Artificial Intelligence Based Task Mapping and Pipelined Scheduling for Checkpointing on Real Time Systems with Imperfect Fault Detection

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*—Fault-tolerance is emerging as one of the important optimization objectives for designs in deep submicron technology nodes. This paper proposes a technique of application mapping and scheduling with checkpointing on a multiprocessor system to maximize the reliability considering transient faults. The proposed model incorporates checkpoints with imperfect fault detection probability, and pipelined execution and cyclic dependency associated with multimedia applications. This is solved using an Artificial Intelligence technique known as *Particle Swarm Optimization* to determine the number of checkpoints of every task of the application that maximizes the confidence of the output. The proposed approach is validated experimentally with synthetic and real-life application graphs. Results demonstrate the proposed technique improves the probability of correct result by an average 15% with imperfect fault detection. Additionally, even with 100% fault detection, the proposed technique is able to achieve better results (25% higher confidence) as compared to the existing fault-tolerant techniques.**

I. Introduction

To accommodate the ever increasing computing demand and to address scalability, multiple processing elements (PEs) are interconnected using an interconnection network (such as bus, networks-on-chip, etc.) to form multiprocessor systems-on-chip (MPSoCs) [1]. Shrinking transistor geometries, increasing transistor density, and aggressive voltage scaling are negatively impacting the dependability of these processing elements and the communication backbone by increasing the probability of faults [2][1]. Such faults can be classified as transient, intermittent and permanent. This paper deals with transient faults, which are temporary deviation of a circuit's output from its reference or expected output. These faults are caused by single-event upsets due to particle strikes, electrical noises, cross talks, or other environmental effects. A traditional technique of transient fault-tolerance is to replicate some (or all) the tasks of an application, with the different replicas of a task being executed on different processing elements [3]–[5]. Stringent area and power budget is prohibiting the use of redundancy in today's MPSoCs. One economic transient fault-tolerant technique is **checkpointing** [6]–[13]. In this process, the state of an executing task (known as **checkpoint**) is periodically saved to a local or remote memory at regular (or irregular) time intervals (known as **checkpoint intervals**). When a failure is detected, the process rolls back to the last valid checkpoint and resumes execution. Thus, checkpointing allows task to progress execution in spite of failures.

Mapping and scheduling of real-time applications on multiprocessor platforms have been widely studied in literature focusing on independent tasks as well as on tasks with precedence constraints. These applications are characterized by performance requirement, the violation of which is undesirable. Multimedia applications constitute a significant fraction of these real-time applications. This class of application is often associated with a throughput requirement, violation of which impacts the quality-of-service provided to end users. **Pipelining** is a technique to exploit temporal parallelism associated with these applications. Prior studies have confirmed that pipelined scheduling of multimedia applications offer a distinctive performance improvement over sequential scheduling, and is therefore adopted as the scheduling technique for this work. Determining the optimal number of checkpoints for real-time applications that maximizes reliability while satisfying the throughput requirement is an NP-hard problem. Significant attention has been drawn over the past decades to solve this problem efficiently. Most of these techniques have focused on independent tasks [6]–[10]. Although there are techniques for dependent tasks, such as [11]–[13], they suffer from the following limitations.

First, in all the existing techniques, 100% fault detection is assumed i.e., a fault is assumed to be detected completely at the end of the checkpoint interval. In these models, a fault always triggers the re-execution of the checkpoint segment. This assumption is optimistic in determining the overhead in execution time of a task with checkpoints, and leads to sub-optimal scheduling for scenarios with imperfect fault detection. On the contrary, this paper analyzes the execution time of a task incorporating the probability of fault detection in computing the overhead, and thus provides a realistic bound on the schedules obtained. Second, all existing techniques on checkpoint-based fault-tolerance consider **directed acyclic graphs (DAGs)** with sequential application execution. Multimedia applications require modeling cyclic dependency, multi-input tasks, multi-rate tasks, and pipelined execution. In this work, we consider **synchronous data flow graphs (SDFGs)** that is generic to represent cyclic and acyclic dependencies. This makes the proposed approach applicable for real-time multimedia as well as non-multimedia applications.

Contributions: To summarize, this paper formulates the task mapping and scheduling problem to determining the optimal number of checkpoints for the constituent tasks with imperfect fault detection. An Artificial Intelligence (AI) based meta-heuristic technique called **Particle Swarm Optimization (PSO)** is utilized to solve the same. Unlike Genetic Algorithms, PSO does not rely on mutations of population, but rather on the social behavior of the individuals. Following are the key contributions in this respect.

1. A checkpoint-based transient fault-tolerance approach

[1]This paper considers malignant faults i.e., the faults that are manifested as errors in the system.

considering imperfect fault detection;
2. A PSO-based heuristic to solve the formulated problem;
3. A task mapping and scheduling problem considering SDFG that allow modeling cyclic dependency, multi-input tasks, multi-rate tasks, and pipelined execution.

Experiments conducted on a set of synthetic and real-life SDFGs demonstrate that the proposed approach is able to improve the reliability of an MPSoC by an average 15% by exploiting the performance slack with imperfect fault detection. Additionally, even with 100% fault detection, the proposed approach is able to improve the confidence of the produced result by 25% as compared to the existing techniques.

The rest of this paper is organized as follows. A brief introduction to the related works is presented in Section II. This is then followed by preliminaries on checkpointing with imperfect fault detection in Section III and the problem formulation in Section IV. The PSO-based heuristic is introduced in Section V. Experimental results are presented in Section VI and the paper is concluded in Section VII.

II. Related Works

Transient faults have received significant attention in recent years due to their adverse effects in deep sub-micron technology nodes. Several techniques exist in literature for transient fault-tolerance, such as replication [3]–[5], checkpointing [6]–[13] and rollback recovery [11]–[13]. We consider checkpoint-based transient fault-tolerance, which can be classified into two categories – online and offline. Online checkpointing techniques select the number and interval of the checkpoints in an adaptive manner [14]. Offline techniques on the other end adopt a task-centric view where the number and interval of checkpoints for each task are determined offline. Offline techniques can be further classified into techniques dealing with independent [6]–[10] and dependent tasks [11]–[13]. The techniques with independent tasks determine the number of checkpoints for every tasks such that reliability is maximized while satisfying their deadline requirements (for real-time tasks). However, these techniques result in sub-optimal solutions for applications with dependent tasks, since optimum number of checkpoints for individual tasks do not guarantee reliability optimality for the entire application. Hence, references [11]–[13] are discussed in details. A Tabu Search heuristic is proposed in [11] to statically schedule the tasks of an application on a multiprocessor system. Several policies are proposed, such as task re-execution, replication or checkpointing. The heuristic selects the one that maximizes the reliability of application execution. The proposed technique selects sub-optimal checkpoints for the tasks to determine the global reliability optimum point for the entire application. A similar approach is proposed in [12] for mixed critical applications. A reliability analysis technique is proposed in [13] to determine the schedulability of tasks with dependency considering checkpoint-based transient fault-tolerance. Table I summarizes the existing works.

III. Checkpointing

This section formulates the checkpointing problem with imperfect transient fault detection for a single task. This is later extended to multiple tasks of a real-time application. 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_s
3. time for recovery from a checkpoint, T_r
4. fault arrival rate, λ

TABLE I
Summary of related Works

Related Works	Throughput Degradation	Imperfect Fault Detection	Application Model
Huang et al. [5]	×	√	DAGs
Pop et al. [11]	×	×	DAGs
Saraswat et al. [12]	×	×	DAGs
Axer et al. [13]	√	×	DAGs
Proposed	√	√	SDFGs

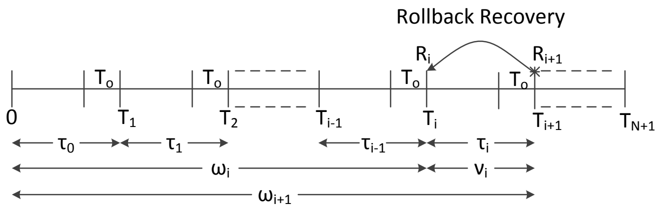

Fig. 1. Checkpointing of a task

Following are the assumptions regarding checkpointing-based transient fault-tolerance similar, to [6]–[13].

- 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 system rolls back to the last valid checkpoint as soon as a fault is detected.
- The probability of multiple transient faults in each checkpoint segment is negligible.

A. *Confidence of Result with Imperfect Fault Detection*

Figure 1 shows an example task execution with N checkpoints. Let X be a random variable denoting the interval between two successive transient faults. The probability density function (PDF) of X is $f_X(t) = \lambda e^{-\lambda t}$. Let τ_i define the i^{th} checkpoint interval (time between i^{th} and $i+1^{th}$ checkpoint) composed of the actual computation (denoted by I_i) and the checkpoint overhead ($T_o = T_s + T_r$). The probability of failure during this interval τ_i is $F_i = Prob[X \leq \tau_i] = 1 - e^{-\lambda \tau_i}$. Let D denote the probability of fault detection[2] i.e., ($0 \leq D \leq 1$), and R_i the confidence of the output result i.e., the probability of reliable result at the beginning of τ_i. Then, the reliability at the beginning of τ_{i+1} is the probability $P(A \cup B)$, where A is the event that the result is reliable at the beginning of τ_i and no fault occurs and B is the event that faults occur in the interval and gets detected triggering a restart recovery.

$$R_{i+1} = R_i(1 - F_i) + R_{i+1} F_i D \qquad (1)$$

The above equation can be rewritten as

$$R_{i+1} = \frac{R_i(1 - F_i)}{1 - F_i D} \qquad (2)$$

The above equation is used recursively to determine the probability of correct result with N checkpoints (E_{N+1}) with the initial condition E_0 set to zero.

[2]D = 1 implies 100% fault detection.

B. Execution Time with Checkpoints

Let Y_i denote the actual computation with one or more faults in interval τ_i. The probability density function of Y_i is

$$f_{Y_i}(t) = \frac{f_X(t)}{F_i} = \frac{\lambda e^{-\lambda t}}{1 - e^{-\lambda \tau_i}} \tag{3}$$

The mean of Y_i (denoted as $\bar{Y_i}$) is

$$\bar{Y_i} = \int_0^{\tau_i} t f_{Y_i}(t) dt = \frac{1}{\lambda} - \frac{\tau_i e^{-\lambda \tau_i}}{1 - e^{-\lambda \tau_i}} \tag{4}$$

Let ω_i denote the execution time from time $t = 0$ to the completion of the i^{th} checkpoint. Thus, $\omega_{i+1} = \omega_i + \nu_i$, $\forall i \in [0, N]$ (refer Figure 1), where ν_i is the length of the checkpoint interval τ_{i+1} determined probabilistically i.e.,

$$\nu_i = \begin{cases} \tau_i & \text{with probability } (1 - F_i D) \\ \bar{Y_i} + T_o + \nu_i & \text{with probability } F_i D \end{cases} \tag{5}$$

Using Equation 5 the following solution can be derived,

$$\omega_{i+1} = \omega_i + \tau_i + \frac{F_i D(\bar{Y_i} + T_o)}{1 - F_i D} \tag{6}$$

The above equation can be used recursively to determine the mean execution time with N checkpoints as ω_{N+1} with initial condition ν_0 equal to 0. Clearly, the checkpoint intervals are non uniform and is dependent on the fault coverage. A point to note here is that when real-time applications are considered with multiple tasks, a superscript (indicating the task id) is introduced on ω_{N+1} and R_{N+1}.

C. Difference with Existing Checkpointing Techniques

The existing works on checkpoints assume the following – perfect fault detection (i.e. $D = 1$) and equidistant checkpoints (i.e. $\omega_i = i * \omega$ where ω is the average length of each segment); both are difficult to achieve in reality. This work overcomes these restrictions by considering imperfect fault detection and non-uniform checkpoint intervals.

IV. Problem Formulation

A. Introduction to Synchronous Data Flow Graphs

Synchronous Data Flow Graphs (SDFGs, see [15]) are often used for modeling modern DSP applications [16] and for designing concurrent multimedia applications implemented on multiprocessor systems. Both pipelined streaming and cyclic dependencies between tasks can be easily modeled in SDFGs. SDFGs allow analysis of a system in terms of throughput and other performance properties e.g., latency and buffer requirements [17].

The nodes of an SDFG are called **actors**; they represent functions that are computed by reading **tokens** (data items) from their input ports and writing the results of the computation as tokens on the output ports. The number of tokens produced or consumed in one execution of an actor is called port **rate**, and remains constant. The rates are visualized as port annotations. Actor execution is also called **firing**, and requires a fixed amount of time, denoted with a number in the actors. The edges in the graph, called **channels**, represent dependencies among different actors.

Figure 2 shows an example of a SDFG. There are three actors in this graph. In the example, $\boldsymbol{a}_0$ has an input rate of 1 and output rate of 2. An actor is called **ready** when it has sufficient input tokens on all its input edges and sufficient buffer space on all its output channels; an actor can only fire when it is ready. The edges may also contain **initial tokens**, indicated by bullets on the edges, as seen on the edge from actor $\boldsymbol{a}_2$ to $\boldsymbol{a}_0$ in Figure 2. Formally, an SDFG is a directed graph $\mathcal{G}_{app} = (A, C)$ consisting of a finite set A of actors and a finite set C of channels.

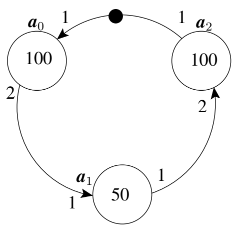

Fig. 2. Application SDFG.

One of the most interesting properties of SDFGs relevant to this thesis is throughput. Throughput is defined as the inverse of the long term period i.e., the average time needed for one iteration of the application. (An iteration is defined as the minimum non-zero execution such that the original state of the graph is obtained.) This is the performance parameter used in this thesis. The following properties of an SDFG are defined.

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

Definition 2: (Application Period) *Application Period* Per(A) *is defined as the time SDFG,* $\mathcal{G}_{app} = (A, C)$ *takes to complete one iteration on an average.*

The period of an SDFG can be computed by analyzing the maximum cycle mean (MCM) of an equivalent homogeneous SDFG (HSDFG). The period thus computed gives the minimum period possible with infinite hardware resources e.g. buffer space. If worst-case execution time estimates of each actor are used, the performance at run-time is guaranteed to meet the analyzed throughput. Self-timed strategy is widely used for scheduling SDFGs on multiprocessor systems. In this technique, the exact firing of an actor on a core is determined at design-time using worst-case actor execution-time. The timing information is then discarded retaining the assignment and ordering of the actors on each core. At run-time, actors are fired in the same order as determined from design-time. Thus, unlike fully-static schedules, a self-timed schedule is robust in capturing the dynamism in actor execution time. The self-timed execution of an SDFG consists of a transient phase followed by a periodic or steady-state phase [18].

A point to note is that, the conversion of an application to an equivalent SDFG is not the focus of this paper. Interested readers can refer to [19]. Instead, this paper focuses on the reliability optimization methodology as outlined below.

An SDFG is mathematically represented as a directed graph $G_{app} = (V_{app}, E_{app}, T_c)$, where V_{app} is the set of nodes representing tasks of the application, E_{app} is the set of edges $\{e_{ij} \mid 1 \leq i, j \leq |V_{app}|\}$, representing data dependency among tasks and T_c is the throughput constraint of G_{app}. Let $n_{app} = |V_{app}|$. Each task $v_i \in V_{app}$ is a tuple $\langle t_i, D_i \rangle$, where t_i is the execution time of v_i and D_i is the data produced at every iteration of v_i. D_i is the set $\{d_{ij} \mid 1 \leq j \leq |V_{app}|\}$, representing the data produced on edge e_{ij}.

The architecture model for this work consists of homogeneous cores interconnected in a mesh-based topology. Such an architecture is represented as a graph $G_{arc} = (V_{arc}, E_{arc})$, where V_{arc} is the set of nodes representing processors and E_{arc} is the set of edges representing communication channels among the processors. Let $n_{arc} = |V_{arc}|$.

Denoting $R^k_{N_c(k)}$ as the probability of erroneous output of task v_k with $N_c(k)$ checkpoints, the overall optimization problem becomes

Minimize $\prod_{k=1}^{n_{app}} R^k_{N_c(k)}$
Subject to

- All tasks meet their respective deadlines
- All control/data dependencies are satisfied
- Application throughput constraint is satisfied

where $n_{app} = |V_{app}|$ is the number of tasks of application.

V. Optimization Technique

The proposed optimization problem formulation can be solved using standard solvers. However, the execution time grows super exponentially with the number of tasks and cores. This paper proposes a fast heuristic to solve the optimization problem using particle swarm optimization to determine the number of checkpoints for each task that maximizes the confidence of the result under imperfect fault detection while satisfying performance.

A. Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an artificial intelligence technique [20] influenced by the social behavior of animals, such as flocking of swarms and schooling of fish. Since its introduction, PSO has been used extensively in solving linear and nonlinear optimization problems. A **PSO** system is characterized by the existence of a number of **particles**, analogous to **populations** in genetic algorithms. However, unlike the genetic algorithm, PSO does not rely on **mutation** of population, but rather on the **movement** of the particle swarms, which is coordinated by its position and velocity. In the quest for an optimal landing position, a particle adjusts its position according to its local experience (best position of the particle, $pbest$) and global experience (best position by the flock of particles, $gbest$). A **fitness function** is used to evaluate the metric of interest corresponding to the positions of the particles. Thus, each particle has a **fitness** value assigned to it at every generation of the algorithm. The **velocity** and the **position** of a particle are determined according to the following equations (refer [20]).

$$\begin{aligned} V^{k+1}(i) &= W * V^k(i) + c_1 * rnd_1 \left(pbest_i - Pos^k(i)\right) \\ &\quad + c_2 * rnd_2 \left(gbest - Pos^k(i)\right) \qquad (7) \\ Pos^{k+1}(i) &= Pos^k(i) + V^{k+1}(i) \end{aligned}$$

where $V^k(i)$ and $Pos^k(i)$ are respectively, the velocity and position of i^{th} particle in generation k, W is the inertial weight, c_i, c_2 are the acceleration coefficients, and rnd_1 and rnd_2 are random numbers in $[0, 1]$.

B. Mapping Problem in PSO System

Figure 3 shows an example transformation of a task mapping instance to a position in the PSO system. The problem consists of mapping three tasks (denoted as v_1, v_2, v_3) to an architecture consisting of two cores c_1 and c_2. A mapping in the task mapping domain corresponds to a position in the PSO System. The number of dimensions of the PSO system is equal to the number of tasks (three in this example). The cores to which the tasks are mapped, form the coordinate of the particle's position. As an example, the three tasks in the first mapping are mapped to c_1, c_1 and c_2, respectively. This corresponds to $(1, 1, 2)$ in the PSO coordinate system. The particle moves to its new position $(2, 1, 2)$ in accordance with Equation 7. This corresponds to the new mapping where the three tasks are mapped to c_2, c_1 and c_2, respectively.

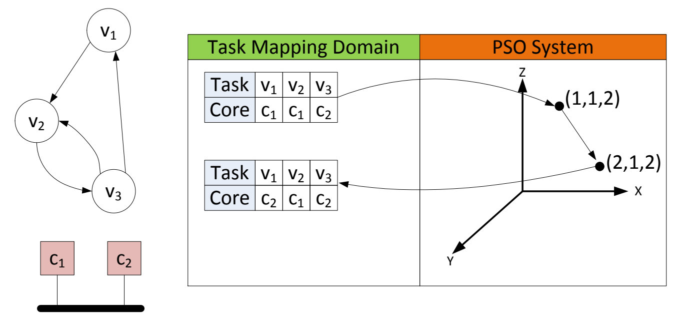

Fig. 3. Transforming to PSO system

ALGORITHM 1: $FEval$: Fitness Evaluation

```
Input: Application and architecture graph $G = [G_{app}, G_{arc}]$ and mapping $M$
Output: Fitness value $R_b$
1  $N_c[1 : n_{apps}] = 0$ and $\delta = SDF^3(G, M) - T_c$;
2  while $\delta > 0$ do
3      for $v_i \in G_{app}$ do
4          $N_c(i) = N_c(i) + 1$, $t_i = \omega^i_{N_c(i)}$ and determine $E^i_{N_c(i)}$ $\forall i$;
5          $\delta_i = SDF^3(G, M) - T_c$;
6          $R_i = \prod_{i=1}^{n_{app}} \prod_{j=1}^{RV(i)} R^i_{N_c(i)}$ //$RV(i)$ denotes the number of
           invocations of task $v_i$ in one period of the application;
7          Define $grad_i = \frac{R_i}{\delta_i}$ if $\delta > 0$ or $-\infty$ otherwise;
8          $N_c(i) = N_c(i) - 1$;
9      end
10     Determine actor $j$ such that $grad_j$ is maximum;
11     $N_c(j) = N_c(j) + 1$, $t_j = \omega^j_{N_c(j)}$;
12     $R_b = \prod_{i=1}^{n_{app}} \prod_{j=1}^{RV(i)} R^i_{N_c(i)}$;
13     $\delta = SDF^3(G, M) - T_c$;
14 end
```

C. Fitness Evaluation

The reliability evaluation corresponding to a mapping (or equivalently, the position of a particle) is performed by a fitness function, which incorporates an existing tool (SDF^3 [21]) to determine the schedule of an SDFG. The throughput slack from the resultant pipelined scheduling is exploited to selectively insert checkpoints in the tasks. The overall reliability of the output result is determined and returned as fitness value. This is shown as pseudo-code in the Algorithm 1. The algorithm iterates as long as there is throughput slack available (outer while loop lines 2 - 14). At each iteration, the algorithm checks the impact of increasing the number of checkpoints of every task of the set G_{app} to evaluate a metric $grad$, defined as the reliability per unit throughput slack (lines 4 - 8). The $grad$ value is assigned $-\infty$ when the throughput is violated. The algorithm selects the task for which $grad$ is maximum (line 10). The number of checkpoints for the corresponding task is incremented, and reliability and the throughput slack are re-computed (lines 11

ALGORITHM 2: Particle Swarm Optimization

```
Input: Application and architecture graph $G = [G_{app}, G_{arc}]$, set of particles
       $S_p$ and maximum generations $MGen$
Output: Best position $gbest$
1  for $p_i \in S_p$ do
2      $Pos(i, j) = rand(1 : n_{app})$;
3      Initialize $V(i)$ and $pbest_i = Pos(i)$;
4  end
5  $gbest = Pos(j)$ s.t. $FEval\,(G, Pos(j))$ is maximum;
6  while $NGen \leq MGen$ do
7      for $p_i \in S_p$ do
8          Determine $V(i)$ and $Pos(i)$ according to Equation 7;
9          if $FEval\,(G, Pos(i)) > FEval\,(G, pbest_i)$ then
10             $pbest_i = Pos(i)$;
11         end
12     end
13     $gbest_{temp} = pbest_j$ s.t. $FEval(G, pbest_j)$ is maximum;
14     if $FEval(G, gbest_{temp}) > FEval(G, gbest)$ then
15         $gbest = gbest_{temp}$;
16     end
17     $NGen = NGen + 1$;
18 end
```

- 13). The probability of fault and the modified execution time for a task with checkpoints are determined using Equations 2 and 6, respectively.

D. PSO-Based Algorithm

Algorithm 2 provides the pseudo-code of the PSO. The first step in the algorithm is to initialize the dimensions of the particles and their corresponding velocity and best position. The best position in the set is recorded. Following this, the algorithm iterates for a maximum number of generations. At each generation, the position Pos and the velocity V are updated according to Equation 7 and the local and global best are determined. Interested readers can refer to [20] for general PSO algorithm.

VI. Results

Experiments are conducted with synthetic and real-life application SDFGs on Intel Xeon 2.4 GHz server running Linux. Fifty synthetic SDFGs are generated with the number of tasks in each application selected randomly from the range 8 to 100. Additionally, fifteen real-life applications are considered with seven from streaming and the remaining eight from non-streaming domain. The streaming applications are obtained from the benchmarks provided in the SDF^3 tool [21]. These are *H.263 Encoder*, *H.263 Decoder*, *H.264 Encoder*, *MP3 Decoder*, *MPEG4 Decoder*, *JPEG Decoder* and *Sample Rate Converter*. The non-streaming application graphs considered are *FFT*, *Romberg Integration* and *VOPD* from [22] and one application each from *automotive*, *consumer*, *networking*, *telecom* and *office automation* benchmark suite [23]. These applications are executed on an MPSoC consisting of 9 homogeneous cores arranged in a 3×3 mesh-based topology.

A. Complexity Analysis

The complexity of the Algorithm 1 is computed as follows. Let η denotes the average number of times the outer loop is executed. The complexity of Algorithm 1 is therefore

$$C_1 = O\left(\eta \times O\left(SDF^3\right)\right) = O\left(\eta \times n_{app} \times n_{arc}\right) \qquad (8)$$

where the complexity of the SDF^3 tool is given by $O\left(n_{app} \times n_{arc}\right)$ (refer [17]). This fitness evaluation algorithm is called in the main PSO algorithm (Algorithm 2) for each particle in each generation. Therefore, the complexity of Algorithm 2 is

$$C_2 = \left(|S_p| \times MGen \times \eta \times n_{app} \times n_{arc}\right) \qquad (9)$$

Rest of the results section is organized into three sections – comparison of the proposed checkpointing technique with (1) other fault-tolerant techniques considering imperfect fault detection; (2) existing checkpointing techniques with 100% fault detection; and (3) other heuristics (alternative to PSO) to establish its superiority for the problem class.

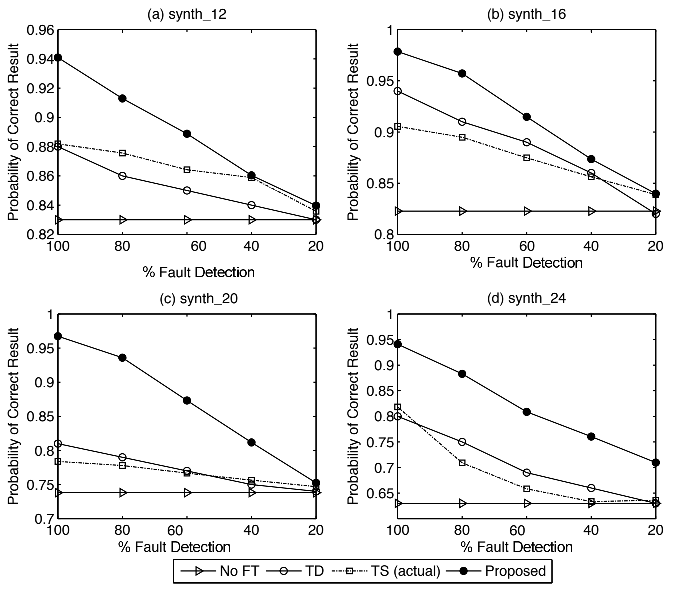

Fig. 4. Performance with varying fault detection

B. Results With Imperfect Fault Detection

Figure 4 plots the performance of the proposed technique with varying fault detection for four synthetic task-graphs selected randomly from the set of 50 graphs considered. The number of tasks in each graph is indicated in the corresponding name of the graph and the results are average of 20 runs. The result of the proposed technique is compared with a technique with no transient fault-tolerance incorporated (indicated as *no FT* in the figure), the technique of task duplication with imperfect fault detection [5] (referred in the figure as *TD*) and the tabu-search based technique of [11] (referred to as *TS*). The *TS* technique assumes 100% fault detection. The reliability results obtained using this technique is multiplied with the fault detection probability to obtain the actual reliability (this is indicated as *TS (actual)* in the figure). The probability of correct result for *no FT* technique is (1 - probability of fault) and is independent of the fault detection. The fault arrival rate (λ) is 10^{-6} (similar to that considered in [3]–[14]). The parameters used for PSO are as follows: number of particles = number of tasks and number of generations = 20. The choice of these parameters are justified in later subsections. It can be seen from the figure that, as the probability of fault detection decreases, there is a decrease in the probability of correct result (for *TS, TD* and proposed). This is expected because with decrease in the probability of fault detection, the number of silent faults (those that are undetected by fault detection mechanisms) increases. Therefore, there is a higher probability

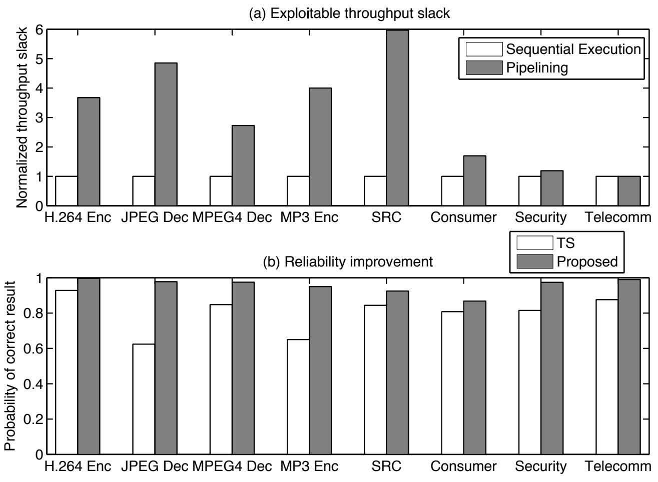

Fig. 6. Performance of the proposed technique

that the output result is erroneous. This trend is consistent with that predicted from Equation 2. The proposed technique achieves 1% to 37% higher reliability than *TD* (average 15% for all 50 applications). In comparison with *TS*, the proposed technique achieves on average 20% higher reliability.

C. Results with Perfect Fault Detection

As established previously, all the existing checkpointing techniques consider 100% fault detection. To establish a fair comparison of the proposed technique with these techniques, the fault detection is assumed to be 100% in the proposed formulation (i.e. $D = 1$). Figure 6 plots the throughput slack and reliability improvement of the proposed technique in comparison to the tabu-search based checkpointing technique of [11] (referred in the figure as *TS*) for 8 applications (5 streaming and 3 non-streaming). Results reported in this figure are the average of 20 runs. A point to note here is that, the *TS* technique is applicable to Directed Acyclic Graphs (DAGs) only and therefore all the streaming SDFGs are first converted to homogeneous SDFG (HSDFG) representation before applying *TS* on them. The conversion of SDFGs to HSDFGs is known to be of exponential complexity and results in an exponential increase in the number of tasks in the resultant HSDFG. Therefore, the proposed technique is the first checkpointing technique for SDFGs.

There are a few trends to follow from this figure. First of all, pipelining results in significant increase in the throughput obtained especially for the streaming applications (*H.264 Enc, JPEG Dec, MPEG4 Dec, MP3 Enc* and *SRC*). This is shown as the increases in the throughput slack (throughput constraint – throughput obtained) in Figure 6(a) with pipelining. On average, for all the streaming applications considered (including those not shown in the figure), pipelining results in an average 3 fold increase in throughput slack as compared to the sequential execution. This is crucial as the slack is exploited for improving the quality of result. For non-streaming applications such as *Consumer, Security* and *Telecomm*, pipelining is not able to exploit higher slacks as these are highly sequential applications. On average, pipelining is able to extract 30% higher throughput slack. Secondly, the reliability (i.e. the probability of correct result) is higher in the proposed technique as compared to *TS*. For streaming applications, the proposed technique improves reliability by 3% to 60% (average 25% for all applications). This improvement can be attributed primarily to the higher slack (due to pipelining) and in parts to PSO-based optimization. For non-streaming applications, the proposed technique increases reliability by 1% to 19% (average 10% for all applications). This improvement can be attributed mostly to the PSO-based technique as the throughput slack improvement for these applications is nominal. The important conclusion to make from these observations is that, the combination of pipelining and PSO-based optimization improves confidence of the produced result significantly for both streaming and non-streaming applications.

D. Convergence of PSO

To establish the suitability of the PSO-based optimization for the problem class, experiments are conducted with the same set of applications using the standard meta heuristics – *Simulated Annealing (SA)*, *Tabu-Search (TSearch)*, *Genetic Algorithm (GA)* and a *greedy search heuristic (GS)* similar to the one proposed in [24]. These heuristics are implemented using the same fitness function as the one for the PSO (ref Algorithm 1). The following parameters are used for *GA*: Crossover Probability = 0.9 and Mutation Probability = $\frac{1}{\text{number of tasks}}$. The size of the population is same as that of PSO i.e. twice the number of tasks in each application. Candidates are selected using *Roulette* wheel based selection and the best solution is preserved across generations. Further, *NSGA-II* is used to solve the problem and it replaces Algorithm 2. For *SA*, the following parameters are used: initial temperature = 100, end temperature = 10^{-5}, cooling rate = 0.95 and the number of random moves for each temperature = 100. The *SA* is solved using *Matlab* integrating Algorithm 1 for fitness evaluation. These parameters are chosen based on extensive evaluation and the results are omitted here. For *TS* and *GS*, the parameters are same as that used in [11] and [24] respectively.

As can be seen from the figure, the *GA* achieves better result (higher probability of correct result) among all the alternative heuristics. The PSO-based optimization achieves better results (for *H.264 Dec*) than *GA*. For the remaining 3 applications, the result of PSO and *GA* are the same. Among all 65 applications considered (synthetic and real) including those shown on the figure, *PSO* and *GA* achieves similar results for 43 applications (35 synthetic and 8 real-life). For the remaining 22 applications, PSO achieves higher probability of correct result by an average 5%. Another important consideration to make is that, the number of generations of *GA* is usually higher than that of PSO by an average 20%. This results in a higher execution time of *GA* as compared to PSO. These results justifies the selection of PSO for this problem class.

VII. Conclusion

This paper presents an artificial intelligence based task mapping technique to determine the number of checkpoints of a real-time application mapped on a homogeneous multiprocessor system with imperfect fault detection. Experiments conducted with synthetic and real-life application graphs demonstrate that the proposed technique improves the probability of correct result by 15% with imperfect fault detection. Even with 100% fault detection, the proposed technique is able to improve the confidence of the produced result by an average 25% and outperforms other meta-heuristic/greedy approaches

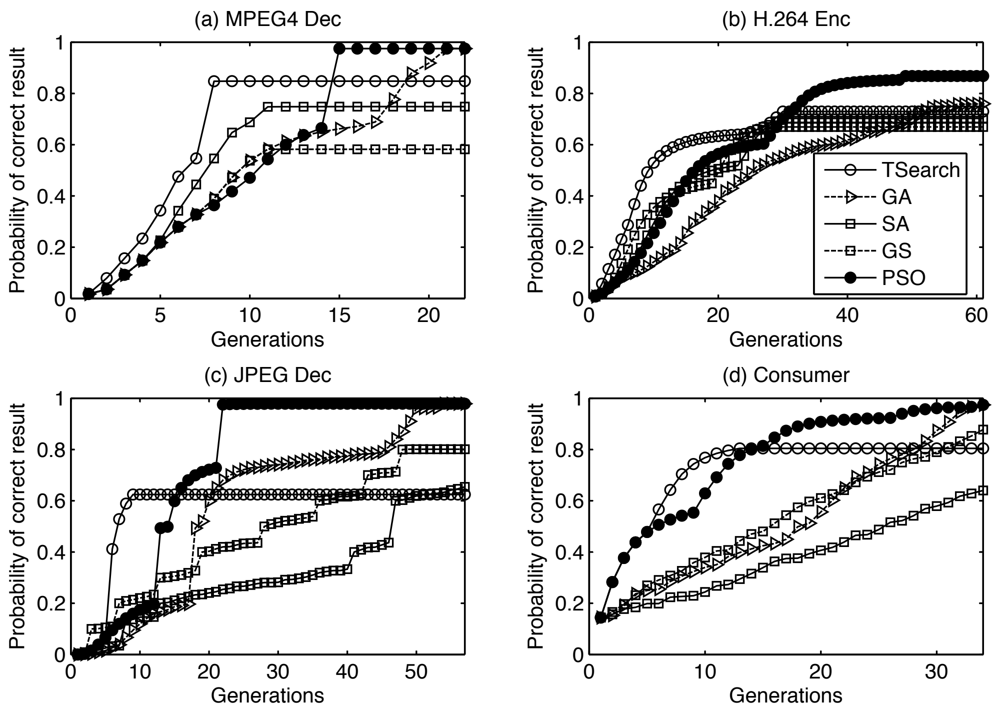

Fig. 5. Convergence of the proposed PSO technique

in terms of solution quality and/or execution time. As a future extension, heterogeneous architectures can be considered.

REFERENCES

[1] W. Wolf, "The future of multiprocessor systems-on-chips," in *Design Automation Conference (DAC)*. ACM, 2004, pp. 681–685.

[2] S. Borkar, T. Karnik, and V. De, "Design and reliability challenges in nanometer technologies," in *Design Automation Conference (DAC)*. ACM, 2004, pp. 75–75.

[3] C. Bolchini, A. Miele, and D. Sciuto, "An adaptive approach for online fault management in many-core architectures," in *Conference on Design, Automation and Test in Europe (DATE)*, 2012, pp. 1429–1432.

[4] J. Huang, J. O. 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)*. ACM, 2011, pp. 247–256.

[5] J. Huang, K. Huang, A. Raabe, C. Buckl, and A. Knoll, "Towards fault-tolerant embedded systems with imperfect fault detection," in *Design Automation Conference (DAC)*. ACM, 2012, pp. 188–196.

[6] K. Shin, T.-H. Lin, and Y.-H. Lee, "Optimal checkpointing of real-time tasks," *IEEE Transactions on Computers*, vol. C-36, no. 11, pp. 1328–1341, 1987.

[7] C. Krishna and A. Singh, "Reliability of checkpointed real-time systems using time redundancy," *IEEE Transactions on Reliability (TR)*, vol. 42, no. 3, pp. 427–435, 1993.

[8] A. Ziv and J. Bruck, "Performance optimization of checkpointing schemes with task duplication," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1381–1386, 1997.

[9] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Systems*, vol. 20, no. 1, pp. 83–102, 2001.

[10] 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 (TR)*, vol. 50, no. 3, 2001.

[11] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 17, no. 3, pp. 389–402, 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, pp. 89–98.

[13] 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, pp. 149–158.

[14] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," in *Conference on Design, Automation and Test in Europe (DATE)*, 2003, pp. 918–923.

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

[16] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors; Scheduling and Synchronization*. New York, NY, USA: Marcel Dekker, 2000.

[17] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Design Automation Conference (DAC)*. ACM, 2006, pp. 899–904.

[18] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Modeling static-order schedules in synchronous dataflow graphs," in *Conference on Design, Automation and Test in Europe (DATE)*, 2012, pp. 775 –780.

[19] D. Zaretsky, G. Mittal, R. Dick, and P. Banerjee, "Generation of control and data flow graphs from scheduled and pipelined assembly code," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, E. Ayguad, G. Baumgartner, J. Ramanujam, and P. Sadayappan, Eds. Springer Berlin Heidelberg, 2006, vol. 4339, pp. 76–90.

[20] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *International Symposium on Micro Machine and Human Science (MHS)*, 1995, pp. 39–43.

[21] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *IEEE Conference on Application of Concurrency to System Design (ACSD)*, 2006, pp. 276–278. [Online]. Available: http://www.es.ele.tue.nl/sdf3.

[22] 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)*, vol. 16, no. 2, pp. 113–129, 2005.

[23] R. Dick. (2013) Embedded system synthesis benchmarks suite (e3s). [Online]. Available: http://ziyang.eecs.umich.edu/~dickrp/e3s/

[24] A. Das, A. Kumar, and B. Veeravalli, "Aging-aware hardware-software task partitioning for reliable reconfigurable multiprocessor systems," in *ACM Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2013.