance considering throughput, computation and communication energy optimization jointly, there are no existing works for comparison. However, results of this work are compared with some of the existing *reactive* fault-tolerant techniques such as the throughput maximization technique of [Lee et al. 2010] (referred to as *TMax*), the migration overhead minimization technique of [Yang and Orailoglu 2007] (referred as *OMin*), the energy minimization technique of [Hu and Marculescu 2004] (referred as *EMin*), the throughput constrained migration overhead minimization technique of [Das and Kumar 2012] (referred as *TConOMin*) and the throughput constrained communication energy minimization technique of [Das et al. 2012] (referred as *TConCMin*). The technique proposed here minimizes total energy (computation and communication energy) with throughput as a constraint and is referred as *TConEMin*. The objective of these comparisons is to establish the fact that the existing techniques when applied to MPSoC can lead to sub-optimal results in terms of energy consumption and throughput per unit energy metric.

7.1. Complexity analysis of algorithms

There are three algorithms proposed in this paper – fault-tolerant mapping generation algorithm (Algorithms 1, 2 and 3), the initial mapping generation algorithm (Algorithm 4) and the schedule generation algorithm (Algorithm 5). The complexity of Algorithm 1 is calculated as follows. The number of iterations of the algorithm is determined by the number of fault scenarios with F faults. This is given by Equation 14.

$$n_{FS} = \sum_{f=1}^{F} {}^{n_{arc}} P_f \tag{14}$$

At each iteration, the $genMinEnergyMap$ algorithm is invoked. The overall complexity of Algorithm 1 is given by Equation 15 where $O(C_2)$ is the complexity of $genMinEnergyMap$ (Algorithm 2).

$$O(C_1) = O\left(n_{FS} * O(genMinEnergyMap)\right) = O(n_{FS} * O(C_2)) \tag{15}$$

The complexity of Algorithm 2 is governed by two factors – parameter $maxMap$ and the routine $RemapActor()$. Core and frequency assignments for an actors are accomplished in constant time. Assuming the $RemapActor()$ routine to be executed η times on average for each value of $numIter$, the complexity of Algorithm 2 is

$$O(C_2) = maxMap * \eta * O(RemapActor) \tag{16}$$

The $RemapActor()$ routine remaps each actor on each functional core at each frequency to determine if the throughput constraint is satisfied and the energy is lower than the minimum energy obtained so far. If actor assignment operations take unit time and the complexity of the SDF^3_M engine is denoted by $O(SDF^3_M)$, the overall complexity of Algorithm 3 is given by Equation 17 where n_{freq} is the average number of frequency levels supported on a core.

$$O(RemapTask) = O(C_3) = O(n_{app} * n_{arc} * n_{freq} * O(SDF^3_M)) \tag{17}$$

Combining Equations 15, 16 and 17, the complexity of the fault-tolerant mapping generation algorithm is given by equation 18.

$$O(C_1) = O(n_{FS} * maxMap * \eta * n_{app} * n_{arc} * n_{freq} * O(SDF^3_M)) \tag{18}$$

The complexity of the schedule generation algorithm (Algorithm 5) is calculated as follows. The rank computation for all the uni-processor schedules can be performed in $O(n_s * n_{FS})$ time where n_s is the number of uni-processor schedules constructed and n_{FS} is the number of fault-scenarios. The highest throughput rank can be selected in $O(n_s)$ and lines 15-19 can be performed in $O(n_{FS} * O(SDF^3_{MS}))$. Finally, the outer while loop (lines 3-23) is repeated n_{FS} times in the worst case. Combining,

$$O(C_5) = O(n_{FS} * [n_s * n_{FS} + n_s + n_{FS} * O(SDF^3_M)]) = O(n^2_{FS} * O(SDF^3_M))$$

7.2. Selection of initial mapping

Figure 5 plots the throughput and the energy performance of the proposed technique in comparison with the three prior research works on *reactive* fault-tolerance. The starting mapping selection criteria for these works are highest throughput (*TMax* of [Lee et al. 2010]), migration overhead minimization with throughput constraint (*TConOMin* of [Das and Kumar 2012]) and communication energy minimization with throughput constraint (*TConCMin* of [Das et al. 2012]) respectively. Additionally, to determine the energy overhead incurred in considering throughput in the optimization process, the proposed technique is also compared with the minimum energy starting mapping (*EMin* of [Hu and Marculescu 2004]).

Figure 5(a) plots the normalized total energy consumption per iteration of 8 real-life applications for the existing and the proposed techniques. The energy values are normalized with respect to those obtained using *TMax*. As can be seen from the figure, the energy consumption of the proposed technique (*TConEMin*) is the least among all the existing *reactive* fault-tolerant techniques. This trend is also true for all the 42 remaining applications considered (not shown explicitly here). On average, for all the applications, *TConEMin* achieves 30%, 25% and 16% less energy as compared to the *TMax, TConOMin* and *TConCMin* respectively. The energy savings with respect to *TConCMin* is lower as compared to the other two techniques because *TConCMin* minimizes communication energy component of the total energy while the other two

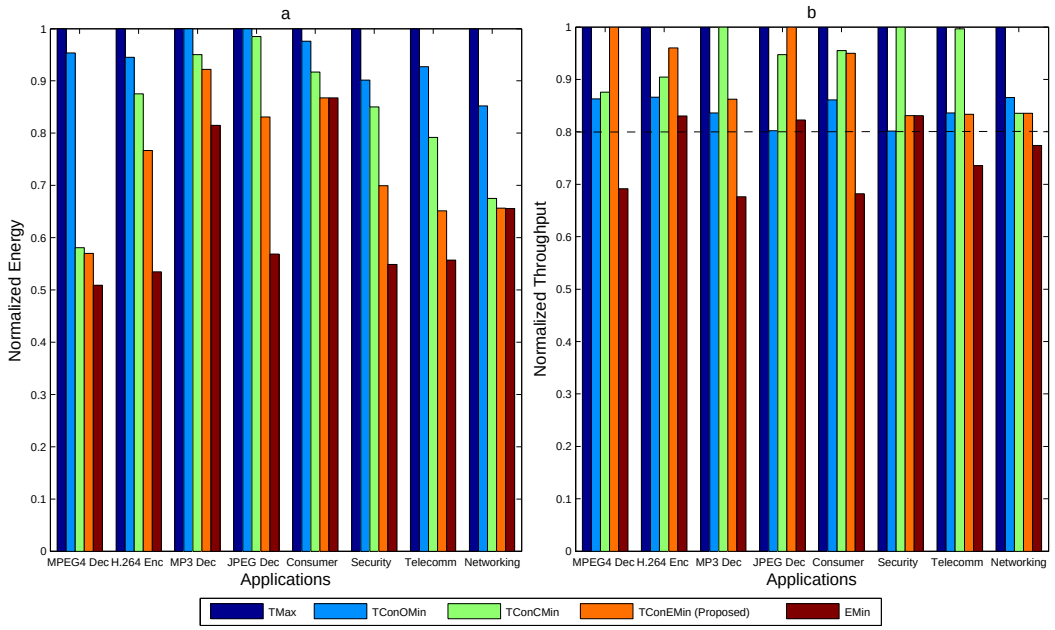


Fig. 5: Energy and performance for different applications

Table I: Number of mappings in exhaustive search

Actors	Homogeneous	Heterogeneous	
	1 core type	2 core types	3 core types
2	2	6	12
4	15	94	309
6	203	2,430	12,351
8	4,140	89,918	681,870
10	115,975	4,412,798	48,718,569
14	190,899,322	20,732,504,062	461,101,962,108

techniques do not consider energy optimization. Finally, the *TConEMin* consumes 15% more energy than *EMin* which does not consider throughput degradation.

Figure 5(b) plots the normalized throughput of all the techniques. The throughput constraint is shown by the dashed line in the figure. As previously indicated, the *EMin* does not consider throughput degradation and therefore throughput constraint is violated for most applications (a total of 45 out of all 50 applications).

Another aspect of the starting mapping generation algorithm is the execution time. The *reactive* fault-tolerant techniques in [Lee et al. 2010; Das and Kumar 2012; Das et al. 2012] search the design space exhaustively to select a starting mapping. Although this is solvable for homogeneous cores with limited number of actors and/or cores, the same becomes computationally infeasible even for small problem size as the cores become heterogeneous. Table I reports the growth in the size of the design space (number of mappings evaluated) as the number of actors scales. The number of cores in the table is same as the number of actors. If the SDF^3 engine takes an average $10\mu S$ to compute the schedule of a mapping, the design space exploration time for 14 actors on 14 cores with three types of heterogeneous cores is 54 days. The heuristic proposed in this paper solves the same problem in less than 2 hours.

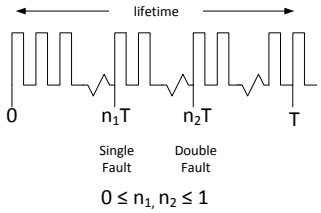


Fig. 6: Simulation environment

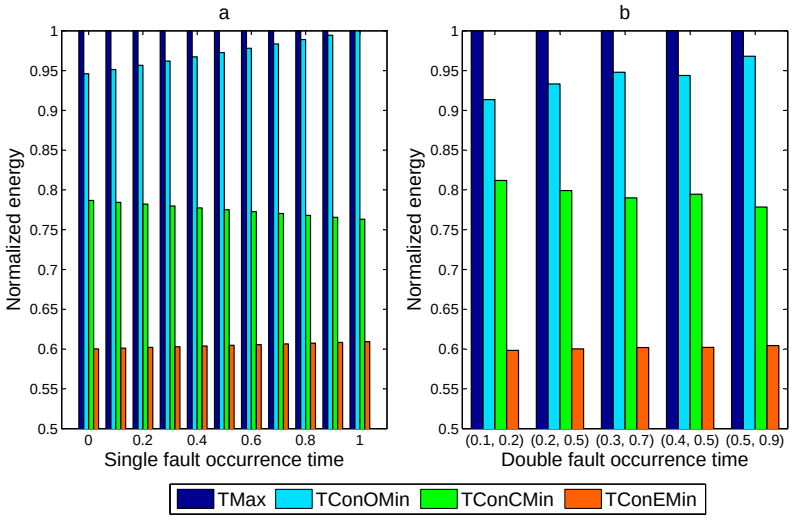


Fig. 7: Lifetime energy consumption of MPSoC with single and double faults

7.3. Energy savings with core fault scenarios

This section introduces the energy savings obtained during the overall lifetime of an MPSoC as one or more permanent faults occur. Experiments are conducted with the same set of applications (50 in total) and executed on an architecture with 2×3 cores. The number of faults is restricted to 2. These are forced to occur after $n_1 * T$ and $n_2 * T$ years respectively from the start of the device operation, where T is the total lifetime of the device and $0 \leq n_1, n_2 \leq 1$. Figure 6 represents the simulation environment. During 0 to $n_1 * T$ years, energy is consumed by the starting mapping i.e. the no-fault mapping obtained in Subsection 7.2; during $n_1 * T$ years to $n_2 * T$ years and $n_2 * T$ years to T years, energy is consumed by single fault-tolerant and the double fault-tolerant mappings respectively. The cores affected by faults are selected randomly and the results presented here is average of all single and double faults for all applications.

Figure 7(a) plots the result for single fault scenario i.e. assuming only single fault occurs during the lifetime of the device. The average energy per iteration of the application is plotted with n_1 varied from 0 to 1. A lower value of n_1 implies a fault occurs in the early life of the device while a higher value indicates faults occurring at later stages. Since *EMin* does not consider fault scenarios, only *reactive* fault-tolerant techniques (with throughput consideration) are included for comparison.

As can be seen from this figure, the energy consumption of *TMax* and *TConOMin* techniques are comparable and is higher than that consumed by the other two techniques. This is due to the non-consideration of computation and communication energy

Table II: Execution time (in secs) of existing and proposed technique

Actors	Homogeneous (1 core type)				Heterogeneous (3 core types)			
	Existing		Proposed		Existing		Proposed	
	4 cores	9 cores	4 cores	9 cores	4 cores	9 cores	4 cores	9 cores
4	110	1,450	100	1,210	150	2,150	150	1,900
8	630	6,770	410	3,100	1,810	17,440	720	5,980
12	80,100	–	1,320	6,600	2,47,000	–	2,280	12,700
16	–	–	9,700	10,600	–	–	16,750	22,400

for optimization. Although *TConOMin* minimizes migration overhead (energy), this energy is one time overhead (incurred during fault) and is negligible compared to the total energy consumed in the lifetime of the device. *TConCMin* considers communication energy and throughput jointly and therefore the energy is lower than *TMax* and *TConOMin* by average 23% and 20% respectively. The proposed *TConEMin* minimizes the total energy by achieving an average 22% savings as compared to *TConCMin*.

Figure 7(b) plots the result for double fault scenarios. A fault-coordinate (n_1,n_2) refers to the time for the first and the second fault occurrence respectively. Although, experiments are conducted for all values of the fault-coordinates, results for few of the coordinates are plotted. Similar to the single fault results, the proposed *TConEMin* also achieves 30% lower energy as compared to the existing techniques for MPSoC with two faults. These results prove that considering energy in the mapping selection for fault-tolerance is crucial for the overall MPSoC energy consumption.

7.4. Execution time comparison of the proposed fault-tolerant algorithm

As established previously, all the prior fault-tolerant works search for a suitable mapping exhaustively for different fault-scenarios. A dynamic programming is proposed in [Lee et al. 2010] to compute the minimum migration overhead incurred in moving from an initial mapping to the fault-scenario mapping. An ILP approach is proposed as an alternative in [Das and Kumar 2012; Das et al. 2012] to compute the minimum migration overhead. However, selection of the fault-tolerant mapping is based on exhaustive search. Although, dynamic programming and ILP are computationally feasible for small problem sizes, the bottleneck is in the exhaustive mapping selection process (which grows exponentially with the number of actors and cores) limiting their adaptability for large problem size and heterogeneous architectures. The work in this paper addresses this problem by proposing a heuristic algorithm with worst case complexity given by Equation 18. Table II reports the execution time of the existing approaches in comparison with the proposed heuristic for homogeneous and heterogeneous architecture with different actors and core count. For the existing approaches, the execution time reported in the table includes mapping generation time, mapping evaluation time (throughput computation) and the dynamic programming (or ILP) time for fault-tolerant mapping selection. For the proposed approach, the execution time is the sum of the execution time of Algorithms 1 and 4.

There are a few trends to be followed from this table. First of all, the execution time on heterogeneous architecture is more than that on homogeneous architecture for all actor-core combinations. This trend is same for the proposed approach as well as the existing approaches. This is due to the fact that with core heterogeneity, the execution time of an actor is different on different cores and therefore more actor-core combinations are evaluated. Secondly, the execution time of the proposed approach is comparable with that of the existing approaches for fewer actors (4 in the table) due to the fewer number of exhaustive mappings. Third, as the number of actors increases, the number of actor-core combinations (mappings) grows exponentially leading to an exponential growth in the execution time for the existing approaches. Beyond 12 ac-

Table III: Migration overhead performance

		Migration Energy (nJ)	Total Energy (nJ)	Migration Overhead Savings (nJ)	Extra Energy Per Iteration (nJ)	Iterations to recover
H.264 Encoder	*OMin*	1.1×10^9	7.2×10^5	7×10^8	3.2×10^5	$2,188$
	TConOMin	1.7×10^9	4.6×10^5	1×10^8	6×10^4	$1,667$
	TConEMin	1.8×10^9	4.0×10^5	–	–	–
MP3 Decoder	*OMin*	7.0×10^8	2.9×10^6	1.7×10^9	1.4×10^6	$1,215$
	TConOMin	1.3×10^9	2.0×10^6	1.1×10^9	5×10^5	$2,200$
	TConEMin	2.4×10^9	1.5×10^6	–	–	–

tors, these techniques fail to provide a solution due to the high memory requirement (to store the mappings) of the host CPU. The proposed technique scales well with the number of actors and cores. For 12 actors mapped on 4 cores of three different types, the proposed technique results in 100x reduction in execution time with less than 10% variation from the optimal solutions obtained by solving the Equations 9 and 12 directly while satisfying the application throughput requirement. Further, the proposed technique achieves upto 15x reduction in execution time as compared to simulated annealing based heuristic. Details are omitted for space limitations.

7.5. Migration overhead performance

Table III reports the migration overhead (measured as energy) and total energy of two existing techniques (*OMin* and *TConOMin*) in comparison with the proposed technique for two different applications (*H.264 Encoder* and *MP3 Decoder*) with 5 and 14 actors on an MPSoC with 6 cores arranged in 2×3. The core heterogeneity is fixed to 2. The other existing techniques (*TMax, EMin* and *TConCMin*) are not included for comparison as they do not optimize migration overhead. Columns 3 and 4 report the migration overhead incurred when faults occur and the average energy consumption per iteration of the application graph respectively. These numbers are average of single and double faults values. Column 5 reports the savings in migration overhead achieved by *OMin* and *TConOMin* with respect to the proposed *TConEMin*. Column 6 reports the extra energy (computation + communication) incurred in selecting the same two techniques with respect to *TConEMin*.

As can be seen from the table, significant savings in migration overhead are possible with *OMin* technique. However, this technique is associated with energy penalty (column 6). For application *H.264 Encoder* for example, the migration overhead savings in *OMin* is $7 \times 10^8 nJ$ while the energy penalty is $3.2 \times 10^5 nJ$ per iteration. As established previously, migration is one time overhead and energy is consumed in every iteration of the application graph (both pre- and post-fault occurrence). Therefore, the savings in migration overhead is compensated in $\frac{7 \times 10^8}{3.2 \times 10^5} = 2,188$ iterations ($\approx 146s$ with a 500MHz clock at encoding rate of 15 frames per sec). This is shown in column 7 of the table. Interpreting this in reverse manner, selecting *TConEMin* as the fault-tolerant technique results in an extra migration overhead of $7 \times 10^8 nJ$ which is amortized in the next (post-fault) $2,188$ iterations of the application graph.

For most of the multimedia applications, actors are executed periodically. Examples of these applications on a mobile phone include decoding of frames while playing video and fetching emails from server. Typically, these applications are executed countably infinite times in the entire lifetime of the device. If N denotes the total iterations of a device post-fault occurrence, then the first $2,188$ iterations will be used to recover the migration overhead loss while the remaining $(N - 2188)$ iterations will fetch energy savings ($3.2 \times 10^5 nJ$ per iteration). As $N \rightarrow \infty$, the energy savings obtained $= (N - 2188) \times 3.2 \times 10^5 \approx N \times 3.2 \times 10^5 nJ$. This substantial energy gain clearly justifies the non-consideration of migration overhead in the fault-tolerant mapping selection.

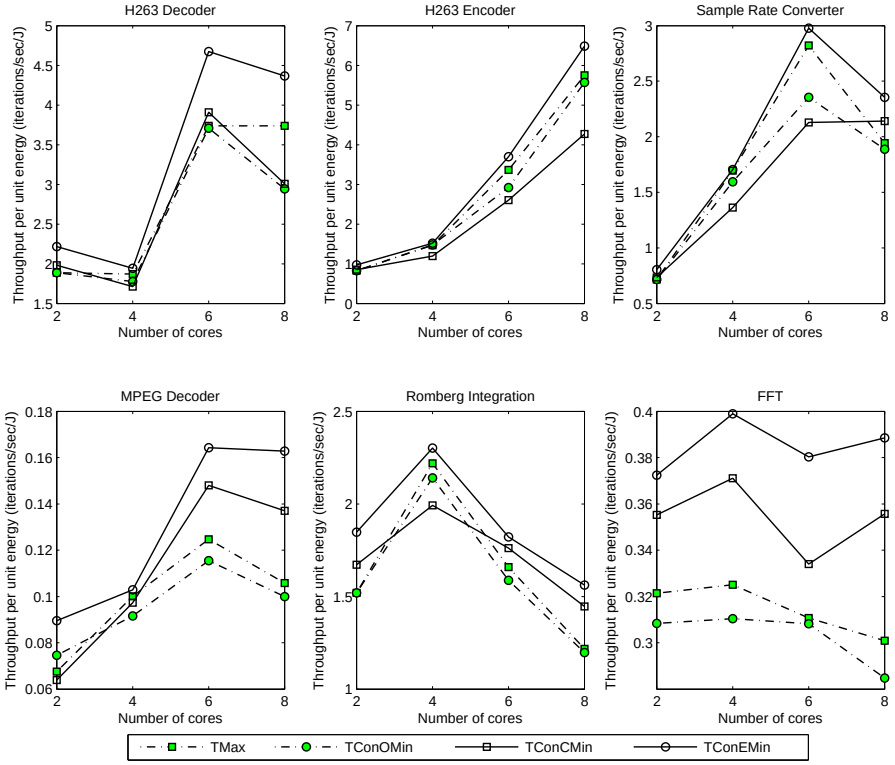


Fig. 8: Throughput-energy joint performance for real-life applications

7.6. Scalable throughput performance

Streaming multimedia applications can be broadly classified into two categories – applications, those benefiting from scalable QoS and those requiring a fixed throughput. Majority of the streaming applications such as video encoding/decoding falls in the latter category. The results of the previous sections are based on performance (throughput) as constraint. However, to signify the importance of the proposed technique for scalable throughput applications, a metric is defined (*throughput per unit energy*). The proposed and the existing techniques are compared based on this metric (Figure 8). Experiments are conducted with a set of 6 real applications on an architecture with the number of cores varying from 2 to 8. Core heterogeneity of the architectures is limited to 2 as the existing techniques fail to provide a solution for the applications with higher core heterogeneity. The results reported in the figure are average of all single and double fault scenarios. A common trend from these plots is that for most applications (except *H.263 Encoder*) the throughput per unit energy initially increases with the number of cores. However, beyond a certain core count, the throughput per unit energy decreases. This behavior is same for all the techniques i.e. *TMax, TConOMin, TConCMin* and *TConEMin*. As the number of cores increases, the throughput of an application increases. At the same time, the two energy components (computation and communication) also increase. For lower core count, the growth in throughput dominates causing an increase in the overall throughput per unit energy. As the core count increases beyond 6 cores (4 cores for *Romberg Integration* and *FFT*), the energy growth dominates over throughput growth and therefore the throughput per unit energy drops. Although, *H.263 Encoder* shows a growth in throughput per unit en-

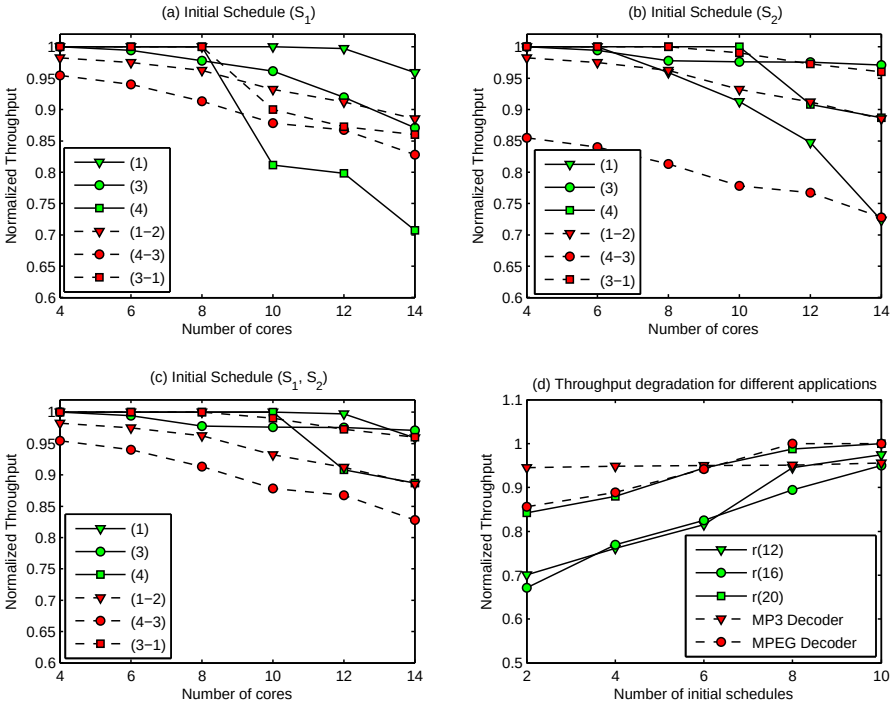


Fig. 9: Normalized throughput using the proposed self-timed execution

ergy upto 8 cores, the drop-off point is observed for *TConEMin* with 16 cores. However, the results are omitted as the exhaustive search based existing techniques – *TMax, TConOMin* and *TConCMin* fail to give a solution for that value of core count.

As can be seen, the throughput per unit energy of *TConEMin* is the highest among all existing techniques delivering on average 30% better throughput per unit energy.

7.7. Throughput performance of the proposed self-timed execution schedule

Figure 9 (a,b,c) plots the throughput obtained in the proposed self-timed execution based scheduling technique for six fault-scenarios (3 single and 3 double) of application *MP3 Decoder* as the number of cores is varied from 4 to 14. There are two initial uni-processor schedules considered ($N_s = 2$). The multiprocessor throughput obtained using these uni-processor schedules are normalized with respect to the throughput obtained using the SDF^3 tool and are plotted in Figure 9 (a,b)

As can be seen from Figure 9 (a) (with initial schedule as S_1), for all fault-scenarios, the normalized throughput decreases with increase in the number of cores. This is expected as uni-processor schedule fails to capture the parallelism available with multiple cores. Among the six fault-scenarios considered, the throughput degradation for fault-scenario *(4)* is maximum ($\approx 30\%$), while for others this is less than 20%. Similarly, for Figure 9 (b) (corresponding to initial schedule S_2), fault-scenarios *(1)* and *(4-3)* suffers the maximum throughput degradation of 25%. If the two schedules (S_1 and S_2) are considered to be available simultaneously and the one which gives the highest throughput for a fault-scenario is selected as the initial schedule, the throughput degradation can be bounded (predicted) at design time. This is shown in Figure 9 (c) where S_1 is selected as the initial schedule for fault-scenarios *(1) and (4-3)* and S_2 as the initial schedule for the remaining fault-scenarios. The maximum throughput

Table IV: Schedule storage overhead and computation time

Parameters	*H.263 Encoder*			*MP3 Decoder*		
	Storage based	Construction based	Proposed	Storage based	Construction based	Proposed
Mapping and schedule storage overhead (Kb)	892.1	68.6	68.6	1464	91.5	91.5
Run-time schedule construction time (s)	0	0.42	0.027	0	3.06	0.035
Design-time schedule construction time (s)	34.44	34.44	2.75	80	80	3.66

degradation obtained using this technique is 18%. Figure 9 (d) plots the throughput degradation obtained as the number of initial schedules is increased from 2 to 10 for five different applications. The results reported in this plot are the average of all single and double fault-scenarios. As can be seen from this figure, the throughput degradation decreases with an increase in the number of initial schedules. On average for all five applications considered, the throughput degradation is within 5% from the throughput constructed using SDF^3 with 10 initial schedules. A point to note here is that, choosing more initial schedules results in an increase in the storage complexity. Results indicate that $N_s = 10$ (i.e 10 initial schedules) offer the best trade-off with respect to storage and throughput degradation.

7.8. Schedule storage overhead and schedule computation time performance

Table IV reports the schedule storage overhead (in Kb) and the schedule computation time using the proposed self-timed execution technique in comparison with the storage-based and the construction-based techniques for two applications (*H.263 Encoder* and *MP3 Decoder*) with 5 and 14 actors on an architecture with 12 cores arranged in 3×4. The results are reported for 3 fault-tolerant systems. The *construction-based* and the *proposed* technique require storing the fault-tolerant mappings only while the *storage-based* technique stores the schedule of actors on all cores and for all fault-scenarios alongside the fault-tolerant mappings.

The run-time storage construction overhead is 0 for the *storage-based* technique because schedule needs to be fetched from a database[9]. The *construction-based* technique results in a execution time of 0.4s. The construction time increases exponentially with the number of actors and/or cores (column 3 & 6). This large schedule construction time can potentially lead to deadline violations. The proposed technique results in a linear growth of execution-time and is scalable with the number of actors and cores.

Finally, the reported execution-time of the design-time analysis phase for the *storage-based* and the *construction-based* techniques involve construction of the schedule for all the fault-scenarios. Although schedules are not stored in the *construction-based* technique, they are still computed at design-time for verification. The corresponding number for the *proposed* technique denotes the time to construct initial schedules only. On average for all fifty applications considered, the *proposed* technique reduces storage overhead by 10x (92%) with respect to the *storage-based* technique and execution time by 20x (95%) as compared to the *construction-based* technique.

8. CONCLUSION

This paper presents a design-time technique to generate mappings of an application on an architecture for all possible core fault-scenarios. The technique minimizes the energy consumption while satisfying the application throughput requirement. Experiments conducted with real and synthetic application graphs on heterogeneous MPSoC platform with different core counts clearly demonstrate that the proposed technique is able to minimize the energy consumption by 22%. Additionally, the technique also achieves 30% better throughput per unit energy performance as compared to the exist-

[9]Fetching of a schedule from a database is faster.

ing *reactive* fault-tolerant techniques. A scheduling technique is also proposed based on self-timed execution, to minimize the schedule construction and storage overhead. Experimental results indicate that the proposed approach achieves 95% less time at run-time for schedule construction. This is crucial to meet real-time deadlines. Finally, the scheduling technique also minimizes the storage overhead by 92% which is an important consideration especially for multimedia applications. An open source tool release is planned to help researchers world-wide to benefit from our work by generating fault-tolerant mappings for a given application on a given architecture.

ACKNOWLEDGMENTS

This work was supported in part by Singapore Ministry of Education Academic Research Fund Tier 1 with grant number R-263-000-655-133 and Singapore DSO funding with grant number R-263-000-A17-592.

REFERENCES

AL FARUQUE, M. A., KRIST, R., AND HENKEL, J. 2008. Adam: run-time agent-based distributed application mapping for on-chip communication. In *ACM Design Automation Conference (DAC)*.

ARTIERI, A., ALTO, V., CHESSON, R., HOPKINS, M., AND ROSSI, M. 2003. Nomadik open multimedia platform for next-generation mobile devices. *STMicroelectronics Technical Article TA305*.

BERTOZZI, D., JALABERT, A., MURALI, S., TAMHANKAR, R., STERGIOU, S., BENINI, L., AND DE MICHELI, G. 2005. Noc synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 16,* 2, 113–129.

BLAZEWICZ, J. 1976. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proceedings of the International Workshop organized by the Commision of the European Communities on Modelling and Performance Evaluation of Computer Systems*.

CHOU, C.-L. AND MARCULESCU, R. 2011. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In *IEEE Conference on Design, Automation and Test in Europe (DATE)*. 1 –6.

CONSTANTINESCU, C. 2003. Trends and challenges in VLSI circuit reliability. *IEEE Micro 23,* 4, 14–19.

CUMMING, P. 2003. The ti omap platform approach to soc. *Winning the SOC Revolution*.

DAMAVANDPEYMA, M., STUIJK, S., BASTEN, T., GEILEN, M., AND CORPORAAL, H. 2012. Modeling static-order schedules in synchronous dataflow graphs. In *IEEE Conference on Design, Automation and Test in Europe (DATE)*.

DAS, A. AND KUMAR, A. 2012. Fault-Aware Task Re-Mapping for Throughput Constrained Multimedia Applications on NoC-based MPSoC. In *IEEE Symposium on Rapid System Prototyping (RSP)*.

DAS, A., KUMAR, A., AND VEERAVALLI, B. 2012. Energy-aware communication and remapping of tasks for reliable multimedia multiprocessor systems. In *International Conference on Parallel and Distributed Systems (ICPADS)*.

DAS, A., KUMAR, A., AND VEERAVALLI, B. 2013. Reliability-Driven Task Mapping for Lifetime Extension of Networks-on-Chip Based Multiprocessor Systems. In *IEEE Conference on Design, Automation and Test in Europe (DATE)*.

DAVIS, R. I. AND BURNS, A. 2011. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR) 43,* 4, 35:1–35:44.

DE OLIVEIRA, J. AND VAN ANTWERPEN, H. 2003. The philips nexperia digital video platform. *Winning the SoC Revolution*, 67–96.

DERIN, O., KABAKCI, D., AND FIORIN, L. 2011. Online task remapping strategies for fault-tolerant Network-on-Chip multiprocessors. In *IEEE/ACM Symposium on Networks on Chip (NoCS)*.

DICK, R. 2013. Embedded system synthesis benchmarks suite (e3s).

GARY, M. AND JOHNSON, D. 1979. Computers and intractability: A guide to the theory of np-completeness.

GHAMARIAN, A., GEILEN, M., STUIJK, S., BASTEN, T., MOONEN, A., BEKOOIJ, M., THEELEN, B., AND MOUSAVI, M. 2006. Throughput analysis of synchronous data flow graphs. In *IEEE Conference on Application of Concurrency to System Design (ACSD)*.

GOH, L., VEERAVALLI, B., AND VISWANATHAN, S. 2009. Design of fast and efficient energy-aware gradient-based scheduling algorithms heterogeneous embedded multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 20,* 1, 1–12.

HU, J. AND MARCULESCU, R. 2004. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *IEEE Conference on Design, Automation and Test in Europe (DATE)*.

HUANG, J., BLECH, J., RAABE, A., BUCKL, C., AND KNOLL, A. 2011. Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In *IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

HUANG, L. AND XU, Q. 2010. Energy-efficient task allocation and scheduling for multi-mode MPSoCs under lifetime reliability constraint. In *IEEE Conference on Design, Automation and Test in Europe (DATE)*.

IRANI, S., SHUKLA, S., AND GUPTA, R. 2003. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM Transactions on Embedded Computing Systems (TECS) 2,* 3, 325–346.

JIASHU, L., DAS, A., AND KUMAR, A. 2012. A design flow for partially reconfigurable heterogeneous multiprocessor platforms. In *IEEE International Symposium Rapid System Prototyping (RSP)*.

KIM, M., BANERJEE, S., DUTT, N., AND VENKATASUBRAMANIAN, N. 2008. Energy-Aware Cosynthesis of Real-Time Multimedia Applications on MPSoCs Using Heterogeneous Scheduling Policies. *ACM Transactions on Embedded Computing Systems (TECS) 7,* 2, 9:1–9:19.

KOREN, I. AND KRISHNA, C. 2007. *Fault-tolerant systems*. Morgan Kaufmann.

KUMAR, A., MESMAN, B., CORPORAAL, H., AND HA, Y. 2010. Iterative Probabilistic Performance Prediction for Multi-Application Multiprocessor Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 29,* 4, 538–551.

KUMAR, R., TULLSEN, D. M., RANGANATHAN, P., JOUPPI, N. P., AND FARKAS, K. I. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *IEEE International Symposium on Computer Architecture (ISCA)*.

KWOK, Y.-K. AND AHMAD, I. 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys (CSUR) 31,* 4, 406–471.

LEE, C., KIM, H., PARK, H., KIM, S., OH, H., AND HA, S. 2010. A task remapping technique for reliable multi-core embedded systems. In *IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

LEE, E. AND MESSERSCHMITT, D. 1987. Synchronous data flow. *Proceedings of the IEEE 75,* 9, 1235–1245.

MANDELLI, M., OST, L., CARARA, E., GUINDANI, G., GOUVEA, T., MEDEIROS, G., AND MORAES, F. 2011. Energy-aware dynamic task mapping for noc-based mpsocs. In *IEEE International Symposium on Circuits and Systems (ISCAS)*.

MEIJER, M. AND DE GYVEZ, J. P. 2008. Technological boundaries of voltage and frequency scaling for power performance tuning. In *Adaptive Techniques for Dynamic Processor Optimization*. Springer, 25–47.

POPOVICI, K., GUERIN, X., ROUSSEAU, F., PAOLUCCI, P. S., AND JERRAYA, A. A. 2008. Platform-based software design flow for heterogeneous mpsoc. *ACM Transactions on Embedded Computing Systems (TECS) 7,* 4, 39:1–39:23.

QUAN, G. AND HU, X. S. 2007. Energy efficient dvs schedule for fixed-priority real-time systems. *ACM Transactions on Embedded Computing Systems (TECS) 6,* 4.

RAKHMATOV, D. AND VRUDHULA, S. 2003. Energy management for battery-powered embedded systems. *ACM Transactions on Embedded Computing Systems (TECS) 2,* 3, 277–324.

SCHRANZHOFER, A., CHEN, J.-J., AND THIELE, L. 2010. Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms. *IEEE Transactions on Industrial Informatics 6,* 4, 692–707.

SINGH, A. K., SRIKANTHAN, T., KUMAR, A., AND JIGANG, W. 2010. Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms. *Journal of Systems Architecture (JSA) 56,* 7, 242–255.

SRIRAM, S. AND BHATTACHARYYA, S. 2000. *Embedded Multiprocessors; Scheduling and Synchronization*. Marcel Dekker.

STUIJK, S., GEILEN, M., AND BASTEN, T. 2006a. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *ACM Design Automation Conference (DAC)*.

STUIJK, S., GEILEN, M., AND BASTEN, T. 2006b. SDF^3: SDF For Free. In *IEEE Conference on Application of Concurrency to System Design (ACSD)*.

TOPCUOGLU, H., HARIRI, S., AND WU, M. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 13,* 3, 260–274.

WEI, T., CHEN, X., AND HU, S. 2011. Reliability-Driven Energy-Efficient Task Scheduling for Multiprocessor Real-Time Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 30,* 10, 1569–1573.

WOLF, W., JERRAYA, A., AND MARTIN, G. 2008. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 27,* 10, 1701 –1713.

YANG, C. AND ORAILOGLU, A. 2007. Predictable execution adaptivity through embedding dynamic reconfigurability into static MPSoC schedules. In *IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

YE, T., BENINI, L., AND DE MICHELI, G. 2003. Packetized on-chip interconnect communication analysis for MPSoC. In *IEEE Conference on Design, Automation and Test in Europe (DATE)*.

ZHANG, Y., HAO, Z., XU, X., ZHAO, W., AND WANG, Z. 2010. Workload-balancing schedule with adaptive architecture of MPSoCs for fault tolerance. In *IEEE Conference on Biomedical Engineering and Informatics (BMEI)*.

ZHAO, W. AND CAO, Y. 2007. Predictive technology model for nano-cmos design exploration. *ACM Journal on Emerging Technologies in Computing Systems (JETC) 3,* 1.

ZHU, D. 2011. Reliability-aware dynamic energy management in dependable embedded real-time systems. *ACM Transactions on Embedded Computing Systems (TECS) 10,* 2, 26:1–26:27.