Improving MPSoC Reliability through Adapting Runtime Task Schedule based on Time-Correlated Fault Behavior*

Laura A. Rozo Duque, Jose M. Monsalve Diaz, and Chengmo Yang
Electrical and Computer Engineering
University of Delaware
140 Evans Hall, Newark, DE 19716
Email: {lrozo, josem, chengmo}@udel.edu

***Abstract*—The increasing susceptibility of multicore systems to temperature variations, environmental issues and different aging effects has made system reliability a crucial concern. Unpredictability of all these factors makes fault behavior diverse in nature, which should be considered by the runtime task scheduler to improve overall system reliability. To achieve this goal, this paper proposes a fault tolerant approach to model core reliability at runtime and tune resource allocation accordingly. Given variations in fault duration, we propose a reliability model capable of tracking not only faults appeared in each core but also their correlation in time. Taking this model as an input, a runtime scheduling algorithm that allocates critical and vulnerable tasks to reliable cores is also proposed. Experimental results show that the proposed adaptive technique delivers up to 56% improvement in application execution time compared to other techniques.**

I. Introduction

The increasing complexity of multicore systems, together with the unending demand for higher performance and less energy consumption, keeps pushing the trend of shrinking device sizes and increasing core counts on a single chip. This trend is accompanied however with an increasing susceptibility of these chips to faults due to different factors [1], such as radiations, noise fluctuations, and aging effects. Not only do these factors lead to elevated fault rates in todays systems, but furthermore their correlation makes the fault behavior more diverse in nature. Temperature variations due to increasing power densities are accelerating aging effects (e.g. Electro migration [1], [2], Thermal cycling [3], etc), which in turn increase the number of intermittent and permanent faults. Meanwhile, environmental issues are also affecting fault behavior, producing more transient faults. Together these factors cause fault durations to vary over nanosecond to second time scales.

With faults occurring continuously and their durations varying significantly, it is desirable for fault resilience solutions to deliver runtime adaptability. Previous solutions [4], [5], [6] proposed solely for permanent or transient faults will no longer suffice as they assume a fault consistently manifesting or never re-manifesting itself. Instead, the combined effects of both types of faults need to be bear in mind, and runtime fault tolerant solutions need to be able to monitor fault duration, adaptively recover task execution, and adjust resource allocation. Transient and intermittent faults, for example, affect a processing unit for a while and then disappear. During that time period, it is desirable to decrease the number of tasks allocated to that unit. Once the fault disappears, tasks can be allocated to that unit at the normal speed. In comparison, processing units with permanent faults should be isolated, since allocating tasks to them will have an adverse impact on system performance.

*This work is supported by NSF grant #1253733.

Adaptively allocating resources, while desirable, requires a cost-effective solution to characterize fault behavior promptly at runtime. Yet current solutions [7] have a very limited capability to react to faults due to the small number of defined system states. As the only recognized resource states are healthy and unhealthy, the reaction is just to exclude those unhealthy units from being used, which is quite abrupt and only happens upon identifying a permanent fault. In comparison, we propose to let the system react to faults in a much smoother way. As faults appear, intermediate system states will be defined, allowing the system to progressively reduce the frequency of job assignment to a processing unit until it becomes non-functional.

The proposed fault tolerant framework is composed of two parts. First, we develop a *core reliability model* capable of tracking not only faults appeared in each core but also their correlation in time. Through monitoring fault occurrence frequency, healthy, unhealthy and intermediate states are defined for cores, allowing them to be ranked for task allocation. Second, we propose a *reliability-driven scheduling* approach to adaptively allocate the most critical and vulnerable tasks to those most reliable cores. To relieve runtime overhead, application information regarding task criticality, vulnerability, and mobility is statically extracted and delivered to the runtime system. In this way, the amount of computational overhead imposed by the proposed fault tolerant framework can be minimized.

The rest of this paper is organized as follows. Section II reviews the related work in fault tolerant scheduling. Section III presents the adaptive runtime fault tolerant technique. Section IV presents the experimental methodology and obtained results, while Section V concludes the paper.

II. Related Work

Existing work in fault tolerant scheduling techniques can be divided into three categories: static, dynamic and hybrid approaches. Static approaches [8], [9] rely on offline analysis to generate fixed task schedules capable of tolerating up to a fixed number of faults. Solutions for tolerating both transient [8] and permanent [9] 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.* [10], [11] have developed adaptive static schedules to attain runtime adaptation in the face of resource variations in MPSoCs. No spare resource is required. Instead, these schedules are partitioned into regular yet shiftable band structures, thus maximizing resource

978-3-9815370-4-8/DATE15/©2015 EDAA

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.* [4] 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.* [5] 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 large amount of data processing to make reconfiguration decisions, making them less suitable for applications with strict time constraints.

Bolchini *et al.* [7] 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 over 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 [12], 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.* [6] 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.

III. Fault-duration aware task scheduling

This section presents the proposed fault tolerant framework that monitors time-correlated fault behavior to define reliability levels of different resources, and gradually tunes task allocation based on such information. We first give a brief overview of the framework, and then describe the proposed core reliability models, as well as task vulnerability and criticality models. Taking these models as inputs, an adaptive task scheduling is presented at the end.

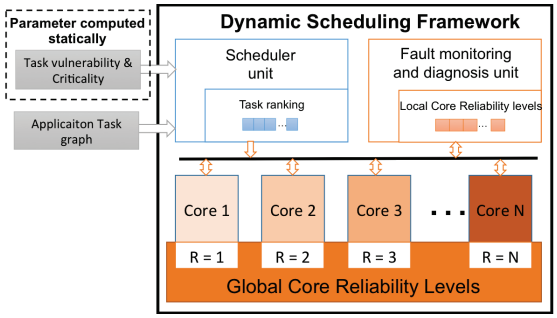

Fig. 1. Runtime fault tolerance framework

A. Framework overview

Our fault tolerant framework is proposed for multicore platforms with homogenous Processing Units (PU). However, these PUs are diverse not only in their fault rates but also in their fault duration, which in turn influence scheduling decisions. The target set of faults includes transient, permanent, and intermittent faults with varying durations. It is assumed that the system already has mechanisms [13] such as acceptance test, control flow checking, algorithm-based fault tolerance, or redundant threading to detect faults in task results. Once a faulty result is detected, the task will be re-executed on a different PU.

The overarching goal of the proposed reliability framework is to schedule tasks to cores with non-constant fault behavior in a way that minimizes the execution time of the entire application. When a fault corrupts the result of a task, the task has to be re-executed by another PU, and the start time of the dependent tasks has to be delayed as a result. In this scenario, three factors can be used to model the impact of faults on application execution time:

- **Core reliability** indicates the fault rate of a core.
- **Task vulnerability** indicates the probability that a fault will corrupt task results.
- **Task criticality** indicates the impact of a faulty task result on application execution time.

Among these three factors, core reliability is expected to be affected by execution conditions, while task vulnerability and task criticality are specific to an application. To relieve most of the computational overhead at runtime, the proposed fault tolerant framework extracts the last two factors based on application task graph information *a priori*. Meanwhile, a cost-effective core reliability model is developed for the runtime system to monitor time-correlated fault behavior of each core. In this way, the runtime scheduler can easily assign more vulnerable and critical tasks to more reliable cores.

Fig. 1 presents an overview of the runtime fault tolerant framework, composed of a *scheduler unit* and a *Fault Monitoring and Diagnosis (FMD) unit*, together with the set of cores available in the system. The scheduler is responsible for monitoring task execution and upon a fault, initiating re-execution and postponing dependent tasks. The FMD unit is responsible for recording fault occurrences and updating core reliability levels periodically. As can be seen, task criticality and vulnerability information is considered as an external input to the scheduling framework.

B. Modeling Core Reliability

In the proposed fault tolerance framework, the reliability level of each core is computed by the FMD unit. Instead

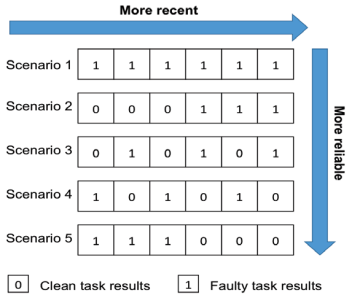

Fig. 2. Fault scenarios that each shows a fault history.

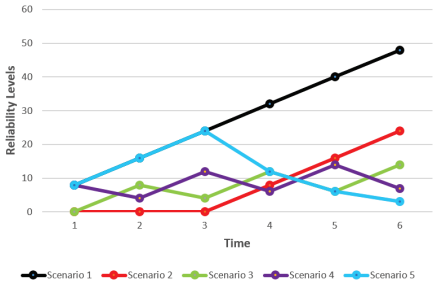

Fig. 3. Changing reliability levels of the fault cases in Fig. 2, computed with Algorithm 1. A smaller value implies more reliable.

of simply counting the number of occurred faults, the FMD unit models time-correlated fault behavior. This implies that faults occurred at different time should have different impact to core reliability level. To illustrate this time-correlated behavior, Fig. 2 depicts a set of scenarios, ranked in ascending order in terms of how reliable they are. Each scenario shows a 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 elapsed, 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 should keep decreasing. In an extreme case, such as Scenario 1 shown in Fig. 2, the core has not produced any clean result within the observed window, and hence should be considered "dead".

Another important consideration is that consecutive occurrences of faults should have a larger impact than faults occurred in a random and discrete manner. This can be observed through a comparison between Scenarios 2 and 3 in Fig. 2. While their numbers of faults are equal, faults in Scenario 2 occurred in a continuous manner, indicating high correlation among fault behavior (probably 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 elapsed. This situation is shown in Fig. 2 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 recent task results, the higher the reliability of the core.

Overall, the scenarios depicted in Fig. 2 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 long time ago.
- Continuously occurred faults should be given a higher weight than random and discrete faults.
- A clean task result should suppress the accumulating impact of previous faults.

To fulfill these criteria, we propose Algorithm 1, which can be efficiently implemented in hardware. Every time a task result is checked, the algorithm is invoked to update *core reliability level*, and a smaller value indicates that the core is more reliable. If the task result is clean, the reliability level of the corresponding core is divided by 2. Otherwise, the result is faulty, and the algorithm adds a specific $faultWeight$ to the current reliability level. 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 effect. Using $faultWeight = 8$ as an example, the core reliability levels computed with Algorithm 1 for the five scenarios in Fig. 2 are shown in Fig. 3.

One significant advantage of Algorithm 1 is that it can be efficiently implemented in hardware. As shown in Fig. 4, only an adder, a shifter, a 2-to-1 multiplexer, and a register are needed, as the division operation can easily be accomplished through shifting operations.

Furthermore, Algorithm 1 also indicates that the maximum possible value of the reliability level equals $faultWeight * FHLength$, with $FHLength$ denoting the maximum length of fault history to keep track. Accordingly, a register of $\log_2(faultWeight * FHLength)$ bits is sufficient for recording the reliability level of each core. As a concrete example, if $faultWeight = 8$ and $FHLength = 8$, the size of the register is only 6 bits. Note that the fault history information shown in Fig. 2 is only for presenting those scenarios. It is not recorded in the proposed fault tolerant framework, and the FMD unit only needs to maintain the reliability Level for each core.

Finally, it needs to be noted that Algorithm 1 only updates the *local* core reliability level. As shown in Fig. 1, the *global* core reliability levels, which are accessed by the scheduler unit, are periodically updated by the FMD unit. This excludes any potential race condition between the scheduler and the FMD unit in accessing reliability levels. This also implies that

Algorithm 1 Local Reliability Level

```
1: Input: taskResult ∈ {faulty, clean}
2: Output: LocalReliabilityLevel_i
3: procedure LocalReliabilityLevel(taskResult)
4:     if taskResult = faulty then
5:         LocalReliabilityLevel_i += faultWeight
6:     else
7:         LocalReliabilityLevel_i = LocalReliabilityLevel_i / 2
8:     end if
9: end procedure
```

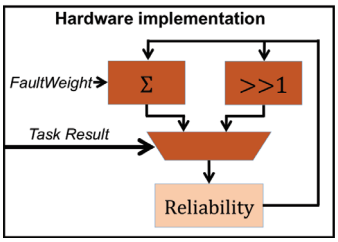

Fig. 4. Hardware implementation of FMD unit

between two consecutive updates to the global reliability levels, the scheduling policy is relatively stable.

C. Modeling task criticality and vulnerability

As mentioned before, task vulnerability and criticality are application-specific factors. Before executing an application, the proposed fault tolerant framework will invoke a procedure to analyze the application task graph and extract such information, relieving in this way most of the runtime computational overhead.

In a nutshell, the static analysis procedure computes criticality and vulnerability factors for each task, as well as a unified priority metric that combines these two factors in a single quantity. The inputs to this procedure are the application task graph and a reliability weight, α, in the range of $[0,1]$. This parameter allows the user to define the relative importance of reliability over performance. As shown in the following equation, a larger value of α will make the runtime system prioritize reliability over performance, and vise versa.

$$priority_i = \alpha * vulnerability_i + (1-\alpha) * criticality_i \quad (1)$$

Task vulnerability measures the probability that a fault will corrupt task results. Since the proposed fault tolerant framework targets to faults occurring in task execution, the longer the execution time is, the more possible that the task result may become faulty. In light of this observation, the vulnerability factor of task i can simply be computed as follows:

$$vulnerability_i = \frac{ET_i}{Max(ET_i)} \quad (2)$$

Here, ET_i represents the execution time of task i, while $Max(ET_i)$ retrieves the maximum execution time among all tasks in the task graph. This equation indicates that the value of $vulnerability_i$ is in the range of $[0,1]$, and linearly proportional to task execution time. A higher value indicates a longer task and hence, a higher possibility for faults to corrupt the result. The task with the largest execution time will be most vulnerable and therefore will have a vulnerability factor of 1.

The second factor, **task criticality**, measures the impact on application performance that a task has, if it is delayed due to faults or resource unavailability. Intuitively, this factor can be measured from two perspectives. First, a task with a smaller slack will have a larger impact on the overall schedule length if it is delayed. Second, a task with more dependents will have its delay impact amplified. Both aspect have a direct effect on application performance. It is therefore necessary for the static analysis procedure to take both of them into account when modeling task criticality.

Slack measures the amount of flexibility for a task to be delayed without degrading application performance. The static analysis procedure obtains the slack of a task by computing the difference between the Earliest Start Time (EST) and Latest Start Time (LST) of the task in the task graph. Subsequently, the obtained value is normalized to the maximum slack value to get a fixity value in the range of $[0,1]$:

$$fixity_i = 1 - \frac{slack_i}{Max(slack_i)} \quad (3)$$

Here, $fixity_i$ and $slack_i$ respectively represent the fixity and slack of task i, while $Max(slack_i)$ computes the maximum slack value among all the nodes in the task graph. A higher value of $fixity_i$ indicates that application performance is more sensitive to the delay of task i. For all the tasks on the critical path, their fixity value is 1.

Once task i is delayed, its dependent task k, if any, will probably be delayed as well. A task with more dependents will have this impact amplified. Taking such a propagation effect into consideration, the overall criticality factor is computed by summing up the fixity of a task as well as its dependent tasks, as shown below:

$$criticality_i = \left(fixity_i + \frac{\sum_{k \in children_i} fixity}{Max(OutDegree_i)} \right) / 2 \quad (4)$$

Here, $OutDegree_i$ is the number of dependent tasks of a node i and $Max(OutDegree_i)$ retrieves the maximal number of out-degree in the task graph. The first term models the effect of delaying tasks, while the second term models the number of dependent tasks that will be affected because of this delay. Since both the fixity value and the second term are in the range of $[0,1]$, a normalization with respect to 2 delivers a criticality factor in the range $[0,1]$. A higher value of $criticality_i$ indicates that if the task is delayed, it will have a larger probability to degrade application performance through one or multiple dependences chains.

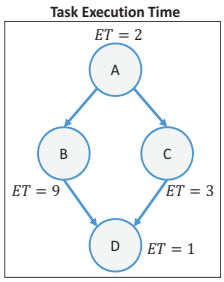

Fig. 5. A sample task graph. ET refers to execution time.

TABLE I. VULNERABILITY AND CRITICALITY OF FIG. 5 TASKS.

	Vulnerability	Slack	Fixity	Criticality
A	0.222	0	1	0.75
B	1	0	1	0.75
C	0.333	6	0	0.25
D	0.111	0	1	0.5

As a concrete example, Fig. 5 shows a graph of four tasks, their execution time and inter-task dependencies. The values of vulnerability, slack, fixity, and criticality of each task are shown in Table I.

D. Adaptive task scheduling

With the core reliability levels computed and updated by the FMD unit, as well as task criticality and vulnerability

extracted based on the task graph, the dynamic scheduler is able to prioritize task assignment. This process is shown in Algorithm 2. As the FMD unit already delivers a core ranking to the scheduler unit, the scheduler only needs to rank tasks according to their priority. To do so, The scheduler first checks all the tasks in the pending list to identify ready tasks (lines 1 to 7). A task is considered "ready" if it has no predecessor or all of its predecessors have been scheduled. These tasks are inserted to the ready queue according to their predefined scheduling priority, extracted with Equation (1). After that, the algorithm schedules the first task in the ready queue, that is, the one with the highest priority (lines 8 to 12), by mapping it to the most reliable idle core at the time. Then this task is added to the scheduled list, and some dependent constraints can possibly be cleared and more tasks may be added to the ready queue.

Algorithm 2 Dynamic Scheduling Algorithm

```
Input: pendingTasks, task priority, cores, core reliability level;
Output: scheduledTasks;
 1: while pendingTasks.isNotEmpty or readyTasks.isNotEmpty do
 2:     for all task ∈ pendingTasks do
 3:         if PredecessorScheduled(task) then
 4:             readyTasks.insert(task, task.priority);
 5:             pendingTasks.delete(task);
 6:         end if
 7:     end for
 8:     toSchedule = readyTasks.first();
 9:     bestCore = findMostReliableAndIdleCore(toSchedule);
10:     bestCore.scheduleTask(toSchedule);
11:     scheduledTasks.add(toSchedule);
12:     readyTasks.delete(toSchedule);
13: end while
```

Algorithm 2 confirms that the scheduler unit does not need to perform any intensive computation. Instead, since all the ready tasks and idle cores are ranked, the algorithm imposes minimum overhead when tuning task allocation as cores in the system become more or less reliable. Another noticeable point is that if a task fails during execution (because a fault is observed), it will be re-inserted to the head of the ready queue. Then the scheduler will check its original core assignment and will assign the task to a different and more reliable core.

IV. Results

A. Methodology

To evaluate the proposed scheduling framework, we have developed a simulator that injects faults as it executes tasks. The simulator implements the algorithms depicted in previous sections in C++. To model time-correlated fault behavior, the failure probability of each core is not constant, but model as a Weibull Distribution [13], with parameters $\beta = 0.7$ and λ varying between 0.002 and 0.02. More specifically, the failure probability of each core will be given by:

$$F(t) = 1 - e^{-\lambda t^{\beta}} \tag{5}$$

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

To insert faults, the simulator generates random numbers that follow the distribution given in Equation (5). Failures are checked both at the beginning and at the end of each execution.

TABLE II. Standard task graphs used

Task Graph	Number of nodes	Average computation/ communication time
Fast Fourier Transform (FFT)	39	20/10
Fork/ Join application	43	20/10
Gaussian elimination	49	20/10
Laplace Equation	42	20/10
LU Decomposition	44	20/10
Out Tree task graph	63	20/10
In Tree task graph	63	20/10

If a fault occurs during task execution, the task will be re-executed, and all the dependent tasks will be delayed as a result. In this way, the simulator is able to model runtime schedule length, the system architecture and varying fault rates for each core.

To show the advantage of the proposed reliability model and the adaptive scheduler, it is compared against two other schedules:

1) A baseline approach that considers only task criticality and vulnerability but not core reliability when mapping tasks into cores.
2) A dynamic technique that takes into consideration task criticality and vulnerability, but computes core reliability levels simply by counting the number of faults occurred. Cores with higher counts of faults will be less reliable and vice versa. With this approach, time correlation across faults is not taken into consideration.

To mitigate the impact of randomness and obtain more accurate results, hundreds of simulations are performed for each task graph under the same system configuration. At the end, the average schedule lengths obtained in different simulations and the average number of faults per simulation are reported for each approach. The reductions in schedule lengths compared to the baseline approach are reported as well.

We use different standard task graphs as the input set, with their characteristics summarized in Table II. Regarding system architecture, four cores with λ values of 0.002, 0.005, 0.01 and 0.02 are used during runtime simulation. The goal of using this fault rate distribution is to make the impact of lower/higher reliability levels more observable. The proposed scheduling technique will try to minimize the number of faults and re-executions by mapping tasks with the highest priorities to the most reliable cores. On the other hand, the scheduler that does not consider core reliability will tend to produce higher number of faults at runtime and hence, will degrade overall schedule lengths.

B. Results

The experimental results for the different task graphs are shown in table III. A total number of 300 simulations have been made by the runtime simulator for each task graph and each scheduling approach. Due to lack of space, this table only shows results for task prioritized based on $\alpha = 0.5$, indicating that vulnerability and criticality are equally important for ranking tasks.

From the average schedule length values in Table III, it can be seen that the proposed technique improves schedule length in almost all cases, compared to the baseline which does not take into consideration core reliability levels. The proposed technique achieves the highest reduction ratio of 56% for LU

TABLE III. COMPARISON OF A RELIABILITY-UNAWARE TECHNIQUE, THE PROPOSED TECHNIQUE, AND A FAULT-COUNT BASED TECHNIQUE.

	Baseline		Proposed technique			Fault count based		
Task Graph	Fault count	Schedule length	Fault count	Schedule length	Reduction (%)	Fault count	Schedule length	Reduction (%)
FFT	541	241.47	471	236.04	2.25	486	237.69	1.56
Fork/ Join	664	357.03	589	350.38	1.86	583	348.50	2.39
Gaussian elimination	812	411.22	630	404.30	1.68	699	410.58	0.16
Laplace Equation	296	426.59	38	396.00	7.17	375	412.65	3.27
LU Decomposition	8523	1080.91	488	469.61	56.55	3439	742.93	31.27
Out Tree	93	406.79	0	404.00	0.69	5	404.20	0.64
In Tree	518	394.34	61	378.03	4.14	228	383.90	2.65

Decomposition task graph. Furthermore, for all benchmarks, the number of faults occurred in the baseline approach is larger than that of the proposed technique. These results confirm that by tuning task allocation frequency according to core fault behavior, both the schedule length and the number of faults occurred in the system can be reduced as well.

When compared to the fault count based approach, the proposed technique also outperforms it in most of the cases. This is because while the fault count based technique takes into consideration core fault behavior, it overlooks time correlation between faults. In other words, the proposed core reliability metric is more accurate, especially for modeling transient/intermittent faults that have an unpredictable nature. For instance, transient faults are not related to the actual healthy status of the systems. Instead, they are due to external factors that affect processing units during a short period and then disappear. Under this scenario, a fault count based approach will end up providing misleading information about a core that was affected by a transient fault at some point but is healthy now. In contrast, the proposed reliability level will quickly bring a core back to the healthy state once it starts to produce clean results consecutively. Therefore, in real systems where transient, intermittent and permanent faults appear indiscriminately, the proposed approach can achieve much better results than the fault count based approach.

Finally, for some benchmarks, such as *LU decomposition*, the difference in the number of faults between the fault count approach and the proposed approach is very large but not proportionally reflected in the schedule length. This is due to the fact that not all the faulty results will affect tasks on the critical path. Instead, the impact of faults is not constant and depends on the task where it was produced. Faulty tasks located on the critical path will have a greater impact on the schedule length than those faulty but non-critical tasks.

V. CONCLUSION

This paper have presented an adaptive fault tolerant technique that dynamically tunes resource allocation based on the time correlation of faults detected in the system. The proposed technique effectively models the impact of diverse fault behavior in terms of the frequency at which faults occurred and their time correlation. Task vulnerability and criticality is also considered to improve schedule quality, by assigning task priorities based on these two parameters and sorting tasks dynamically based on their priorities. In the experiments, the proposed technique is compared against two scheduling techniques: one does not consider core reliability and one only counts faults while ignoring their time-correlation. The results show that by modeling time correlated fault behavior and prioritizing tasks based on their criticality and vulnerability, our technique can reduce runtime schedule length by up to 56%.

REFERENCES

[1] 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*, June 2004, pp. 177–186.

[2] R. Malucci, "The impact of electro-migration on various contact materials," in *IEEE 59th Holm Conference on Electrical Contacts (HOLM)*, Sept 2013, pp. 1–8.

[3] J. Roberts, S. Hussain, J. Suhling, R. Jaeger, and P. Lall, "Characterization of die stresses in microprocessor packages subjected to thermal cycling," in *13th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, May 2012, pp. 1003–1014.

[4] 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)*, March 2013, pp. 1373–1378.

[5] 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*, Sept 2007, pp. 312–321.

[6] 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)*, March 2008, pp. 49–54.

[7] 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)*, March 2012, pp. 1429–1432.

[8] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Scheduling and optimization of fault-tolerant embedded systems with transparency/performance tradeoffs," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 3, pp. 61:1–61:35, Sep. 2012.

[9] 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)*, Oct 2011, pp. 247–256.

[10] 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)*, Sept 2010, pp. 396–401.

[11] C. Yang and A.Orailoglu, "Tackling resource variations through adaptive multicore execution frameworks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 132–145, Jan 2012.

[12] 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)*, May 2013, pp. 1–7.

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