Task Recomputation Based Reliability Improvements in Heterogeneous Multi-Core Systems

Hakduran Koc† and Bayan Nimer
KocHakduran@uhcl.edu Bayan.Nimer@gmail.com
University of Houston-Clear Lake, Houston, TX, USA, †Corresponding Author

Summary
Reliability issues in embedded systems are becoming limiting factors while the demand for high performance and computational complexity continue to increase. Especially, systems operating under harsh conditions are prone to generate erroneous results. As embedded systems already house large datasets, traditional techniques to improve reliability such as task redundancy can be very costly in terms of memory space consumption and hence may cause performance degradation. In this paper, we propose a task-recomputation based approach that utilizes idle time frames of computational resources in order to iteratively recompute tasks to increase reliability of overall design without incurring any additional execution latency, area or extra memory space. We present our experimental evaluation and compare our results with a state-of-the-art technique. The experimental results collected using both task graphs extracted from benchmarks and automatically-generated task graphs show the effectiveness of the proposed approach.

Keywords:
Embedded systems, heterogeneous (hybrid) systems, multiprocessor systems, recomputation, reliability, soft errors.

1. Introduction

As the demand for high performance and computational complexity continue to increase in embedded systems, reliability issues are becoming limiting factors on application scalability and long-term survivability in which such systems are operated. Given that erroneous results generated by super-fast power-aware architectures have no meaning, improving reliability of such systems has gathered much attention in recent years. In today's technology, reliability is considered as a primary optimization metric in computing systems.

Embedded systems are widely used in controlling variety of applications ranging from portable devices to safety-critical medical, automotive and space applications. Such systems are required to have extended life-times and long-term availability with little or no maintenance. That is because such systems require very short repair time, closely available personnel and extensive data collection capabilities which can be very costly.

With the recent advances in technology such as lower voltage values, higher clock frequencies and increased functionality, integrated circuits (ICs) have gotten denser causing lower capacitance inside the cells resulting in decreased natural resistance against transient errors, namely soft errors known as single event upsets (SEU). Soft errors are random and occur at unpredicted intervals; they are exposed as voltage glitches that cause computational errors in the output of functional units or bit flips in memory cells, but not a permanent damage to the circuit [28]. As a result, providing reliable (correct) execution of tasks on respective processing elements is becoming increasingly critical in embedded systems and de-signers are required to incorporate reliability into the overall system development cycle.

Even though such faults affect systems in general, embedded systems designers are faced with unique challenges in order to meet user demands of increased functionality and reliability under tight constraints. First, these systems impose tight performance and area constraints requiring the execution of a set of tasks to meet a deadline with limited amount of resources. Second, high performance means high power consumption which limits these devices' operational lifetime since they are mostly dependent on batteries. Therefore, techniques such as check-pointing and roll back and recovery cannot be applied because they are costly in terms of execution delay. Third, since these systems are cost sensitive and limited by small packaging size and weight, they use small attached memory or diskless designs that have no means of expanding other than upgrading which may cost more than the system itself. Additionally, the dataset sizes and software in such systems have grown to occupy a considerable memory space. As a result, standard techniques to improve reliability such as duplication [21] (i.e., per-forming the computation more than once and storing the result in memory) can be very costly in terms of memory space consumption.

In this paper, we propose to utilize task recomputation in order to improve the overall system reliability of heterogeneous multi-core embedded architectures without incurring any overhead on performance, area or memory space consumption. Given the task graph representation of an application, our approach iteratively tries to recompute the outputs of tasks (from least reliable to most) using available input values and idle computation-al resources; and then, pass the recomputed outputs to the successor

Manuscript received August 5, 2020
Manuscript revised August 20, 2020

task(s) without storing it in memory. By doing so, our goal is to create a parallel execution path and use this output if the original task fails to produce the correct result. Our approach makes use of a technology library that offers alternatives to each task with different reliability and delay metrics. After initial schedule of the tasks on available processing elements (PEs) using a reliability-centric scheduling algorithm [6], our approach iteratively searches for idle time frames of PEs and available inputs to recompute the output of a task before each of its successors start executing. If the out-puts of predecessor tasks are alive in memory, our approach assigns the recomputation of the task to an avail-able PE; otherwise, it searches if there is available idle time on the same or different PE to recompute the output of the predecessor task(s) that are not available. If the idle time available is smaller than the task's execution time, we recompute the task and adjust the schedule accordingly. If the resulting latency is greater than the given execution deadline, the task is not recomputed. This process is iterated until all tasks in the task graph are visited.

The rest of this paper is organized as follows. Section 2 briefly reviews previous research and sets the stage for our work. Section 3 provides a background on types and sources of failures in embedded systems and how they occur. Section 4 presents the way in which the problem is formulated. Section 5 illustrates our approach through an example. Then, the details of the proposed approach are given in Section 6. After presenting our experimental evaluation in Section 7, we conclude the paper and give future directions in Section 8.

2. Related Work

As reliability has become a bottleneck on the next generation of embedded systems [10, 29], researchers proposed several hardware and software based solutions at different abstraction levels [16, 22, 26, 27]. At the system level, a recent work by Bolchini and Miele [15] provide a reliable design flow through improving the standard system-level synthesis process. They suggest an adaptive technique that adds another layer before operation scheduling and resource mapping to introduce reliability-awareness. Many notable research efforts based on HW/SW co-design [24, 25] are reported in the literature. Tosun et al. [7] introduce a hardware/software co-design solution, which considers reliability as a main optimization metric under performance and area constraints during the scheduling phase by making use of a component library with multiple versions of the same task with different reliability, area and latency measures. Huang et al. [14] propose a combined hardware/software replication technique using multi-objective evolutionary algorithms (MOEA) that result in an optimized solution to map tasks to various PEs. Their technique gives an exact task and message schedule and applicable amount of hard-ware/software redundancy. Li et al. [11] introduce reliability into co-design through allocation and scheduling algorithms that selectively duplicate tasks during idle times of functional resources, taking advantage of mutually exclusive tasks that have overlapping execution times. Bolchini et al [17] illustrate methodologies for integrating fault detection properties by introducing critical sections (i.e., tasks that require reliability) into system specifications and performing partitioning to provide an optimized solution under area, latency and power constraints. Their approach targets at designing redundant elements, such as checker and fault tolerant communication links. Glaß et al. [9] introduce a technique that synthesizes reliable datapaths that are optimal with respect to multiple objectives. They start from behavioral description and select resources with different area, latency and reliability values simultaneously, and then, optimize these parameters and implement resource sharing in or-der to allow redundancy for improved design reliability.

In high level synthesis, Tosun et al. [6] propose an algorithm that utilizes most reliable version of each component in the design bounded by area and latency constraints. Furthermore, [8] put forward an integer linear programming (ILP) formulation that uses a characterization library with different versions of hardware components to schedule operations in order to increase reliability. Moreover, several researchers [20, 23, 28] propose allocation and scheduling algorithms to increase reliability in heterogeneous embedded systems. He et al. [12] present two heuristic scheduling algorithms that iteratively schedule nodes to processing elements in such a way that a minimum reliability cost is obtained under timing constraints. Qin et al. [13] describe a reliability cost driven scheduling scheme for real-time distributed heterogeneous systems that result in improved system reliability with reduced schedule at no extra area cost.

Recomputation technique was studied first in the con-text of register allocation by Briggs et al [31]. Their meth-od rematerializes (recomputes) a register value when it is cheaper than storing and retrieving it from memory. Kandemir et al. [33] utilize recomputation to reduce memory space consumption of data intensive applications in memory-constrained embedded processing. Koc et al [18] use data recomputation to reduce the memory space consumption of data-intensive applications. Then, they aim at reducing the number of off-chip memory accesses in order to improve the performance of Chip Multi-Processors in [32]. Most prior research on reliability focuses mainly on optimizing reliability by keeping area-performance-power parameters at minimum and adding a lot of complexity while the recomputation concentrates on

improving the performance or memory space consumption of computing systems. To our knowledge, there is no research that utilizes recomputation to improve reliability of heterogeneous multi-core embedded systems. Our approach improves reliability without incurring any performance, area or latency overhead.

3. Sources and Types of Failures in Embedded Systems

The accelerating pace in technology scaling (65nm and beyond) fueled by reduction in transistor size and voltage levels allowed for tremendous advances in embedded systems functionality but with concomitant increase in failures. That is, the reduced transistor size and the increased clock frequencies have minimized the effects of electrical and latching window masking resulting in an increased threat of errors in a processor's functionality [29]. Such errors, in embedded systems, can be induced through different radiation sources. More specifically, when cosmic rays collide with environmental particles, they produce both high energetic proton and neutron precipitations; however, neutrons are of greater concern because they are capable of penetrating most of human made circuit designs. Another source of radiation occurs when low energy cosmic neutrons, which are uncharged particles, interact with circuit material and become electrically charged. Moreover, packaging material such as solder paste and molding compounds result in alpha particle radiation; which can also be emitted by radioactive elements that are found in abundance in our environment [3]. Another significant source of radiation is thermal neutrons; however, they are effectively accounted for by removing borophosphosilicate glass in recent processors [4]. The collision of these energetic particles with a sensitive area in a node result in different types of errors, namely single event effects (SEE), that can be classified as either soft or hard errors.

Hard errors are ones that cause permanent damage to the circuit and occur in the form of single event latchups (SEL), produced when a particle strike cause an increased current in the circuit. Also, they can be manifested as single event burnouts (SEB), which are realized when a particle strike effects a power transistor resulting in a rupture [30]. However, hard failures are not of concern to us in this work because they should be accounted for by testing the hardware in the field, using simulation tools to analyze designs, and applying design improvements and mitigation schemes such as the ones in [27].

Soft errors, on the other hand, are single event upsets (SEU), which are wrong signals in the circuit (faults) that are random, occurring at unpredicted intervals, temporary, and do not cause a permanent damage to the circuit. They take the form of a single bit upset (SBU) caused by a bit flip in an element of the circuit (mostly memory); multiple bit upset (MBU), which causes more than one bit flip but is less likely to take place; and single event transients (SET) that cause voltage glitches (disruptions) [30]. The focus of this work is to increase resilience of embedded systems against soft errors, which have higher occurrence probabilities. According to the research done by IBM [35], a soft error rate of 4000 FIT (unit for expressing failure rate in electronic devices and one FIT equals one failure per billion hours) can occur in a processor's silicon, of which half will affect the processor's logic and half will affect the memory/cache. This number has been exponentially increasing because of decreased capacitance of today's denser circuits; as a result, energetic particles have greater linear energy transfer (LET) effect (ionizations per unit distance) in circuits [5]. Such ionizations cause a charge collection, $Q_{collected}$ on or near a sensitive region of the node (such as p-n junction) creating a voltage glitch and therefore altering the value produced by a processing unit or stored in memory. Soft errors occur when the collected charge, $Q_{collected}$, exceeds the critical charge, $Q_{critical}$, required to retain data, which has been decreasing resulting in a decreased natural resistance against soft errors.

4. Problem Formulation

4.1 System Modelling

The functionality of an embedded system application is depicted as a task graph. A task graph is a directed acyclic graph, $G_s(V, E)$, (e.g., one in Fig. 4). Each vertex node v_i in $V = \{v_i; i = 0, 1, 2, 3, \ldots, n\}$ represents an independent task performing various operations. The intercommunication between tasks is denoted by edges. The edges represent dependencies between tasks, $E = \{(v_i, v_j); i, j = 0, 1, \ldots, n\}$. A task can start executing only after all its predecessors complete their executions. The result of a predecessor task is passed to its successors upon completion. Tasks are executed on various processing elements. Our target architecture is a multi-core embedded system composed of heterogeneous CPUs that have different reliability, latency and area values communicating through a software-managed memory. These tasks are allocated and mapped to different PEs using a reliability-centric approach that makes use of a technology library to give the best reliability-optimized schedule under latency constraints. This algorithm is given in [7] and represents, in a sense, the state-of-the-art.

4.2 Reliability Calculations

We calculate the reliability of the entire system considering reliability of each component. In its simplest form, reliability of a given task is defined as the probability that the task generates its intended output for a given time period knowing that it was functioning correctly at start time t_0. It is a value from 0 to 1, where 1 represents 100% reliable output. Reliability models are formulated to match the system operation to a logical structure (that is used to calculate the reliability of the system) expressed as combinatorial reliabilities of its individual components [2]. The tasks are performed in a series or parallel pattern depending on when their inputs are available during the course of execution. However, in order to guarantee the successful execution of an application (represented by a task graph); the correct operation of each and every task is required. As a result, the original system has a series configuration *in terms of reliability* (i.e., failure of any single task interrupts the path and causes the whole system to generate an erroneous output) such as the one shown in Fig.1. The reliability of a single task is represented by the probability of success of the task, P_i, and is independent of the rest of the system. The reliability of the whole system, R, is the intersection of the probability of success of each task, which is expressed in (1), and then, can be re-written as shown in (2). Given that each task's failure rate is constant and is independent of their usage time, Equation (2) simplifies to Equation (3), where R_s represents the system reliability, R_i is the reliability of component *i* and *n* is the number of tasks in the task graph.

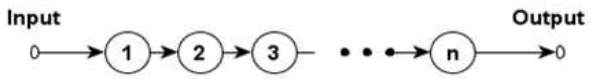

Fig. 1. Series reliability configuration graph.

$$R = P_s = P(x_1 x_2 x_3 \dots x_n) \quad (1)$$

$$P_s = P(x_1)P(x_2)P(x_3)\dots P(x_n) \quad (2)$$

$$R_s = \prod_{i=1}^{n} R_i \quad (3)$$

The series configuration is the lowest bound in terms of reliability. In order to improve reliability, we need to add redundancy (i.e., add redundant tasks that would take over and perform computation in case of failure of the primary task). In our approach, redundancy is added by recomputing the outputs of tasks using idle times of processing elements. The reliability of a purely parallel system, such as the one shown in Fig.2, can be expressed as in (4), which reduces to (5).

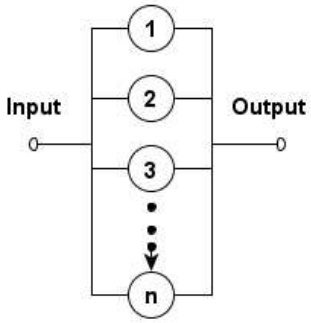

Fig. 2. Parallel reliability configuration graph.

$$R = P_s = P(x_1 + x_2 + x_3 + \dots + x_n) \quad (4)$$

$$R_s = 1 - \prod_{i=1}^{n}(1 - R_i) \quad (5)$$

As a result, a recomputed task adds an alternative path that could be taken in case the original computation fails. In this paper, the reliability of the system after applying the proposed recomputation technique is calculated using a combination of series and parallel configuration. A recomputed task with its original task is considered as a parallel subsystem connected in series to the rest of the system.

4.3 Memory Space Consumption Cost

In order to determine the minimum memory space consumption required to store results of a scheduled task graph, the well-known left-edge algorithm [1] is employed. This algorithm was originally developed to find the minimum number of tracks used to connect points in channel routing [36]. With this algorithm, we start out by finding lifetime intervals of each result generated by each task. Lifetime intervals are defined by a starting position (left edge) on the x-axis representing the clock cycle at which the result is generated as an output of a task (its birth) and an ending position on the x-axis (right edge) depicting the latest time at which the result is referenced as an input to another successor task (its death). The intervals are assigned to tracks such that no two intervals in a track overlap with each other. Intervals are sorted in increasing order of left edge; that is, the interval with the least value (in clock cycles) of left edge is assigned to the first memory space (track), then the rest of the intervals are scanned to find the next interval whose birth time is greater than or equal to the death time of the previous interval (one assigned to first memory space) and, if found, gets assigned to the same memory space. This process is repeated until no more intervals can share the same track or memory space.

Afterwards, one color is considered at a time and assigned to each packed interval (sorted intervals on the same track). Then, the colored graph is created where each of the intervals represent one of the vertices with its assigned color and an edge between pairs corresponding to intervals that overlap. The minimum amount of memory space used is obtained by minimum coloring of the graph where each color represents the cost of memory space and all vertices with the same color are assigned to the memory space that represents the color. In the case when memory space consumption is assumed to be the same for all tasks, the total cost of memory space consumption is found by multiplying the total number of colors used to color the graph by the memory space cost. Nevertheless, in the case where different tasks implemented on different PEs have different memory cost values, the total memory space consumption is determined by the maximum resultant value of adding each set of overlapping intervals (given that each interval has a different memory space cost).

4.4. Task Recomputation

CDFG Task recomputation is used to reduce the amount of data that must be saved during an execution of an application by recovering this data from a small subset of available saved data when needed by another task instead of storing it in memory. As mentioned earlier, a task graph representation is used to demonstrate a given embedded system application. The execution latency of such a task graph is determined by the critical path, which is the path from source to sink node for which the sum of latency and communication costs is the maximum [34]. Depending on the properties of the task graph, such as amount of dependencies between tasks, a poor utilization of processing elements (PEs) may exist; i.e., not all computing resources are fully used all the time since they will be waiting on an existing task to be completed. As a result, it might be possible to perform re-computations using those idle time frames without incurring performance overhead. Hence, if a result of a task will not be needed for a certain amount of time, then it does not need to be stored in memory and, instead, can be recomputed before its successor task(s) if there exists enough idle time for the needed recomputations to take place. Thus, performing careful recomputations can result in a reduction of memory space consumption at no extra latency cost. In many cases, further memory reduction can be achieved by introducing a small amount of latency overhead, which can lead to a significant improvement in systems that are limited by their memories. In some cases, it is possible to recompute a certain task once and store it in memory for a certain number of clock cycles in order to be used by other successor tasks or recomputations and result in an improvement in memory space consumption.

4.5. Target Architecture

The target architecture used in the experiments is illustrated in Fig.3, which depicts a commonly used heterogeneous multi-core architecture in embedded systems. It is composed of a number of heterogeneous CPUs and a software-managed memory. CPUs (PEs) have different area, latency and reliability values and communicate through a shared bus. Each task in the task graph can have different implementation option on each available CPU but with different area, performance and reliability values (technology library). We assume that each task requires the same amount of memory in each PE. Given the task graph representation of an embedded application, performance deadline and technology library (such as the one given in Table 1), our method produces a schedule with allowed recomputations for a certain number and type of CPUs.

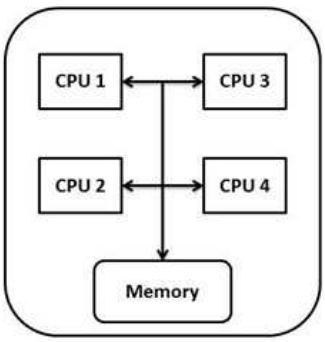

Fig. 3. Target heterogeneous multi-core embedded architecture.

5. Illustrative Example

In order to show how task recomputation is utilized to increase reliability, let us consider the task graph given in Fig. 4 with 7 tasks and a latency deadline of 120 clock cycles. In this illustrative example, let us assume a heterogeneous embedded architecture with two processing elements, namely CPU1 and CPU2 (similar to one in Fig. 3 with 2 CPUs). The technology library, which specifies performance (delay) and reliability values of each task on the different PEs, is given in Table 1. Execution delays are given in clock cycles.

The initial task allocation and scheduling are based on the reliability-centric co-design framework in [7]. First, each task is allocated and scheduled on the most reliable PE. Then, after finding earliest start times (*EST*) and latest start times (*LST*), freedom (*F*) of each task is calculated ($F_i = LST_i - EST_i$, where F_i is the freedom of vertex v_i). The freedom is used to determine the priority (order) of

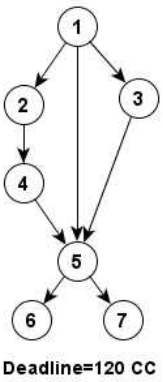

Fig. 4. Example task graph with seven tasks.

Table 1. Reliability and Execution Delay Values for Tasks in Fig. 1 on Different Processing Elements

		Tasks						
		1	2	3	4	5	6	7
CPU 1	Rel.	0.979	0.975	0.999	0.983	0.989	0.980	0.980
	Delay	20	45	50	19	18	15	12
CPU 2	Rel.	0.973	0.966	0.989	0.978	0.985	0.985	0.985
	Delay	13	15	15	17	15	17	14

competing tasks to schedule to earlier control steps. If the resulting schedule exceeds the given latency deadline, performance optimization techniques are applied. To decrease overall latency, we identify the tasks that can be scheduled concurrently (but are running on the same CPU one after another) and assign them to different CPUs without violating dependency conditions. If the deadline is still not achieved, we iteratively assign slowest tasks in the critical path to faster CPUs. This step is repeated until the performance deadline is met. The initial schedule of the example task graph after performance optimizations is given in Fig. 5.

As seen in Fig. 5, there is a considerable amount of idle time on CPU2 that can be utilized for recomputation in order to increase reliability. First, we incrementally check for each task starting with the least reliable one whether the available idle time allows us recomputing that specific task. There must be enough idle time on the CPU to recompute a single task before its successors. In addition, the results of predecessors of the task to be recomputed should be alive as not to increase memory space consumption. We assume that communication between tasks running on different CPUs takes 1 clock cycle, while communication between tasks on the same CPU is done in the same clock cycle. In this example, the least reliable task is task 2. There exists enough idle time to recompute this task before task 4, which is its only successor. The next least reliable task is task 1, which needs to have enough idle time on CPU2 to be recomputed before its successor tasks, namely tasks 2, 3 and 5. Also, in order for our reliability

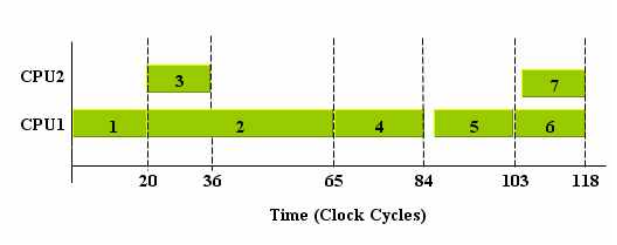

Fig. 5. The initial schedule for the task graph in Fig. 1 after performance optimization.

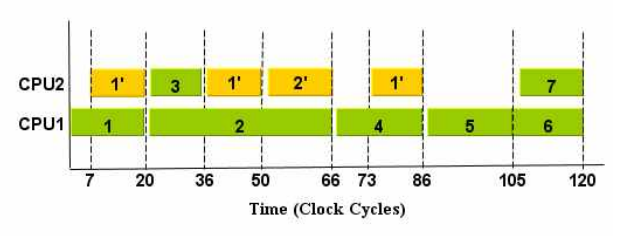

Fig. 6. The schedule for the task graph in Fig. 1 after possible recomputations.

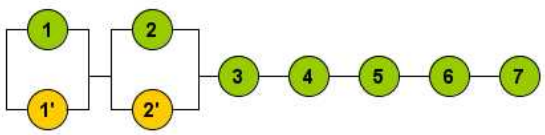

Fig. 7. Reliability configuration graph for recomputed schedule.

calculation to hold true, task 1 needs to be recomputed before task 2' (recomputed task 2). None of the other tasks can be recomputed because of deadline and dependency constraints. Fig. 6 illustrates the schedule of the task graph after allowed recomputations. Note that tasks 1 and 2 are recomputed and the corresponding parallel execution paths to improve reliability are shown in Fig. 7. Using the combination of series and parallel reliability models, our method results in a reliability value of 0.926814 (compared to the original schedule with reliability of 0.885922). Please note that this improvement comes with no performance or memory overhead. We use left-edge algorithm described earlier in order to determine the memory space consumption. Assuming each task needs 10 units of memory space to store its result, our design requires 30 units of memory space (same as the original one), which corresponds to the chromatic number of the corresponding conflict graph.

6. Details of the Approach

In multi-core architectures, some processing elements sit idle during the course of execution due to dependences or lack of parallelism. In this paper, we propose to utilize these idle time frames to iteratively recompute the outputs

of least reliable tasks in order to improve the reliability of the system by strengthening the weakest link in the chain. Three conditions need to be considered in order to apply task recomputation: 1) There must be available processing elements sitting idle to perform recomputation, 2) dependency conditions should not be violated, and 3) given constraints (e.g., latency, performance, memory space consumption, etc.) should be met.

A sketch of the high-level algorithm describing the proposed approach is given in Fig. 8. In order to use a parallel reliability model, the recomputed versions of the same task need to be implemented on the same processing element before all of its successors. Consequently, we first identify the least reliable task in the schedule, and then check if there are available idle time frames to recompute before all its successors. If so, we check if the outputs of predecessors of the task to be recomputed are currently alive until the latest task to be recomputed. Then, we schedule a recomputation and calculate the respective latency to ensure that the performance deadline is met. If the outputs of any of the predecessors are not available, we try to find idle time slots to recompute them, and then, perform recomputation and calculate latency of the new schedule. If there are not enough idle times to recompute a task before all its successors, we check the next least reliable task and so on. We do this until all tasks in the task graph are visited. The recomputations are performed regardless if an error is present in the original computation or not. A simple switch or checker is used to determine the correct result to be used.

If there exist two or more tasks that have the same reliability value and only one can be recomputed, the one with the highest criticality value gets selected. That is because a soft error in a critical task can have a higher probability of propagating to later tasks and having a more serious impact on the system, which leads not only to corrupt the output data but also to cause a loss of functionality and critical failures of processing elements. In this work, the criticality of a task depends on the number of tasks whose correct functionality is subject to the task in question. It is also possible that criticality of a task is given in the specification. Sometimes, it is possible to recompute a task before one of its successors and store it in memory for a few clock cycles to serve as alternative path for another successor if it does not cause an increase in memory space consumption.

In some cases, we are faced with two possibilities: recomputing a task more than once or recomputing another task with a higher reliability value. After performing extensive experiments, we conclude that recomputing a different task achieves higher reliability improvement. More specifically, after recomputing a task once, its reliability increases (due to different parallel paths) and becomes higher than the rest of the tasks that have not been recomputed yet. So, treating the task and its recomputed version as one component and comparing it with the next candidate task to be recomputed, we find that the candidate task has lower reliability value. In order to have better reliability gains, our approach recomputes as many different tasks as possible to create alternative paths. Please note that our algorithm can easily be modified to work with an allowed performance and memory space overhead.

```
1. INPUT: Gs(V, E), technology library, memory (Mmax), area (Amax) and
   performance (Lmax) bounds.
2. OUTPUT: modified schedule with possible recomputations
3. Schedule Gs(V, E) using a reliability-centric approach
4. for all vi := 1 to n
5.    calculate criticality(vi)
6. for each vi with reliability from low to high
7.    Search for idle time slots before successors of vi
8.    if (idle time slot & predecessors' outputs are available)
9.       Recompute task before its successors and update schedule
10.   else if (idle time slots are available to recompute task and
   predecessors' inputs)
11.   Recompute task and its predecessors and update schedule
12.   else if (idle time slots and predecessor' inputs are not available)
13.     Add recomputation to schedule and calculate new latency, lnew
14.     if (lnew > lmax)
15.       Delete the recomputation
16.       Report that task cannot be recomputed
17.     end if;
18.   end if;
19.end for;
```

Fig. 8. A high-level algorithm to schedule task recomputations.

6. Experimental Evaluation

In order to show the effectiveness of our approach, we conducted experiments using two sets of task graphs: task graphs extracted from benchmarks and automatically-generated task graphs using TGFF tool [19]. Our target architecture is shown in Fig. 3 which is a four-core heterogeneous embedded system in which processors communicate through a shared software-managed memory. Data transfers between tasks running on different CPUs takes 1 clock cycle, while communication between tasks on the same CPU is performed in the same clock cycle.

Automatically-generated task graphs contain 9 to 49 tasks. Each task needs 10 units of memory space and has a different execution time and reliability value for each processor (technology library). The important characteristics of these graphs are given in the first four columns of Table 2. Column 1 gives the task graph label; column 2 indicates the number of tasks; column 3 gives the latency constraint that should be met; and, column 4 reports the memory space consumption. Reliability values calculated using the approach in [7] are reported in column 5. Please note that this approach represents, in a sense,

Table 2. Reliability Values for Automatically-Generated Task Graphs Using the Approach in [6] and the Proposed Approach.

Task Graph Label	Number of Tasks	Latency Bound	Memory Consump.	Reliability				
				Ref. [6]	Task Recomputation		Improvement (%)	
					Same Mem.	25% Increase	Same Mem.	25% Increase
TG 1	9	600	40	0.878744	0.928221	0.938390	5.630	6.788
TG 2	10	115	30	0.832658	0.923863	0.923863	10.95	10.95
TG 3	12	120	40	0.792224	0.840325	0.88768	6.072	12.05
TG 4	14	140	40	0.870059	0.934611	0.944731	7.419	8.582
TG 5	16	153	40	0.828850	0.894701	0.914316	7.945	10.31
TG 6	20	180	60	0.822328	0.916983	0.945175	11.51	14.94
TG 7	22	700	70	0.762702	0.801795	0.801795	6.720	6.720
TG 8	49	420	130	0.579133	0.652257	0.741849	12.63	28.10

Table 3. Reliability values for task graphs extracted from benchmarks using the approach in [6] and the proposed approach.

Benchmark	Latency Bound	Memory Consump.	Reliability				
			Ref. [6]	Task Recomputation		Improvement (%)	
				Same Mem.	25% Increase	Same Mem.	25% Increase
Diff. Equation Solver	55	30	0.87227	0.954305	0.954305	9.405	9.405
EW Filter	88	70	0.655482	0.67777	0.765061	3.400	16.72
AR Filter	75	50	0.545696	0.607155	0.636386	11.26	16.62
FIR Filter	165	40	0.706534	0.871886	0.914509	23.40	29.44

state-of-the-art. Then, next column reports the reliability values after applying our task recomputation approach. We also report, in the next column, the reliability values of our approach if 25% increase in memory consumption is allowed (e.g., a task is recomputed before some of its successors, and then, is stored in memory to be used for the rest of successors). Last two columns show the reliability improvements our approach achieves over Reference [6]. As can be seen in the table, our approach brings between 5.6% - 12.6% reliability improvements (around 8.6% on the average) without any overhead and between 6.7% - 28.1% reliability improvements (around 12.3% on the average) with an allowed increase in memory consumption. Our approach achieves higher reliability gain for TG 2 (compared to TG 1 and TG 3) even though TG 2 utilizes less memory space and all have similar number of tasks. It is because the tasks in TG 1 have less dependences; hence, allowing our algorithm to generate more recomputation opportunities. It is similar for TG 6 and TG 7.

The second set of experimental evaluation using task graphs extracted from benchmarks is given in Table 3, which follows a similar format as that of Table 2. Column 1 gives the name of the benchmark and next two columns show latency constraint and memory space consumption, respectively, for each benchmark. Column 4 reports the reliability values calculated using the approach in [7]. Then, columns 5 and 6 give the reliability values after applying our approach. Last two columns reports the reliability improvements over Reference [7]. Our approach brings between 3.4% - 23.40% improvements (around 11.9% on the average) without any overhead and between 9.4% - 29.4% reliability improvements (around 18.1% on the average) with an allowed increase in memory consumption. When we compare the reliability values of EW Filter (last two columns), we see significant improvement because increasing memory consumption for this benchmark creates more recomputation opportunities for the successor tasks.

Notice that we did not include a separate column for execution latency in both tables as none of the methods exceeds the latency bound. Also, it is worth mentioning that recomputation would bring more significant improvement if applied to a homogeneous embedded system. As mentioned earlier, in order for the reliability calculation to work correctly, a task has to be recomputed in a similar manner before each successor which places a constraint on our algorithm; thus not all idle time frames

are utilized properly. However, in a homogeneous architecture, all PEs will have the same metrics giving the chance for tasks to be recomputed on any available PE before each successor.

As seen in both sets of experimental evaluations, our approach gives, in most cases, higher reliability gains if it is allowed to utilize extra memory space. It is because this extra space is used to store recomputed values in memory and keep them available for recomputations of successor tasks. However, Diff. Equation, TG2 and TG7 have the same reliability values with/without memory increase (last two columns in both tables). This indicates that our algorithm utilized the recomputations at the highest degree for the given latency bound.

8. Conclusion and Future Work

Reliability concerns have increased with the technology scaling, latest power management techniques and radioactive impurities present in new device materials. All existing work done on reliability ignores memory space consumption constraint imposed by data intensive embedded systems. In order to remedy this deficiency, we present a task recomputation based approach that utilizes idle times of the processing elements in a heterogeneous multi-core embedded architecture to increase overall reliability of the system. That is, if a task fails to produce the intended result, the output of the recomputed task is passed to successor tasks that need it without storing in memory as long as area and latency requirements of an application are satisfied. Our experimental evaluation using automatically-generated task graphs and benchmarks shows the effectiveness of the proposed approach in improving reliability without incurring any performance, memory space or area overheads.

Our future work is to apply the proposed approach in hardware/software co-design framework where a full degree of parallelism can be exploited by allowing recomputation of tasks to be performed on either hardware (ASICs) or software (general purpose CPUs) components regardless of the processing element on which they were originally scheduled to increase reliability of overall design while meeting system level objectives.

In conclusion, even though guaranteeing proper functionality while meeting constraints in complex embedded systems presents a big challenge when dealing with safety critical applications, our recomputation method presents opportunity that leads to optimized reliability while meeting the design constraints.

References

[1] G. D. Micheli, Synthesis and Optimization of Digital Circuits, 3rd ed., New York: McGraw-Hill Higher Education, 1994, pp.51-67.

[2] M. L. Shooman, Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design. New York: John Wiley & Sons Inc., 2002.

[3] R. C. Baumann, "Soft errors in advanced computer systems," *IEEE Trans. Design & Test of Computers*, vol. 22, no. 3, June 2005, pp. 258-266.

[4] R.C. Baumann, T. Hossain, S. Murata and H. Kitagwa, "Boron compounds as a dominant source of alpha particles in semiconductor devices," *Proc. 33rd Annual Int'l Symposium on Reliability Physics*, IEEE Press, 1995, pp. 297-302.

[5] A. Javanainen, T. Malkiewicz, J. Perkowski, W. H. Trzaska, A. Virtanen, G. Berger, W. Hajdas, R. Harboe-Sorenson, H. Kettunen, V. Lyapin, M. Mutterer, A. Pirojenko, I. Riihimaki, T. Sajavaara, G. Tyurin, and H. J. Whitlow, "Linear energy transfer of heavy ions in silicon," *IEEE Trans. Nuclear Science*, vol. 54, no. 4, Aug. 2007, pp. 1158-1162.

[6] S. Tosun, N. Mansouri, E. Arvas, M. T. Kandemir, and Y. Xie, "Reliability-centric high-level synthesis," *Proc. Conf. Design, Automation and Test in Europe*, IEEE Press, March 2005, pp. 1258-1263.

[7] S. Tosun, N. Mansouri, E. Arvas, Y. Xie, and W. L. Hung, "Reliability-centric hardware/software co-design," *Proc. 6th Int'l Quality of Electronic Design Symp*. (ISQED 05), March 2005, pp. 375-380.

[8] S. Tosun, O. Ozturk, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, and W. L. Hung, "An ILP formulation for reliability-oriented high-level synthesis," *Proc. 6th Int'l Quality of Electronic Design Symp.* (ISQD 05), IEEE Press, March 2005, pp. 364-369.

[9] M. Glaß, M. Lukasiewycz, T. Streichert, C. Haubelt and J. Teich, "Reliability-aware system synthesis," *Proc. Design, Automation and Test in Europe (DATE 07),* IEEE CS Press, April 2007, pp. 1-6.

[10] M. Glaß, M. Lukasiewycz, F. Reimnn, C. Haubelt, J. Teich, "Symbolic reliability analysis and optimization of ECU networks." *Proc. Design, Automation and Test in Europe (DATE 08)*, IEEE CS Press, March 2008, pp. 158-163.

[11] L. Li, Y. Xie, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, "Reliability-aware co-synthesis for embedded systems." *Proc. 15th Int'l Conf. Application-Specific Systems, Architectures and Processors*, IEEE Press, Sept. 2004. pp. 41-50.

[12] Y. He, Z. Shao, B. Xiao, Q. Zhuge, and E. Sha, "Reliability driven task scheduling for heterogeneous systems," *Proc. 15th Int'l Conf. Parallel and Distributed Computing and Systems*, vol.1, Nov. 2003, pp. 465-470.

[13] X. Qin, H. Jiang, C. S. Xie, and Z. F. Han, "Reliability-driven scheduling for real-time tasks with precedence constraints in heterogeneous systems," *Proc. 12th Int'l Conf. Parallel and Distributed Computing and Systems*, 2000.

[14] J. Huang , J.O. Blech, A. Raabe and A. Knoll, "Reliability-aware design optimization for multiprocessor embedded

systems," *Proc. 14th Euromicro Conf. Digital System Design (DSD)*, IEEE Press. Aug. 2011, pp. 239-246.

[15] C. Bolchini and A. Miele, "Reliability-driven system-level synthesis for mixed-critical embedded systems," *IEEE Trans. On Computers*, no. 99, 2012.

[16] C. Bolchini, A. Miele and C. Pilato, "Combined architecture and hardening techniques exploration for reliable embedded system design." *Proc. 21st Great Lakes VLSI Symp.*, ACM Press. May 2011, pp. 301-306.

[17] C. Bolchini, L. Pomante, F. Salice, and D. Scuito, "Reliability properties assessment at system level: a co-design framework," *Proc. 7th Int'l On-Line Testing Workshop*, IEEE Computer Soc., Oct. 2001, pp. 165-171.

[18] H. Koc, S. Tosun, O. Ozturk, M.T. Kandemir, "Reducing memory requirements through task re-computation in embedded systems," *Proc. Annual Emerging VLSI Technologies and Architectures Symp. (ISVLSI),* March 2006.

[19] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," Proc. 6th Int'l Hardware/Software Co-design Workshop, *IEEE Computer Soc.*, March 1998, pp. 97-101.

[20] L. Huang, F. Yuan, and Q. Xu, "Lifetime reliability-aware task allocation and scheduling for MPSoC," *Proc. Conf. Design, Automation & Test in Europe*, European Design and Automation Association, April 2009, pp. 51-56.

[21] G. Chen, F. Li, M. Kandemir and I. Demirkiran, "Increasing FPGA resilience against soft errors using task duplication," *Proc. Conf. Asia South Pacific Automation*, ACM Press, Jan. 2005, pp. 924-927.

[22] K. Mohanaram and N.A. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits." *Int'l Test Conf.*, Sept. 2003, pp. 893-901.

[23] G. Chen, M.T. Kandemir, S. Tosun and U. Sezer, "Reliability-conscious process scheduling under performance constraints in FPGA-based embedded systems," *Proc. 19th Int'l Parallel and Distributed Processing Symp.*, IEEE Press, Sept. 2005, pp. 162a-162a.

[24] B. Xia, F. Qiao, H. Yang and H. Wang, "A fault-tolerant structure for reliable multi-core systems based on hardware-software co-design." *The 11th Int'l Symp. Quality Electronic Design (ISQED)*, IEEE Press. March 2010, pp. 191-197.

[25] F. Vargas, E. Bezerra, L. Wulff and D. Barros Jr., "Optimizing HW/SW co-design towards reliability for critical-application systems." *Proc. 7th Asian Test Symp. (ATS 98),* IEEE Press, Dec. 1998, pp. 52-57.

[26] N. Wattanapongsakorn, S.P. Levitan, "Reliability optimization models for embedded systems with multiple applications", *IEEE Trans. Reliability*, vol. 53, no. 3, 2004, pp. 406-416.

[27] M. Nicolaidis, "Design for soft error mitigation", *IEEE Trans. Device and Material Reliability*, vol.5, no. 3, 2005, pp. 405-418.

[28] A. Dogan and F. Ozguner, "Reliable matching and scheduling of precedence-constrained tasks in heterogeneous distributed computing", *Proc. 2000 Conf. Parallel Processing International*, IEEE Press, 2000, pp. 307-314.

[29] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," *Proc. Int'l Conf. Dependable Systems and Networks (DSN 02)*, 2002, pp. 389-398.

[30] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger and L. Alvisi. "Modeling the impact of device and pipeline scaling on the soft error rate of processor elements," Computer Science Department, University of Texas at Austin, 2002.

[31] P. Briggs, K. D. Cooper, and L. Torczon, "Rematerialization," *ACM SIGPLAN Notices*, vol. 27, no. 7, July 1992, pp. 311-321.

[32] H. Koc, M. Kandemir, E. Ercanli, and O. Ozturk, "Reducing off-chip memory access costs using data recomputation in embedded chip multi-processors," *Proc. 44th Conf. Annual Design Automation*, ACM Press. June 2007, pp. 224-229.

[33] M. T. Kandemir, F. Li, G. Chen, G. Chen, and O. Ozturk. "Studying storage-recomputation tradeoffs in memory-constrained embedded processing," *Proc. Conf. Design, Automation and Test in Europe*, IEEE Press, March 2005, pp. 1026-1031.

[34] T. N'takpe and F. Suter, "Critical path and area based scheduling of parallel task graphs on heterogeneous platforms." *Proc. 12th Int'l Conf. on Parallel and Distributed Systems*, 2006.

[35] J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, T. J. O'Gorman and J. M. Ross. "Accelerated testing for cosmic soft-error rate," *IBM Journal of Research and Development*, vol. 40, no. 1, 1996, pp. 51-72.

[36] A. Ghosh, S. Devadas and A. Newton, "Sequential Logic Testing and Verification". Boston: Kluwer Academic Publishers, 1992.

Hakduran Koc received his B.S. degree in Electronics Engineering from Ankara University in 1997. After working in the industry for two years, he joined Syracuse University, NY where he received his M.S. and Ph.D. degrees in Computer Engineering in 2001 and 2008, respectively. During his graduate study, he was at the Pennsylvania State University as visiting scholar. He is currently chair and an associate professor of Computer Engineering at University of Houston-Clear Lake. His research is in the areas of digital design, embedded systems, and computer architecture.

Bayan Nimer received her B.S. degree Telecommunications Engineering at University of Texas at Dallas in 2009; and her M.S. degree from Computer Engineering at University of Houston-Clear Lake in 2013. Her research interests include digital design, embedded systems and cyber physical systems.