Aging-aware Hardware-Software Task Partitioning for Reliable Reconfigurable Multiprocessor Systems

Anup Das, Akash Kumar and Bharadwaj Veeravalli
Department of Electrical and Computer Engineering
National University of Singapore
Singapore – 117583
Email: {akdas,akash,elebv}@nus.edu.sg

***Abstract*—Homogeneous multiprocessor systems with reconfigurable area (also known as Reconfigurable Multiprocessor Systems) are emerging as a popular design choice in current and future technology nodes to meet the heterogeneous computing demand of a multitude of applications enabled on these platforms. Application specific mapping decisions on such a platform involve partitioning a given application into software tasks (executed on one or more of the general purpose processors, GPPs) and the hardware tasks (realized as dedicated hardware on the reconfigurable area) to optimize and/or satisfy design constraints such as reliability, performance and design cost. Improving the reliability considering transient faults by increasing the number of checkpoints negatively impacts the reliability considering permanent faults. This trade-off is ignored in all prior studies on task mapping and scheduling. This paper proposes an optimization technique to decide the optimal number of checkpoints for the software tasks which minimizes aging of the GPPs while maximizing the transient fault-tolerance of the overall platform (GPPs and the reconfigurable area) and satisfying design cost and performance. Experiments conducted with synthetic and real-life application task graphs (cyclic and acyclic) demonstrate that the proposed technique minimizes aging and improves the platform lifetime by an average 60% as compared to the existing transient fault-aware techniques. Further, a gradient-based heuristic is proposed to minimize the design space exploration time by upto $500\times$ with less than 5% deviation from optimal solution.**

I. Introduction

Multiprocessor systems are becoming the obvious design choice in current and future technology nodes [1] to accommodate the ever increasing demands of applications and to address scalability. A growing trend in multiprocessor research is the integration of reconfigurable area (such as field programmable gate arrays, FPGAs and programmable logic devices, PLDs) with general purpose processors (GPPs) and digital signal processors (DSPs). Such architectures (commonly referred to as reconfigurable multiprocessor systems) offer significant power and performance benefits due to heterogeneity achieved in executing an application. Although several variants of reconfigurable architecture exist in literature, this research considers multiprocessor system with homogeneous GPPs and FPGA as shown in Figure 2. However, the techniques proposed here are generic and applicable to other architectures.

Recently, significant research attention is directed to run-time reconfigurable processors by implementing custom instructions [2], [3] or custom logic [4] on the reconfigurable area. Application mapping decisions on a reconfigurable architecture consist of hardware-software partitioning of the application to decide on the tasks that need to be executed on one or more GPPs and those that require dedicated hardware realized on the reconfigurable area (RA). Prior research studies on these architectures have mostly focused on improving application performance measured as throughput or makespan and/or minimizing the energy consumption of the platform while satisfying the design cost and area requirements [5], [6]. An area of growing concern in multiprocessor design is concerning error free execution of an application on the platform. Shrinking transistor geometries, growing transistor density and aggressive voltage scaling is negatively impacting the dependability of semiconductor devices by increasing the probability of transient, intermittent and permanent faults [7], [8]. The existing studies on reliability-aware application mapping suffer from the following limitations.

First of all, the existing checkpointing-based design methodologies consider transient faults in the GPPs alone. The solutions from these approaches guarantee or maximize fault-free task execution on the GPPs while exploiting the execution slack arising from the hardware execution of certain tasks. The area/performance penalty of adding redundant hardware (duplicating/triplicating) for the reconfigurable area are not accounted in these techniques. Further, the extent of fault-tolerance achieved within an allocated reconfigurable area in a co-design framework is not addressed. A complete solution to fault-tolerance needs to incorporate the transient fault-tolerance overhead for both the software and hardware tasks while satisfying the design performance constraints and reconfigurable area availability. Secondly, none of the existing hardware-software co-design techniques consider aging of the GPPs and transient faults simultaneously in the application task mapping and scheduling. As shown in Section III, improving transient fault-tolerance by increasing the number of checkpoints negatively impacts aging of the GPPs. A balance of the two is essential to tolerate transient and permanent faults jointly for multiprocessor systems. Finally, most multimedia applications such as H.264 encoder/decoder, JPEG decoder, etc. are characterized by cyclic dependency of tasks and require a fixed throughput to be satisfied to guarantee quality-of-service to end users. Existing fault-tolerant research studies for reconfigurable multiprocessor systems are based on acyclic graph modeling of applications with no consideration of throughput degradation. These techniques require significant modification (if not entirely in-applicable) for streaming multimedia applications represented as cyclic graphs.

This paper presents a design-time technique to decide the hardware/software partitioning of an application i.e. deciding on the tasks to be executed on the GPPs and those on reconfigurable area. The overall objective of the approach is to improve fault-tolerance of the platform which considers three effects – transient faults in the GPPs, single event upsets in the logic configuration bits of the reconfigurable area and the aging of the GPPs. The transient faults in the GPPs are mitigated using checkpointing technique. Single event upsets in the logic configuration bits of a reconfigurable area (like Xiling FPGA) manifest as permanent faults and render the affected logic useless, unless reprogrammed. The current approach does not consider reprogramming the reconfigurable

area within an application execution and therefore redundancy-based techniques such as duplication (for tasks requiring fault-detection only) and triplications (tasks requiring correction) are assumed for the single event upsets. The corresponding area overhead is incorporated in the problem formulation. Finally, aging of the GPPs is mitigated using intelligent task mapping and scheduling.

Contributions: Following are the key contributions.

- Application task mapping and scheduling with joint consideration of permanent and transient faults for reconfigurable multiprocessor systems.
- Formulation of aging of GPPs and checkpointing based transient error recovery problem in the hardware-software task partitioning framework with reconfigurable area as a constraint.
- A gradient-based fast design space exploration for task mapping considering aging and transient faults simultaneously.
- Use of cyclic and acyclic graphs for hardware-software task partitioning.

The mapping flow proposed in this paper generates the following decisions for each application (represented as directed graph) enabled on the reconfigurable multiprocessor platform.

1. Tasks to be mapped on GPPs (software tasks) and tasks to be implemented on reconfigurable area (hardware tasks).
2. Number of checkpoints for each software tasks
3. Mapping and scheduling of software tasks to maximize transient fault-tolerance while minimizing GPP aging.

These decisions are used at run-time as and when an application is enabled. Experiments conducted on synthetic and real-life applications represented as directed acyclic graphs (DAGs) and synchronous data flow graphs (SDFGs) demonstrate that the proposed methodology minimizes the aging of GPPs by an average 60% for a low transient fault-tolerant system and average 10% for high transient fault-tolerant system. Further, the proposed gradient-based heuristic reduces design space exploration time by upto $500\times$ with less than 5% deviation from the optimal solution.

To the best of our knowledge, this is the first work on hardware-software application task partitioning that considers transient fault-tolerance of GPPs and reconfigurable area simultaneously with aging of GPPs.

The rest of this paper is organized as follows. A brief introduction on the related research is provided in Section II followed by mathematical foundation on transient and permanent fault-tolerance in Section III highlighting the trade-off missing in earlier works. The reliability-aware hardware-software co-design problem is formulated in Sections IV and V. The mapping flow is introduced in Section VI and the experimental results in Section VII. Finally, Section VIII concludes this paper with discussion on future directions.

II. Related Works

Since its introduction, hardware-software co-design has received significant attention among researchers starting from the classical optimization metric such as performance, power and cost to the recent ones such as reliability. Details of the performance, cost and power driven co-design techniques is beyond the scope of the current paper. Interested readers are urged to refer to [9]. Some of the key studies on reliability-aware co-design are presented here.

The first category of research studies is focused on application mapping on static multiprocessor architectures with GPPs only. An ant-colony optimization technique is proposed in [10] to generate a task mapping on static multiprocessor systems which satisfies the useful life requirement. This technique incorporates aging in the application mapping but is limited to permanent faults only. A design optimization with task duplication and processor hardening for intermittent and transient faults is proposed in [11]. Aging of GPPs is not considered. A checkpointing-based technique is proposed in [12] for transient fault-tolerance of hard/soft real-time tasks on a static multiprocessor system. A genetic algorithm is proposed in [13] to optimize for cost, time and dependability. Tasks are replicated to improve transient fault-tolerance and task graphs with replicated tasks are scheduled on a multiprocessor system. A multi-objective evolutionary algorithm is proposed in [14] to determine the mapping of tasks to processing elements considering the occurrence of permanent and transient faults. However, wear-outs of GPPs are not considered. A fault-aware resource management is proposed in [15] to determine spare core placement while dealing with transient, intermittent and permanent faults explicitly incorporating processor aging. A common limitation of these techniques is the non-consideration of platform re-programmability and the associated reliability trade-offs in terms of aging and transient fault tolerance. Further, throughput degradation is not considered.

The next class of research focuses on mapping of applications on GPPs with one or more custom modules. A system-level synthesis flow is proposed in [16] to mitigate the impact of transient faults. Fault management requirements (e.g. fault detection/tolerance) of different tasks of an application are incorporated in the task graph representation and a scheduling technique is proposed to map the application graph on a multiprocessor system (consisting of GPPs and/or dedicated hardware modules) satisfying the performance requirement. However, the proposed methodology does not consider constraints on the reconfigurable area availability and the corresponding fault-tolerance trade-off. Further, aging of GPPs and the use of cyclic graphs are lacking. A multi-objective evolutionary algorithm is proposed in [17] to jointly optimize reliability, area and latency while deciding on application mapping on GPPs and custom modules. The proposed technique optimizes for permanent faults on the processors only and limited details are provided on the consideration of aging. Moreover, this research considers static multiprocessor systems with no transient fault prevention. A hardware-software co-design approach is presented in [18], [19]. Every task of an application is specified by different implementation alternatives such as GPP or ASIC. Each implementation differs by area, cost and reliability values. Both these techniques suffer from the four limitations discussed in Section I.

The third category of research is application mapping on GPPs and FPGAs. In a recent study [20], fault-tolerance of hardware-software hybrid tasks is proposed. Fault-tolerance is incorporated as rollback and recovery for software tasks (running on GPP) and triple modulo redundancy (TMR) for hardware tasks. However, this technique is not focused on hardware-software partitioning i.e. this technique is built on the pre-computed decisions of tasks that need to be run on GPPs and those requiring hardware implementation. Further, aging of processors, transient and permanent fault-tolerance trade-off and throughput degradation are not addressed. A task partitioning technique is proposed in [21] to tolerate transient faults in GPPs. This technique suffers from the same limitations.

The last category of research is related to reliability-aware

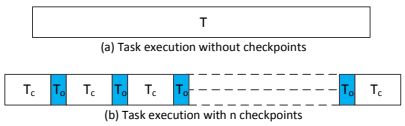

Fig. 1. Task execution with and without checkpoint

application mapping on FPGA-based systems [22]. Transient faults are dominating threats for such system as GPPs are incorporated as soft cores in the FPGA fabric and can be easily reconfigured using spare lookup tables on the detection of permanent faults. Thus, the impact of permanent faults (and hence aging) is insignificant.

III. Background on Reliability

Transient and/or permanent faults on a processor degrade the reliability of an application/thread/task running on the processor. Transient faults are single event upsets and the impacted processor can be recovered, while permanent faults are non-recoverable defects rendering a processor unusable. One of the primary reasons for permanent faults is processor aging. This section introduces the mathematical foundation to model the trade-off involved in reliability of execution under the impact of transient faults and aging.

A. Transient fault-tolerance

Transient faults have received significant attention in recent years due to the growing rate at deep sub-micron nodes. Some of the commonly used transient fault-tolerance technique are *Checkpointing* [12], [23]–[25], *Rollback-Recovery* [27] and *Duplication with Comparison and Re-execution*. For the current analysis, *checkpointing* based technique is assumed.

Checkpoint refers to the state of the system at a particular instance of time. The process of *checkpointing* involves periodically storing the checkpoint (in local or remote memory) during the execution of a task. In the event of a transient fault, execution is continued from the last valid checkpoint. One important parameter of *checkpointing* is *checkpoint overhead* which is defined as the increase in the execution time. This overhead is dependent on

1. number of checkpoints, N
2. time for checkpoint capture and storage, T_o
3. time for recovery from a checkpoint, T_r
4. fault arrival rate, λ

Following are the assumptions regarding checkpointing-based transient fault-tolerance similar to the works in [12]–[14], [17], [23].

- Transient faults follow Poisson distribution with a rate of λ failures per unit time.
- Transient faults are point failures i.e. these faults induce an error in the system and disappear.
- The probability of multiple transient faults in each checkpoint segment is negligible.
- Checkpoints can be inserted anywhere in the execution time. This assumption although difficult to accomplish in practice, gives a first order approximation on the problem at hand.

Figure 1 shows an example task execution with N checkpoints. Let T denote the total execution time of the task and T_c, the execution time of the task in each checkpoint segment. Clearly, $T_c = \frac{T}{N+1}$. The probability of atleast one fault in inter checkpoint interval $(T_c + T_o)$ is $P_e = 1 - e^{-\lambda(T_c+T_o)}$. Assuming fault arrival follows Poisson process, the probability of more than one fault in the inter checkpoint interval is negligible as $\lambda T_c \ll 1$. Using first order approximation, the expected length of checkpoint segment $E[T_c]$ is calculated as

$$\begin{aligned} E[T_c] &= P\{\text{no error in segment}\} * \text{ normal checkpoint interval } + \\ &\quad P\{\text{error in segment}\} * \text{ modified checkpoint interval} \end{aligned} \quad (1)$$

When there are no errors in checkpoint segment, the checkpoint interval (normal) is $T_c + T_o$ where T_o is the time for checkpoint computation and storage (refer Figure 1). Let τ denote the time of the first fault from the start of a checkpoint segment. Since a fault can occur at any time in the checkpoint segment, τ is uniform random variable in the range 0 to T_c+T_o with an average value of $\frac{T_c+T_o}{2}$. Hence, under the assumption of single failure, modified checkpoint interval is given by

$$\text{modified checkpoint interval} = \tau + T_r + (T_c + T_o) \quad (2)$$

where the first component is the time lost since the fault occurrence, the second component is the time for recovery from the last valid checkpoint and the last component is the re-execution time of the checkpoint segment starting from the last valid checkpoint. The recovery time includes the overhead for fetching the checkpoint from the local/remote memory and applying to the processor. Thus, Equation 1 can be written as

$$\begin{aligned} E[T_c] &= (1 - P_e) * (T_c + T_o) + P_e * (\tau + T_r + T_c + T_o) \\ &= (T_c + T_o)e^{-\lambda(T_c+T_o)} + \\ &\quad \left(\frac{3(T_c + T_o)}{2} + T_r\right)\left(1 - e^{-\lambda(T_c+T_o)}\right) \\ &= \frac{3(T_c + T_o)}{2} + T_r - \left(\frac{T_c + T_o + 2T_r}{2}\right) e^{-\lambda(T_c+T_o)} \end{aligned} \quad (3)$$

The expected length of the last checkpoint segment ($E[T_c^L]$) is computed from Equation 3 by replacing $(T_c + T_o)$ with T_c. This is because there is no checkpointing overhead for the last segment. The expected execution time of the task is given by

$$E[T] = N * E[T_c] + E[T_c^L] \quad (4)$$

Reliability of a task with checkpointing-based fault-tolerance increases with an increase in the number of checkpoints upto a saturation point, beyond which the reliability drops-off [24], [25]. This point of negative returns (in terms of reliability of checkpointing) is dependent on the workload execution time and the checkpointing overhead (T_o). All discussions in this paper are limited to the region where reliability improves with checkpoints. The reliability of a task i (in this region) with checkpointing is given by

$$\begin{aligned} R_i(t) &= (1 - P_e)^{N+1} + \binom{N+1}{1} P_e (1 - P_e)^{N+1} \\ &\quad + \binom{N+2}{2} P_e^2 (1 - P_e)^{N+1} + \cdots \end{aligned} \quad (5)$$

where the first term on the right hand side is the reliability with no faults; the second term is the reliability with one transient fault in any of the $N+1$ checkpoint segment; the third term is the reliability with two faults in $N+2$ segments ($N+1$ original segments and 1 re-execution segment of the segment where the first fault occurs) and so on. Assuming infinite faults in the task execution, the above expression reduces to[1]

$$R_i(t) = \sum_{\omega=0}^{\infty} \binom{N+\omega}{\omega} P_e^{\omega} (1 - P_e)^{N+1} = 1 \quad (6)$$

[1]Proof ommitted for space limitation.

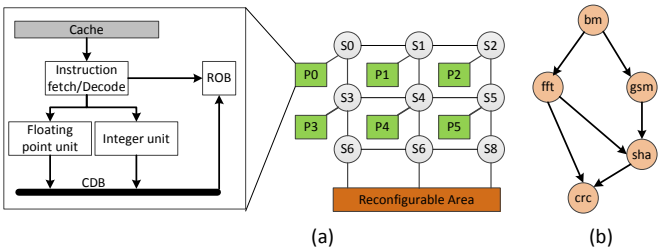

Fig. 2. Architecture and application model (DAG)

This is intuitive because if segment re-execution is allowed every time a fault is detected, the task will eventually be executed successfully. However, for real time systems with task deadlines, infinite faults will lead to deadline violation. Therefore, for real time systems, the sum in Equation 6 is evaluated till ζ_i, where ζ_i is the maximum number of faults for the task i such that its deadline is satisfied. The reliability of task i is

$$R_i(t) = \sum_{\omega=0}^{\zeta_i} \binom{N+\omega}{\omega} P_e^{\omega}(1-P_e)^{N+1} \tag{7}$$

For an application consisting of multiple interconnected tasks, the overall reliability and the mean time between failures (MTBF) are given by

$$\begin{aligned} R(t) &= \prod_{i=1}^{N_a} R_i(t) \\ MTBF &= \int_{t=0}^{\infty} R(t)dt \end{aligned} \tag{8}$$

where N_a is the number of tasks and $R_i(t)$ is the reliability of task i. The above equation is derived based on the assumption that transient fault occurrences are independent of each other and an application is successful when all tasks of the application executes successfully.

B. Permanent fault-tolerance

There are four dominant wear-out effects for ICs: electromigration (EM), time-dependent dielectric breakdown (TDDB), stress migration (SM) and thermal cycling (TC). For the current research, EM related wear-out failures are assumed, however, any other effects can be easily incorporated either standalone or using Sum-of-Failure Rate (SOFR) model for any combination of the above failure effects. The lifetime reliability of a processor at the end of the first period of an application graph is calculated according to the following equations (ref. [28]–[30]).

$$\begin{aligned} R(t_p) &= e^{-(A)^{\beta}} \text{ where} \\ t_p &= \text{period of the application} \\ A &= \text{Aging effect of processor } = \sum \frac{\Delta t_i}{\alpha(T_i)} \\ \Delta t_i &= \text{time intervals within period } t_p \\ \alpha(T_i) &= \frac{A_0(J-J_{crit})^{-n} e^{\frac{E_a}{KT_i}}}{\Gamma\left(1+\frac{1}{\beta}\right)} \end{aligned} \tag{9}$$

where β is the slope parameter of the Weibull distribution. The reliability after m periods of the application graph and the closed form expression for the mean time to permanent fault (MTTF) are given by the following equations.

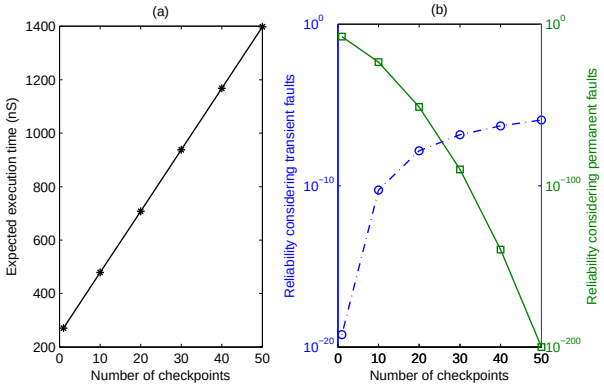

Fig. 3. Trade-off

$$R(t_{mp}) = e^{-(m\times A)^{\beta}} \tag{10}$$

$$MTTF = \sum_{i=0}^{\infty} e^{-(i\times A)^{\beta}} \times t_p \tag{11}$$

For the remainder of the paper *MTBF* is used to denote mean time between transient faults and *MTTF or lifetime* is used to denote the mean time to permanent faults.

C. Transient and permanent fault-tolerance trade-off

Figure 3 (a) plots the expected execution time of a task as the number of checkpoints are increased. The expected execution time is computed according to Equation 4. The parameters used for simulation are execution time with no-checkpoints, $T = 150nS$, the checkpoint computation and storage overhead $T_o = 15nS$, the recovery time $T_r = 0.5nS$ and fault rate $\lambda = 10^{-6}$ i.e. one fault every 10^6 seconds. As can be seen from the figure, the expected execution time increases with an increase in the number of checkpoints. Figure 3 (b) plots the reliability trade-off as the number of checkpoints is increased. The reliability considering transient faults (shown in the figure by blue dotted line) increases with an increase in the number of checkpoints. This is in line with that predicted by Equation 7. On the other hand, the reliability considering permanent faults (shown by green solid line) decreases due to an increase in the expected execution time (Figure 3 (a)) which increases processor aging (ref. Equation 9). Analysis are also conducted on multitask applications and a similar trend is observed. The results for the same are omitted here for space limitation.

IV. Problem Formulation for DAGs

A. Architecture model

Figure 2 (a) represents an architecture model assumed for this research. The architecture consists of N_p homogeneous processors connected to a shared reconfigurable area. The processor cores have private L1 data and instruction caches while the L2 cache is shared among all the cores. The reconfigurable area assumed in this work is a one dimensional (1D) model and is divided into N_c equal sized columns. A column is a basic unit for reconfiguration (e.g. Xilinx Virtex 6 FPGA). The architecture is represented as $G_{arc} = (V_{arc}, E_{arc})$, where V_{arc} is the set of processors $\langle p_1, p_2, \cdots p_{N_p}\rangle$ and E_{arc} is the set of links connecting the processors. Thus, $N_p = |V_{arc}|$. For the ease of problem formulation, the reconfigurable area is considered as a virtual processor (different from the homogeneous ones) and is indexed by the subscript $N_p + 1$. The following restrictions/relaxations apply for the virtual processor p_{N_p+1}.

1. Tasks mapped to this processor have associated area cost. This is because a task mapped on the virtual processor implies dedicated hardware for the task (in reality) which consumes few columns of the reconfigurable area.
2. Multiple tasks can be executed at the same time on the virtual processor. This is because, one or more hardware tasks implemented on the different regions of the reconfigurable area can run independently.
3. Execution time of a task on the virtual processor is usually less than the execution time of the task on a GPP
4. A task mapped on the virtual processor does not need software-based protection technique such as check pointing/rollback and replication. Instead, the protection is provided by replicating the hardware implementation.

B. Application model

Figure 2 (b) plots the application model represented as directed acyclic graphs (DAGs). Later in Section V, model for cyclic graphs is presented. Mathematically, an application DAG is represented as $G_{app} = (V_{app}, E_{app})$, where V_{app} is the set of nodes representing tasks of the application and E_{app} is the set of directed edges representing data dependency among various tasks. Let N_a $(= |V_{app}|)$ denote the number of tasks and L the set of leaf nodes for the application.

Every task $v_i \in V_{app}$ is a tuple $\langle a_i, n_i, T_i \rangle$, where a_i is the area required to implement v_i on the reconfigurable area and n_i is the time taken by v_i to execute on the dedicated hardware. If a task does not support hardware implementation, the value of these parameters are set to infinite. The overhead for hardware-based transient fault-tolerance for the task is incorporated into a_i. For tasks requiring fault-detection only, a_i is the area of duplicating the logic in hardware and the area of a checker circuit. For those tasks requiring fault-mitigation, a_i includes the area for triple-modulo redundancy (TMR) and the voter circuit. T_i is the set of execution time of v_i with different number of checkpoints. Specifically, $t_{i,f} \in T_i$ is the expected execution time with f checkpoints.

C. Mapping representation

The mapping of G_{app} on G_{arc} is an $N_a \times (N_p + 1)$ matrix given by

$$M = \langle m \rangle = \begin{pmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,(N_p+1)} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,(N_p+1)} \\ \vdots & \vdots & \ddots & \vdots \\ m_{N_a,1} & m_{N_a,2} & \cdots & m_{N_a,(N_p+1)} \end{pmatrix}$$

where the binary variable $m_{i,j}$ is given by

$$m_{i,k} = \begin{cases} 1 & \text{if task } v_i \text{ is mapped on processor } p_k \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

A task is mapped to a single processor only. Therefore

$$\forall i \in [1, 2, \cdots N_a], \quad \sum_{k=1}^{N_p+1} m_{i,k} = 1 \quad (13)$$

D. Variables for problem formulation

Following variables are defined for ease of problem formulation.

$$x_{i,k,f} = \begin{cases} 1 & \text{if task } v_i \text{ is mapped on processor } p_k \text{ with } f \text{ checkpoints} \\ 0 & \text{otherwise} \end{cases}$$

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

s_i = start time of task v_i

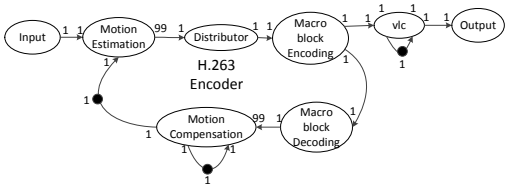

Fig. 4. SDFG of H.263 Encoder

A task can have only a fixed number of checkpoints i.e.

$$\forall i, k \ \sum_f x_{i,k,f} = 1 \quad (14)$$

The variable x is related to m according to

$$m_{i,k} = \sum_f x_{i,k,f} \quad (15)$$

E. Constraints of the problem formulation

1. Every task must be assigned to a single processor with a single fault-tolerant technique. Combining Equations 13, 14 and 15,

$$\forall i \in [1, 2, \cdots N_a], \quad \sum_{k=1}^{N_p+1} \sum_f x_{i,k,f} = 1 \quad (16)$$

2. Finish time of the leaf tasks is less than the application deadline (D).

$$\forall v_i \in L, \ p_k \in V_{arc}, \quad s_i + et(i,k,f) \leq D + (1 - x_{i,k,f})W \quad (17)$$

where W is a very large number and $et(i,k,f)$ is given by

$$et(i,k,f) = \begin{cases} t_{i,f} & \text{for } 1 \leq k \leq N_p \\ n_i & \text{for } k = N_p + 1 \end{cases} \quad (18)$$

3. All tasks must satisfy the dependency constraints.

$$\forall (i,j) \in E_{app}, \ \forall k, \quad s_i + et(i,k,f) \leq s_j + (1 - x_{i,k,f})W \quad (19)$$

4. Independent tasks mapped on the same processor (excluding the virtual one) must not be executed simultaneously.

$$\begin{aligned} &\forall (i,j) \notin E_{app}, \quad \forall k \in [1, 2, \cdots, N_p], \quad \forall f, f' \\ &s_i + et(i,k,f) \leq s_j + (3 - x_{i,k,f} - x_{j,k,f'} - d_{i,j,k})W \\ &s_j + et(j,k,f') \leq s_i + (2 - x_{i,k,f} - x_{j,k,f'} + d_{i,j,k})W \end{aligned} \quad (20)$$

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

5. Area consumed by tasks mapped to virtual processor should satisfy the reconfigurable area constraint.

$$\sum_i a_i \times x_{i,N_p+1,f} \leq RA \quad (21)$$

where RA is the total columns of the reconfigurable area.

F. Objective function

The objective of the optimization problem is to determine the number of checkpoints for each task which is balanced in terms of transient and permanent fault-tolerance. A cost function is defined

$$MTTF_J = a \times MTTF_P + b \times MTBF_T \quad (22)$$

where $MTTF_P$ and $MTTF_T$ are the MTTF due to permanent and transient fault-tolerance respectively, a and b are two user-defined weights assigned to these metrics respectively. Clearly, setting $a = 0$, optimizes for transient fault-tolerance only, setting $b = 0$ optimizes for permanent fault-tolerance.

V. Problem Formulation for SDFGs

Synchronous Data Flow Graphs (SDFGs, see [31]) are often used for modeling modern DSP applications and for designing concurrent multimedia applications implemented on a multi-processor 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 an actor is called port $rate$, and remains constant. The rates are visualized as port annotations. Actor execution is also called $firing$, and requires a fixed amount of time, denoted with a number in the actors. The edges in the graph, called $channels$, represent data communicated from one actor to another.

Figure 4 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 4. 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.

A. Changed variable definitions and additional constraints

Let $G_{app} = (V_{app}, E_{app})$ represent an application SDFG with V_{app} actors and E_{app} edges. The following are defined.

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

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

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

$$\forall i, j : \sum_{u=1}^{r(i)} x_{ik,u} = 0 \text{ or } r(i) \tag{23}$$

  where $r(i)$ is the repetition vector of task i.
- *Auto-concurrency of actors* (multiple iterations of an actor are not enabled simultaneously)

$$\forall i, k \text{ and } 2 \leq u \leq r(i) : s_{ik,u} \geq s_{ik,u-1} + et(i) \tag{24}$$

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

$$\forall k, l \in P \text{ and } \forall (j, i) \in E : s_{ik,u} \geq e_{jl,m} \tag{25}$$

  where m is defined as follows

$$m = \lfloor \frac{kp}{q} + init(i,j) \rfloor$$

$$p = \text{tokens produced by actor } i \text{ on edge } (i,j)$$

$$q = \text{tokens consumed by actor } j \text{ from edge } (i,j)$$

$$init(i,j) = \text{initial token on edge } (i,j)$$

B. Objective function

The objective function is same as that for the acyclic graph derived in Section IV.

Algorithm 1 $GDSE()$: Gradient based mapping generation

Input: G_{app}, # of processors (N), RA
Output: Mapping of G_{app} on N processors, M_N

1: Initialize $HList = \emptyset$
2: **while** $\sum_{\forall v_k \in HList} a_k \leq RA$ **do**
3: **for all** $v_i \in V_{app} \setminus HList$ **do**
4: $HList.push(v_i)$
5: $[M \quad \Pi^i_T] = SGen(V_{app}, N, HList)$
6: $HList.pop(v_i)$
7: **end for**
8: Find $v_j \in V_{app} \setminus HList$ such that Π^j_T is maximum
9: $HList.push(v_j)$
10: **end while**
11: $[M_N \quad \Pi_T] = SGen(V_{app}, N, HList)$

VI. Application Mapping Flow

The objective function for the joint optimization is non-linear and therefore the execution time increases exponentially with an increase in the number of tasks and/or processors. A gradient-based design space exploration ($GDSE$) technique is proposed to reduce the exploration time. A joint metric (*Reliability Gradient,* R_G) is defined as follows.

$$R_G = \frac{\Delta R_P(t)}{\Delta R_T(t)} \tag{26}$$

where $R_P(t)$ and $R_T(t)$ are the reliability due to permanent faults (aging) and transient faults respectively. The *reliability gradient* is interpreted as ratio of the change (decrease) in reliability due to permanent faults per unit change (increase) in reliability due to transient faults. The objective of the $GDSE$ is to minimize the *reliability gradient*. This is shown as pseudo-code in Algorithm 1.

A list ($HList$) is defined to store the list of tasks which are mapped on the reconfigurable area i.e. tasks which are implemented as hardware logic (these tasks are referred as hard tasks). The algorithm iterates (lines 2-10) as long as the available reconfigurable area constraint is satisfied (line 2). In every iteration, the algorithm selects one of the tasks from the set $V_{app} \setminus HList$ (i.e. selects one of those tasks which are not marked as hard tasks) and assigns it to the reconfigurable area (line 4). The mean time between transient failures for this assignment is determined (line 5). The one assignment with the highest mean time between transient failures is selected and permanently marked as hard task and pushed in the $HList$ (line 9). An important component of this algorithm is the generation of a mapping (and schedule) along with the mean time between transient failures. This is performed in the $SGen$ subroutine whose pseudo-code is shown in Algorithm 2.

The first step of Algorithm 2 is the generation of an initial mapping (line 1). The choice of initial mapping is crucial in determining the overall quality of the proposed $GDSE$ algorithm. For this work, the task mapping with the highest mean time to permanent fault is selected. This choice is justified in Section VII. The algorithm continues to remap each task to a processor with different checkpoints to determine the *reliability gradient* (lines 5-10). If the *reliability gradient* obtained for an assignment is lower than the best value obtained thus far, the best values are updated (lines 11-13). After iterating for all tasks, the mapping is changed with the best value of task, processor and checkpoints. This process is continued (starting with this changed mapping) as long as no further remapping is possible without violating the performance requirement. When this happens, the best

Algorithm 2 $SGen()$: Gradient-Based Task Re-Mapping

Input: $V_{app},\ N,\ HList$
Output: Mapping, M and mean time between transient faults, Π_T

1: $\hat{M} = \langle \hat{m} \rangle$ = initial mapping; $x_{i,k,0} = \hat{m}_{i,k},\ \forall i,k$
2: $\forall v_j \in HList,\ m_{j,N_p+1} = 1;\ runIter = 1$
3: **while** $runIter > 0$ **do**
4: $t_b = 0;\ p_b = 0;\ f_b = 0;\ r_b = \infty$
5: **for all** $v_i \in V_{app} \setminus HList$ **do**
6: **for** $k = 1$ to N_p **do**
7: **for** $f = 1$ to N_f **do**
8: $M = \hat{M}$; Determine $\langle x \rangle$ from $\langle m \rangle$
9: $x_{i,k,f} = 1$
10: $s = SCE(G_{app}, \langle x \rangle);\ r = calcRG(s, r_b)$
11: **if** $r < r_b$ **then**
12: $p_b = V_{app}(i);\ t_b = k;\ f_b = f;\ r_b = r$
13: **end if**
14: $x_{i,k,f} = 0$
15: **end for**
16: **end for**
17: **end for**
18: **if** $t_b > 0$ **then**
19: Change mapping $\hat{M}$ with task t_b assigned to processor p_b with f_b checkpoints
20: **else**
21: $runIter = 0$
22: **end if**
23: **end while**
24: $M = \hat{M};\ \Pi_T = calcMTBF(\hat{M})$

value of task, processor and checkpoints are all empty. The algorithm proceeds to the else section (lines 20-22) where the terminating condition is asserted. Finally, the new mapping is returned together with the mean time between transient faults which is calculated in the $calcMTBF$ routine. The $SCE()$ subroutine computes the schedule for a given mapping and implements the performance requirement. This is the same as the scheduling technique proposed in [32] for DAG and SDFG.

There are two points to note from these two algorithms. First of all, for multiprocessor systems with no reconfigurable area, it is sufficient to execute $SGen$ only with $HList = \emptyset$. Thus, the technique proposed here is generic and can be applicable to both static and reconfigurable multiprocessor architecture. Secondly, the schedule of actors (or tasks) on a processor for SDFG may consist of multiple firings of mapped actors. If the actor firing orders at run-time differ from those at design-time, the throughput obtained at run-time can be significantly different than those analyzed/guaranteed from design-time. Thus, the analyzed schedule needs to be stored and used at run-time. However, multiple such schedules are required based on the availability of GPPs (due to multiple applications running concurrently) and the reconfigurable area size. This poses significant storage overhead. Further, constructing a new schedule at run-time which guarantees throughput requirement can lead to deadline misses for large number of tasks and/or processors. This makes the existing techniques for reliability-aware hardware-software task partitioning infeasible especially for SDFGs. The self-timed execution based scheduling proposed in [32] solves both problems by generating a master schedule at design-time which can be reused at run-time.

VII. Results

Experiments are conducted on a quad-core Intel Xeon 2.4GHz server running Linux with fifty synthetic application task graphs generated using *TGFF* tool [33]. The number of tasks range from 4 to 32 and the targeted MPSoCs consist of 2 to 8 homogeneous cores with 100 columns of reconfigurable area. Additionally, a set of real-life applications are considered both from streaming and non-streaming domain [34] [35]. The following parameters are used for computing MTTF [28]–[30]: current density $J = 1.5 \times 10^6 A/cm^2$, activation energy $E_a = 0.48eV$, slope parameter $\beta = 2$ and $n = 1.1$. The scale parameter of each core is normalized so that its MTTF is 10 years for the characterization temperature of $300K$.

A. Algorithm complexity

The complexity of Algorithm 1 is computed as follows. Let η be the average number of tasks that can be accommodated in the given reconfigurable area constraint. There are therefore η iterations of the outer while loop (lines 2-10). At each iteration, lines 4-6 are executed for all tasks in $V_{app} \setminus HList$. This can be upper bounded by N_a, the total number of tasks in the application graph. Line 8 finds the minimum from a list of N_a elements. Assuming the complexity of push and pop from a list as constant, the complexity of Algorithm 1 is given by

$$C_1 = \eta \times (N_a \times O(SGen) + N_a) + O(SGen) \tag{27}$$

where $O(SGen) = C_2$ is the complexity of the $SGen$ routine. This complexity is computed as follows. The SCE engine computes the schedule starting from a given mapping. This can be performed in $O\left(N_a \log N_a + N_a * \pounds\right)$ (ref. [32]) where $\pounds$ is the average number of successors of a task. The *reliability gradient* can be computed in $O\left(N_a + N_p\right)$. Assuming the outer while loop executes for χ times on average, the complexity of Algorithm 2 is

$$\begin{aligned} C_2 &= \chi N_a N_p N_f \left(O(SCE) + O(calcRG)\right) \\ &= \chi N_a N_p N_f \left(N_a \log N_a + N_a * \pounds + N_a + N_p\right) \\ &= O\left(\chi N_a^2 N_p N_f (\log N_a + \pounds) + \chi N_a N_p^2 N_f\right) \end{aligned} \tag{28}$$

Combining Equations 27 and 28, the overall complexity of Algorithm 1 is given by

$$\begin{aligned} C_1 &= O\left(\eta \times N_a \times C_2\right) \\ &= O\left(\chi \eta N_a^2 N_p N_f \left(N_a \log N_a + N_a \pounds + N_p\right)\right) \end{aligned} \tag{29}$$

B. Transient and permanent fault reliability trade-offs

Figure 5 plots the mean time to permanent failures and mean time between transient failures for different design solutions (application mapping) obtained from the proposed design space exploration for H.263 Decoder application. The transient fault rate assumed for this work is 300 faults every 10^6 sec [36]–[38]. The mean time to failure requirements for both faults are shown in the figure by solid lines.

Point A in this figure corresponds to task mapping obtained using the permanent fault preventive technique of [10] incorporating transient fault-tolerance in the form of task duplication. As can be seen, the permanent fault aware task mapping leads to a low transient fault reliability (no checkpoints) implying a low mean time between transient faults. On the other hand, the transient fault aware solution from [12] selects point B in the figure. Although mapping corresponding to point B satisfies the performance requirement (throughput/makespan), the platform lifetime resulting from same mapping is lower,

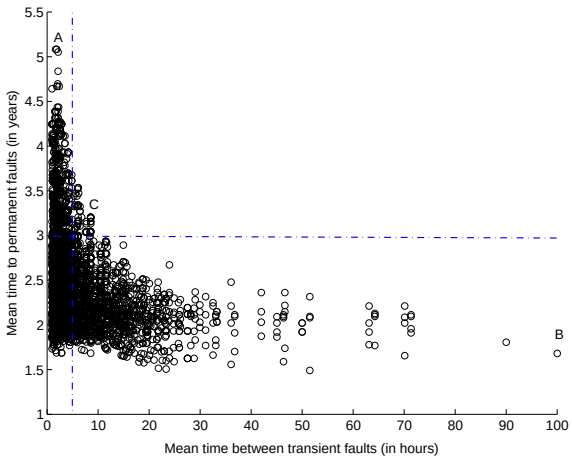

Fig. 5. Mean time to failure for transient and permanent faults for H.263 Decoder

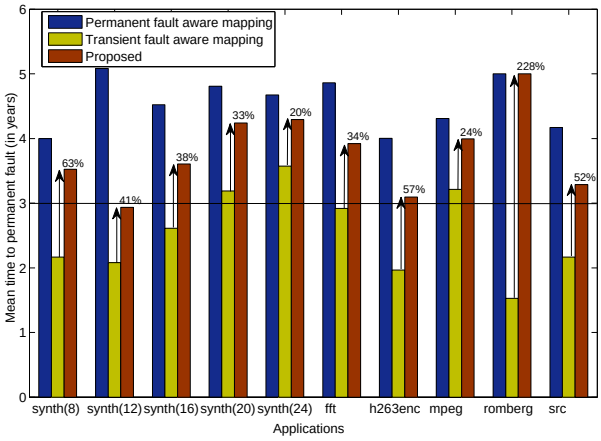

Fig. 6. Mean time to permanent failure for 10 applications

leading to violation of mean time to permanent faults requirement. The proposed $GDSE$ selects the point marked C in the figure. This mapping satisfies the permanent and transient fault requirements and results in minimum *reliability gradient* i.e. the minimum degradation of reliability considering permanent faults for a maximum increase in the reliability considering transient faults. Further, this mapping improves the mean time to permanent faults by more than 100% (2x improvements) and satisfies the mean time to failure requirement for both transient and permanent faults. Although not shown explicitly, this trend is observed for 90% (45 out of 50) applications considered. For two of the remaining five applications, the mapping corresponding to points B and C both violate useful life requirement while satisfying the mean time between transient fault requirements. For the three remaining applications, the points B and C both satisfy the mean time to transient and permanent fault requirement. The important conclusion to draw from this figure is that if aging is not considered to determine the number of checkpoints for transient fault tolerance, some processors can age faster than others leading to a significant reduction in platform lifetime.

C. Mean time to permanent fault performance

Figure 6 plots the mean time to permanent faults of ten (out of fifty considered) different applications for two of the existing category of research in comparison to the proposed solution. The permanent fault preventive technique of [29]

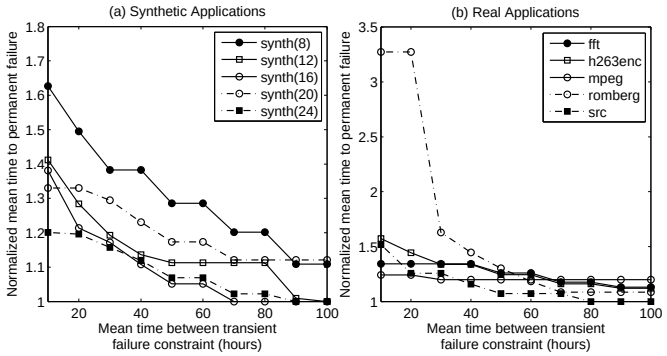

Fig. 7. Normalized mean time to permanent fault with varying transient fault-tolerance constraint

and the transient fault preventive technique of [12] are used for comparison. The transient fault tolerance requirement for the application is set as at most one fault every ten hours of execution and the fault arrival rate is set to one fault every two hours. The platform lifetime requirement is set as 3 years and is shown as the dark solid line. There are five synthetic and five real-life applications considered. The synthetic applications are labeled as *synth(n)* where n denotes the number of tasks of the application; the five real life applications are *fft, H.263 Encoder, MPEG, Romberg Integration* and *Sample Rate Converter*. These ten applications are executed on a homogeneous multiprocessor systems consisting of four cores with a fixed reconfigurable area of 100 columns.

There are a few trends to observe from this figure. First of all, both the proposed and the transient fault-tolerant technique lead to a reduction in mean time to failure (permanent) as compared to the permanent fault preventive technique. This is expected due to the conflicting nature of the two fault types as established in Section III. Secondly, for applications *synth(20)*, *synth(24)* and *mpeg*, both the proposed and the transient fault aware techniques satisfy the useful life requirement whereas for the remaining eight applications, the transient technique violates the useful life requirement. It is important to note that for the synthetic application *synth(12)* both the proposed and the existing technique violate the useful life requirement. However, the lifetime from the proposed technique is 41% better than the existing technique. Thirdly, the mean time to failure obtained using the proposed technique is higher than the transient technique by 20% to 228% for all applications considered with an average improvement of 60%. Finally, the lifetime of the proposed technique is within an average 15% of the maximum lifetime obtained using the permanent fault aware technique. For some of the applications such as *romberg integration*, the proposed and the permanent fault aware mapping technique yield similar result. Another important point to note for this application is that, the lifetime using the proposed technique is 228% better than the transient fault tolerant technique. There are two factors contributing to this effect. First of all, the transient fault-tolerant technique selects a higher number of checkpoints which results in an increase in the expected length of each task of the application. Secondly, the aging unaware mapping obtained from this technique consists of more cycles where all the processors of the system are operational resulting in an increase in temperature. Both these factors negatively impact processor aging.

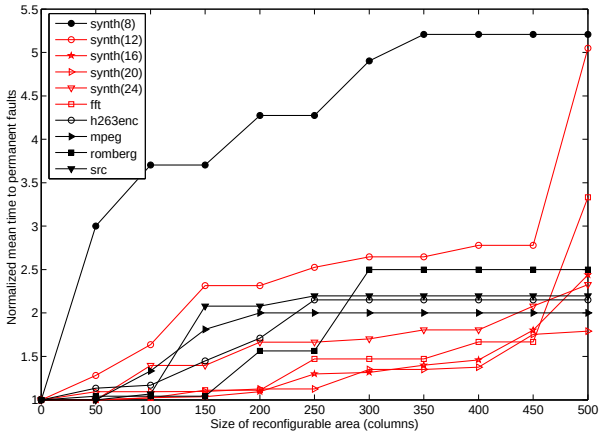

Fig. 8. Mean time to permanent fault size of reconfigurable area trade-off

D. Permanent fault-tolerance with varying constraint for transient fault-tolerance

Figure 7 plots the normalized mean time to permanent faults obtained using the proposed technique with varying transient fault-tolerance constraint for all the ten applications considered. The mean time to permanent fault obtained using the proposed technique is normalized with respect to that obtained using the transient fault-tolerant technique. The transient fault tolerance constraint is measured as the mean time between two transient faults and is varied from 1 fault every 10 hours to 1 fault every 100 hours. This range represents the varying reliability requirement of safety and non-safety critical applications. The trend to observe from these figures is that as the transient fault tolerance becomes more and more stringent (higher fault tolerance constraint), the normalized mean time to permanent fault drops. This is expected due to conflicting nature of the requirement for permanent and transient faults. For critical applications where high reliability is desired (e.g. 1 fault in every 100 hours of operation), the mean time to permanent fault obtained using the proposed technique is close to that obtained using the transient fault-tolerant technique. For some of the applications such as *synth(12), synth(16), synth(24)* and *src* the proposed and the existing technique yields similar result in terms of mean time to permanent faults. For all the remaining six applications, the proposed technique performs better. On average for all fifty applications considered, the proposed technique outperforms (in terms of platform lifetime) the existing technique by 10% even at a high transient fault rate requirement of 1 fault every 100 hours. On the other end, for less reliability requirement of 1 fault every 10 hours, the proposed technique provides an average 60% lifetime improvement. These results clearly suggest the importance of considering processor aging in the checkpoint selection for transient fault-tolerance.

E. Reliability and reconfigurable area trade-off

Figure 8 plots the normalized mean time to permanent fault of the ten previous applications as the size of the reconfigurable area is increased. The mean time to permanent fault obtained using the proposed $GDSE$ algorithm for an application with reconfigurable area is normalized with respect to the mean time to permanent fault obtained when no reconfigurable area is available in the system. As can be seen from the figure, the mean time to failure or in other words the system reliability improves with increase in the size of reconfigurable area. This is expected as higher the availability of reconfigurable area, more the number of tasks that can be mapped to the same with an overall reduction of time spent on the GPPs. This has positive impact on the temperature and hence on reliability. The second point to note from the figure is that the improvement saturates beyond a certain size. This is due to the limited lifetime improvement possible after remapping most of the tasks of an application. For some of the applications such as *src* and *h263enc*, the saturation point is at lower size of reconfigurable area. These applications are marked in the figure by black solid lines. For other applications, the saturation point is beyond 500 columns. The third key point is that for some of the applications such as *synth(16), synth(20)* and *fft*, the improvement of lifetime is less upto a reconfigurable area size of 300 columns. The improvement is observed as the reconfigurable area size is increased beyond 300 columns. For other applications such as *synth(8), synth(12)* and *src*, lifetime improvement is possible even with small reconfigurable area. The important conclusion to derive from these results is that different applications exhibit different trade-off with respect to reconfigurable area and lifetime performance. It is essential to characterize each application during design phase to explore such trade-off. Such knowledge can be applied at run-time during application mapping and reconfigurable area distribution among multiple simultaneous applications. As an example, if the reconfigurable area available at a given time during operation is 100 columns and application *fft* and *src* needs to be mapped, it is better to reserve the reconfigurable area for *src* which provides significant improvement to lifetime than *fft*.

TABLE I
EXECUTION TIME (IN SEC) OF THE PROPOSED $GDSE$

Tasks	RA size = 100				RA size = 300			
	cores = 2	cores = 4	cores = 6	cores = 8	cores = 2	cores = 4	cores = 6	cores = 8
8	50	75	100	125	90	145	150	160
16	150	670	720	775	500	1,140	2,140	3,000
24	785	1,860	3,300	5,575	2,580	7,100	12,150	13,560
32	1,140	3,020	5,130	6,790	2,850	7,500	12,700	14,180

F. Execution time performance

Table I reports the execution time of the proposed $GDSE$ algorithm as the number of tasks and processors are scaled for reconfigurable area of 100 and 300 columns. As can be seen, the execution time increases with increase in the number of tasks and processors. However, the time growth can be accommodated as the analysis are performed at design time. Moreover, with increase in the size of the reconfigurable area, the run-time also increases. Here, two factors are coming into effect. First of all, with increase in the reconfigurable area size, the number of iterations of the outer while loop (lines 2-10) of Algorithm 1 increases. However at each iteration, the number of tasks to be analyzed i.e. the iterations of Algorithm 2 reduces as more and more tasks are marked as hard tasks.

A point to note is that the optimization problem formulated in Sections IV and V can be solved using standard solver e.g. CPLEX. However, the execution time grows exponentially with the number of tasks and processors. The solver fails to provide results beyond 8 tasks mapped on 6 processors even after running for more than five hours. For applications for which the optimization terminates within the given time frame of five hours, the proposed $GDSE$ algorithm provides upto $500\times$ reduction in execution time (average of $200\times$ for all applications considered).

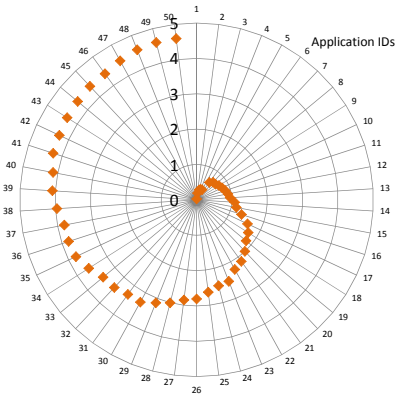

Fig. 9. Variation from optimal point for different applications

G. Distance from optimality

For determining the closeness of the proposed technique to the optimal solution, execution time is restricted to five hours. A separate set of fifty synthetic applications are generated with the number of tasks varying from four to eight (so that the solver terminates within five hours). The mean time to permanent fault obtained using the proposed $GDSE$ algorithm is compared with that obtained from the solver. Figure 9 plots the distance (measured as percentage deviation) of the proposed algorithm from the optimal solution. As can be seen, only thirteen out of fifty applications deviate from optimality with the maximum deviation of 4.5%. For the remaining thirty seven applications, the mean time to fault obtained from the solver and the proposed technique are same or comparable.

VIII. Conclusions

In this paper, a gradient-based technique is proposed to improve the lifetime of a homogeneous reconfigurable multiprocessor system while optimizing for transient fault tolerance. Experiments conducted with variable fault-tolerance requirement demonstrate that the proposed solution improves lifetime by 10% to 60% as compared to the state-of-art transient fault-tolerant technique. The gradient-based technique provides upto $500\times$ reduction in design space exploration time with less than 5% distance from optimality. Exploring run-time dynamic reconfiguration feature of modern FPGA devices and consideration of heterogeneous processors are left as future works.

Acknowledgment

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

References

[1] A. Jerraya and W. Wolf, *Multiprocessor systems-on-chips*. Morgan Kaufmann, 2004.
[2] L. Chen and T. Mitra, "Shared reconfigurable fabric for multi-core customization," in *Design Automation Conference (DAC)*, 2011.
[3] M. Shafique, L. Bauer, W. Ahmed, and J. Henkel, "Minority-game-based resource allocation for run-time reconfigurable multi-core processors," in *Conference on Design, Automation and Test in Europe (DATE)*, 2011.
[4] L. Jiashu, A. Das, and A. Kumar, "A design flow for partially reconfigurable heterogeneous multi-processor platforms," in *International Symposium on Rapid System Prototyping (RSP)*, 2012.
[5] D. Gohringer, M. Hubner, M. Benz, and J. Becker, "A design methodology for application partitioning and architecture development of reconfigurable multiprocessor systems-on-chip," in *Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2010.
[6] M. Kim, S. Banerjee, N. Dutt, and N. Venkatasubramanian, "Energy-aware cosynthesis of real-time multimedia applications on mpsocs using heterogeneous scheduling policies," *ACM Transactions on Embedded Computing (TECS)*, 2008.
[7] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, 2003.
[8] J. W. McPherson, "Reliability challenges for 45nm and beyond," in *Design Automation Conference (DAC)*, 2006.
[9] J. Teich, "Hardware/Software Codesign: The Past, the Present, and Predicting the Future," *Proceedings of the IEEE*, 2012.
[10] A. Hartman, D. Thomas, and B. Meyer, "A case for lifetime-aware task mapping in embedded chip multiprocessors," in *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010.
[11] V. Izosimov, I. Polian, P. Pop, P. Eles, and Z. Peng, "Analysis and optimization of fault-tolerant embedded systems with hardened processors," in *Conference on Design, Automation and Test in Europe (DATE)*, 2009.
[12] P. Saraswat, P. Pop, and J. Madsen, "Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.
[13] A. Jhumka, S. Klaus, and S. Huss, "A dependability-driven system-level design approach for embedded systems," in *Conference on Design, Automation and Test in Europe (DATE)*, 2005.
[14] J. Huang, J. Blech, A. Raabe, C. Buckl, and A. Knoll, "Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems," in *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011.
[15] C.-L. Chou and R. Marculescu, "FARM: Fault-aware resource management in NoC-based multiprocessor platforms," in *Conference on Design, Automation and Test in Europe (DATE)*, 2011.
[16] C. Bolchini and A. Miele, "Reliability-Driven System-Level Synthesis of Embedded Systems," in *Symposium on Defect and Fault Tolerance in VLSI Systems (DFTS)*, 2010.
[17] M. Glaß, M. Lukasiewycz, T. Streichert, C. Haubelt, and J. Teich, "Reliability-aware system synthesis," in *Conference on Design, Automation and Test in Europe (DATE)*, 2007.
[18] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, and W.-L. Hung, "Reliability-centric hardware/software co-design," in *International Symposium on Quality Electronic Design (ISQED)*, 2005.
[19] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Reliability-aware co-synthesis for embedded systems," *Journal of VLSI Signal Processing*, 2007.
[20] J. Yin, H. Song, L. Yuan, and Q. Cui, "A real-time fault-tolerant scheduling algorithm for software/hardware hybrid tasks," in *Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, 2011.
[21] J. Kim, K. Lakshmanan, and R. Rajkumar, "R-BATCH: Task Partitioning for Fault-tolerant Multiprocessor Real-Time Systems," in *Conference on Computer and Information Technology (CIT)*, 2010.
[22] C. Bolchini, A. Miele, and C. Sandionigi, "A novel design methodology for implementing reliability-aware systems on sram-based fpgas," *IEEE Transactions on Computers*, vol. 60, no. 12, 2011.
[23] P. Axer, M. Sebastian, and R. Ernst, "Reliability analysis for mpsocs with mixed-critical, hard real-time constraints," in *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011.
[24] C. Krishna and A. Singh, "Reliability of checkpointed real-time systems using time redundancy," *IEEE Transactions on Reliability*, 1993.
[25] S.-W. Kwak, B.-J. Choi, and B.-K. Kim, "An optimal checkpointing-strategy for real-time control systems under transient faults," *IEEE Transactions on Reliability*, 2001.
[26] M. Väyrynen, V. Singh, and E. Larsson, "Fault-tolerant average execution time optimization for general-purpose multi-processor system-on-chips," in *Conference on Design, Automation and Test in Europe (DATE)*, 2009.
[27] A. Das, A. Kumar, and B. Veeravalli, "Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems," in *Conference on Design, Automation and Test in Europe (DATE)*, 2013.
[28] L. Huang, F. Yuan, and Q. Xu, "Lifetime reliability-aware task allocation and scheduling for MPSoC platforms," in *Conference on Design, Automation and Test in Europe (DATE)*, 2009.
[29] A. Das, A. Kumar, and B. Veeravalli, "Communication and migration energy aware design space exploration for multicore systems with intermittent faults," in *Conference on Design, Automation and Test in Europe (DATE)*, 2013.
[30] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, 1987.
[31] A. Das, A. Kumar, and B. Veeravalli, "Energy-aware task mapping and scheduling for reliable embedded computing systems," National University of Singapore, Tech. Rep., 2013.
[32] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," in *Workshop on Hardware/software Codesign (CODES)*, 1998.
[33] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "Noc synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2005.
[34] 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.
[35] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.
[36] H. Liu, M. Cotter, S. Datta, and V. Narayanan, "Technology assessment of si and iii-v finfets and iii-v tunnel fets from soft error rate perspective," in *International Electron Devices Meeting (IEDM)*, 2012.
[37] G. Lu, Z. Zheng, and A. A. Chien, "When is multi-version checkpointing needed?" in *Workshop on Fault-tolerance for HPC at extreme scale (FTXS)*, 2013.