RELIABILITY-AWARE RUNTIME ADAPTION THROUGH A STATICALLY GENERATED TASK SCHEDULE

by

Laura Rozo

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Fall 2016

RELIABILITY-AWARE RUNTIME ADAPTION THROUGH A STATICALLY GENERATED TASK SCHEDULE

by

Laura Rozo

Approved: _____
Chengmo Yang, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical & Computer Engineering

Approved: _____
Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____
Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Device scaling, increasing number of components in a single chip, varying environmental issues, and aging effects have brought severe reliability challenges that impose tight constraints on the operation of a system. To cope with these challenges this thesis proposes a reliability aware scheduling framework that combines static and dynamic analysis to improve the overall system resiliency to different kind of faults (i.e. intermittent, transient, and permanent). The static analysis technique employs genetic algorithms to optimize the overall system reliability by considering Reliability Level (RL) as an intermediate scheduling dimension, and creating a task-to-RL mapping. This enables the RL-to-core mapping to be efficiently adapted at runtime according to fault rate variations, while the task-to-RL mapping can still be reused. The dynamic analysis tracks faults appearing in each core and measures the time correlation of those faults to update the RL-to-core mapping. The proposed reliability aware framework is implemented in a state of the art runtime system, DARTS, so as to quantitatively show the advantages of using the overall framework in existing multicore platforms. Experimental results show that the proposed technique delivers up to 30% improvement in application execution time and up to 72% improvement in faults occurring at runtime.

Chapter 1

INTRODUCTION

Trends in multicore systems such as device scaling and increasing number of components in a single chip have brought severe reliability challenges for current and future computer systems. Device scaling brings more variation in the operational conditions of transistors, leading to great differences in how fast they wear out due to aging mechanisms, such as Time Dependent Dielectric Breakdown (TDDB), Negative Bias Temperature Instability (NBTI), etc. These aging mechanisms cause a gradual decline in performance and increase the rate of intermittent and permanent faults [20, 4]. *Permanent faults* are unrecoverable and continue to exist until the faulty component is replaced [14]. *Intermittent faults* appear at random times during the operation of a system and unpredictably oscillate between being dormant and active [14]. Device scaling also increases the system's vulnerability to transient faults caused by cosmic and terrestrial radiation, noise due to crosstalk, and parameter variations [6, 11]. *Transient faults* are random and cause components to malfunction for periods of time and then disappear [14]. Moreover, the significant growth in component count amplifies the effect of the aforementioned issues since it increases the number of failure sources. These issues in combination contribute significantly to increasing overall system fault rates.

Due to the diverse nature of different types of faults occurring continuously with varying duration over wide time scales, it is desirable for fault resilience solutions to deliver runtime adaptability. Previous solutions [5, 8, 7] proposed exclusively for permanent or transient/intermittent faults will no longer suffice as they assume faults either consistently manifesting or never re-manifesting. Instead, the combined effects of all types of faults need to be considered, and runtime fault tolerant solutions should

monitor fault duration, adaptively recover task execution, and adjust resource allocation. For example, during the period of time in which an intermittent or transient fault is active, it is desirable to decrease the number of tasks allocated to the unit affected by the fault. Once the fault disappears, tasks can then be allocated to that unit at the normal rate whereas, processing units with permanent faults should be isolated since allocating tasks to them will have an adverse impact on system performance.

Because power, computing performance, and cost have become of critical importance, adaptive resource allocation must efficiently, promptly, and economically characterize faulty behavior. However, dynamic state-of-the-art solutions such as [5, 8] rely on heavy runtime computations to optimize system reliability. Even though they are able to adapt resource allocation as faults occur at runtime, by redistributing tasks in the system, they impose high overheads in terms of execution times and power consumption. Furthermore, since the decisions made by a dynamic scheduler are only locally optimal, they usually have an unpredictable impact on overall system workload distribution and hence cannot deliver a predictable worst-case performance.

To minimize runtime overhead and improve system-wide predictability, this thesis integrates the approaches in [18, 17] into a single framework. Traditional static fault tolerant techniques [13, 12] do not take the non-constant fault behavior into consideration as they generate schedules based on predefined worst-case conditions. To avoid this, the technique in [18] defines an intermediate scheduling dimension: Reliability Level (RL). During the static scheduling process, tasks are mapped to RLs rather than cores. This enables RL-to-core mapping to be adapted at runtime to match the monitored core fault rates. Meanwhile, to monitor and update the RL of each core and apply the pre-generated static schedule, this thesis will employ the heuristic proposed in [17]. The reliability model in [17] is capable of tracking not only faults appearing in each core but also their correlation in time. Through monitoring fault occurrence frequency, healthy, unhealthy, and intermediate states are defined for cores. This allows the system to progressively reduce the frequency of job assignment to a processing unit until it becomes non-functional. This approach outperforms previous solutions [3]

which have a limited capability to react to faults due to the small number of defined system states. As they only classify resources as healthy and unhealthy, their reaction is to exclude unhealthy units from being used, which is quite abrupt and only happens upon identifying a permanent fault.

Not only does this thesis integrate the approaches in [18, 17] into a single framework, but it also enhances them in two aspects. First, the static heuristic in [18] is replaced with a genetic algorithm which makes a broader exploration of the solution space in order to obtain better average execution times for applications in the presence of faults. Second, the reliability model proposed in [17] is implemented in a state-of-the-art runtime system, DARTS [21]. DARTS is an open source implementation of the codelet execution model [24] that aims to provide high performance execution of fine-grained workloads. With the reliability model in [17], this runtime system is further improved to monitor faults, update the core-to-RL mapping, and apply the task-to-RL mapping provided by the static genetic algorithm. To summarize, this thesis makes the following contributions:

- It integrates the approaches in [18, 17] into a single framework that implements the mapping chain proposed in [18]. The runtime overhead is minimized by computing the task-to-RL mapping statically while the flexibility and adaptability of dynamic solutions are kept by allowing the runtime system to dynamically change the core-to-RL mapping.
- It improves the task-to-RL mapping heuristic proposed in [18] so as to make a broader exploration of the solution space. It is quantitatively shown that the proposed genetic algorithm outperforms the static heuristic in [18] in terms of average execution time of applications.
- It implements the reliability model [17] using a state-of-the-art runtime system, DARTS, and quantitatively evaluates the overall framework that combines [18] and [17] using existing multicore hardware platforms. In contrast, the original evaluations in [18, 17] relied upon in-house simulators to obtain their results. Experimental results show that the proposed framework delivers up to 30% improvement in application execution time and up to 72% improvement in the number of faults.

The rest of this thesis is organized as follows: Section 2 reviews the related work in fault tolerant scheduling. Section 3 introduces the overall reliability framework and

the coordination process of all of its pieces. Section 4 and 5 respectively describe the static portion and the dynamic portion of the framework. Section 6 describes the experimental setup, Section 7 presents the experimental results, and Section 8 concludes the thesis.

Chapter 2

RELATED WORK

Existing work in fault tolerant scheduling techniques can be divided into three categories: static, dynamic and hybrid approaches. Static approaches [13, 12] rely on offline analysis to generate fixed task schedules capable of tolerating up to a fixed number of faults. Solutions for tolerating both transient [13] and permanent [12] faults have been proposed. These static schedules require a Fault Tolerant Process Graph (FTPG) to model all possible fault scenarios in advance. While these solutions effectively hide the runtime overhead for making rescheduling decisions, they do not deliver runtime adaptability. Since the generated schedules consider only the worst case scenario, spare resources are required, leading to resource underutilization in most cases. Yang *et al.* [23, 22] have developed adaptive static schedules to attain runtime adaptation in the face of resource variations in multicore systems. No spare resource is required. Instead, these schedules are partitioned into regular yet shiftable band structures, thus maximizing resource utilization in most cases and enabling a regular reconfiguration process to isolate a faulty core at runtime.

Due to the diverse behavior of faults, it would be more desirable to have mechanisms that monitor faults at runtime and adapt task allocation accordingly. Dynamic approaches are able to naturally react to faults by adapting the redundancy level, postponing tasks or redistributing workloads. Chantem *et al.* [5] proposed an on-line task assignment and scheduling algorithm that optimizes system lifetime based on the aging condition of each core, which is computed with temperature information sensed from each core. However, this approach only considers temperature variations while ignoring sporadic environmental issues that cause unpredictable fault behavior. Gottumukkala *et al.* [9] presented another dynamic approach to distribute jobs of different

lengths in a way that maximizes (or minimizes) a pre-defined system reliability function. Again, the effect of intermittent faults is ignored. Furthermore, both approaches involve complex function optimizations and a large amount of data processing to make reconfiguration decisions, making them less suitable for applications with strict time constraints. Haque *et al.* [10] proposed a generic recovery strategy called Δ-idling that minimizes the worst-case overhead for transient faults that occur randomly and over a conitinuous time interval with varying durations. This scheme leaves the CPU idle for Δ units of time, where Δ is the upper bound on the length of the fault burst. However, this approach is designed only for transient faults and its success depends on the scheduling algorithm used in combination with the technique. Bolchini *et al.* [3] proposed an approach that additionally handles permanent faults in the system. By recording the error history of each core, this technique differentiates transient and permanent failures, and avoids allocating tasks to unhealthy cores. However, since the only recognized core states are healthy/unhealthy, the system reacts only upon identifying a permanent fault. In contrast, the proposed fault tolerant framework uses multiple intermediate states to characterize intermittent faults whose duration may vary from nanosecond to second time scales, thus enabling task allocation frequency to be adjusted in a much finer granularity.

Hybrid approaches aim at combining the advantages of static analysis and dynamic adaptation to efficiently tolerate faults. In [16], an on-line adaptive recovery scheme is used to exploit spatial and temporal redundancy based on power and execution time constraints. Another tool is used to create an error vulnerability profile during the design phase that identifies critical functional blocks with the most rigorous recovery constraints. Coskun *et al.* [7] also proposed a hybrid approach to improve system lifetime. A static schedule is generated using ILP (integer linear programing), aiming at minimizing temperature hotspots and balancing temperature distribution across the die. This approach only considers temperature impact but not runtime core failures.

To summarize, existing approaches are limited either by their incapability of

tolerating transient, permanent, and intermittent faults all at the same time, or by the number of states in which they classify resources for task allocation. Those shortcomings make them insufficient for responding to the unpredictable factors that boost fault occurrence including environmental issues, aging effects, and temperature variations.

Chapter 3

FRAMEWORK OVERVIEW

As discussed in Section 2, purely static approaches lack the adaptability to respond to variations in core fault rates at runtime, while pure dynamic approaches require heavy runtime computations and cannot guarantee predictable worst-case performance. To overcome these shortcomings, this thesis integrates the static approach in [18] with the dynamic approach in [17] in order to produce a holistic approach capable of providing performance guarantees with runtime adaptability. The result of which is a static scheduling approach that guides runtime adaption, relieving the runtime overhead that comes with deciding how to map tasks onto available resources. At runtime, an effective online fault diagnosis algorithm is implemented, capable of differentiating between different types of faults and assigning a RL to each core in order to tune its task allocation frequency.

The rest of this chapter is devoted to a discussion of the coordination between the static scheduler and the runtime scheduler, as well as the framework's interaction with DARTS, a codelet based runtime system for shared memory systems.

3.1 System Requirements

The reliability-aware scheduling framework is proposed for multicore platforms with homogeneous Processing Units (PU). However, these PUs are diverse in their fault rates, which in turn necessarily influences scheduling decisions due to the impact of faults in the overall application execution time. The faults are expected to be transient or intermittent faults with short durations that affect the core logic (e.g. bit flips in registers, timing errors in CMOS circuits) and produce incorrect computations at the end of the task. It is assumed that the system already has either software or hardware

mechanisms to check task results and identify faults. Once a faulty result is detected, the task will be re-executed on another core that matches the required RL of the task. Any dependent tasks will be delayed as a result.

3.2 Static and Runtime Coordination

The overarching goal of the proposed reliability framework is to schedule tasks to cores with non-constant fault rates in a way that minimizes the execution time of the entire application. To this end, as indicated in Fig. 3.1, tasks are statically mapped into RLs, and then the runtime system dynamically updates the RL-to-core mapping, adapting the final task-to-core mapping during execution. The underlying reason is because the task-to-RL mapping is determined by the application task graph, while the runtime variations in core fault rates only influence the RL-to-core mapping.

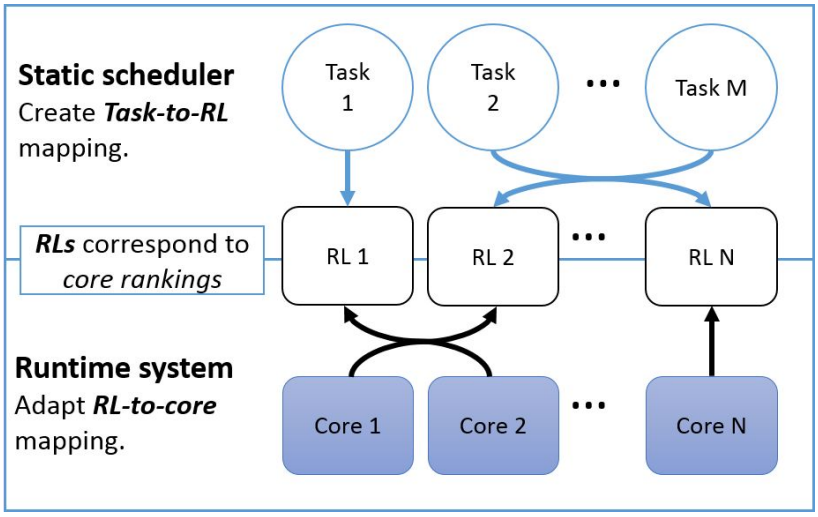

Figure 3.1: Static vs. Dynamic framework functionality

Integrating the static scheduler in [18] and the runtime scheduler in [17] into a coherent framework is nontrivial given the fact that the two approaches employ different reliability metrics. On one hand, the scheduler in [18] models the *expected* execution time of each task as a function of the core fault rate. By doing so, the problem of statically generating an optimal task-to-RL mapping is reduced to finding the minimal

expected schedule length. On the other hand, the approach in [17] does not directly measure fault rates, but instead defines *Fault Vulnerability Factors* (FVFs) whose values are in the range $[0, \infty)$ and do not have a straightforward mapping onto fault rates. To integrate both approaches coherently, RLs in this thesis are *core rankings*, which are not absolute measures of core reliability, but relative measures that compare the reliability metrics of two cores, which could be either fault rates or FVFs. The static scheduler ranks cores based on their predicted fault rates, while the runtime system ranks cores based on the observed FVFs. In this way, the runtime system will be able to appropriately adapt the schedule generated by the static scheduler.

3.3 DARTS and its Interaction with the Framework

The Delaware Adaptive Run-Time System (DARTS) is the University of Delaware's own implementation of the Codelet Model specification [24] for shared memory systems, aiming to provide high performance execution of fine-grained workloads. The codelet model represents applications as task graphs (i.e. codelet graphs) that are embedded inside containers called Threaded Procedures (TPs). A codelet (i.e. a task) is a collection of machine instructions that are scheduled *atomically*, as a non-preemptive, single unit of computation [21]. Dependencies between codelets are defined statically. DARTS embeds an abstract machine model consisting of clusters made up of one Scheduling Unit (SU) and one or more Computation Units (CUs). Each CU, as well as the SU, have a one-to-one mapping to a physical machine core. As input, DARTS receives an application in the form of codelets graphs embedded within a TP. The SU receives the TP and unwraps the codelet graph contained in it. When codelets are ready to fire (i.e., when all their dependencies are met), the SU will dispatch the ready codelets among its CUs and each CU will execute all the codelets assigned to it. Fig. 3.2 shows the integration of the reliability-aware scheduling framework into the DARTS runtime system using a single cluster.

The previous implementation of DARTS did not take into consideration reliability issues. As a result, the implementation has been modified to incorporate a number

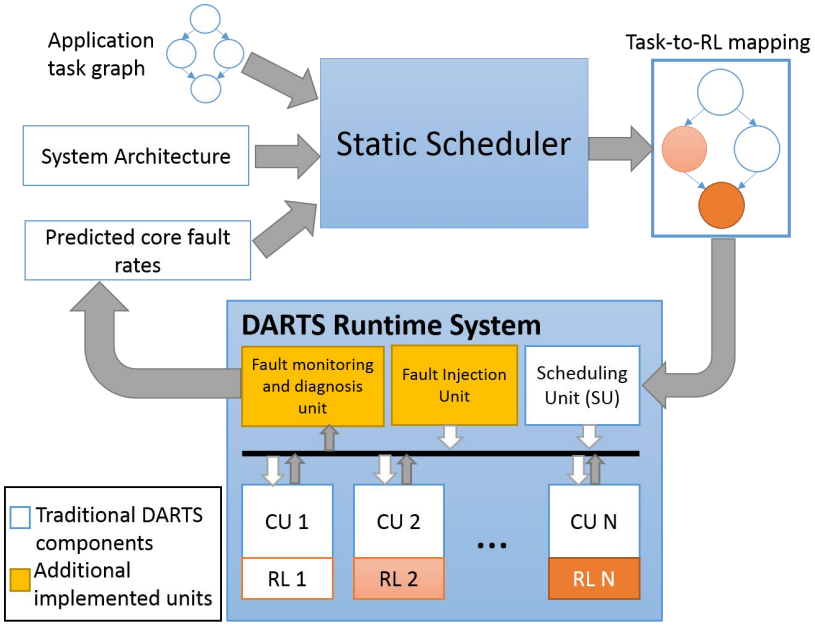

Figure 3.2: Reliability Aware Scheduling Framework

of new required features, summarized as follows:

- Assigning tasks to cores based on the task-to-RL mapping provided by the static scheduler.
- Checking task results to identify faults, if any.
- Monitoring faults occurring in each core and computing FVFs accordingly.
- Ranking cores based on their FVFs.
- Re-executing a faulty task on a core with the required Reliability Level (RL).

Fig. 3.2 summarizes the main components in DARTS, together with the incorporated new components. Specifically, to implement the new features and to test the proposed framework, three new components have been incorporated into DARTS:

1. ***An RL attribute for each task (or codelet)***, which is used to determine the core ranking required by the task according to the static scheduler.
2. ***A Fault Monitoring and Diagnosis (FMD) unit***, which is responsible for recording fault occurrences and updating FVFs. At the beginning of execution, although all cores are equally reliable, an initial consecutive ranking will still be assigned by the scheduler. As the execution continues, the FMD unit will update the FVF of each core based on the number of faults and their time correlation.

3. ***A Fault Injection Unit***, which is responsible for randomly injecting faults into the system so as to simulate different fault behaviors. The number of faults in each core depends on its pre-specified fault rate, while the injected faults follow Weibull distribution [14].

Additionally, the scheduler unit in DARTS has been augmented to periodically sort cores based on the FVFs, provided by the FMD unit, so as to update the RL of each core (i.e. core ranking). It needs to be noted that the core sorting process only takes place at the end of each application period rather than when one of the FVFs is modified. This policy minimizes the runtime overhead imposed by the proposed framework and improves its stability. DARTS scheduler unit has also been modified to assign tasks to CUs with the required RLs, monitor task execution, and upon a fault initiate re-execution and postpone the firing of any dependent tasks. Note that the fault detection mechanisms are just simulated in DARTS: a codelet is consider faulty if and only if a fault is injected into the corresponding core during its execution. This policy is sufficient to evaluate the proposed work whose contribution is not a new fault detection scheme but a resiliency framework capable of proactively reacting to faults on top of existing fault detection mechanisms.

Chapter 4

RELIABILITY AWARE STATIC SCHEDULING

This chapter explains the reliability-aware static scheduling approach in detail. We first discuss the reliability model during static scheduling. Subsequently, we formulate the scheduling problem using Integer-Linear-Programming (ILP) and show two approaches to solve this problem: a polynomial time heuristic employed in [18] and a genetic algorithm approach used in this thesis. We then compare the two approaches and show that the genetic algorithm outperforms the polynomial heuristic in [18] in most of the cases.

4.1 Reliability Modeling During Scheduling

Since the static scheduler centers on mapping tasks to reliability levels (RLs), it is necessary to develop a model so that different RLs influence scheduling decisions. At first sight, an intuitive idea is to minimize the probability for a fault to occur during task execution. However, this will result in the scheduler to assign all the tasks to the most reliable core, that is, the core with the lowest fault rate. Therefore, such scheduler will not be able to fully exploit the available hardware resources, since cores with lower RLs will never be used. Instead, our observation is that tasks that are less critical and less vulnerable to faults can be mapped to a less reliable core. Even if a fault corrupts the results, it is possible that re-execution of these tasks may not degrade the overall execution time of the application. In light of this observation, we propose to model the *expected execution time* of a task as a function of core fault rates.

Assume a task with execution time T is executed on a core with constant failure rate f. The probability that a fault will occur during execution is $1 - e^{-fT}$ [14]. If a fault occurs, the task has to be re-executed, thus requiring additional execution time

Table 4.1: Set of model constraints

1	$\forall v_i : \sum_{k=1}^{N} VtoC_{ik} = 1$	All tasks must be bound to a unique core.
2	$Order_{ij} + Order_{ji} = \sum_{k=1}^{N}(VtoC_{ik} * VtoC_{jk})$	The order relation is mutually exclusive. If tasks v_i and v_j are scheduled to the same core, the right-hand side of the equation is 1. If v_i goes before v_j, then v_j cannot go before v_i and vice versa.
3	$\forall e_{ij} : Order_{ji} = 0$	A task cannot be scheduled before any of its predecessors.
4	$\forall v_i, v_j, v_k \ such \ that \ i \neq j \neq k : Order_{ij} \geq Order_{ik} * Order_{kj}$	There is a transitive relation in task orders. If tasks v_i, v_j and v_k are all mapped to the same core, v_i goes before v_j and v_j goes before v_k, then v_i must go before v_k.
5	$\forall v_i : tf_i = ts_i + \frac{1}{e^{-\sum_{k=1}^{N}(VtoC_{ik} * f_k) * T_i}}$	The finish time of a task is given by its start time plus its expected execution time, which is influenced by core fault rate f_k.
6	$\forall e_{ij} : ts_j \geq tf_i + w_{ij} * (1 - Order_{ij})$	The start time of a task is constrained by the finish time of all its predecessors plus the communication cost, if any.
7	$\forall v_i, v_j \ such \ that \ i \neq j : ts_j \geq Order_{ij} * tf_i$	For any pair of tasks v_i and v_j, if they are mapped into the same core and v_i goes before task v_j, then $Order_{ij} = 1$, and the start time of v_j is constrained by the finish time of v_i.

Table 4.2: Variables defined for the ILP formulation

v_i	A task in the task graph, with $i = 1 \ldots M$, and M denoting the total number of tasks.
T_i	Execution time of task v_i.
ts_i	Start time of task v_i.
tf_i	Finish time of task v_i.
e_{ij}	An edge in the task graph, representing a dependence between tasks v_i and v_j, with $i \neq j$.
w_{ij}	Communication cost of edge e_{ij}, given that v_i and v_j are mapped to different cores.
c_k	A core in the system architecture, with $k = 1 \ldots N$ and N denoting the total number of cores.
f_k	The pre-defined fault rate of core c_k.
$VtoC_{ik}$	A set of binary variables. $VtoC_{ik} = 1$ *iff* task v_i is mapped to core c_k; $VtoC_{ik} = 0$ otherwise.
$Order_{ij}$	A set of binary variables. $Order_{ij} = 1$ *iff* task v_i and task v_j are mapped to the same core and v_i is before v_j. $Order_{ij} = 0$ otherwise.

of T. The *expected task execution time*, denoted as T_{EXE}, can be modeled using the following formula:

$$T_{EXE} = T + (1 - e^{-fT}) * (T + (1 - e^{-fT} * (T + ...))) = \frac{T}{e^{-fT}} \quad (4.1)$$

Clearly, both a higher fault rate f and a longer execution time T will result in a larger T_{EXE}. By modeling the execution time of each task in this way, the problem of generating an optimal task-to-RL-to-core mapping is reduced to the problem of finding the minimum expected schedule length, which is equal to the expected finish time of the last task.

4.2 Scheduling Problem Formulation

Given an application task graph and a set of cores with different fault rates, the reliability-aware scheduling problem can be modeled using Integer Linear Programming (ILP). Table 4.2 summarizes the set of variables defined for the ILP formulation. Overall, the static scheduler takes three sets of inputs: applications task graph, system architecture, and predicted core fault rates. The task graph is a directed periodic

graph $G = \{V, E\}$, where each vertex $vi \in V$ represents a task. Each task has an execution time T_i, a start time ts_i, and a finish time tf_i. Each edge $e_{ij} \in E$ represents a dependency between tasks v_i and v_j. It has a weight w_{ij}, representing the communication cost between v_i and v_j, if they are mapped to different cores. Finally, the system architecture is represented as a set of cores $\{c_1, c_2, ..., c_N\}$, with N denoting the total number of cores. Each core c_k has a fault rate f_k, denoting its reliability level.

To model task binding, a set of binary variables $VtoC_{ik}$ is used. $VtoC_{ik} = 1$ if a task v_i is bound to core c_k, while $VtoC_{ik} = 0$ otherwise. Additionally, to ensure that tasks mapped to the same core will not overlap in their execution time, another set of binary variables $Order_{ij}$ is defined. $Order_{ij} = 1$ only when tasks v_i and v_j are mapped to the same core and v_i goes before v_j, while $Order_{ij} = 0$ otherwise.

Using the variables defined in Table 4.2, the scheduling problem can be formulated with a set of ILP constraints, as summarized in Table 4.1. Constraint 1 models task-to-core binding, constraints 2 to 4 model the precedence ordering and non-overlapping execution relations between tasks, and constraints 5 to 7 model the start and finish time of each task as a function of core fault rates. As can be seen in the 5th constraint, for task v_i, its finish time tf_i is the sum of its start time ts_i and its expected execution time, computed using Equation (4.1). Since $VtoC_{ik} = 1$ only for a single core, the expression $\sum_{k=1}^{N}(VtoC_{ik} * f_k)$ ensures that the fault rate of one and only one core is counted.

Finally, Equation (4.2) shows the optimization goal of the ILP problem, that is, minimizing the overall schedule length which equals the finish time of the last task. Since the ILP model already expresses the expected execution time of every task as a function of the core fault rate, by optimizing the schedule length, the model implicitly takes the impact of fault rate into consideration.

$$Minimize(\max_{1 \leq i \leq M} tf_i) \tag{4.2}$$

Table 4.3: Standard task graphs used

Task Graph	Nodes	Average execution times (us)
Fast Fourier Transform (FFT)	39	4992
Fork/ Join application	43	5680
Gaussian elimination	49	5625
Laplace Equation	42	6000
LU Decomposition	44	6666
Out Tree task graph	63	6333
In Tree task graph	63	5833

4.3 List Scheduling Heuristic

The heuristic in [18] schedules task to cores through list scheduling [15]. List scheduling algorithms are typically composed of two phases, namely, a task prioritization phase, wherein the scheduling order of each task is determined, and a core assignment phase, wherein each task is assigned to a core that minimizes its start time. In the previous scheduling heuristic [18], core fault rates will influence both phases. When scheduling a task, instead of minimizing its earliest start time, the heuristic minimizes its earliest finish time (EFT). The task to schedule is placed on the core that delivers the shortest EFT, based on the core fault rate and its availability. In this way, the impact of core fault rates on task execution time is taken into account. Moreover, if a task is scheduled on a less reliable core, its expected execution time will become longer, and the start time of its descendant tasks may be delayed as well. In the task prioritization phase, the heuristic gives a larger chance for these descendant tasks to be scheduled on more reliable cores, by dynamically ranking all the to-be-scheduled tasks based on the execution time of already-scheduled tasks. More details about the algorithm can be found in [18].

4.4 Genetic Algorithm-based Scheduling Heuristic

One shortcoming of the list scheduling heuristic [18] is that it only generates a single schedule. To effectively search the possible solution space, in this thesis we

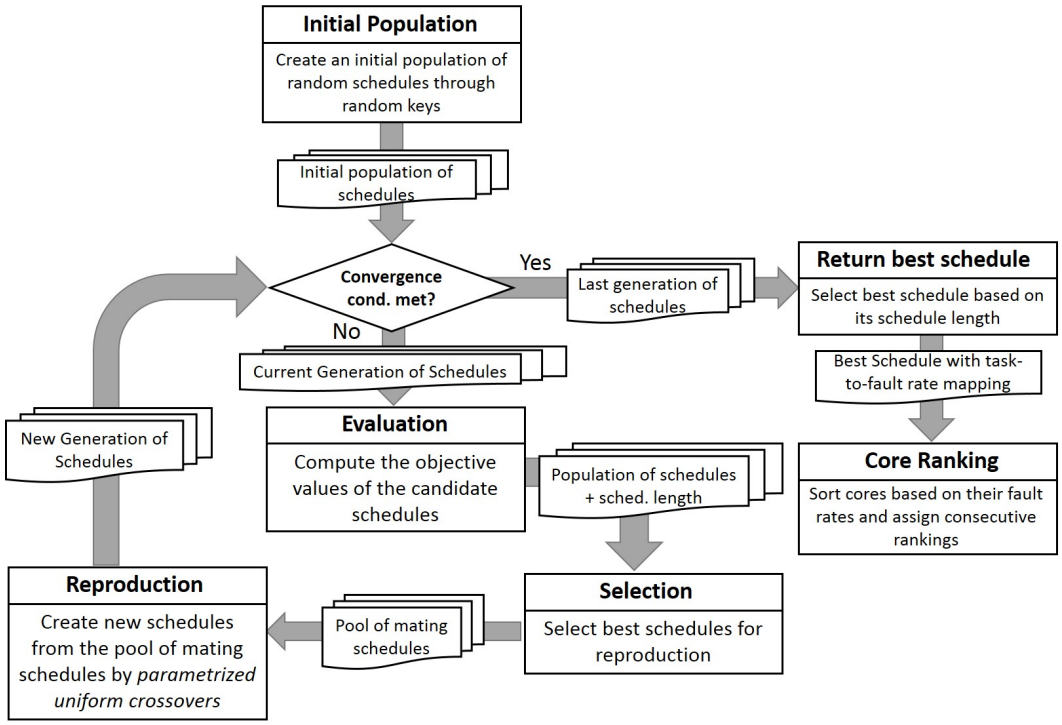

Figure 4.1: Overview of genetic algorithm phases

propose a genetic algorithm [2] based approach. Same as the list scheduling heuristic, the inputs to the genetic algorithm are the application task graph, the set of cores in the system, and the predicted fault rate of each core, f_k.

Fig. 4.1 presents the flow of the genetic algorithmm which is composed of four phases: 1) Generation of initial population, 2) Evaluation of the population, 3) Selection, and 4) Reproduction. To ***generate the initial population***, the algorithm randomly generates a set of schedules by randomly mapping tasks to cores based on the *random keys* approach [1]. Random keys solves the feasibility problem in genetic algorithms by guaranteeing the feasibility of all offspring or solutions without creating any additional overhead. Each schedule is represented as an array of rational numbers, with each element in the array representing a task in the task graph. The integer part of each element encodes the task id while the fractional part encodes the RL of the core to which it was mapped. Once these set of encoded schedules are generated, the

evaluation of the population takes place. Here, the objective function, which in this case corresponds to the expected schedule length (i.e.,the expected finish time of the last task), is computed for each of the schedules. Then, the ***selection*** phase identifies the schedules with minimum expected schedule lengths. These schedules form the pool of mating schedules, to be given as input for the reproduction phase. Finally, during the ***reproduction*** phase, new schedules are created from the pool of mating schedules based on parametrized uniform crossovers [19]. The process is repeated from the evaluation phase until the convergence condition is met. The convergence condition is met after the reproduction phase is no longer able to find better schedules, with respect to previous iterations, for a fixed number of iterations.

Upon completion of the scheduling process, cores are sorted and consecutively ranked based on their fault rates to translate the task-to-fault rate mapping into a task-to-RL mapping, where RLs represent the core ranking. The final output is a static schedule with core binding and RL binding information. Note that the exact start time of a task is not needed, since the runtime system is responsible for monitoring inter-task dependencies and starting a task once its inputs are ready and the corresponding core is idle.

Table 4.4: Comparison of List Scheduling (LS) Genetic Algorithm (GA) for the follwoing sets of system fault rates: FR={0.01, 0.02, 0.04} and FR={0.01, 0.03 0.09}

	FR = 0.01, 0.02, 0.04			FR = 0.01, 0.03 0.09		
Task Graph	LS	GA	Redt. (%)	LS	GA	Redt. (%)
FFT	457	365	20.14	986	428	56.60
Fork/ Join	476	407	14.46	1098	527	52.00
Gaussian elimination	555	427	23.07	1003	523	47.81
Laplace Equation	712	607	14.71	3472	803	76.87
LU Decomposition	1081	652	39.71	4135	810	80.41
Out Tree	990	1174	−18.62	2399	1579	34.16
In Tree	706	720	−2.08	1608	929	42.19

Table 4.5: Comparison of List Scheduling (LS) Genetic Algorithm (GA) for the follwoing sets of system fault rates: FR={0.01, 0.04, 0.16} and FR={0.01, 0.05, 0.25}

	FR = 0.01, 0.04, 0.16			FR = 0.01, 0.05, 0.25		
Task Graph	LS	GA	Redt. (%)	LS	GA	Redt. (%)
FFT	4690	469	90.00	19424	550	97.17
Fork/ Join	6400	571	91.08	64027	661	98.97
Gaussian elimination	3434	621	81.92	18396	661	96.41
Laplace Equation	30342	892	97.06	440354	974	99.78
LU Decomposition	31764	956	96.99	449137	989	99.78
Out Tree	18767	1678	91.06	220055	1984	99.10
In Tree	11575	1333	88.48	166058	6180	96.28

4.5 Comparison of scheduling heuristics

The proposed Genetic Algorithm (GA) (Section 4.4) is compared with the previously used List Scheduling (LS) heuristic (Section 4.3) in terms of the expected schedule lengths for different application task graphs and under different system configurations. The studied applications and their main characteristics are shown in Table 4.3. All system configurations have 3 cores, but they vary in terms of their reliability levels. Configuration 1, for example, starts with a fault rate of 0.01 (i.e., 1% fault rate per unit time) and uses an increasing factor of 2x, while the other three configurations use the same initial fault rate but different increasing factors of 3x, 4x and 5x, respectively.

Tables 4.4 and 4.5 depicts the expected schedule lengths for each heuristic as well as the reduction ratio of the Genetic Algorithm with respect to List Scheduling. It can be seen that the genetic algorithm outperforms the List Scheduling heuristic in most cases by a significant amount. This trend is even more evident as the increasing factor of the fault rates becomes higher. These results clearly confirm that by making a broader exploration of the solution space, the Genetic Algorithm is capable of finding better suited solutions than the List Scheduling heuristic that just employs a greedy algorithm.

Chapter 5

RELIABILITY AWARE RUNTIME SYSTEM

This chapter describes the runtime perspective of the proposed fault tolerance framework, which monitors time-correlated fault behavior to define reliability values of different processing units and gradually tunes task allocation based on such information. It first presents the employed core reliability model and then the adaptive task scheduler which takes the task-to-RL mapping from the static scheduler as input and maps tasks to cores based on it.

5.1 Modeling Core Reliability

In the proposed fault tolerance framework, the reliability level of each core is computed by the FMD unit with a goal of modeling time-correlated fault behavior. Instead of simply counting the number of occurring faults, faults occurring at different times should be given different weights in the model. To illustrate this time-correlated behavior, Fig. 5.1 depicts a set of scenarios, ranked in ascending order in terms of how reliable they are. Each scenario shows the failure history of a single core: a "1" represents a faulty task result, while a "0" represents a clean result.

A "1" in the rightmost bit position indicates that a fault was observed for the most recently executed task. As the fault may have not disappeared, it is necessary to reduce task allocation frequency to the corresponding core, implying that faults occurring more recently should have greater impact on core reliability. Unless the core starts to produce clean task results, task allocation frequency will keep decreasing. In an extreme case, such as Scenario 1 shown in Fig. 5.1, the core has not produced any clean result within the observed window, and hence no more tasks should be allocated to that core.

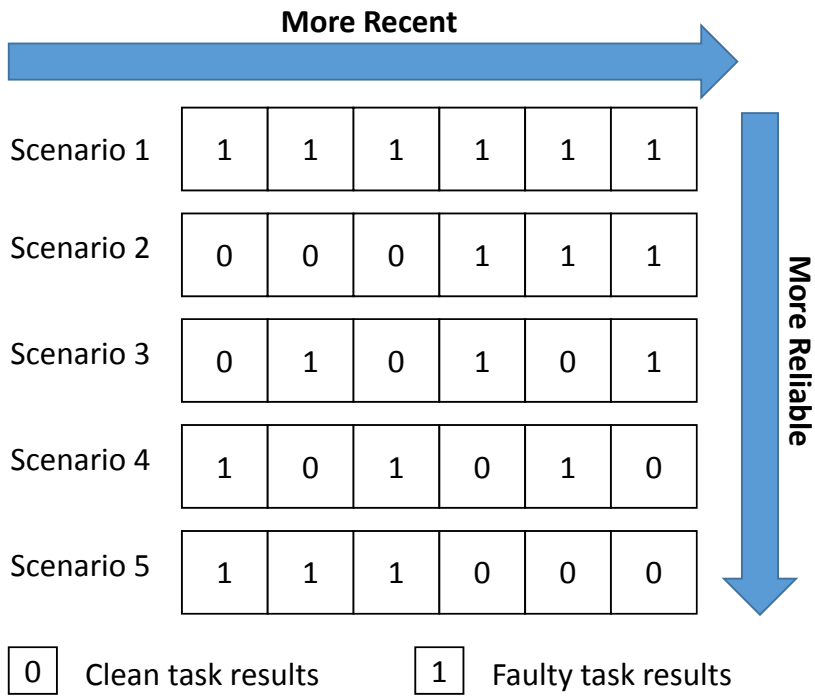

Figure 5.1: Examples of fault scenarios, each showing different fault histories

Another important consideration is that consecutive occurrences of faults should have a larger impact than faults occurring at random. This can be observed through a comparison between Scenarios 2 and 3 in Fig. 5.1. While their numbers of faults are equal, faults in Scenario 2 occurred in a continuous manner, indicating high correlation among fault behavior (possibly an intermittent fault of long duration). As a result, Scenario 2 is considered less reliable than Scenario 3.

Once a core starts to produce clean results, its task allocation frequency is gradually increased. This is because a clean result is a good sign that indicates the fault affecting the core has probably disappeared. This situation is shown in Fig. 5.1 with the rankings of Scenarios 3, 4 and 5. Scenario 4 is more reliable than Scenario 3 because of the clean task result recorded in the rightmost bit position, while Scenario 5 is considered most reliable due to the consecutive clean results for the most recent executions. The higher the number of clean task results observed recently, the higher

Algorithm 1 Computation of Fault Vulnerability factors

```
1:  Input:  taskResult ∈ {faulty, clean}
2:  Output:  FVF_i – FVF of core i
3:  procedure FVFactor(taskResult)
4:      if taskResult is faulty then
5:          FVF_i = FVF_i + faultWeight;
6:          faultCon = faultCon + 1;
7:          if faultCon = Max_FAULTS then
8:              Mark core i as unrecoverable;
9:          end if
10:     else
11:         FVF_i = FVF_i/2;
12:         faultCon=0;
13:     end if
14: end procedure
```

the reliability of the core.

Overall, the scenarios depicted in Fig. 5.1 indicate that to effectively model time-correlated fault behavior, an algorithm should prioritize cores based on the following criteria:

- A *more recent fault* should be given a higher weight than a fault occurred a long time ago.
- *Continuously occurring faults* should be given a higher weight than random and discrete faults.
- A *clean task result* should suppress the accumulating impact of previous faults.

Based on these criteria, we develop Algorithm 1 to compute the Fault Vulnerability Factor (FVF) of a core upon finishing executing a task on it and checking its result. Smaller FVF values indicate that the core is more reliable. If the task result is clean, the FVF of the corresponding core is divided by 2 (line 11). Otherwise, the result is faulty and the algorithm adds a specific $faultWeight$ to the current fault vulnerability factor (line 5). The exact value of $faultWeight$ can be determined by the user. A larger $faultWeight$ makes the system more sensitive to a faulty result, as more clean executions are required to suppress the effect of a single fault. More specifically, $faultWeight$ equal to n requires $log_2(n)$ consecutive clean task results to

completely suppress the fault's effect. As a concrete example, Fig. 5.2 shows the FVF values computed by Algorithm 1 for the five scenarios presented in Fig. 5.1.

Compared to the FVF computing algorithm used in [17], Algorithm 1 is augmented with the capability of detecting cores with permanent failures. In Line 7, a consecutive fault threshold, denoted as MAX_FAULTS, is used to represent the maximal amount of consecutive faults that the system allows before considering a core as unrecoverable. If $faultCon$ in Algorithm 1 reaches MAX_FAULTS, the core is marked as "unrecoverable" and will not be used by the scheduling unit. Furthermore, as the RL ranking associated with the unrecoverable core becomes invalid, for load balancing purposes, tasks originally mapped to that ranking will be evenly re-distributed among the remaining healthy cores. This optimization will be further explained in Section 5.2.1.

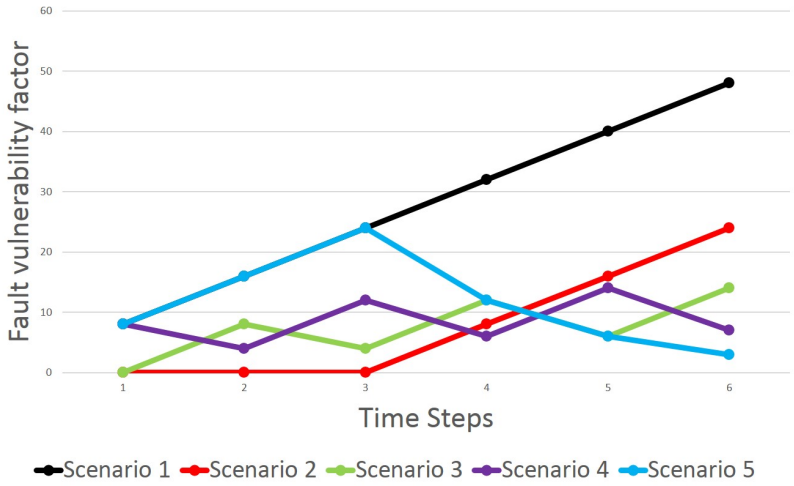

Figure 5.2: Computed fault vulnerability factors of the fault scenarios in Fig. 5.1

Finally, it needs to be highlighted that the FVFs computed by the FMD unit are later used by the Scheduling Unit to rank the cores and generate the RLs required to integrate the static and dynamic schedulers, thus accurately reflecting the current state of the system. This differs from the technique in [17], which uses local and global FVFs instead.

5.2 Adaptive Task Scheduling

As described above, the proposed resiliency framework employs both static and runtime reliability rankings for cores. Reliability Levels (RLs) are defined for each task by the static scheduler, while Fault Vulnerability Factors (FVFs) are computed for each core by the FMD unit at runtime. When integrating this framework into DARTS, these two rankings are respectively handled by the two-level schedulers. At the outer level, the Scheduling Unit (SU) is responsible for periodically ranking cores, updating their RLs, and mapping tasks onto cores according to their statically predefined RLs. At the inner level, the Computing Units (CUs) are responsible for executing the codelets assigned by the SU, checking codelet results, updating the FVF using Algorithm 1, and inserting faulty codelets back into the ready queue of the SU. This collaborative scheduling and execution process is shown in Algorithm 2 and Algorithm 3.

5.2.1 DARTS Scheduling Unit (SU)

As mentioned in Section 3.3, applications in DARTS are expressed as codelet graphs that are embedded inside Threaded Procedures (TPs). As a result, the SU has two queues: the *TP queue* that stores application TPs pending to execute, and the *codelet queue* that stores the pending codelets. Since the application task graphs in the proposed resiliency framework are intended to be periodic, everytime a TP finishes its execution (i.e., all of its codelets have been executed), the TP invokes a replica of itself to be re-executed by inserting a new pending TP in the TP queue. Meanwhile, all the codelets of the new pending TP are inserted into the codelet queue as well. This process is repeated for a fixed number of iterations. In Algorithm 2, the condition in line 2 is satisfied since at the end of the last iteration, a new TP is placed into the TP queue. New codelets are added to the codelet queue in line 6. In line 4, cores are sorted and ranked to reflect the latest changes to the FVFs made by the FMD unit. The sorting process is protected by a lock, denoted as *RLlock*, which prevents CUs from modifying core FVFs when the sorting process is taking place.

Algorithm 2 Algorithm for DARTS Scheduling Unit (SU)

Input: pendingTPs, pendingCodelets, cores, core reliability level;
Output: scheduledCodelets;

```
1: while alive do
2:     if TPQueue.notEmpty() then
3:         RLlock.lock();
4:         sortAndRankCores();
5:         RLlock.unlock();
6:         TPQueue.pop()
7:     end if
8:     Codelet toExecute = popCodelet();
9:     requiredRL = toExecute.getRanking();
10:    if requiredRL > (TotalCUs - DeathCUs) then
11:        requiredRL = getRoundRobinRL();
12:    end if
13:    bestCore = getCoreByRL(requiredRL);
14:    if core.isDisabled() then
15:        requiredRL − −;
16:        bestCore = getCoreByRL(requiredRL);
17:    end if
18:    bestCore.scheduleCodelet(toExecute);
19: end while
```

Lines 8–19 of Algorithm 2 are repetitively executed as long as there are pending codelets in the codelet queue. Each codelet has an attribute that specifies the required core Reliability Level, which is pre-computed by the static scheduler. As it can be seen, the SU assigns codelets to cores according to the pre-computed RLs in line 9, and the codelet is issued to the chosen CU in line 18. Since all the cores are ranked, the algorithm does not need to perform any intensive computation when tuning task allocation as cores in the system become more or less reliable. However, unlike the scheduling heuristic in [17], Algorithm 2 uses a set of extra criteria to adaptively re-schedule tasks on top of the pre-computed task RLs:

- **Criterion 1**, if a CU becomes unrecoverable and therefore cannot be used anymore, tasks pre-mapped to that RL ranking should be evenly re-distributed among the remaining healthy cores to balance the workload. This criterion is implemented in lines 10–11 of Algorithm 2. The SU checks if the required RL is higher than the number of healthy cores in the system. If this is true, the SU picks an RL belonging to one of the healthy cores using a Round Robin policy.

Algorithm 3 Algorithm for DARTS Computing Unit (CU)

```
Input: pendingCodelets, faultTrace;
 1: while alive do
 2:     Codelet toExecute ⇐ popCodelet();
 3:     toExecute.fire();
 4:     if toExecute.executionStatus = FAULTY then
 5:         RLlock.lock();
 6:         this.computeFVF(FAULTY_RESULT);
 7:         RLlock.unlock();
 8:         this.disableCore();
 9:         toExecute.decreaseRanking();
10:         scheduler.pushCodelet(toExecute);
11:     else
12:         RLlock.lock();
13:         this.computeFVF(CLEAN_RESULT);
14:         RLlock.unlock();
15:     end if
16: end while
```

- **Criterion 2**, if a CU has recently experienced a fault, the SU will let the CU rest for a fixed amount of time to prevent the system from executing tasks in a CU that may still be affected by a fault. This criterion is called Δ-*idling* recovery and is taken from [10]. It is implemented in lines 14–17 of Algorithm 2. If the originally chosen core is disabled due to an ongoing fault, a more reliable core (with a smaller RL value) is chosen to execute the task by decreasing the $requiredRL$ attribute in line 13.

- **Criterion 3**, if the execution of a task on a specific core fails, to avoid repetitive failures, the task will be re-executed on a different CU. Since RL rankings are unique for each CU, this criterion requires a change of the corresponding RL ranking for any faulty task so as to assure that the re-execution is handled by another core. This criterion is implemented within the computing unit (CU) since it is responsible for checking task results and re-inserting faulty tasks into the codelet queue of the SU.

5.2.2 DARTS Computing Unit (CU)

DARTS CUs execute tasks assigned by the SU and check for faults upon completing a task. Algorithm 3 depicts its main functions. The unit has two inputs: a fault trace generated by the Fault Injection Unit, which determines the exact time of

a fault occurring and its duration, as well as the list of pending codelets which defines the set of codelets that should be executed by the CU.

In lines 2–3, the codelet is popped from the queue and executed. Then the CU checks the execution status of the codelet. If the result is faulty, the CU updates the FVF value (lines 5–7), disables the core for Δ units of time to fulfill Criterion 2 (line 8), decreases the RL ranking of the codelet to fulfill Criterion 3 (line 9), and inserts the codelet back into the codelet queue of the SU (line 10). On the other hand, if the result is clean, the CU simply updates the FVF value in lines 12–14 and proceeds to execute the next codelet. As in the SU algorithm, a lock is used to protect the update of FVFs, so as to prevent the core FVFs to be updated while the sorting process takes place.

Chapter 6

EXPERIMENTAL SETUP

This chapter describes the experimental setup employed to collect the results in section 7. We start with a description of the fault injection module, followed by the task graph inputs and state of the art approaches used for comparison.

6.1 Fault injection unit

To evaluate the proposed scheduling framework, we developed a Fault Injection Unit that statically generates **fault traces** for each Computing Unit within the system. A **fault trace** is an array that determines if a fault is occurring at each specific unit of time. When a task is running, this array is checked. If the time interval of the task overlaps with the occurrence of a fault in the fault trace then a faulty task execution occurs. To generate these fault traces, the failure probability of each core is modeled as a Weibull distribution [14] with parameters $\beta = 0.7$ and λ taking different values inside the interval $[0, 1]$. More specifically, the failure probability of each core is given by:

$$F(t) = 1 - e^{-\lambda t^{\beta}} \tag{6.1}$$

With t defined as the time elapsed since the last fault occurred in the core. Since $\beta < 1$, core fault rates decrease as t increases. In other words, a recently failed core tends to produce more failures.

To produce a fault trace, per each unit of time, the simulator generates random numbers that follow the distribution in Equation (6.1). Based on the produced number, one of the following two procedures are followed:

- If a fault is generated, the duration of the fault is set based on a uniform distribution that generates random numbers between $[1, \Delta]$. In this way, the duration of a fault is not constant, but follows a random pattern over a time window, imposing a continuous fault disturbance over a given time frame to the computational activities of the core.

- If no fault occurs, the fault injection unit inserts a *clean time frame* in the fault trace. This *clean time frame* defines a time window where the core produces clean task results. The duration of this time frame is constant and is defined based on the core fault rate λ: the higher the fault rate, the lower the value of the clean time frame and vice versa.

Using this approach, the Fault Injection unit statically generates the fault traces for each computing unit in the system up until the end of the simulation time interval. This method allows the use of the same fault traces for each tested approach to ensure fair comparison between different fault tolerance techniques.

6.2 State-of-the-art techniques used for comparison

The reliability framework proposed in this thesis is compared against three state-of-the-art scheduling techniques, all of which are implemented inside of DARTS:

- ***DARTS round robin scheduling [21]***, under this approach, the Scheduling Unit pushes codelets into each Computing Unit using a Round Robin distribution without any consideration for core reliability. This approach is used as the baseline.

- ***DARTS with Δ-Idling [10]***, under this approach, when a fault is detected within a CU, the affected CU is temporarily disabled in the system to prevent the system from executing additional tasks in a CU that may still be affected by a fault.

- ***Fault counter with Δ-Idling [10]***, under this approach, the proposed static task-to-RL mapping is used as input, but ranks cores based on a naive reliability metric that simply counts the number of faults that have occurred. The technique also classifies cores as healthy/unhealthy based on pre-defined fault thresholds, and temporarily disables cores upon observing a fault [10]. Cores with higher fault counts are less reliable and vice versa. Time correlation across faults is not taken into consideration.

Table 6.1: Machine specifications

Processor	Frequency	RAM	Cores
Intel Xeon CPU E5-2670	$2.60GHz$	$65.77GB$	8

6.3 Task graph inputs and system architecture

The input task graphs and the system architecture are shown in Table 4.3 and Table 6.1, respectively. Each task graph is an abstract representation of the actual applications they are built upon. This means that the task graphs are not performing any real computations. Instead, each node in the task graph only invokes a function that suspends the execution for a fixed amount of time to simulate different task execution times. Each of the task graphs models a different shape of task dependencies. For example, in the out tree task graph, there is a single root node and at each level of the tree the amount of parallelism increases.

Chapter 7

EXPERIMENTAL EVALUATION

To collect the results, different fault rate configurations are studied. A total of 8 cores are used in all scenarios. Per each fault rate configuration and application task graph, 10 different fault traces are defined per each core with varying λ values as input to reflect different fault rate distributions among the cores and therefore make the impact of lower/ higher reliability levels more observable. An increasing factor of $3X$ is employed during all simulations to emphasize the differences, in terms of faults, among different cores. For example, a possible core fault rate configuration for a system with 8 cores could be: $[0.0004, 0.0012,\ 0.0037, 0.0111, 0.0333, 0.1000, 0.3000, 0.9000]$, where each fault rate is 3 times bigger than the previous one. In order to account for variance due to scheduling and other OS noise, 10 simulations are carried out per fault trace. The average execution times and fault counts are computed across all simulations and the 10 different fault traces.

7.1 Overall framework performance

Tables 7.1, 7.2, 7.3 and 7.4 show the overall framework performance in terms of execution times and fault counts compared to other state-of-the-art techniques. The λ values used for this set of experiments are $[0.0004, 0.0012, 0.0037, 0.0111, 0.0333, 0.1000, 0.3000, 0.9000]$, corresponding to each core's fault rate. The reduction ratios for each approach are computed using DARTS as a baseline.

The proposed reliability framework tries to minimize the number of faults and re-executions by mapping the most critical tasks in the task graph to cores with the highest reliability ranks. To do this, it assigns FVFs to each of the cores in the system at runtime and produces a ranking based on these values. The accuracy of these

Table 7.1: Overall execution times of DARTS and DARTS with Δ- Idling.

	DARTS		DARTS with Δ- Idling		
Task Graph	Average (s)	Worst (s)	Average (s)	Worst (s)	Reduction (%)
FFT	9.07	9.17	6.65	6.76	26.75
Fork/ Join	10.45	10.49	8.66	8.79	17.12
Gaussian elimination	10.81	10.85	9.44	9.52	12.70
Laplace Equation	10.29	10.35	8.69	8.84	15.59
LU Decomposition	11.30	11.37	10.40	10.45	7.89
Out Tree	9.39	9.48	7.73	7.84	17.64
In Tree	9.86	9.95	8.13	8.20	17.52

measurements at runtime, determines the effectiveness of the static task-to-RL mapping. From the average execution times in Tables 7.1 and 7.2, it can be seen that the proposed technique improves execution times in all cases in comparison to DARTS, which only considers computing performance and not reliability issues. For the FFT task graph, the proposed technique achieves the highest reduction ratio of 30.70%. On the other hand, the fault counter approach performs poorly compared to DARTS in most of the cases. This is fundamentally due to the reliability metric employed by the fault counter approach, which does not necessarily reflect the actual state of cores in terms of reliability in most cases, as is highlighted in Section 5.1. Given the inaccurate reliability metric of the fault counter approach, the static task-to-RL mapping becomes less useful since the static RLs no longer correspond with the dynamic RLs, leading to poor performance. Additionally, the fault counter approach tends to mark cores as unrecoverable at a higher rate in comparison to the proposed technique. The reason is that the fault counter approach does not provide a fault weighting mechanism and instead treats all faults regardless of when they occurred within a core to be of equal weight. This in turn, means that cores that may be recovering can be erroneously marked as unrecoverable. Whereas, the proposed approach tackles this issue by reducing the FVF exponentially every time a clean task result is produced, as explained in

Table 7.2: Overall execution times of the fault counter approach and the proposed approach

	Fault counter with Δ- Idling			Proposed reliability framework		
Task Graph	Average (s)	Worst (s)	Reduction (%)	Average (s)	Worst (s)	Reduction (%)
FFT	7.44	7.52	18.03	6.29	6.38	30.70
Fork/ Join	9.23	9.35	11.69	7.63	7.74	26.97
Gaussian elimination	11.16	11.32	−3.19	9.34	9.63	13.59
Laplace Equation	10.30	10.41	−0.10	8.99	9.18	12.64
LU Decomposition	11.71	11.78	−3.65	9.33	9.42	17.43
Out Tree	12.94	13.29	−37.84	8.00	8.04	14.84
In Tree	12.06	12.78	−22.34	8.28	8.39	16.02

Section 5.1. As a result, cores are less likely to be marked as unrecoverable when they are infact recovering. Finally, in terms of performance, DARTS with Δ-idling exhibits execution times slightly higher than the proposed technique on average. However, as shown in Tables 7.3 and 7.4, in terms of fault counts, DARTS with Δ-idling performs much worse than the proposed technique.

Tables 7.3 and 7.4 shows average fault counts of different state of the art techniques compared to our approach. It can be seen that the proposed reliability framework improves fault counts in all cases in comparison to DARTS. The proposed technique achieves the highest reduction ratio of 72.17% for the Fork/Join application task graph. The fault counter approach also outperforms DARTS and has higher reduction ratios in comparison to the proposed technique. However, since the fault counter approach overloads the most reliable cores in the system as faulty cores become unrecoverable at a faster rate, this approach minimizes fault counts at the cost of higher execution times. DARTS with Δ-idling also improves fault counts in comparison to DARTS. However, as shown in Table 7.3, the fault counts of DARTS combined with Δ-idling are considerably higher than those produced with the proposed technique by

Table 7.3: Fault counts of DARTS and DARTS with Δ- Idling

	DARTS		DARTS with Δ- Idling		
Task Graph	Average	Worst	Average	Worst	Reduction (%)
FFT	2457.49	2476.60	1394.28	1438.10	43.26
Fork/ Join	2380.47	2417.20	1597.19	1635.80	32.90
Gaussian elimination	2476.60	2501.00	1734.68	1761.5	29.96
Laplace Equation	2505.61	2538.60	1540.25	1599.4	38.53
LU Decomposition	2173.95	2201.80	1554.42	1577.40	28.50
Out Tree	3875.20	3949.10	2308.55	2364.20	40.43
In Tree	3820.56	3858.20	2442.59	2478.70	36.07

almost two fold in most cases. Because higher fault counts trigger a higher number of task re-executions as well as system recovery mechanisms, this adversely affects application performance.

One of the goals of using a statically generated task-to-RL mapping is to maintain high stability across different runs at runtime. The static task-to-RL mapping retains intact regardless of runtime core reliability variations, which assures that the performance impact of faults, and therefore re-executions, is minimal. This can be seen from the worst case and average case execution times and fault counts in Tables 7.2 and 7.4. The proposed technique exhibits a low variance and therefore high stability among differing program executions and fault traces. Finally, the results also show how the proposed reliability framework achieves a balance between execution times and fault counts, unlike other techniques, such as the fault counter approach and DARTS with Δ-idling, that exhibit a one sided behavior. These approaches provide an interesting case study and show the spectrum that fault tolerance approaches may run from undercompensating for faults to overcompensating for faults: on one hand, an approach may largely reduce faults at the cost of overloading the reliable cores and degrading performance; on the other hand, it may not reduce enough faults and hence trigger considerable amount of re-executions that also degrade performance.

Table 7.4: Fault counts of the fault counter approach and the proposed approach

	Fault counter with Δ- Idling			Proposed reliability framework		
Task Graph	Average	Worst	Reduction (%)	Average	Worst	Reduction (%)
FFT	410.00	420.90	83.32	779.96	855.50	68.26
Fork/ Join	434.92	450.10	81.73	662.47	692.60	72.17
Gaussian elimination	421.77	442.00	82.97	971.50	1131.20	60.77
Laplace Equation	421.80	435.70	83.17	822.86	880.60	67.16
LU Decomposition	424.98	442.40	80.45	726.62	743.20	66.58
Out Tree	450.80	465.10	88.37	1149.06	1195.00	70.35
In Tree	441.23	452.70	88.45	1142.63	1245.10	70.09

7.2 Overhead Analysis

Table 7.5 shows the time overhead (in milliseconds) associated with each approach. While such overhead is already included in the execution times shown in Tables 7.1 and 7.2, this section provides a detailed discussion of these overheads and breaks them down by approach. The overhead shown in the table measures the amount of time the dynamic scheduler spends in scheduling tasks to computational units using the associated reliably technique. These measurements constitute a worse case scenario because they do not account for the fact that during actual execution, scheduling may occur in parallel with computation. For DARTS, the overhead shown in the table only reflects the time required to perform scheduling decisions since DARTS does not take into consideration reliability issues. For the fault counter approach and the proposed reliability framework, the time overhead reflects the time required to make scheduling decisions, as well as, time required to sort the cores in the system based on the values of the reliability metric, so as to compute the rankings needed to map tasks onto cores based on the static task-to-RL mapping provided as input. We note here that the values of the reliability metrics cannot be modified by Computational Units

Table 7.5: Time overheads of the evaluated techniques

Task graph/ Approach	DARTS (ms)	DARTS with Δ-idling (ms)	Fault counter (ms)	Proposed (ms)
FFT	0.225	0.471	1.928	0.819
Fork/ Join	0.245	0.533	1.971	0.881
Gaussian elimination	0.184	0.545	1.016	0.794
Laplace Equation	0.202	0.546	1.955	0.742
LU Decomposition	0.213	0.616	2.251	1.161
Out Tree	0.323	0.921	3.426	2.046
In Tree	0.302	0.893	3.511	2.181

while the sorting process takes place in the Scheduling Unit, as mentioned in Section 5.2.1. To assure this condition, the scheduling algorithm employs a lock that imposes some serialization during the execution of the application. The time penalty for using the lock during the sorting process depends on the time required to sort the reliability metrics of the cores. Finally, the overhead for DARTS with Δ-idling only reflects the time the dynamic scheduler spends making scheduling decisions.

From Table 7.5, it can be seen that DARTS has the lowest overhead, which is expected, since it does not perform additional computations to take reliability into consideration. DARTS with Δ-idling also exhibits relatively low overhead in comparison to other techniques because the additional computations added for Δ-idling are minimal.The fault counter approach and the proposed approach have the highest performance overheads because they consider additional criteria such as core rankings, Δ-idling, and must avoid re-executing a task using the same core when a fault is detected, as explained in Section 5.2.1. Although these two approaches employ similar scheduling techniques, the proposed approach exhibits a much lower overhead than the fault counter approach. The primary difference between them is the reliability metric

used to rank cores in the presence of faults. This difference can have a significant impact on the number of swaps performed by the sorting algorithm to sort the reliability metrics in an increasing order to produce core rankings. An approach that minimizes changes to reliability values from one period to another will take less time to sort since element positions will swap less often resulting in a lower serialization penalty, because of the lock, and ultimately lower overhead.

Additionally, from Table 7.5 and Tables 7.1 and 7.2, it can be seen that in comparison to overall execution time, overhead is negligible, ranging from 0.002% of the execution time for DARTS and DARTS with Δ-idling, and 0.01% of the execution time for the fault counter and the proposed approaches. Although the time overhead imposed by the proposed technique is higher than some of the other approaches, the overhead represents such a small fraction of the time spent during execution that our proposed technique outperforms both DARTS and DARTS with Δ-idling, as shown in section 7.1. The fault counter approach also has a very small overhead, but its performance is not as good as the proposed technique in terms of overall execution times.

7.3 Fault Rate Scalability

Fig. 7.1 and 7.2 respectively show the normalized average execution times and fault counts of different schemes under different fault rate configurations. The x axis in both figures represents the highest core fault rate in the system which varies from 0.1 to 0.9. An increasing factor of $3X$ and a total core count of 8 are still employed. For example, when the highest core fault rate is 0.9, the fault rate configuration of the 8 cores is: $[0.0004, 0.0012,\ 0.0037, 0.0111, 0.0333, 0.1000, 0.3000, 0.9000]$. Values in the figures are the average execution time and fault count values gathered over all task graphs, which are further normalized to the maximum execution time and fault rate obtained.

From Fig. 7.1 and 7.2, it can be seen that as system fault rates increase, execution times and fault counts slowly increase as is expected but remain stable for all

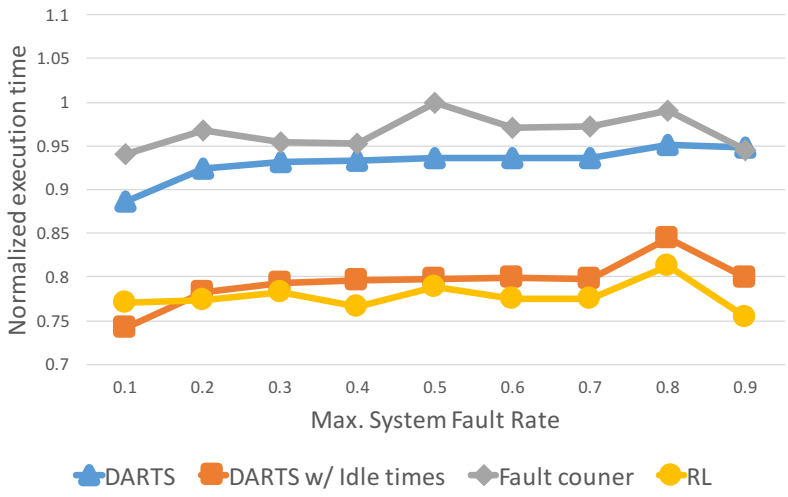

Figure 7.1: Normalized application execution times as fault rates increase

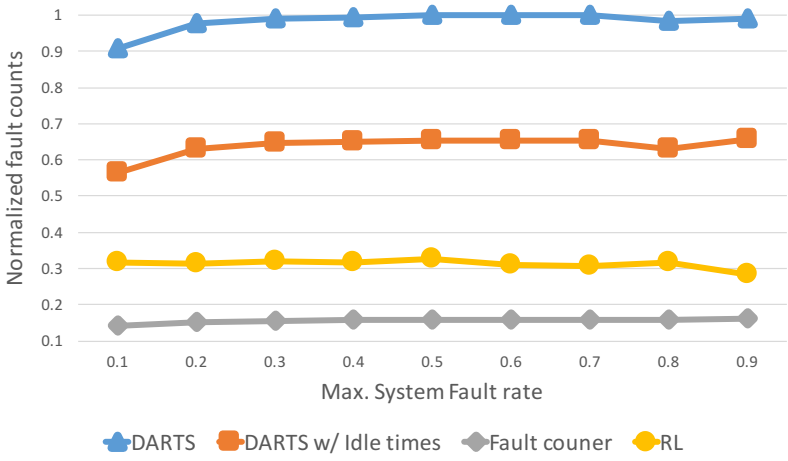

Figure 7.2: Normalized fault counts as system fault rates are increased

approaches, showing that the behavior is predictable across different system fault rates. However, as shown in Fig. 7.1, the proposed approach outperforms both DARTS and the fault counter scheme by considerable margins. Since the proposed scheme tries to minimize faults occurring at runtime by allocating the most critical tasks to the most reliable cores in the system, the impact of increasing fault rates is minimal. In contrast, the fault counter approach has the lowest fault counts but the highest execution

times among all the approaches. As explained in Section 7.1, this is due to the fact that the fault counter approach overloads the most reliable cores in the system with many tasks, which minimizes fault counts at the expense of longer execution times. In comparison, although the proposed reliability framework has higher fault counts , it is capable of making a more even distribution of tasks among cores. Since DARTS does not account for faulty cores, it performs poorly as fault rates increase. From Fig. 7.2, it can be seen that DARTS has the highest fault counts in all scenarios, which is expected since it does not take reliability into consideration. DARTS with Δ-idling exhibits similar behavior to the proposed reliability framework in terms of execution times, and in most cases performs slightly worse than the proposed technique. However, in terms of fault counts, DARTS with Δ-idling is not as good as the proposed scheme. Fig. 7.2 shows that DARTS with Δ-idling has $2X$ more fault counts than the proposed approach, which may lead to performance bottlenecks in realtime systems due to the cost associated with task recovery. Also, since fault traces are randomly generated, some fault traces may affect a higher number of critical tasks, leading to a more observable degradation in execution time than other fault traces. The peaks towards the end of Fig. 7.1 reflect such variation.

To conclude, the results show that the proposed reliability framework is able to keep low execution times and fault counts as system fault rates increase in comparison to other state of the art techniques, as well as, to achieve a balance between execution times and fault counts by optimizing for both.

Chapter 8

CONCLUSION

This thesis presents a reliability-aware task scheduling framework that combines static optimizations and runtime adaptibility to simultaneously minimize the number of faults occurring at runtime as well as the overall application performance. The static scheduling approach models the impact of core fault rates on task execution time to generate a task-to-RL mapping, optimized with a genetic algorithm which proves to be more effective than the heuristic employed in [18]. At runtime, the RL-to-core mapping is updated based on the time correlation of detected faults to reflect the actual conditions of the computational resources, which implicitly modifies the final task-to-core mapping. The runtime scheduling heuristic in [17] is improved to take into consideration additional criteria such as an even distribution of tasks among healthy cores when a core becomes unrecoverable, use of an alternate core for re-execution of tasks, and the inclusion of a state-of-the-art technique named Δ-idling. By combining the scheduling approaches in [18] and [17], most of the computational overhead is relieved from the runtime system since task requirements for core reliability are pre-computed statically, and runtime adaption is performed in a predetermined manner. By implementing the proposed framework in DARTS, a state-of-the-art runtime system, we show that the proposed technique achieves a balance between execution times and fault counts, and outperforms other state-of-the-art techniques that exhibit one-sided behavior by either undercompensating or overcompensating for faults. The proposed technique can reduce execution times by up to 30% and fault counts by up to 72%, compared to the baseline. The proposed framework also scales well and maintains low execution times and fault counts as system fault rates increase.

BIBLIOGRAPHY

[1] J. Bean. Genetics and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6:154–160, 1992.

[2] B. D. Beasley, D. and R. Martin. *An overview of genetic algorithms: Part 1, fundamentals.*, volume 15. Inter-University Committee on Computing, 1993.

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

[4] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, Nov 2005.

[5] T. Chantem, Y. Xiang, X. Hu, and R. P. Dick. Enhancing multicore reliability through wear compensation in online assignment and scheduling. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1373–1378, March 2013.

[6] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *Micro, IEEE*, 23(4):14–19, July 2003.

[7] A. Coskun, T. Rosing, K. Whisnant, and K. Gross. Temperature-aware MPSoC scheduling for reducing hot spots and gradients. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 49–54, March 2008.

[8] N. Gottumukkala, C. Leangsuksun, N. Taerat, R. Nassar, and S. Scott. Reliability-aware resource allocation in hpc systems. In *IEEE International Conference on Cluster Computing*, pages 312–321, Sept 2007.

[9] N. Gottumukkala, C. Leangsuksun, N. Taerat, R. Nassar, and S. Scott. Reliability-aware resource allocation in HPC systems. In *IEEE International Conference on Cluster Computing*, pages 312–321, Sept 2007.

[10] M. Haque, H. Aydin, and D. Zhu. Real-time scheduling under fault bursts with multiple recovery strategy. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 63–74, April 2014.

[11] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai. Impact of cmos process scaling and soi on the soft error rates of logic processes. In *VLSI Technology, 2001. Digest of Technical Papers. 2001 Symposium on*, pages 73–74, June 2001.

[12] J. Huang, J. Blech, A. Raabe, C. Buckl, and A. Knoll. Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 247–256, Oct 2011.

[13] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Scheduling and optimization of fault-tolerant embedded systems with transparency/performance trade-offs. *ACM Trans. Embed. Comput. Syst.*, 11(3):61:1–61:35, Sept. 2012.

[14] I. Koren and M. Krishna. *Fault-Tolerant Systems*. Organ Kaufmann, 2007.

[15] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, May 1996.

[16] T. Li, M. Shafique, J. Ambrose, S. Rehman, J. Henkel, and S. Parameswaran. Raster: Runtime adaptive spatial/temporal error resiliency for embedded processors. In *50th ACM / EDAC / IEEE Design Automation Conference (DAC)*, pages 1–7, May 2013.

[17] L. Rozo, J. M. M. Diaz, and C. Yang. Improving mpsoc reliability through adapting runtime task schedule based on time-correlated fault behavior. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 818–823, San Jose, CA, USA, 2015. EDA Consortium.

[18] L. Rozo and C. Yang. Guiding fault-driven adaption in multicore systems through a reliability-aware static task schedule. In *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, pages 612–617, Jan 2015.

[19] W. Spears and K. D. Jong. On the virtues of parametrized uniform crossover. *Proc. of the fourth International Conference on Genetic Algorithms*, pages 230–236, 1991.

[20] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The impact of technology scaling on lifetime reliability. In *International Conference on Dependable Systems and Networks*, pages 177–186, June 2004.

[21] J. Suettlerlein, S. Zuckerman, and G. R. Gao. An implementation of the codelet model. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, 2013.

[22] C. Yang and A.Orailoglu. Tackling resource variations through adaptive multicore execution frameworks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(1):132–145, Jan 2012.

[23] C. Yang and A. Orailoglu. Fully adaptive multicore architectures through statically-directed dynamic execution reconfigurations. In *18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SoC)*, pages 396–401, Sept 2010.

[24] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.