Fault-Aware Task Re-Mapping for Throughput Constrained Multimedia Applications on NoC-based MPSoCs

Anup Das and Akash Kumar
Department of Electrical & Computer Engineering
National University of Singapore, Singapore
{akdas, akash}@nus.edu.sg

***Abstract*—Shrinking transistor geometry and aggressive voltage scaling are leading to growing concerns on the reliability of multiprocessor systems. Majority of streaming multimedia applications are characterized by fixed throughput requirements; violation of which directly impacts user experience. None of the prior research considers joint treatment of throughput and task-migration overhead, both of which are essential for fault-tolerance of throughput-constrained multimedia multiprocessor systems. In this paper, we propose to remap tasks from faulty processors with the objective of minimizing the migration overhead while satisfying throughput constraints. The proposed technique is based on extensive design-time analysis of different fault scenarios to determine optimal mappings from the throughput-migration overhead Pareto space. These mappings are stored in a table and are looked-up at run-time to migrate tasks as and when faults occur. Applications are modeled using Synchronous Data Flow graphs (SDFG) to consider cyclic dependencies of tasks, typically found in multimedia systems. Experiments performed with synthetic and real application graphs demonstrate that the migration overhead can be reduced by 26% on average while still meeting throughput constraints. Moreover, by selecting an appropriate initial processor-task mapping, migration overhead can be further reduced by 15% on average.**

***Index Terms*—Fault-Tolerance; Task Remapping; Synchronous Data Flow Graph; Linear Programming.**

I. Introduction

To accommodate ever increasing demand of applications and to ease the scalability, multiprocessor systems-on-chip (MPSoCs) are becoming the obvious design choice in current and future technologies with streaming multimedia applications constituting a large fraction of the application space [1] [2]. With reducing feature size and increasing transistor count, modern systems are becoming susceptible to both transient and permanent faults [3]. This research focuses on permanent fault-tolerance techniques in MPSoCs.

Permanent faults are traditionally tackled using hardware redundancy [4]. However, stringent area and power budgets are prohibiting the use of hardware redundancy in today's systems. Software techniques such as task-migration are therefore gaining popularity among research community [4]–[14]. Task migration involves migrating tasks from faulty core(s) to other functional core(s). Many of the task-migration research works have focused on minimizing migration overhead and load balancing [8] [11]. However, they provide no guarantee on the application throughput. There is a study to maximize application throughput under faulty scenarios, but migration overhead is not taken into account in the maximization process and it can therefore increase significantly in this technique [14].

Most streaming multimedia applications, such as H.263 decoder, demand fixed throughput which is manifested as the quality-of-service (QoS) requirement. These applications do not benefit from a higher throughput than required and can even increase buffer requirements at a higher throughput. The task-migration objective for this class of applications is to minimize the migration overhead satisfying the application throughput requirement. Additionally, tasks of most multimedia applications exhibit cyclic dependency which has not been considered in any of the prior research on fault-tolerance.

This paper focuses on software techniques for tolerating permanent faults in homogeneous multiprocessor systems with applications modeled using Synchronous Data Flow Graphs (SDFGs) [15]. The key contributions of this paper are the following.

- Minimization of migration overhead for throughput-constrained streaming multimedia applications for multiple faults.
- Joint consideration of migration overhead and throughput degradation for applications with scalable QoS.
- Use of Synchronous Data Flow Graphs to model streaming multimedia applications.
- An initial processor-task mapping aiming at minimizing migration overhead for all single-fault scenarios.

The technique proposed in this paper performs design-time analysis with all optimal points (including the lowest throughput point which meets the requirement) on the throughput-migration overhead Pareto space. The minimum migration overhead mappings are retained and stored in a table for use at run-time. Experiments conducted with synthetic and real application graphs show that the proposed technique results in 26% reduction of migration overhead while satisfying the throughput requirement. Additionally, with proper choice of an initial mapping, migration overhead can be further reduced by 15%. Moreover, the proposed technique can also minimize throughput degradation and migration overhead jointly.

The rest of the paper is organized as follows. A brief overview of the prior art is provided in Section II which is then followed by a motivating example to emphasize the importance of this work in Section III. Then the task-mapping problem is defined in Section IV and the solution approach is defined in V. Next, Section VI provides complexity analysis of the proposed algorithms and simulation results and finally Section VII concludes the paper with future directions.

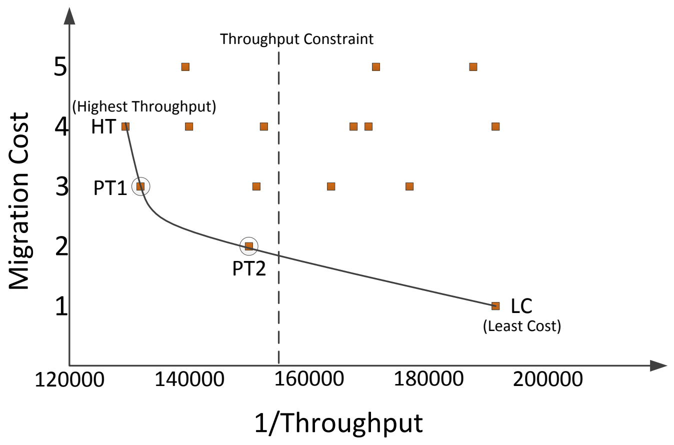

Fig. 1. Throughput-cost variation of different mappings with 5 tiles

II. Related Work

Fault-tolerance is emerging as one of the most desirable features in modern day MPSoCs. The existing fault-tolerant research can be broadly classified into two categories – application level and architecture level. One of the popular architecture level fault-tolerance techniques is to use redundancy-based design [4] [5]. In such a system, critical design components are replicated and results are voted to produce the output. However, replication techniques like DMR, TMR etc. come with high area penalty. Stringent cost budget is increasingly prohibiting the use of redundancy-based designs for MPSoCs.

A cost-effective solution for fault-tolerance involves migration of tasks from faulty cores. Task mapping/scheduling decisions can be pre-computed at design-time or can be done at run-time. Accordingly, task mapping can be categorized as static, dynamic and quasi-static. Static task mapping involves analysis at design-time to maximize system reliability [6] [7] but does not address task migration.

Dynamic approaches monitor system-status and decide on task migration at run-time to minimize migration overhead [8] [9] [10] or balance processor load [11]. However, throughput is not always guaranteed in these techniques. Moreover, migration algorithms need to be simple to minimize computation.

Quasi-static task migration techniques compute task mapping decisions at design-time for different fault-scenarios [12] [13] [14]. As faults occur, these mappings are looked up at run-time to carry out task-migration. The advantage of this technique is that any sophisticated algorithm can be used at design-time despite the associated mapping storage overhead. The research objective of this paper falls in quasi-static class. Hence, references [12] [13] [14] are discussed in detail.

A fixed order Band and Band reconfiguration technique is studied in [12]. Cores of target architecture are partitioned into two bands. When one or more cores become faulty, tasks on these core(s) are migrated to other functional core(s) determined by the band in which the 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 [13]. 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, schedule length can become unbounded for high fault-tolerance systems. Moreover, analysis is based on task graphs and therefore cannot be applied to streaming applications with cyclic task dependencies.

Task remapping technique based on offline computation and virtual mapping is proposed in [14]. Here, task mapping is performed in two steps – determining the highest throughput mapping followed by generation of a virtual mapping to minimize the cost of task migration to achieve this highest throughput mapping. These virtual mappings are computed at design-time based on different fault scenarios. A limitation of this technique is that the migration overhead significantly increases as this is not considered in the initial optimization process. Moreover, throughput constrained streaming applications do not benefit from a throughput higher than required and can increase buffer requirements at output.

III. Motivation

A. *Throughput-Migration Overhead Tradeoff*

Streaming multimedia applications can be broadly classified into two categories – applications, those benefitting 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 importance of migration overhead for multimedia applications [16] enforces the simultaneous consideration of throughput and migration overhead for fault-tolerance analysis of this class of applications.

As faults occur, tasks from faulty core(s) need to be migrated to other functional core(s). Least migration overhead is achieved by remapping only the tasks from faulty core(s) and keeping all other task mappings unchanged. This may degrade throughput below an acceptable limit. Remapping all tasks may result in the highest throughput but can impose significant migration overhead. Trade-off analysis can be performed by considering other (throughput-cost) points by selectively allowing more task remapping. Figure 1 plots the throughput-cost variations of different mappings of an experiment with 6 cores and 8 tasks with one of the cores as faulty. Most of the dynamic task migration policies [8] use the least cost point (LC) of the figure. Authors in [14] use the highest throughput point (HT). But when throughput is a constraint (as shown by the dashed line), there are three optimal points (HT, PT1, PT2) to be considered in the throughput-cost Pareto-space. While HT and PT1 give higher throughput, PT2 gives minimum migration overhead satisfying the throughput requirement.

B. *Consideration of Cyclic Graph*

Throughput-constrained multimedia applications such as H.263 encoders are typically characterized by cyclic dependency among the different tasks (iterative) (refer Figure 2). It is therefore essential to consider cyclic dataflow models for fault-tolerance analysis of streaming multimedia applications. Synchronous Data Flow Graphs (SDFGs, see [15]) are often used for modeling modern DSP applications [17] and for designing concurrent multimedia applications implemented on

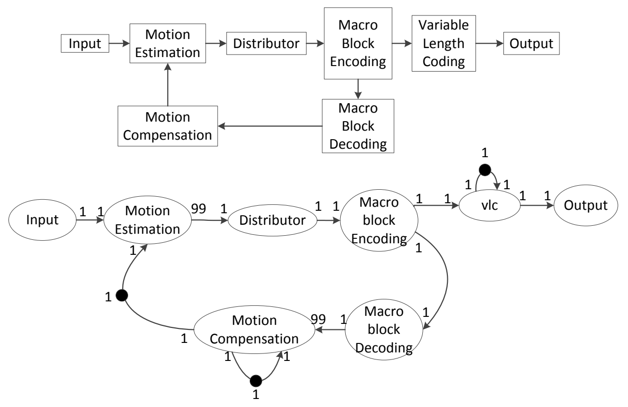

Fig. 2. H.263 Encoder & corresponding SDFG model

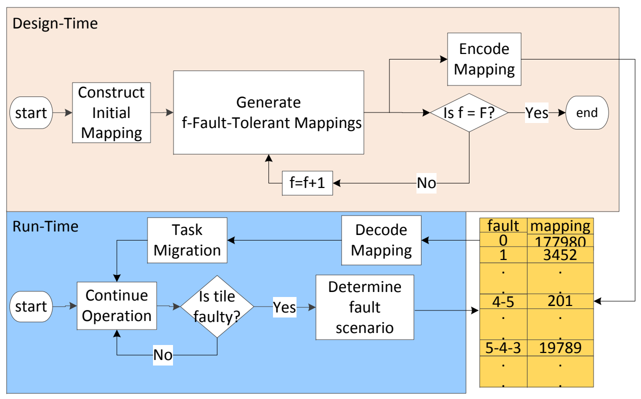

Fig. 3. Design Methodology

an MPSoC. The nodes of an SDFG are called $actors$; they represent tasks that are computed. The edges in the graph, called $channels$, represent communication of data from one actor to another.

IV. Problem Formulation

A. Migration Overhead Measure

Most modern embedded multiprocessors are equipped with local and shared memory. The context of an actor (referred to as state-space) is stored in the local memory of a tile where it is executed. Task migration involves moving this content of local memory from a faulty tile over NoC, provided the content is un-corrupted. Let s_i be the size of the state-space of an actor a_i participating in the task-migration process. If t_a be the initial tile of actor a_i and t_b be the tile after migration, then the migration overhead of a_i is given by

$$C_{mig}(a_i) \propto s_i * dist(t_a, t_b) \qquad (1)$$

where $dist(t_a, t_b)$ is the Manhattan distance between tile t_a and t_b determined using any shortest path algorithm[1].

B. Architecture Platform and Application Graph

Multiprocessor architectures consist of multiple tiles interconnected by network-on-chip. For simplicity, we represent a multiprocessor architecture by a tuple $\Lambda(\gamma, E_t)$ where γ is the set of tiles and E_t is the set of connection between any pair of tiles in γ. An application is represented as SDFG $G(A, E_a)$ where A is the set of actors and E_a is the set of edges between the actors representing data dependency.

C. Objective

Define

$$\begin{aligned} m &= \text{Initial actor-tile mapping} \\ \Lambda^f &= \text{Architecture graph with } f \text{ faulty tiles } \; : f \le F \\ m^f &= \text{Actor-tile mapping with } f \text{ faulty tiles} \end{aligned}$$

Determine $m^f : G \rightarrow \Lambda^f$ such that the migration overhead of moving from m to m^f is minimized while satisfying the throughput constraint.

[1]While a mesh-based topology is assumed for the target MPSoC, the proposed technique is orthogonal to any other topology like torus, tree etc.

V. Fault-Tolerant Task Mapping Methodology

To exploit application specific information and minimize run-time overhead, the objective function has been splitted into two phases. In phase I (design-time),actor-tile mappings for multiple fault-scenarios are generated. These mappings are stored in memory and are used in phase II (run-time) to perform the actual task migration as faults occur.

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

The rest of this section is organized as follows. In Section V-A the details of generating fault-tolerance mappings are provided. This involves determining the minimum migration overhead which is modeled as an integer linear programming (ILP) problem and is described in Section V-B. A linearization technique is developed in Section V-C to represent a mapping which reduces the storage overhead significantly. In Section V-D, an algorithm is developed to select an initial actor-tile mapping which minimizes migration overhead. Finally, Section V-E extends the proposed approach for joint minimization of throughput degradation and migration overhead.

A. Mappings with Variable Functional Tiles

Fault-tolerant mappings are generated using Algorithm 1. For every fault-scenario, a corresponding actor-tile mapping is generated and stored. There are F stages of the algorithm, where F is a user-defined parameter denoting the desired level of fault-tolerance. At every stage f $(1 \le f \le F)$, mappings are generated for fault-scenarios with f faulty tiles. These mappings are stored in the $HashMap$ data structure.

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 ${}^nP_f$,

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

Algorithm 1 Construct fault-tolerant mappings

```
Input: Initial mapping $m$, set of tiles $T$, throughput constraint $C$, fault-tolerance level $F$, HashMap, set of mappings ($M^{n-i},\ i \in [0..F]$)
Output: Minimum cost mappings satisfying throughput constraint for $f = 1$ to $F$ faults
 1: while $f \leq F$ do
 2:     $S^f := genFaultScenarios(f)$
 3:     for $s_f \in S^f$ do
 4:         $(a_{i_1}, a_{i_2}, \cdots, a_{i_f}) := s_f$
 5:         $s_{i_{f-1}} := (a_{i_1}, a_{i_2}, \cdots, a_{i_{f-1}})$
 6:         $m_{f-1} := HashMap[s_{f-1}].getMap()$
 7:         $[c_f, m_f] := getMinCost(m_{f-1}, a_f, M^{n-f})$
 8:         $HashMap[s_f].setCost(c_f)$
 9:         $HashMap[s_f].setMap(m_f)$
10:     end for
11:     $f := f + 1$
12: end while
```

where n is the initial number of tiles. An example set with 2 out of 3 tiles faulty ($f = 2, n = 3$) is the set $S^f = \{(0,1),(1,0),(0,2),(2,0),(1,2),(2,1)\}$[3]. For every scenario of the set S^f, the last tile (a_{i_f}) of the tuple $(a_{i_1}, a_{i_2}, \cdots, a_{i_f})$ is considered as the current faulty tile and a lower order tuple is generated by omitting a_{i_f} (line 5). This gives fault-scenario s_{f-1} with $f-1$ faulty tiles for which the optimal mapping is already computed (and stored in $HashMap$) in the previous stage (i.e. at stage $f-1$). This mapping is fetched (line 6) and used as the initial mapping for the subsequent steps.

An important aspect of Algorithm 1 is the set of feasible mappings (those satisfying throughput constraint C) with t tiles ($(n-F) \leq t \leq n$). This set (denoted by M^t) along with the corresponding throughputs is pre-computed using the SDF^3 tool [18] and stored in memory (*thrDB*). The $getMinCost$ function takes an initial mapping (m_{f-1}), a faulty tile (a_{i_f}) and the set of mappings (M^{n-f}) with $n-f$ functional tiles. The function returns an optimal mapping (m_f) from the set (M^{n-f}) which gives the minimum cost of migrating from m_{f-1} to m_f with tile a_{i_f} as faulty. Details of $getMinCost$ function are provided in the next sub-section.

Once an optimal mapping and the corresponding migration overhead are determined (line 7), the algorithm stores them in the $HashMap$ for the particular fault-scenario (line 8-9). This is repeated for every scenario of the set S^f.

B. Minimum Overhead Mapping

Minimum-cost task mapping problem is to determine an optimal point on the throughput-cost Pareto space for different fault-scenarios. This is detailed in Algorithm 2. A reference mapping (m), a faulty tile (t) and a set of reduced tile mappings (M) are given as input and the algorithm returns an optimal mapping (m_o) from the set M satisfying the throughput constraint. This is formulated as an ILP problem. A matrix CM is used to store the cost of migrating tasks from an initial mapping to a new mapping. The tiles from the initial mapping form the rows (indicated by o_i) and those from the new mapping form the columns (indicated by n_j). The rows (and columns) corresponding to the unused tile(s) and the fault ID f are filled with zeroes. Table I shows a sample CM with tile t_f as faulty. The non-zero entry (o_i, n_j) of M corresponds to the migration overhead associated with the extra actor(s) on tile n_j of new mapping which is (are) not present on tile o_i of initial mapping. This is computed as follows.

$$actors(o_x) = \text{actors mapped on tile } o_x \text{ of initial mapping}$$
$$actors(n_x) = \text{actors mapped on tile } n_x \text{ of new mapping}$$
$$CM(o_i, n_j) = \sum_{\substack{\forall a \in actors(n_j) \\ a \notin actors(o_i)}} C_{mig}(a)$$

Once the cost matrix is computed (line 4), the ILP is solved to obtain the minimum cost of migration (line 5). The cost and the throughput are stored in *costList* and *thrList* respectively. The process is repeated for all mappings of set M which satisfy the throughput constraint (line 3). The *getOptimal* function returns an optimal point on the throughput-cost Pareto space (point PT2 in Figure 1). Finally, a minimum cost mapping is found and returned as output.

ILP Formulation :
Base Variables: $X_{ij},\ i \in [1,k],\ j \in [1,k-1]$
Objective: Minimize z = $\sum_{ij} X_{ij} \times CM(o_i, n_j)$
Constraints: One element from each row and columns is to be selected

$$\sum_{j=1}^{k-1} X_{ij} := 1,\ \forall i \in [1,k] \ : \ \sum_{i=1}^{k} X_{ij} := 1,\ \forall j \in [1,k-1]$$

Algorithm 2 Get minimum cost mapping

```
Input: Mapping list $M$, reference mapping $m$, faulty tile $t$, throughput constraint $C$
Output: Minimum cost mapping from $m$
 1: $V := constructCostVector(G)$
 2: for $m' \in M$ do
 3:     if $thrDB.getThroughput(m') \geq C$ then
 4:         $CM[i][j] := constructCostMatrix(m, m', t, V)$
 5:         $cost := solveILP(CM)$
 6:         $MapList.push(m')$
 7:         $CostList.push(cost)$
 8:         $ThrList.push(thrDB.getThroughput(m'))$
 9:     end if
10: end for
11: $[m_o, c_o] := getOptimal(CostList, ThrList)$
```

TABLE I
COST MATRIX

$CM(o_i, n_j)$	n_1	n_2	n_3	$\cdots$	n_f	$\cdots$	n_{k-1}
o_1	50	205	180	$\cdots$	0	$\cdots$	175
o_2	200	100	180	$\cdots$	0	$\cdots$	200
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$
o_f	0	0	0	$\cdots$	$\cdots$	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ldots$	$\vdots$	$\ldots$	$\vdots$
o_k	165	110	120	$\cdots$	0	$\cdots$	135

[3]A fault-scenario (0,1) implies fault occuring first at tile 0 and then at tile 1. Thus, fault-scenario (0,1) is different from fault-scenario (1,0) forcing a permutation in the fault-scenario computation rather than combinations.

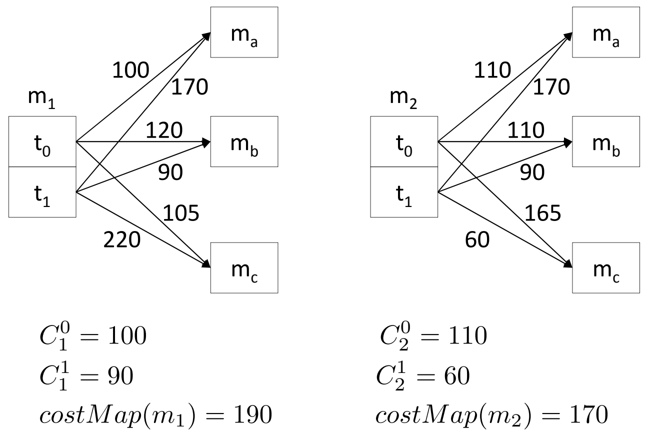

Fig. 4. Selecting an Initial Mapping

C. Encoding of Actor-Tile Mapping

A technique has been developed to encode the actor-tile mappings. This encoding scheme reduces the decoding overhead and is efficient for storing in or retrieving from any data structure (e.g. $HashMap$).

Each mapping m_i is represented by tuple $(t_i^0, t_i^1, .., t_i^{s-1})$ where, t_i^0 is the tile to which actor a_0 is mapped, t_i^1 is the tile where a_1 is mapped and so on. The total number of tasks and tiles are represented by s and n respectively. To uniquely identify each mapping, linearization technique is applied to each tuple and represent them by a mapping ID (mID_i). This is shown in Equation 2.

$$t_i^j \in [0, n-1] \qquad (2)$$
$$mID_i = \sum_{j=0}^{s-1} t_i^j \times n^j$$

D. Construct Initial Mapping

The first step in the design flow is the construction of actor-tile mapping. Simulation results show that the choice of the initial mapping is very crucial and has the potential to reduce migration overhead significantly. Based on this observation and the fact that single fault occurrences are more likely than the occurences of multiple faults, an algorithm has been developed for selecting an initial mapping that minimizes the task-migration of all single-fault scenarios. However, the algorithm can be easily adapted to consider initial mapping which minimizes migration overhead for any number of faults.

The idea of the algorithm is shown in Figure 4 with an example. One of the two mappings m_1 or m_2 is to be selected as the initial mapping. Let $\{m_a, m_b, m_c\}$ be the set of single-fault mappings. The first step of the algorithm is to compute the cost of moving from m_1 (and m_2) to all single-fault mappings considering each of the tiles (t_0 and t_1) to be faulty. The cost is indicated as weight on the edges. Thus, the costs of moving from m_1 to m_a, m_b and m_c are respectively 100, 120 and 105 with tile t_0 as faulty. Similarly, the cost to these three mappings are 170, 90 and 220 respectively considering tile t_1 to be faulty. Once all costs are determined, the minimum costs are retained. C_i^j denotes the minimum cost in moving from mapping m_i to a single-tile mapping with tile t_j as faulty. The minimum cost for mapping m_i is $costMap(m_i) = \sum_j C_i^j$. The mapping with the minimum cost is selected as the initial mapping. In Figure 4, m_2 is selected as it has the least cost (170) out of the two mappings.

Algorithm 3 Construct initial mapping

```
Input: A set of mappings ($M^n$) of $s$ actors on $n$ tiles, a set
    of mappings ($M^{n-1}$) of $s$ actors on $(n-1)$ tiles and the
    set of tiles $\gamma$
Output: Minimum cost start mapping $m$
 1: for $mapping\ m \in M^n$ do
 2:   if $throughput \geq constraint$ then
 3:     $cost := 0$
 4:     for $tile\ t \in \gamma$ do
 5:       $[c_t, m_t] := getMinCost(m, t, M^{n-1})$
 6:       $cost := cost + c_t$
 7:     end for
 8:     $costMap[m] := cost$
 9:   end if
10: end for
11: find $m \in M^n$, such that $costMap[m]$ is minimum
```

The above steps are detailed in Algorithm 3. There are three inputs to this algorithm – the set of mappings of all the actors on n tiles, the set of mappings of all actors on $(n-1)$ tiles and the set of tiles γ. The function $getMinCost(m, t, M^{n-1})$ returns an optimal mapping $m_t \in M^{n-1}$ and the minimum cost (c_t) of moving from m to m_t with tile t as faulty. Steps 3-8 of the algorithm compute the sum of the minimum cost for each single fault scenario for a mapping m. The same is repeated for all valid mappings $\in M^n$ which satisfy throughput constraint (line 2). Finally, in step 11, the minimum cost mapping is selected.

E. Joint Optimization of Throughput and Migration Overhead

As established in Section III-A, a certain category of multimedia applications demands scalable QoS. These applications demand joint treatment of throughput and migration overhead. For this we define an entity *throughput per unit migration*. The objective is to maximize this entity while satisfying the minimum throughput requirement. The change needed to incorporate this objective is in line 4 of Algorithm 2, where the matrix is filled with $\frac{cost}{throughput}$. The rest of the algorithm remains unchanged.

VI. Results

This Section provides an overview of the complexity and simulation results of the proposed alogirithms. Experiments are conducted with synthetic and real application graphs on a quad-core Intel Xeon-2.4GHz server running Linux. Algorithms are implemented in C++ and used in conjunction with the SDF^3 [18] tool for throughput computation. The ILP is solved using Matlab optimization toolbox.

A. Complexity

Time complexity of an algorithm is a measure of the number of computations performed by the algorithm. As established

TABLE II
RUNTIME COMPARISON OF FINDING MINIMUM TASK MIGRATION OVERHEAD WITH SAME NUMBER OF TASKS AND TILES

Tiles	Brute Force (sec)	HT (sec)	ILP Solver (sec)
4	0.002	0.0025	0.0056
8	0.5	0.065	0.0178
12	8.13	0.1579	0.0402
16	–	–	0.0707
20	–	–	0.1033
24	–	–	0.1440
28	–	–	0.1849
32	–	–	0.2388

in Section V-A, the total number of fault-scenarios for a F fault-tolerant system is given by Equation 3.

$$\#fault\_scenarios = \sum_{f=1}^{F} {}^{n}P_{f} \tag{3}$$

The $getMinCost$ function internally calls two functions – $constructCostMatrix$ and $solveILP$ for each mapping considered in the mapping list. The $constructCostMatrix$ function fills the cost matrix with the cost of task migration from an initial mapping to a new mapping. With n tiles, matrix CM (I) contains n^2 elements and therefore the worst case complexity of $constructCostMatrix$ is $O(n^2)$. Table II compares the runtime of the ILP solver against the brute force technique of finding cost using permutations and the dynamic programming based approach of HT [14]. As can be seen from the table, the brute-force and the HT techniques fails 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 HT technique. This clearly signifies the advantage of using the ILP based cost computation technique. The overall complexity of Algorithm 1 is given by Equation 4.

$$\begin{aligned} O(algo\ 2) &= F \times (\sum_{f=1}^{F} {}^{n}P_{f}) \times O(getMinCost) \\ &= (\#mappings)[O(n^2) + O(solveILP)] \end{aligned} \tag{4}$$

The storage associated with a 3-fault-tolerant system with 100 actors is summarized in Table III. The number of fault-scenarios (column 2) corresponding to a tile count is computed using Equation 3. The number of bits required to store a mapping m_i is given by Equation 5.

$$bits = \log_2 mID_i \tag{5}$$

where mID_i is computed using Equation 2. Finally, the total storage is obtained by multiplying the bits per mapping with the total number of homogeneous mappings.

B. Migration Overhead of Throughput-Constraint Applications

In this Section, migration overhead of our technique is compared with the highest throughput-based migration technique of [14] (referred to as **HT**) and the fixed-order band and band reconfiguration technique of [12] (referred to as **BB**). Our base technique is referred to as **TP** and the one including the initial mapping of Algorithm 3 as **TPI**. Experiments were conducted with synthetic application graphs with 6 tiles and 8 actors with throughput as a constraint.

TABLE III
STORAGE REQUIREMENT OF WITH INCREASING TILES FOR A 3-FAULT-TOLERANT SYSTEM WITH 100 ACTORS

Tiles	# Fault Scenarios	Bits per Mapping	Storage (KB)
4	24	200	4.6
8	112	300	32.8
12	264	400	92.4
16	480	400	187.5
20	760	500	320.7
24	1104	500	494.3
28	1512	500	709.8
32	1984	500	968.7

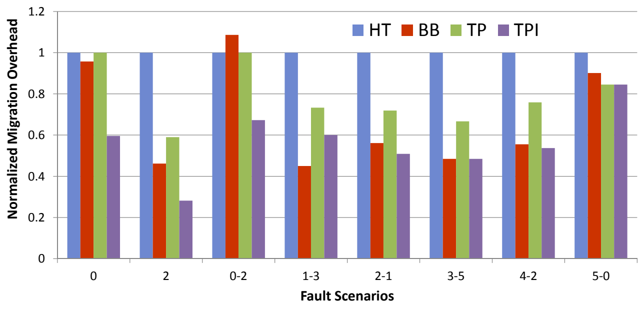

Fig. 5. Migration overhead with constant throughput

Figure 5 plots the migration overhead for all four techniques for different fault-scenarios. A fault-scenario (m-n) implies 2-fault condition with the first fault occurring at tile m and the next at tile n. For ease of representation, few of the single and double-fault conditions are included. The results obtained are normalized with respect to the migration cost obtained using HT technique. For our base technique (TP), the initial mapping is selected as the one which gives the highest throughput. This makes the initial mapping part of the HT and TP algorithm similar. However, for every fault-scenario, HT retains the highest throughput mapping, while TP selects the mapping with the least cost satisfying the throughput requirement.

There are few trends to be observed from the above figure. The BB technique involves migration of only the tasks on faulty tiles. This can potentially degrade the throughput below the constraint as has been observed in the experiments for most of the fault-scenarios. The other three techniques are throughput driven and may have additional migrations to achieve the highest throughput (HT technique) or satisfy throughput constraint (TP/TPI). The TP technique outperforms the HT technique and achieves 26% less migration overhead on an average for all single and double-fault scenarios. Further, the TPI technique based on the initial mapping of Algorithm 3 results in further reduction of migration overhead with an average 41% savings with respect to HT technique.

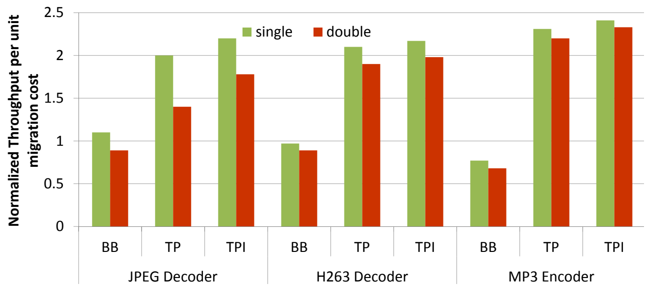

Fig. 6. Throughput per unit migration overhead for 3 real applications

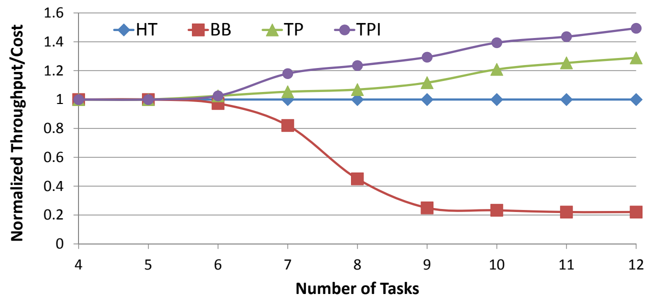

Fig. 7. Normalized throughput/cost with 6 tiles and varying actors

C. Throughput-Migration Cost Joint Performance

Figure 6 plots the normalized throughput per unit migration overhead of BB, TP and TPI with respect to HT technique. Experiments are conducted with three real applications – JPEG decoder (5 actors), H263 decoder (6 actors) and MP3 encoder (14 actors) on a reference architecture with 6 tiles. The result for BB, TP and TPI are normalized with respect to HT and therefore, HT is omitted from the graph. The two bars for each category correspond to the average throughput/cost of all single and double fault scenarios.

As can be seen from the figure, the TPI based technique in general has the highest throughput/cost among all the other techniques. The performance of BB degrades with increase in the number of actors (MP3 encoder for example). This is expected because, although the migration overhead is low for BB, throughput degradation is not accounted. The performance of TP and TPI are better than that of HT and BB.

Although the above observations demonstrate the advantage of our approach with respect to the existing techniques, it is worth mentioning that both TP and TPI achieve similar results as the HT technique when throughput is considered as the only objective.

D. Throughput-Cost Performance with Increasing Tasks

The scalability of our technique is studied by varying the number of actors on architecture with 6 tiles. Beyond 12 actors, the HT algorithm becomes infeasible due to the exponential growth of the possible solutions. Hence results are included up to 12 actors. Figure 7 plots the average throughput/cost of all 1, 2 and 3-fault scenarios for the algorithms normalized with respect to that obtained using HT algorithm.

As can be seen in the figure, the performances of all four algorithms are comparable for a small number of actors. The performance difference becomes significant as the number of tasks exceeds the number of tiles. As expected, throughput/cost of the BB technique is least and the TP technique performs better than the HT technique. Finally, TPI outperforms all the other techniques significantly due to fault-aware initial mapping. From the above results it can concluded that considering Pareto-optimal points and with a proper choice of an initial mapping, migration overhead can be reduced significantly.

VII. Conclusions & Future Direction

In this paper, SDF graph is used for fault-tolerance analysis of cyclic multimedia applications. Different optimal points are considered on the throughput-cost Pareto space to decide on a mapping for a fault-scenario. This is essential for modelling the streaming multimedia applications where throughput often comes as a constraint. Results show that the technique proposed in this paper can reduce the migration overhead by 26% while continuing to satisfy the application throughput requirement. Further, the proposed technique outperforms the existing technique in terms of joint metric like cost/throughput. Moreover, by proper choice of an initial mapping, a further reduction of 15% in migration overhead is achieved. There are however certain areas for improvement. Consideration of heterogeneous architecture and minimization of storage overhead associated with the different mappings are left as a future work.

References

[1] A. Jerraya *et al.*, *The what, why and how of MPSoC*. The Morgan Kaufmann Series in Systems on Silicon, 2005.
[2] W. Wolf, "Multimedia applications of multiprocessor systems-on-chips," in *DATE*, 2005.
[3] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *Micro*, 2003.
[4] I. Koren and C. Krishna, *Fault-tolerant systems*. Morgan Kaufmann.
[5] Y. Xie *et al.*, "Reliability-aware co-synthesis for embedded systems," *JSP*, 2007.
[6] A. Dogan *et al.*, "Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing," *TPDS*, 2002.
[7] L. Huang *et al.*, "Lifetime reliability-aware task allocation and scheduling for MPSoC platforms," in *DATE*, 2009.
[8] V. Izosimov *et al.*, "Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems," in *DATE*, 2005.
[9] T. Streichert *et al.*, "Dynamic task binding for hardware/software reconfigurable networks," in *SBCCI*, 2006.
[10] O. Derin *et al.*, "Online task remapping strategies for fault-tolerant Network-on-Chip multiprocessors," in *NOCS*, 2011.
[11] Y. Zhang *et al.*, "Workload-balancing schedule with adaptive architecture of MPSoCs for fault tolerance," in *BMEI*, 2010.
[12] C. Yang *et al.*, "Predictable execution adaptivity through embedding dynamic reconfigurability into static MPSoC schedules," in *ISSS-CODES*, 2007.
[13] J. Huang *et al.*, "Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems," in *ISSS-CODES*, 2011.
[14] C. Lee *et al.*, "A task remapping technique for reliable multi-core embedded systems," in *ISSS-CODES*, 2010.
[15] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*.
[16] M. Pittau *et al.*, "Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation," in *Estimedia*, 2007.
[17] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors; Scheduling and Synchronization*. Marcel Dekker, 2000.
[18] S. Stuijk *et al.*, "SDF^3: SDF For Free," in *ACSD*, 2006. [Online]. Available: http://www.es.ele.tue.nl/sdf3