Communication and Migration Energy Aware Task Mapping for Reliable Multiprocessor Systems

Anup Das, Akash Kumar, Bharadwaj Veeravalli

Department of Electrical and Computer Engineering

National University of Singapore

Email: {akdas, akash, elebv}@nus.edu.sg

Abstract

Heterogeneous multiprocessor systems-on-chip (MPSoCs) are emerging as a promising solution in deep sub-micron technology nodes to satisfy design performance and power requirements. However, shrinking transistor geometry and aggressive voltage scaling are negatively impacting the dependability of these MPSoCs by increasing the chances of failures. This paper proposes an offline (design-time) task remapping technique to minimize the communication energy and task migration overhead of an application mapped on a heterogeneous multiprocessor system for all processor fault-scenarios. The proposed technique involves two steps – 1) Communication Energy driven Design Space Exploration (*CDSE*) to select an initial mapping and 2) Communication energy and Migration overhead aware Task Mapping (*CMTM*) for different fault-scenarios. The *CDSE* is formulated as a Mixed Integer Quadratic Programming (MIQP) problem and solved using an energy-gradient based heuristic. The *CMTM* problem is solved using a fast heuristic with the starting mapping selected using *CDSE* step. The proposed two steps technique is compared with state-of-the-art approaches through rigorous simulations with synthetic and real application graphs. Results demonstrate that the proposed *CDSE* reduces design space exploration time by 99% with a maximum variation of 5% from the optimal solution obtained by solving the MIQP problem directly. Further, the proposed *CMTM* reduces communication energy by an average 35% and migration overhead by an average 20% for all single and double fault-scenarios as compared to the existing fault-tolerant techniques. The *CMTM* also achieves over 30x reductions in execution time for large problem sizes with a maximum deviation of 15% from the minimum communication energy achievable with the given application on a given architecture. For streaming multimedia applications, the proposed technique delivers 50% higher throughput per unit energy as compared to the existing approaches.

Keywords: Fault-Tolerance, Heterogeneous MPSoCs, Design Space Exploration, Task Mapping, Mixed Integer Quadratic Programming

1. Introduction

To accommodate the ever increasing demands of applications and for the ease of scalability, multiprocessor systems-on-chip (MPSoCs) are becoming the obvious design choice in current and future technologies [1]. With reducing feature size and increasing transistor count, MPSoCs are becoming susceptible to faults [2, 3]. There are three classes of faults studied for Integrated Circuits (ICs) – permanent, intermittent and transient. Permanent faults are irrecoverable damages to the circuit caused by phenomena such as electro-migration, dielectric breakdowns, broken wires etc. These faults are caused during manufacturing or during the product lifetime due to component wear-outs. Behavior of a system under permanent faults is time invariant. Intermittent faults are also hardware faults occurring frequently but irregularly over a period of time due process, voltage and temperature (PVT) variations. Intermittent faults usually persist for few cycles, if not for a few seconds or more. Transient faults are single event upsets occurring due to alpha or neutron particles from cosmic radiations. This paper focuses on permanent and intermittent faults which are jointly referred as *hardware faults* throughout the rest of this paper.

Hardware faults are traditionally tolerated using redundancy-based designs [4]. However, this is only applicable for hardware-software co-design methodology [5] where optimization results determine MPSoC architecture. Software techniques like task remapping [6, 7, 8] have shown significant promise to tolerate hardware faults especially for platform-based MPSoC design [9] where fault-tolerance needs to be incorporated on a fixed architecture. Existing platform-based fault-tolerant task-migration research studies have focused on minimizing migration overhead [6] and load balancing [7] or maximizing the reliability of a system [8]. These techniques provide no guarantee on the application communication energy.

Another research direction for multiprocessor systems is concerning energy consumption. This is partly due to the slow growth in battery technology over the past decades. Researchers have shown that carefully selecting an application task mapping can significantly reduce task communication en-

ergy [10, 11, 12, 13] which constitutes a large fraction ($\approx$ 60% according to [11]) of the overall energy consumption. This is orthogonal to dynamic voltage or frequency scaling capabilities of an MPSoC which can further reduce the task computation energy. These works do not target fault-tolerance.

When one or more processors of an MPSoC become faulty, tasks from these processor(s) are migrated to new locations (functional processors). Tasks migrated further away from their dependent tasks consume more communication energy per iteration of the application graph. Recently, there are studies to integrate fault-tolerance and energy minimization [14, 15, 16]. However, either they are limited to transient faults or they do not address different processor fault-scenarios. This paper attempts to solve the following problem. Given a heterogeneous MPSoC architecture and a set of high performance applications modeled using directed graph.

- Generate a starting mapping of an application on the multiprocessor platform such that task communication energy is minimized and
- Generate task mappings for different processor fault-scenarios with the joint objective of minimization of migration overhead and task communication energy.

In a recent work [17], these challenges are solved for multimedia applications running on homogeneous MPSoCs. Task mappings are generated exhaustively and evaluated. The minimum communication energy mappings are retained for the optimization of migration overhead for different fault-scenarios. One limitation of this technique is that the application domain is restricted to multimedia programs running on homogeneous processors. Additionally, as the number of tasks and/or processors increase, there is an exponential growth in the number of mappings which can lead to longer design cycles.

Contributions: Following are the key contributions.

- A Mixed Integer Quadratic Programming (MIQCP) formulation of the Communication Energy based Design Space Exploration (*CDSE*).
- A energy-gradient based heuristic to minimize the design space exploration time.
- A communication energy and migration overhead aware fast heuristic for task mapping (*CMTM*) for different processor fault-scenarios.
- Consideration of streaming and non-streaming applications on heterogeneous MPSoC.

The *CDSE* is solved at design-time to generate a starting mapping of a given application on the given platform which minimizes the task communication energy. Starting from this mapping, the *CMTM* generates a set of task mappings for all fault-scenarios which are Pareto-optimal in terms of communication energy and migration overhead. These mappings are stored in a table for use at run-time as and when faults occur. Experiments are conducted with synthetic and real application graphs, both from streaming and non-streaming domain demonstrate that the proposed energy-gradient heuristic is able to reduce analysis time by 99% with a maximum of 5% deviation from the communication energy optimum value obtained by solving the MIQP problem directly. Moreover, the overall fault-tolerant technique minimizes communication energy by an average 35% and migration overhead by 20% as compared to the existing fault-tolerant techniques.

The rest of the paper is organized as follows. A brief overview of the prior research works is provided in Section 2 followed by a motivational example in Section 3. Next, the proposed fault-tolerant design methodology is highlighted in Section 4. Different components of this methodology are discussed in Sections 5 and 6. Experimental setup and results are discussed next in Section 7 and the paper is concluded in Section 8 with key future directions.

2. Related Works

Reliability and energy efficiency are the two primary optimization objectives for multiprocessor systems in deep submicron technologies. Research works for both the objectives are carried independently over decades until recently, when efforts are directed towards joint optimization of energy and fault-tolerance. This section provides an overview of some key research results for each of these research directions.

2.1. Fault-tolerance related studies

There are two broad categories of hardware fault-tolerance research – *proactive* and *reactive* as shown in Figure 1. The *proactive* techniques prevent (or delay) the occurrence of failures [8, 18, 19]. The *reactive* techniques deal with task migration after faults have occurred [6, 7, 20, 21, 22, 23]. Task remapping decisions can be pre-computed at compile-time analyzing all possible fault-scenarios (*static*) or can be decided at run-time as and when faults occur (*dynamic*).

Static task migration techniques compute task mapping decisions at compile-time for different fault-scenarios [20, 21, 22]. The band and band reconfiguration technique of [20] minimizes migration overhead while deciding new locations (cores) for the tasks on faulty cores. Communication energy is not guaranteed in this technique. The technique of [21] maximizes application throughput but provides no guarantee on the task migration overhead and communication energy. The authors in [22] established the importance of migration overhead and throughput for multimedia applications and proposed a joint optimization technique for multimedia MPSoCs. Communication energy is not optimized in this technique either. Moreover, the number of mappings explored at compile-time grows exponentially with the number of tasks and/or cores making this technique computationally in-feasible for large scale computing.

Dynamic approaches monitor system-status and decide to migrate tasks at run-time to minimize migration overhead [6] or balance processor load [7]. A limitation of these techniques is that migration algorithms need to be simple to minimize computation overhead. Optimization for communication energy and throughput can potentially lead to deadline misses.

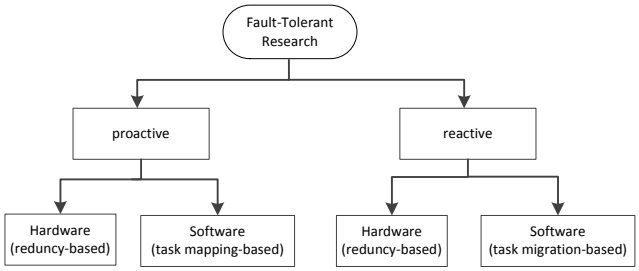

Figure 1: Taxonomy of fault-tolerance research

2.2. Energy-aware task mapping

To accommodate the ever increasing demand of performance and features in MPSoCs, energy budget is becoming more stringent. Researchers have focused on every aspect of energy reduction techniques. These techniques can be classified into circuit-level approaches (power gating for example) and software approaches such as energy-aware task scheduling.

Recently, software approaches have gained lot of interest among research community. A dynamic voltage scaling technique is proposed in [10] to minimize the energy consumption. The slack budgeting technique of [11] distributes execution time slack of a task among other tasks, to reduce their frequency of operation. A gradient-based energy minimization technique is proposed in [12]. However, none of these research works address task mappings for different processor fault-scenarios.

2.3. Energy-reliability joint optimization

In recent years, some studies have been made to jointly optimize fault-tolerance and energy. An ILP based approach is presented in [14]. Energy optimization is performed under the constraint of task-execution time which incorporates fault-tolerance overhead using check-pointing based recovery model. This technique is not suitable for permanent failures as it does not address task migration for different fault-scenarios. A global scheduling based reliability-aware energy management is proposed in [15]. However, this paper also focuses on transient faults and assumes independent tasks and is therefore not suitable for permanent fault-tolerance of applications with dependent task models. A lifetime-reliability aware scheduling technique is proposed in [16] to minimize energy consumption. Tasks are scheduled on processors equipped with dynamic voltage scaling (DVS) capabilities. However, task migration is not addressed in this work either. Recently, the authors in [17] proposed a technique to address minimization of throughput degradation, communication energy and migration-overhead jointly for multimedia applications on homogeneous MPSoCs. An ILP approach is proposed as an alternative (against dynamic programming of [21]) to compute the minimum migration overhead. However, selection of the fault-tolerant mappings is based on exhaustive search. Although, dynamic programming or ILP are computationally feasible for certain problem size, the bottleneck is in the exhaustive mapping selection process (which grows exponentially with the number of tasks and/or processors) limiting its adaptability for large scale computing

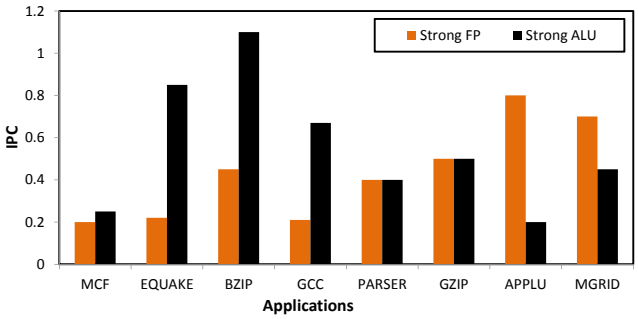

Figure 2: IPC performance for some SPEC2000 benchmarks

and heterogeneous architectures. The work in this paper addresses this problem by proposing a heuristic algorithm to generate minimum energy mappings with polynomial complexity.

3. Motivation

3.1. Heterogeneous MPSoC architecture

Modern MPSoCs are designed to execute a wide range of applications with different characteristics. Figure 2 plots the instructions per cycle (IPC) performance of some of the SPEC2000 [24] benchmarks on floating point and integer intensive processors referred in the figure as *strong FP* and *strong ALU* respectively. As can be seen from the figure, for some applications like *PARSER* and *GZIP*, the IPC performance are same when executed on *strong FP* and *strong ALU* respectively. Running integer intensive applications like *EQUAKE* on a *strong FP* processor can result in 75% reduction in IPC. Similar degradation is observed by running floating point intensive applications like *APPLU* on *strong ALU* processor. From these results it can be concluded that heterogeneous architecture (with *strong FP* and *strong ALU* in this example) can benefit the wide range of applications supported on modern MPSoCs.

3.2. Importance of hop distance

Figure 3(a) shows a synthetic application with 9 tasks mapped on an architecture with 6 processors. The no-fault task mapping is shown in Figure 3(b) with the number in parenthesis against each task indicating the size of its *state space*. Figure 3(c) and 3(d) show two different task mappings satisfying the application throughput requirement with processor c_3 as faulty. The migration overhead for Figure 3(c) involves migrating 180 units of *state space*[1] of task F from processor c_3 to processor c_0 through one hop and 120 units for task I from processor c_3 to processor c_1 through two hops. Thus, the total overhead is $180 + 2 \times 120 = 420$ units. The migration overhead for tasks F and I of Figure 3(d) are 180 and 120 units respectively through one hop each. The total overhead is therefore $180 + 120 = 300$ units. If only *state space* is considered for migration, the two configurations 3(c) and 3(d) are equally good to be selected (*state space* are 300 units each). However, selecting 3(c) results in 40% extra migration overhead in reality than 3(d) due to extra hops.

[1] State-space of a task is the size of its program and data memory.

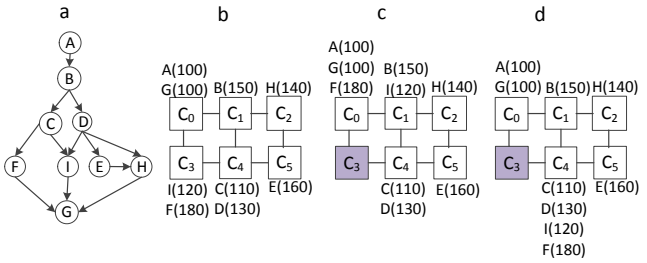

Figure 3: Importance of hop-count and communication energy

Table 1: Communication Energy Estimate

Links	Hop Distance		Token size of	Comm. Energy	
	cfg 3(c)	cfg 3(d)	source task	cfg 3(c)	cfg 3(d)
$C-F$	2	0	32	$64E_{bit}$	0
$C-I$	1	0	32	$32E_{bit}$	0
$D-I$	1	0	64	$64E_{bit}$	0
$F-G$	0	2	64	0	$128E_{bit}$
$I-G$	1	2	64	$64E_{bit}$	$128E_{bit}$
Total				$224E_{bit}$	$256E_{bit}$

3.3. Importance of task communication energy

With reference to Figure 3(c, d) five dependency relations are affected due to task remapping: $C-F$, $C-I$, $D-I$, $F-G$ and $I-G$. In Figure 3(c), the two tasks F and I of processor c_3 are mapped to processors c_0 and c_1 respectively. In Figure 3(d) however, the two tasks F and I are both mapped to processor c_4. Table 1 reports the hop distance, amount of data communicated and the communication energy for the two mappings. Thus, after processor c_3 becomes faulty, configuration 3(c) and 3(d) consume $224E_{bit}$ and $256E_{bit}$ units of extra communication energy at every subsequent iteration of the graph[2]. Clearly, the migration destination (processor) for a task on a faulty processor is crucial for the overall energy consumption.

3.4. Importance of mapping reduction

Table 2 plots the exponential growth in the possible mappings as the number of tasks and type of processors (heterogeneity) are scaled. Assuming the schedule construction and energy computation time as fixed (and equal to $50\mu s$ on average obtained from experiments), the total execution time of the algorithm in [17] for 14 tasks on 14 processors of 3 different types is 6,404 hours (equivalent to 266 days). Clearly, efficient design space pruning is needed to reduce the analysis time.

4. Design methodology

The reliability-aware task mapping methodology consists of two phases - analysis at compile-time and execution at run-time. The focus of this research is on the compile-time analysis; however, for the sake of completeness, a brief overview is provided on how to use the compile-time analysis result at run-time.

[2] E_{bit} is energy required to communicate every bit of information through the communication fabric (e.g. NoC)

Table 2: Number of mappings in exhaustive search

Tasks	Homogeneous	Heterogeneous	
	1 processor type	2 processor types	3 processor 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

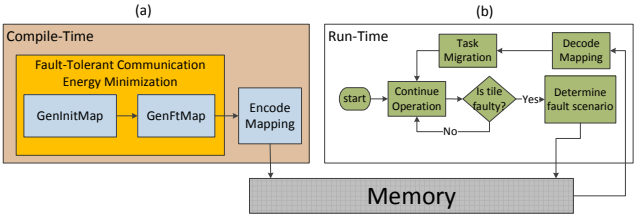

Figure 4: Design methodology

4.1. Compile-time analysis

The compile-time analysis stage is highlighted in Figure 4(a). The *GenInitMap* block implements the *CDSE* algorithm to generate a starting mapping. The *GenFtMap* block implements the *CMTM* algorithm and generates a set of fault-tolerant mappings. These mappings are stored in memory for use at run-time. Sections 5 and 6 provide details of the two blocks *GenInitMap* and *GenFtMap* respectively.

4.2. Run-time execution

Figure 4(b) represents the run-time execution phase. When an application is enabled on the platform, the entire set of processors is dedicated to the application. The optimum mapping (in terms of communication energy) is fetched from memory and applied. When one or more processors fail, the fault-scenario is identified and the best mapping (in terms of communication energy and migration overhead) for the scenario is fetched and applied. Thus, as processor fails either for a short duration (intermittent failures) or permanently (permanent failures), minimum overhead is incurred in task migration to a new mapping which minimizes communication energy.

5. Generate initial mapping

Mapping and scheduling of applications on multiprocessor platform is an NP hard problem [25]. Many heuristics have been developed over years to schedule dependent tasks on heterogeneous platforms [26, 27]. A closely related problem to this is the optimization criteria directed pruning of design space. There are different criteria studied in literature such as minimum energy, throughput and least resource usage. The *GenInitMap* block formulates the task mapping problem with communication energy as the optimization objective.

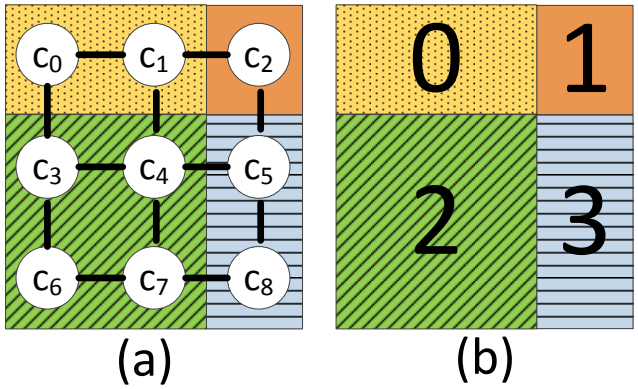

Figure 5: Conceptual architecture model

5.1. Application model

An application is a directed graph $G_{app} = (V_{app}, E_{app})$, where V_{app} is the set of nodes representing tasks of the application and E_{app} is the set of edges $\{e_{ij} \mid 1 \leq i, j \leq |V_{app}|\}$, representing data dependency among tasks. Each task $v_i \in V_{app}$ is a tuple $\langle T_i, S_i, D_i \rangle$, where T_i is the execution time of v_i, S_i is its state space and D_i is the data produced at every execution of v_i. Let h be the types of heterogeneous cores in the platform. The execution time T_i is a set $\{t_{ik} \mid 1 \leq k \leq h\}$, representing the execution time of the task v_i on h cores type. For homogeneous system $h = 1$ and therefore execution time of a task on all the cores are the same. D_i is the set $\{d_{ij} \mid 1 \leq j \leq |V_{app}|\}$, representing the data produced on edge e_{ij}.

5.2. Architecture model

The conceptual architecture model for the target platform is shown in Figure 5(a) with the processing cores interconnected in a mesh-based topology. The scope this paper is limited to fault-tolerance and energy performance of an application on a given architecture. Factors such as deciding the number of cores required or the placement of the different heterogeneous cores in the architecture are not considered. Figure 5(b) shows the floorplan assumed in this work, where different zones represent heterogeneity. The cores within each zone are homogeneous. An architecture is represented as a graph $G_{arc} = (V_{arc}, E_{arc})$, where V_{arc} is the set of nodes representing processors and E_{arc} is the set of edges representing communication channels among the processors. Each processor $c_i \in V_{arc}$ is associated with a heterogeneity type (h_i).

5.3. Mapping representation

For the ease of representation and the algorithm formulation a linearization technique is applied where mapping of an application on the architecture is represented by a tuple $M = \langle M(1), M(2), \cdots M(n_{app}) \rangle$ where n_{app} ($= |V_{app}|$) is the number of tasks and $M(k)$ is the processor on which task v_k is mapped. An ID is given to the mapping as calculated in Equation 1.

$$mID = \sum_{j=1}^{n_{app}} M(j) \times n_{arc}^{j} \tag{1}$$

where n_{arc} ($= |V_{arc}|$) is the number of processors in the given architecture. Clearly, every mapping can be uniquely represented using this linearization technique.

5.4. Communication energy modeling

Energy modeling for NoC-based MPSoCs has received significant attention in recent years. In [28], bit energy (E_{bit}) is defined as the energy consumed when one bit of data is communicated through the routers and links of a NoC.

$$E_{bit} = E_{S_{bit}} + E_{L_{bit}} \tag{2}$$

where $E_{S_{bit}}$ and $E_{L_{bit}}$ are the energy consumed by the switch and the link respectively. The energy per bit consumed in transferring data between processor p and processor q, situated $n_{hops}(p, q)$ away is given by Equation 3 according to [11].

$$E_{bit}(p,q) = \begin{cases} n_{hops}(p,q)E_{S_{bit}} + (n_{hops}(p,q) - 1)E_{L_{bit}} & \text{if } p \neq q \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $n_{hops}(p, q)$ is the number of routers between processors p and q. The total communication energy is therefore given by

$$CommEnergy = \sum_{(i,j) \in E_{app}} d_{ij} \times E_{bit}(M(i), M(j)) \tag{4}$$

5.5. Problem formulation for Directed Acyclic Graphs (DAGs)

5.5.1. Variables for MIQP formulation

$$x_{ik} = \begin{cases} 1 & \text{if task } v_i \text{ is mapped on processor } c_k \\ 0 & \text{otherwise} \end{cases}$$

$$d_{ij,k} = \begin{cases} 1 & \text{task } v_i \text{ and } v_j \text{ are mapped on processor } c_k \\ & \text{and } v_i \text{ starts execution before } v_j \\ 0 & \text{otherwise} \end{cases}$$

s_{ik} = start time of task v_i on core c_k

5.5.2. Constraints for MIQP formulation

- Every task must be assigned to a single processor

$$\forall v_i \in V_{app} : \sum_{k=1}^{n_{arc}} x_{ik} = 1 \tag{5}$$

- Finish time of every leaf task is less than the application deadline

$$\forall v_i \in L, c_k \in V_{arc} : s_{ik} + et(i,k) \leq D + (1 - x_{ik})\Gamma \tag{6}$$

where $et(i, k)$ is the execution time of task v_i on processor c_k, D is the deadline and Γ is a very large number

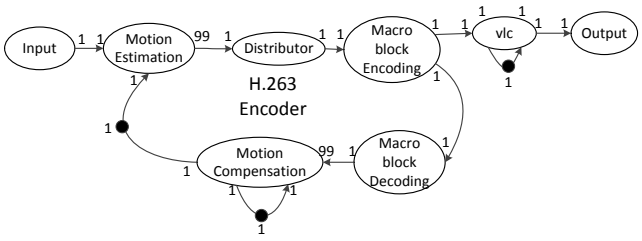

Figure 6: SDFG of H.263 Encoder

- A task can start only after the finish time of its dependent tasks

$$\forall (v_i, v_j) \in E_{app} \text{ and } c_k, c_l \in V_{arc} : s_{ik} + et(i,k) \leq s_{jl} \quad (7)$$

- Independent tasks mapped on the same processor must not be executed simultaneously

$$\begin{aligned} &\forall (v_i, v_j) \notin E_{app} \text{ and } c_k \in V_{arc} \\ &s_{ik} + et(i,k) \leq s_{jk} + (3 - x_{ik} - x_{jk} - d_{ij,k})\Gamma \\ &s_{jk} + et(j,k) \leq s_{ik} + (2 - x_{ik} + x_{jk} + d_{ij,k})\Gamma \end{aligned} \quad (8)$$

where the first equation constraints the starting time of v_i before v_j and the second with v_j before v_i.

5.5.3. Objective function

The communication energy of Equation 4 can be expressed in terms of the assignment variables x_{ij} as shown.

$$\begin{aligned} &\forall (v_i, v_j) \in E_{app}, \text{ and } c_k, c_l \in V_{arc} \\ &CommEnergy := \sum x_{ik} * x_{jl} * E_{bit}(k,l) \end{aligned} \quad (9)$$

The objective function can be written as

$$\begin{aligned} &\text{minimize } f(x) = CommEnergy \\ &\text{subject to Equations } [5-8] \end{aligned} \quad (10)$$

5.6. Problem formulation for SDF Graphs

Synchronous Data Flow Graphs (SDFGs, see [29]) are often used for modeling modern DSP applications and for designing concurrent multimedia applications implemented on a multiprocessor system-on-chip. 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 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 data that is communicated from one actor to another.

Figure 6 shows the SDF Graph of H.263 encoder. There are eight actors in this graph. In the example, actor *motion estimation* has an input rate of 1 and output rate of 99. 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 *motion compensation* to *motion estimation* in Figure 6. A set $Ports$ of ports is assumed, and with each port $p \in Ports$ a finite rate $Rate(p) \in N \setminus \{0\}$ is associated. When an actor $\boldsymbol{a}$ starts its firing, it removes $Rate(q)$ tokens from all $(p,q) \in InC(\boldsymbol{a})$. The execution continues for $\tau(\boldsymbol{a})$ time units and when it ends, it produces $Rate(p)$ tokens on every $(p,q) \in OutC(\boldsymbol{a})$[3]. The repetation vector of an actor $\boldsymbol{a}$ (denoted by $r(\boldsymbol{a})$) is the number of times the actor is fired in one iteration of the SDFG. Apart from being cyclic, another difference of SDFG with Directed Acyclic Graphs (DAGs) is that the repetition vector of all tasks in DAGs is one.

5.6.1. Changed variable definitions

Let $G_{app} = (V, E)$ represents an application SDFG with V actors and E edges. Changed variables and additional constraints (with respect to those for DAG) are presented.

$s_{ik,u}$ = start time of u^{th} iteration of actor i on core k

$$d^{uv}_{ij,k} = \begin{cases} 1 & \text{task } i \text{ and } j \text{ are mapped on core } k \\ & \text{and } u^{th} \text{ iteration of } i \text{ starts execution before} \\ & v^{th} \text{ iteration of } j \\ 0 & \text{otherwise} \end{cases}$$

5.6.2. Additional constraints

- *Actor iteration assignment* (iterations of an actor must be assigned to the same core)

$$\forall i, j : \sum_{u=1}^{r(i)} x_{ik,u} = 0 \text{ or } r(i) \quad (11)$$

- *Auto-concurrency of actors* (multiple iterations of an actor are not enabled simultaneously)

$$\forall i, k \text{ and } 2 \leq u \leq r(i) : s_{ik,u} \geq s_{ik,u-1} + et(i) \quad (12)$$

- *Data-dependency of actors* (u^{th} iteration of task i can start only after its dependent task finishes)

$$\forall k, l \in P \text{ and } \forall (j,i) \in E : s_{ik,u} \geq e_{jl,m} \quad (13)$$

where m is defined as follows

$$\begin{aligned} m &= \lfloor \frac{kp}{q} + init(i,j) \rfloor \\ p &= \text{tokens produced by actor } i \text{ on edge } (i,j) \\ q &= \text{tokens consumed by actor } j \text{ from edge } (i,j) \\ init(i,j) &= \text{initial token on edge } (i,j) \end{aligned}$$

[3] $InC(\boldsymbol{a})$ and $OutC(\boldsymbol{a})$ are respectively the incoming and outgoing edges of actor $\boldsymbol{a}$.

Algorithm 1 CDSE(): Communication Energy aware DSE

Input: G_{app}, G_{arc}, $MaxIter$
Output: minimum communication energy mapping
1: $\forall v_i \in V_{app}$, determine $rank(i)$ according to [30]
2: sort tasks according to rank and push to $TaskArr$
3: **for all** $v_i \in TaskArr$ **do**
4: // Assign the fastest processor to the task
5: $M(i) = c_j \in V_{arc}$
6: **end for**
7: $CE_{best} = CommEnergy(M, G_{arc})$
8: $M_{best} = M$
9: **for** $numIter = 1$ to $maxIter$ **do**
10: $[v_i\ c_k] = RemapTask(M)$
11: **if** $v_i \neq \emptyset$ **then**
12: $M(i) = c_k$
13: **else**
14: $CE = CommEnergy(M, G_{arc})$
15: $S = SGen(M, G_{arc})$
16: **if** $S \leq D$ **then**
17: **if** $CE < CE_{best}$ **then**
18: $M_{best} = M$ and $CE_{best} = CE$
19: **end if**
20: **end if**
21: $numIter++$
22: Randomly assign the tasks to different processors
23: **end if**
24: **end for**
25: return M

5.6.3. Objective function

The objective function is same as that for DAGs (Equation 10).

5.7. Heuristic based simplification technique

The MIQP formulation is NP-hard [31] and therefore an energy-gradient based heuristic is proposed to simplify the same. This is shown as a pseudo-code in Algorithm 1. The algorithm has two sections – generation of initial mapping (lines 3-6) and search of global minimum (lines 9-22).

The initial mapping section of the algorithm generates a starting mapping by assigning every task to processors for which the execution time is least. The second part of the algorithm searches for a global solution. There are *maxIter* iterations, where *maxIter* is an user defined parameter. For each iteration, a task is selected to be remapped to another processor to reduce the communication energy while satisfying the deadline requirement using the $RemapTask()$ routine (line 10). If such a task can be found, the mapping is updated (line 12). This process is continued until there is no task remapping possible without violating the deadline. When this happens, the energy consumption cannot be reduced further. If the current schedule is feasible (the completion time of the leaf tasks are less than the application deadline, D) and the mapping produces lower communication energy than the best mapping obtained so far, the best mapping is updated with the current mapping (line 18-21). However, this best mapping may not be the global optimum in terms of communication energy. In order to obtain a better solution, tasks in the current mapping are randomly reassigned to other processors (line 23) and the whole process of task remapping is then repeated. The *CommEnergy()* routine computes the communication energy of a mapping according to Equation 4. The $SGen()$ is the schedule generator engine. For DAGs, the $SGen()$ is same as $CPTO()$ algorithm of [12], whereas for SDFGs, this is same as the SDF^3 tool [32].

Algorithm 2 RemapTask(M,G_{arc}): Communication energy gradient based tasks remapping

Input: Mapping M and G_{arc}
Output: Determine a task to be remapped
1: $T_s = \emptyset$; $C_s = \emptyset$
2: $CE_r = CommEnergy(M, G_{arc})$; $S_r = SGen(M, G_{arc})$
3: $P_s = -\infty$
4: **for all** $v_i \in V_{app}$ **do**
5: **for all** $c_j \in V_{arc}$ **do**
6: $\hat{M} = M$
7: $\hat{M}(i) = c_j$
8: $S = SGen(\hat{M}, G_{arc})$
9: **if** $S \leq D$ **then**
10: $CE = CommEnergy(\hat{M}, G_{arc})$
11: Calculate P_r according to Equation 14
12: **if** $P_r > P_s$ && $P_r > 0$ **then**
13: $P_s = P_r$
14: $T_s = v_i$
15: $P_s = c_j$
16: **end if**
17: **end if**
18: **end for**
19: **end for**
20: return $[T_s\ P_s]$

The $RemapTask()$ routine selects a task and a processor where the selected task is to be remapped. Algorithm 2 provides the pseudo-code for the same. Every task of the application is moved to every processor and an energy-gradient based priority is calculated for the move. The one move which satisfies the application deadline and has the highest priority is selected and returned (line 20). Equation 14 is used to calculate the energy-gradient based priority.

$$P_r = \begin{cases} \frac{CE_r - CE}{S - S_r} & \text{if } S > S_r \\ (CE_r - CE) & \text{otherwise} \end{cases} \quad (14)$$

Here, two cases are considered. In the first case, if the makespan (maximum finish time of all leaf tasks) of the current move is higher than the original makespan, the energy gradient is considered to calculate the priority i.e. the moves with the largest reduction in communication energy with the least increase in makespan are assigned higher priorities. In the second case, if the makespan of the new mapping is less than the original makespan, higher priorities are assigned to moves producing larger reduction of communication energy consumption.

Algorithm 3 CMTM(): Communication and migration energy aware mapping

Input: Initial mapping M, G_{app}, G_{arc}, 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: $s_f = (\boldsymbol{c}_{i_1}, \boldsymbol{c}_{i_2}, \cdots, \boldsymbol{c}_{i_f})$
5: $s_{f-1} = (\boldsymbol{c}_{i_1}, \boldsymbol{c}_{i_2}, \cdots, \boldsymbol{c}_{i_{f-1}})$
6: starting map = $m_{f-1} = HashMap[s_{f-1}].getMap()$
7: $m_f = CMDSE(m_{f-1}, G_{app}, \{G_{arc} \setminus s_f\})$
8: $HashMap[s_f].setMap(m_f)$
9: **end for**
10: **end for**

6. Generate fault-tolerant mapping

The next step in the design methodology is to generate mappings for different processor fault-scenarios such that communication energy and migration overhead are jointly minimized.

6.1. Modeling migration overhead

Migration overhead associated with moving from one mapping to another is governed by two quantities – the state space of the task(s) participating in the migration process and the distance (hops) through which the state space is migrated. To better couple with the communication energy, the migration overhead is represented as energy and is termed as *migration energy*. Let $T(k)$ denotes the set of tasks mapped to processor c_k. The migration energy ($MigEnergy(k)$) incurred in moving the task(s) from processor c_k (equivalently, the tasks from the set $T(k)$) is given by

$$MigEnergy(k) = \sum_{\forall v_i \in T(k)} S_i \times E_{bit}(k, k_i) \tag{15}$$

where c_{k_i} is the new location (processor) for $v_i \in T(k)$.

6.2. CMTM methodology

Communication and migration energy minimum fault-tolerant mappings are generated using Algorithm 3. 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 $S^f = \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 0,2\rangle, \langle 2,0\rangle, \langle 1,2\rangle, \langle 2,1\rangle\}$[4]. 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 needs 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 currently 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 (i.e. output of Algorithm 1). Once the starting mapping is determined, the next step is to generate a task mapping for the fault-scenario which minimizes the communication energy and incurs minimum overhead in migrating from the starting mapping. This is realized in the *CMDSE()* routine which takes the starting mapping, the application graph and the architecture graph excluding the faulty processors. The pseudo-code for *CMDSE()* is similar to Algorithm 1 with two differences.

- The routine *FTRemapTask* is called internally instead of *RemapTask* (line 10).
- Inputs a starting mapping which is used only in the *FTRemapTask* routine.

The pseudo-code for the *FTRemapTask* routine is shown in Algorithm 4. Each tasks of the application is moved to each processor and the communication energy of the move is computed in a similar manner. The migration overhead for the new

Algorithm 4 FTRemapTask(M_s,M,G_{arc}): Communication and migration energy based tasks remapping

Input: Mapping M, M_s and G_{arc}
Output: Determine a task to be remapped

1: $T_s = \emptyset$; $C_s = \emptyset$
2: $CME_r = \infty$
3: **for all** $v_i \in V_{app}$ **do**
4: **for all** $c_j \in V_{arc}$ **do**
5: $\hat{M} = M$
6: $\hat{M}(i) = c_j$
7: $S = SGen(\hat{M}, G_{arc})$
8: **if** $S \leq D$ **then**
9: $CE = CommEnergy(\hat{M}, G_{arc})$
10: $ME = solveILP(M_s, \hat{M}, G_{arc})$
11: $CME = CE + ME$
12: **if** $CME < CME_r$ **then**
13: $CME_r = CME$
14: $T_s = v_i$
15: $P_s = c_j$
16: **end if**
17: **end if**
18: **end for**
19: **end for**
20: return $[T_s\ P_s]$

[4] 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.

Table 3: Energy Table

$\mathbf{EM(o_i, n_j)}$	$\mathbf{n_1}$	$\mathbf{n_2}$	$\mathbf{n_3}$	$\cdots$	$\mathbf{n_k}$
o_1	50	205	180	$\cdots$	175
o_2	200	100	180	$\cdots$	200
o_3	200	175	130	$\cdots$	125
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
o_l	165	110	120	$\cdots$	135

mapping is computed using integer linear programming. This is discussed in Subsection 6.3. The two energy components (communication and migration) are combined and the task-processor combination with minimum total energy is returned.

6.3. ILP-based task migration energy computation

The minimum overhead in migrating from one mapping to another is formulated as a binary integer linear programming.
Given:

- Application graph G_{app}
- Architecture graph $\hat{G}_{arc}$ (note that $\hat{G}_{arc}$ is G_{arc} without the faulty processors)
- Starting mapping M_s
- New mapping M_n

Objective:
Minimize migration overhead in moving from M_s to M_n.
Simplification:
In order to simplify this objective, an energy matrix (Table 3) is formed with processors from mapping M_s forming the rows (indicated by o_i) and the processors from M_n forming the columns (indicated by n_j). The rows (and columns) corresponding to the faulty processors(s) are filled with zeroes. The non-zero entries (o_i, n_j) of the energy matrix (referred hereafter as EM) correspond to the migration energy associated with the extra tasks(s) on tile n_j of M_n which is (are) not present on processor o_i of M_s. This is computed according to Equation 15.
ILP Formulation:
Binary Variables: $X_{ij},\ i \in [1,k],\ j \in [1,k-1]$
Objective: Minimize z = $\sum_{ij} X_{ij} \times EM(o_i, n_j)$
Constraints:
One element from each row and column is to be selected

$$\sum_{j=1}^{k-1} X_{ij} = 1,\ \forall i \in [1,k],\ \sum_{i=1}^{k} X_{ij} = 1,\ \forall j \in [1,k-1] \qquad (16)$$

7. Results and discussions

Experiments are conducted on synthetic and real application graphs on Intel Xeon 2.4 GHz server running Linux. Fifty synthetic applications are generated with number of tasks in each application selected randomly from the range 8 to 32. Additionally, 10 real applications are considered with 5 from streaming and the remaining 5 from non-streaming domain. The streaming applications are obtained from the benchmarks provided in the SDF^3 tool [32]. These applications are executed on MPSoC architectures consisting of 4 to 16 processors of three different types ($h = 3$) arranged in a mesh-based topology.

All algorithms developed in this work are coded in C++ and Matlab. As established in Section 2, there are three categories of research related to this work – throughput maximization, energy minimization and migration overhead minimization. The results of this work are compared with the representative of each of these categories i.e. the throughput maximization technique of [21] (referred as *TMax*, the migration overhead minimization technique of [20] (referred as *OMin*), the energy minimization technique of [11] (referred as *EMin*) and the joint throughput constrained migration overhead minimization technique of [22] (referred as TConOMin). Although *EMin* does not address fault-tolerance, results are compared with it to determine how far the proposed approach is from the minimum energy possible with application on the given architecture.

The rest of this section is organized as follows. Subsection 7.1 provides the time complexity, reduction in design space exploration time and the communication energy performance of the proposed *CDSE* algorithm. Subsection 7.2 provides the time complexity, space complexity, execution time, migration overhead and communication energy performance of the proposed *CMTM()* algorithm. Finally, to signify the importance of this works for streaming applications, throughput performances of the proposed algorithms are presented in Subsection 7.3.

7.1. Performance of the CDSE

7.1.1. Complexity analysis

The complexity of Algorithm 1 is governed by two factors – user defined parameter *maxIter* and the routine *RemapTask*(). Processor assignments for tasks can be accomplished in constant time. The complexity of lines 3-6 is therefore $O(n_{app})$. Assuming the *RemapTask*() routine to be executed η times on average for each value of *numIter*, the overall complexity of Algorithm 1 is given by Equation 17.

$$O(C_1) = n_{app} + maxIter * \eta * O(RemapTask) \qquad (17)$$

The *RemapTask*() routine remaps each task on each functional processor to determine if the makespan of the new mapping is less than the application deadline and the priority is higher than the highest priority obtained so far. If the processor assignment operation takes unit time and the complexity of the schedule generation engine (*SGen*) is denoted by $O(SGen)$, the overall complexity of Algorithm 2 is given by Equation 18.

$$O(RemapTask) = O(C_2) = n_{app} * n_{arc} * O(SGen) \qquad (18)$$

Combining the two equations, the complexity of Algorithm 1 is given by Equation 19.

$$\begin{aligned} O(C_1) &= n_{app} + maxIter * \eta * n_{app} * n_{arc} * O(SGen) \\ &= maxIter * \eta * n_{app} * n_{arc} * O(SGen) \end{aligned} \qquad (19)$$

The worst-case complexity of $SGen$ ($= CPTO$) is

$$O(SGen) = O(n_{app}(\log n_{app} + n_{succ})) \qquad (20)$$

where n_{succ} is the average number of successors for a task.

Table 4: Execution time (in secs) of existing and proposed technique

Tasks	Heterogeneous architecture (2 types)							
	2×2		3×3		4×4		5×5	
	TMax	Proposed	*TMax*	Proposed	*TMax*	Proposed	*TMax*	Proposed
8	30	24.5	6.8×10^3	55.1	2.5×10^5	98.3	6.1×10^6	166.1
16	1.3×10^5	50.1	–	113.8	–	242.3	–	401.3
32	–	71.8	–	167.4	–	312.8	–	526.6
64	–	282.9	–	628.6	–	767.6	–	1322.4
96	–	303.2	–	721.2	–	1370.8	–	2410.6

7.1.2. Design space exploration time

Table 4 reports the execution time of the proposed Algorithm 1 in comparison with *TMax*. The execution time of other existing fault-tolerant techniques (*OMin* and *TConOMin*) are similar to that of *TMax* and are not included in the table.

The time reported for the exhaustive search based existing techniques includes three components –

- time to generate the mappings
- time to compute communication energy of these mappings
- time to search for the minimum energy mappings (sorting)

The time reported for the proposed technique is the execution time of Algorithm 1 with the design parameter *maxIter* set to 100. As can be seen from the table, the execution time of the proposed technique is comparable to that of the existing techniques for small problem sizes (8 tasks mapped on 4 processors) because of the fewer number of exhaustive mappings. However as the number of tasks and/or processors increases, there is an exponential growth in the number of mappings (task-processor combinations). The existing techniques ([17][21][22]) fail to provide a solution beyond 16 tasks mapped on 9 processors. The proposed technique can provide results within satisfactory time even for 96 tasks mapped on 25 processors with heterogeneity of 2. On average for different task-processor combinations (those for which the existing techniques are able to solve), the proposed technique reduces execution time by 99%.

7.1.3. Communication energy performance

Figure 7 plots the communication energy of the proposed technique in comparison with the communication energy achieved using the *EMin* technique to determine the variations of the heuristic from the energy optimality. Additionally, results are compared with *TMax* ([21]), *OMin* ([20]) and *TConOMin* ([22]) to signify the fact that the existing techniques when applied to generate the starting mapping of an application on an architecture can lead to sub-optimal results in terms of communication energy. Experiments are conducted on 50 synthetic application task graphs with number of tasks varying from 8 to 24 on an MPSoC architecture with 9 processors. The number of tasks is limited to 24 as the existing exhaustive search based *reactive* fault-tolerant techniques fail to provide a solution beyond 24 processors. The heterogeneity of the processors

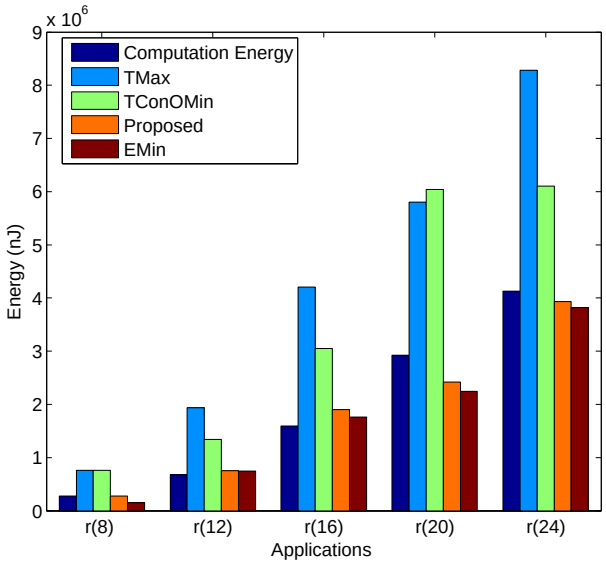

Figure 7: Communication energy performance of the proposed *CDSE()*

is fixed at two. For clarity of representation, results for 5 applications are plotted and the number of tasks in these applications is represented in the name of the application.

There are a few trends to follow from this figure. First, the computation energy for most applications constitutes $\approx 40\%$ of the total energy consumption. Second, the proposed *CDSE()* minimizes the communication energy significantly. On average for all 50 applications considered (including those not shown in the figure), the proposed technique achieves a communication energy savings of 60% and 50% with respect to *TMax* and *TConOMin* respectively. Although not explicitly shown here, these savings constitute 31% and 22.5% respectively of the total application energy. Further, the energy obtained using the proposed *CDSE()* is only within 10% of the minimum communication energy achieved using *EMin*.

An important parameter of the *CDSE()* algorithm is the user specified *maxIter*. This determines the execution time of the algorithm and the solution quality. Figure 8 plots the normalized energy of the *CDSE()* for five synthetic applications as the value of *maxIter* is varied from 10 to 100. The communication energy values obtained using the *CDSE()* for different values of *maxIter* are normalized with respect to the minimum communication energy obtained by solving the *MIQP* problem in

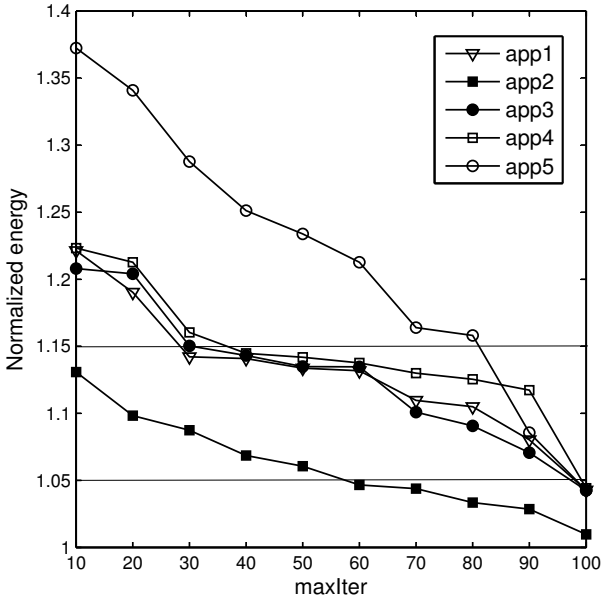

Figure 8: Solution quality and execution time trade-off for different values of *maxIter*

Equation 10 using standard MIQP solvers e.g. *CPLEX* [33].

As can be seen from Figure 8, for all five applications, the variation of the communication energy from the minimum energy point decreases with an increase in the value of *maxIter*. This is due to the increase in the search effort. A point to note here is that the value of *maxIter* is determined by the maximum energy variation allowed for the given set of applications. For all five applications considered, the proposed *CDSE()* achieves a maximum of 5% variation from the global optimum point for *maxIter* = 100. Setting *maxIter* equal to 80 results in 15% variation from the optimum point.

7.2. *Performance of* CMTM

7.2.1. *Time complexity*

The complexity of Algorithm 3 is determined as follows. The number of iterations of the algorithm is determined by the number of fault scenarios with F faults. This is given by

$$n_{FS} = \sum_{f=1}^{F} {}^{n_{arc}}P_f \tag{21}$$

At each iteration, the $CMDSE()$ algorithm is invoked. The overall complexity of Algorithm 3 is given by Equation 22 where $O(CMDSE)$ is the complexity of $CMDSE()$.

$$O(C_3) = O(n_{FS} \times O(CMDSE)) \tag{22}$$

The *CMDSE()* is similar to Algorithm 1 with the exception of *FTRemapTask*. The complexity is given by Equation 23.

$$O(CMDSE) = n_{app} + maxIter * \eta * O(FTRemapTask) \tag{23}$$

The complexity of *FTRemapTask()* is determined by two quantities – *SGen* and *solveILP*. The overall complexity is computed similar to Equation 18 as given by Equation 24.

Table 5: Execution time (in sec) of finding minimum migration overhead

Processors	Brute Force	*TMax*	ILP Solver
8	0.5	0.065	0.0178
12	8.13	0.1579	0.0402
16	–	–	0.0707
32	–	–	0.2388

Table 6: Storage requirement with increasing processors for a 3-fault-tolerant system with 100 tasks

Processors	Number of fault-scenarios	Bits per mapping	Storage (KB)
8	112	300	32.8
16	480	400	187.5
24	1104	500	494.3
32	1984	500	968.7

$$O(C_4) = n_{app} * n_{arc} * (O(SGen) + O(solveILP)) \tag{24}$$

The ILP proposed in this paper is solved using *Matlab* optimization toolbox. Table 5 compares the execution time of the ILP solver against the brute force technique of finding the minimum migration overhead and the dynamic programming based approach of *TMax*. As can be seen from the table, the brute-force and the *TMax* techniques fail beyond 12 tiles due to the high memory requirement. The ILP approach continues to provide an optimal solution even for 32 tasks. Moreover, the computation time of the ILP is 4 times lower than the *TMax* technique.

7.2.2. *Storage complexity*

The storage associated with a 3-fault-tolerant system with 100 tasks is summarized in Table 6. The number of fault-scenarios (column 2) corresponding to a tile count is computed using Equation 21. The number of bits required to store a mapping m_i is given by $bits = \log_2 mID_i$, where mID_i is computed using Equation 1. Finally, the total storage is obtained by multiplying the bits per mapping with the total number of mappings.

7.2.3. *Algorithm execution time*

Table 7 reports the execution time of the proposed technique in comparison with the *TMax* technique of [21] as the number of tasks is varied on an architecture with 4 processors with heterogeneity of 2. The execution time of the other existing techniques such as *OMin* and *TConOMin* are similar to the *TMax* and therefore not shown in the table. All execution times reported in the table are average of single and double fault-scenarios of five different applications. As an example, the time for 16 tasks is the average of 5 synthetic applications each with 16 tasks considering all single and double-fault scenarios possible with four processors.

The execution-time for *TMax* includes the time for following

Table 7: Execution time (in sec) performance of the proposed technique

Tasks	*TMax*			Proposed		
	Start Map	FT Map	Total	$CDSE()$	$CMTM()$	Total
8	30	13	43	24.5	1,015	1,039
16	1.3×10^5	6×10^2	1.3×10^5	50.1	2,534	2,584
20	9.0×10^6	5.0×10^3	9.0×10^6	56.5	3,170	3,226
24	1.8×10^8	1.2×10^5	1.8×10^8	61.8	4,055	4,117

- Exhaustive search for generating the starting mapping (refer column 2 of Table 4)
- Dynamic programming based technique to generate mappings for the fault-scenarios (refer [21])

The execution time of the proposed technique includes the time for the following

- $CDSE()$ to generate the minimum communication energy starting mapping
- $CMTM()$ to generate the fault-tolerant mappings for different fault-scenarios

As can be seen from the table (and also established previously) the bottleneck of the existing techniques is the exhaustive search based starting mapping generation step (column 2). Another point to note from this table is that the execution time of the fault-tolerant mapping generation step of the existing techniques (column 3) grows exponentially with the number of tasks (and/or processors). Finally, the execution time of the proposed $CMTM$ technique grows linearly with the number of tasks (refer Equation 22). The proposed $CMTM$ technique achieves 30x reductions in execution time with 24 tasks mapped on 4 processors. This execution time savings increases as the number of tasks and/or number of processors are scaled further.

7.2.4. Communication Energy performance

Figure 9 plots the average communication energy of the proposed *CMTM* for single and double faults of 3 synthetic applications and 3 real-life applications on an architecture with 12 processors. The synthetic applications are represented by *App_j*, where *j* is the number of tasks. The proposed technique is compared with *TMax* and *EMin* approaches. Further, to signify the potential energy savings possible, *TMax* technique is split into two categories – one resulting in maximum communication energy (*TMax_EMax*) and one resulting in minimum communication energy (*TMax_EMin*). The communication energy of all three techniques is normalized with respect to the minimum communication energy for an application obtained using *EMin*. The percentage change of *TMax_EMin* and *CMTM* with respect to *TMax_EMax* and *TMax_EMin* respectively are indicated on the bars. Although not explicitly captured in the figure, the communication energy for the applications constitute on average 55% of the total energy.

As can be seen from the figure, considering communication energy in the fault-tolerant mapping generation using *TMax* can result in significant communication energy savings (second and third bar for each application). On average for all applications considered (including those not shown in the figure), communication energy aware throughput maximization technique results in 40% savings in communication energy. This is close to 18% savings of the total energy of an application (i.e. the sum of computation and communication energy). The proposed $CMTM$ technique minimizes this further (refer third and fourth bar for each application). For some applications such as *App_24* and *VOPD*, the throughput maximum mapping and the communication energy minimum mapping results in comparable data communication over the networks-on-chip. For these applications, the energy saving using $CMTM$ are less (5.0% and 13.9% respectively). On the other end for application such as *MPEG* the two mappings are significantly different and therefore $CMTM$ is able to achieve the minimum energy resulting in 70% reduction in communication energy. On average for all the applications considered, $CMTM$ minimizes communication energy by 35% as compared to *TMax_EMin*.

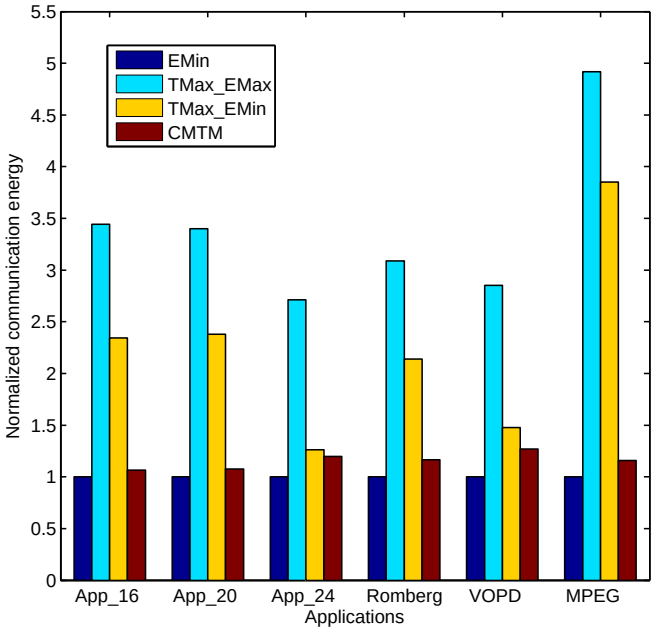

Figure 9: Communication energy performance of *CMTM*

Finally, $CMTM$ achieves an average deviation of 15% from the minimum energy mappings obtained using *EMin* signifying that the heuristic ($CMTM$) is able to achieve satisfactory result quality with significant reduction in execution time.

7.2.5. Migration overhead performance

Migration overhead is incurred once fault occurs. This is a onetime overhead for permanent faults. However, for frequently occurring intermittent faults, this can lead to significant energy penalty as established in this section.

Table 8 reports the migration overhead (measured as energy) and communication energy of the existing techniques (*OMin* and *TMax*) in comparison with the proposed *CMTM* technique for two different applications (*Romberg Integration* and *VOPD*) with 10 and 12 tasks respectively on an MPSoC with 6 processors arranged in 2×3. The core heterogeneity is assumed to be 2. Similar to previous section, the existing techniques can be

Table 8: Migration overhead performance

		Migration Energy (nJ)	Communication Energy (nJ)	Migration Overhead Savings (nJ)	Extra Energy Per Iteration (nJ)	Iterations to recover
Romberg Integration	*OMin_EMin*	1.1×10^9	2.1×10^5	2.0×10^8	3.0×10^4	6,667
	TMax_EMin	1.2×10^9	3.2×10^5	1.0×10^8	1.4×10^5	7,143
	CMTM	1.3×10^9	1.8×10^5	–	–	–
VOPD	*OMin_EMin*	2.8×10^9	4.3×10^5	5.0×10^8	2.0×10^4	25,000
	TMax_EMin	3.0×10^9	4.7×10^5	3.0×10^8	6.0×10^4	5,000
	CMTM	3.3×10^9	4.1×10^5	–	–	–

split into two categories – one resulting in maximum communication energy and the other resulting in minimum communication energy. Results for the minimum communication energy are only included for comparison. Columns 3 and 4 report the migration overhead incurred when faults occur and the average communication 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_EMin* and *TMax_EMin* with respect to the proposed *CMTM*. Column 6 reports the extra communication energy incurred in selecting the same two techniques with respect to *CMTM*.

Considering permanent faults: As can be seen from the table, significant savings in migration overhead are possible with *OMin_EMin* technique. However, this technique is associated with energy penalty (column 6). For application *Romberg Integration* for example, the migration overhead savings in *OMin_EMin* is $2 \times 10^8 nJ$ while the energy penalty is $3 \times 10^4 nJ$ per iteration. As established previously, migration is one time overhead for permanent faults while communication energy is consumed both pre- and post-fault occurrence. The savings in migration overhead is compensated in $\frac{2\times10^8}{3\times10^4} = 6667$ iterations. This is shown in column 7 of the table. Interpreting this in reverse manner, selecting *CMTM* as the fault-tolerant technique results in an extra migration overhead of $2 \times 10^8 nJ$ which is amortized in the following (post-fault) 6667 iterations of the application graph. Typically, applications mapped on a multiprocessor system 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 6667 iterations will be used to recover the migration overhead loss while the remaining $(N - 6667)$ iterations will fetch energy savings ($3 \times 10^4 nJ$ per iteration). As $N \rightarrow \infty$, the energy savings obtained $= (N - 6667) \times 3 \times 10^4 \approx N \times 3 \times 10^4 nJ$. This substantial energy gain clearly justifies the non-consideration of migration overhead in the permanent fault-tolerant mapping selection.

Considering intermittent faults: For this class of defects, the migration energy is incurred every time fault occurs. Figure 10 plots the normalized migration overhead performance of the proposed and the existing techniques for application *MPEG* as the fault-rate is varied. The migration overhead values are normalized with respect to that obtained from *OMin_EMin* technique. As can be seen from the figure, the proposed

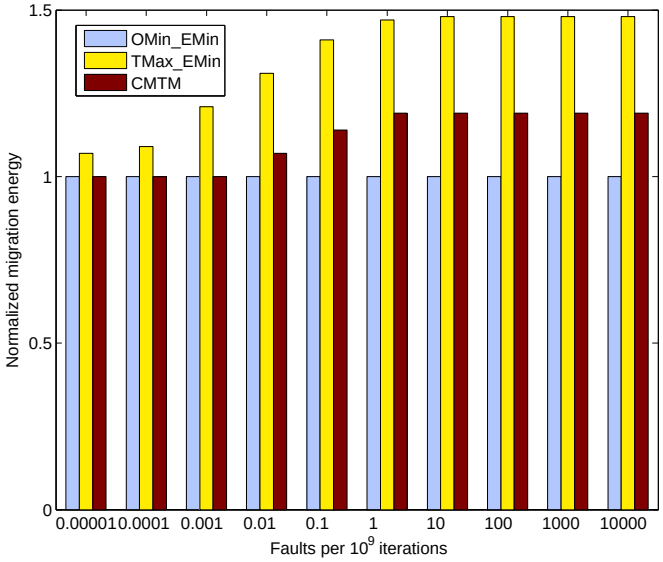

Figure 10: Migration energy performance of *CMTM*

CMTM technique minimizes the migration overhead significantly achieving 20% lower migration overhead as compared to the existing *TMax_EMin*. Further, the proposed technique incurs an average 12% variation from the minimum migration overhead technique (*OMin_EMin*) for all the applications.

7.3. Throughput performance of CMTM

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. Figure 11 (a) plots the throughput performance of proposed technique for six streaming applications – three synthetic (indicated by $s(j)$, where j is the number of actors of the application) and three real-life (*H263 Encoder*, *MP3 Decoder* and *MPEG Decoder*) applications. These applications are executed on a platform with 6 processors and the results are average of all single and double fault-scenarios. The throughput obtained using the proposed technique for an application is normalized with respect to the highest throughput obtained for the application using *TMax* technique. The throughput obtained using *OMin* and *EMin* are also included for comparison. The throughput constraint for the applications is indicated using the dashed line. As can be seen from

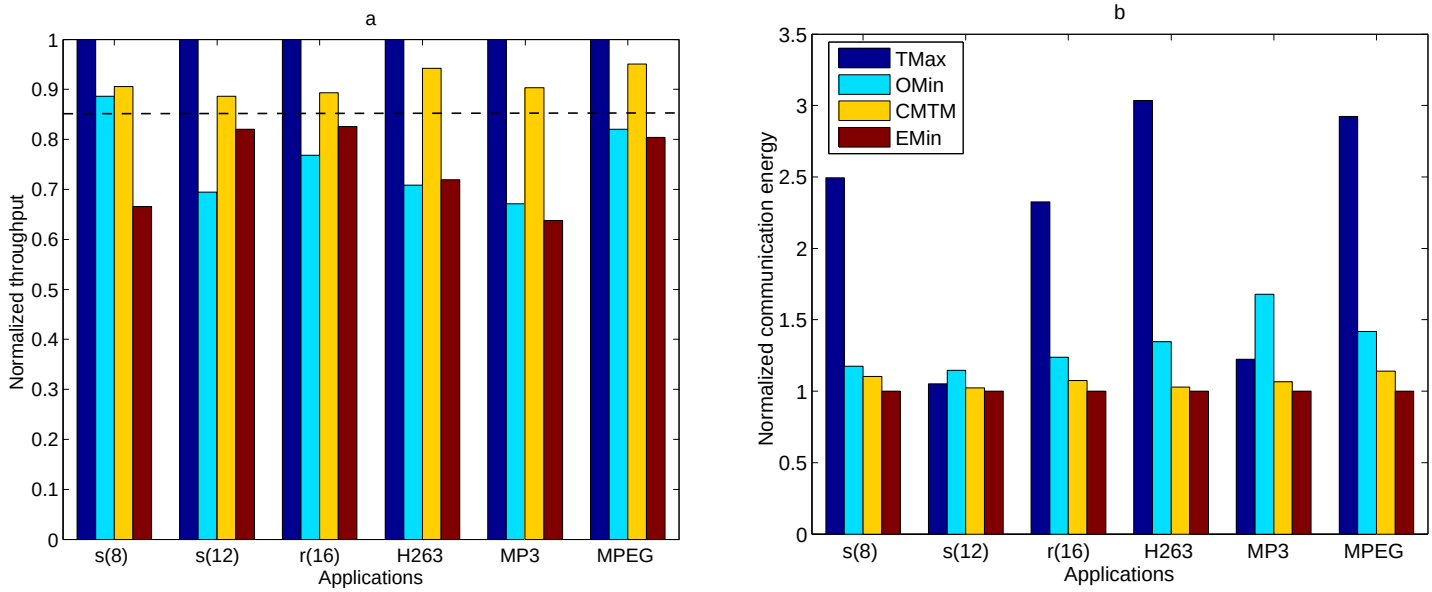

Figure 11: Throughput performance of *CMTM*

the figure and expected, the throughput constraint is violated for most applications for *OMin* and *EMin* techniques. This is due to the non-consideration of throughput degradation in the fault-tolerant mappings generation process. The proposed technique satisfies the throughput requirement for all applications as throughput degradation is explicitly considered in the flow (in the SDF^3 tool). This result signifies the importance of the proposed technique for throughput constrained streaming applications. Finally, the communication energy of the same applications achieved using the proposed and the existing techniques are plotted in Figure 11 (b). Although not shown explicitly, the proposed technique delivers an average 60% and 45% better throughput per unit energy as compared to *TMax* and *EMax* techniques respectively, clearly demonstrating the advantage of the proposed technique for scalable throughput applications.

8. Conclusions

This paper presents a communication and migration overhead aware offline analysis technique to generate fault-tolerant mappings for applications mapped on heterogeneous multiprocessor systems. Two heuristics are proposed to reduce the design space exploration time. Experiments conducted with synthetic and real application graphs demonstrate that the proposed technique is able to minimize the communication energy by 35% and the migration overhead by 20% as compared to the existing fault-tolerant techniques. The analysis time is reduced by 30x with a maximum deviation of 15% from the energy optimal solution. These results signify the adaptability of the proposed technique for large scale problems. Finally, the proposed technique is also shown to deliver an average 50% better throughput per unit energy for streaming multimedia applications. Consideration of task computation energy and minimization of mapping storage overhead are left as future works.

Acknowledgment

This work was supported by Singapore Ministry of Education Academic Research Fund Tier 1 with grant number R-263-000-655-133.

References

[1] W. Wolf, A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 10, pp. 1701 –1713, 2008.

[2] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[3] D. A. Reed, C. da Lu, and C. L. Mendes, "Reliability challenges in large systems," *Elsevier Future Generation Computer Systems (FGCS)*, vol. 22, no. 3, pp. 293 – 302, 2006.

[4] I. Koren and C. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2007.

[5] W. Wolf, "Hardware-software co-design of embedded systems [and prolog]," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967 –989, 1994.

[6] O. Derin, D. Kabakci, and L. Fiorin, "Online task remapping strategies for fault-tolerant Network-on-Chip multiprocessors," in *IEEE/ACM Symposium on Networks on Chip (NoCS)*, 2011.

[7] Y. Zhang, Z. Hao, X. Xu, W. Zhao, and Z. Wang, "Workload-balancing schedule with adaptive architecture of MPSoCs for fault tolerance," in *IEEE Conference on Biomedical Engineering and Informatics (BMEI)*, 2010.

[8] A. Das, A. Kumar, and B. Veeravalli, "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)*, 2013.

[9] K. Popovici, X. Guerin, F. Rousseau, P. Paolucci, A. Jerraya *et al.*, "Platform-based software design flow for heterogeneous MPSoC," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 4, pp. 1–23, 2008.

[10] N. Jha, "Low power system scheduling and synthesis," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2001.

[11] J. Hu and R. Marculescu, "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)*, 2004.

[12] L. Goh, B. Veeravalli, and S. Viswanathan, "Design of fast and efficient energy-aware gradient-based scheduling algorithms heterogeneous embedded multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 20, no. 1, pp. 1–12, 2009.

[13] A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, "Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms," *Elsevier Journal of Systems Architecture (JSA)*, vol. 56, no. 7, pp. 242–255, 2010.

[14] T. Wei, X. Chen, and S. Hu, "Reliability-Driven Energy-Efficient Task Scheduling for Multiprocessor Real-Time Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 30, no. 10, pp. 1569–1573, 2011.

[15] H. Aydin and D. Zhu, "Reliability-aware energy management for periodic real-time tasks," *IEEE Transactions on Computers (TC)*, vol. 58, no. 10, pp. 1382 –1397, oct. 2009.

[16] L. Huang and Q. Xu, "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)*, 2010.

[17] A. Das, A. Kumar, and B. Veeravalli, "Energy-aware communication and remapping of tasks for reliable multimedia multiprocessor systems," in *International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.

[18] ——, "Communication and Migration Energy Aware Design Space Exploration for Multicore Systems with Intermittent Faults," in *IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2013.

[19] E.-G. Talbi, Z. Hafidi, D. Kebbal, and J.-M. Geib, "A fault-tolerant parallel heuristic for assignment problems," *Elsevier Future Generation Computer Systems (FGCS)*, vol. 14, no. 56, pp. 425 – 438, 1998.

[20] C. Yang and A. Orailoglu, "Predictable execution adaptivity through embedding dynamic reconfigurability into static MPSoC schedules," in *IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (ISSS+CODES)*, 2007.

[21] C. Lee, H. Kim, H. Park, S. Kim, H. Oh, and S. Ha, "A Task Remapping Technique for Reliable Multi-core Embedded Systems," in *IEEE/ACM/IFIP Conference on Hardware/Software Codesign and System Synthesis (ISSS+CODES)*, 2010.

[22] A. Das and A. Kumar, "Fault-Aware Task Re-Mapping for Throughput Constrained Multimedia Applications on NoC-based MPSoC," in *IEEE Symposium on Rapid System Prototyping (RSP)*, 2012.

[23] T. Park, N. Woo, and H. Y. Yeom, "An efficient recovery scheme for fault-tolerant mobile computing systems," *Elsevier Future Generation Computer Systems (FGCS)*, vol. 19, no. 1, pp. 37 – 53, 2003.

[24] J. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[25] M. Gary and D. Johnson, "Computers and intractability: A guide to the theory of np-completeness," 1979.

[26] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011.

[27] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[28] T. Ye, L. Benini, and G. De Micheli, "Packetized on-chip interconnect communication analysis for MPSoC," in *IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2003.

[29] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[30] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 13, no. 3, pp. 260–274, 2002.

[31] S. Sahni and T. Gonzalez, "P-complete approximation problems," *Journal of the ACM (JACM)*, vol. 23, no. 3, pp. 555–565, Jul. 1976.

[32] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *IEEE Conference on Application of Concurrency to System Design (ACSD)*, 2006. [Online]. Available: http://www.es.ele.tue.nl/sdf3.

[33] I. CPLEX. (2012) IBM ILOG Optimization Products. [Online]. Available: www-01.ibm.com/software/websphere/products/optimization/.