Improving Reliability of Soft Real-Time Embedded Systems on Integrated CPU and GPU Platforms

Yue Ma, *Student Member, IEEE*, Junlong Zhou, *Member, IEEE*, Thidapat Chantem, *Senior Member, IEEE*, Robert P. Dick, *Member, IEEE*, Shige Wang, *Senior Member, IEEE*, and Xiaobo Sharon Hu, *Fellow, IEEE*

***Abstract*—Multiprocessor systems on a chip consisting of integrated CPUs and GPUs are suitable platforms for real-time embedded applications requiring massively parallel processing. For such applications, lifetime reliability due to permanent faults and soft-error reliability due to transient faults are major concerns. Detailed execution profiling has revealed that a CUDA task's CPU execution time significantly increases if the task executes on a different core than the operating system (OS). Based on this observation, an extended task model is introduced to consider the execution time dependencies among tasks and the OS. A hybrid framework is proposed to improve soft-error reliability while satisfying a lifetime reliability constraint for soft real-time systems executing on integrated CPU and GPU platforms. This framework: 1) reduces the total utilization of cores and improves soft-error reliability via off-line task mapping; 2) achieves a higher lifetime reliability through task migration at run time; and 3) improves soft-error reliability by dynamically scaling frequencies of CPU and GPU cores. The experimental results show that the proposed framework leads to a system that can execute without soft errors for at least 4 days (4 times) and 6 days (6 times) longer, on average, than existing approaches.**

***Index Terms*—CUDA, GPU, lifetime reliability, real-time embedded system, resource management, soft-error reliability.**

I. Introduction

TO HELP meet the performance and power consumption demands of many applications, various heterogeneous multiprocessor systems on a chip (MPSoCs) have been introduced [1]. One type of MPSoCs is composed of integrated CPU and GPU. Thanks to the massively parallel computing capability offered by GPUs and the general-purpose computing capability of CPUs, this type of MPSoC has been widely used in many applications [2], [3]. For such applications, soft-error reliability due to transient faults and lifetime reliability due to permanent faults are typically important design concerns. For example, Nvidia provides a full stack software to support an autonomous driving [3]. The software relies on the GPU's parallel computing capability to implement key features such as detecting obstacles and drivable paths. One big challenge for autonomous vehicles is to achieve high reliability under the harsh automotive conditions [4]. However, the methods to improve soft-error reliability may reduce the lifetime reliability. Since transient faults occur much more frequently than permanent faults [5], in this article, we aim to maximize soft-error reliability under a lifetime reliability constraint for soft real-time applications.

Although most existing papers focus on CPU reliability [6]–[18], there are also several techniques to improve GPU soft-error reliability [19]–[22] and/or lifetime reliability [23]–[25]. However, these techniques are designed to recover from soft errors instead of reducing soft-error rate and are not supported by all GPUs. In addition, they only consider the reliability of the GPU, not that of the CPU. For MPSoCs with integrated CPU and GPU, errors from either can cause failure. Hence, jointly considering the reliability of CPU and GPU is necessary.

This article systematically addresses reliability concerns for tasks running on both CPU and GPU. For a given task, our goal is to maximize soft-error reliability under lifetime reliability and real-time constraints. In addition, to avoid thermal throttling, we require that the system's operating temperature remains lower than a thermal threshold. We consider this problem in the context of systems using CUDA because it is widely used in many real-world applications [2]. A CUDA task uses GPU resources through the driver in the operating system (OS) and may rely on some I/O services[1] to complete. Note that although this article focuses on CUDA

Manuscript received March 25, 2019; revised June 28, 2019; accepted August 14, 2019. Date of publication September 11, 2019; date of current version September 18, 2020. This work was supported in part by the National Science Foundation of the United States under Award CNS-1319904, Award CNS-1319784, and Award CNS-1618979, in part by the National Natural Science Foundation of China under Grant 61802185, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20180470, and in part by the Fundamental Research Funds for the Central Universities under Grant 30919011233. This article was recommended by Associate Editor J. Henkel. *(Corresponding author: Junlong Zhou.)*

Y. Ma and X. S. Hu are with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556 USA (e-mail: yma1@nd.edu; shu@nd.edu).

J. Zhou is with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China (e-mail: jlzhou@njust.edu.cn).

T. Chantem is with the Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Arlington, VA 22203 USA (e-mail: tchantem@vt.edu).

R. P. Dick is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: dickrp@umich.edu).

S. Wang is with the Global Research and Development Center, General Motors, Warren, MI 48090 USA (e-mail: shige.wang@gm.com).

Digital Object Identifier 10.1109/TCAD.2019.2940681

[1]In this article, we refer to the OS as the OS kernel, including hardware drivers, and refer to the I/O services as default services shipped with the OS, such as video and audio services.

tasks, the proposed techniques can be readily applied to other programming models.

In order to solve the above problem, we first explore how the mapping of CUDA tasks affects task CPU times and then develop a hybrid framework called HyFRO (Hybrid Framework for Reliability Optimization). This framework: 1) statically maps tasks to CPU cores[2] to improve soft-error reliability; 2) dynamically migrates tasks among CPU cores to balance the wear states among cores and hence achieve a higher lifetime reliability; and 3) dynamically scales frequencies of CPU and GPU cores to increase soft-error reliability under peak temperature, real-time, and lifetime reliability constraints. This article makes three main contributions.

1) Our experiments on multiple hardware platforms reveal that for a CUDA task, CPU time increases if the OS and/or related I/O services are running on different CPU cores from the task. Based on this observation, we generalize a real-time task model to consider the impact of the OS and I/O services on task execution times. This model captures dependencies among tasks, the OS, and I/O services to assist system design and analysis.
2) We develop an off-line task mapping policy to reduce the execution times of tasks and the utilizations of cores. This lower core utilization leads to a higher soft-error reliability.
3) By considering uneven wear states among cores and unavoidable workload and operating environment variations, we design two on-line algorithms. The first one migrates tasks to balance the wear states among cores, which is optimal to maximize lifetime reliability if cores are homogeneous and have same fault-to-failure rate, but suboptimal in other cases. The second algorithm scales frequencies of CPU and GPU cores to further improve soft-error reliability.

We implemented and evaluated HyFRO on Nvidia's TK1 [26] and TX2 [27] chips. The experimental results show that HyFRO increases the probability that no soft-error occurs for at least 3.9 days (about 5 times) and 7.1 days (about 6 times) more than existing approaches on TK1 and TX2, respectively.

The rest of this article is organized as follows. We review the related work in Section II. Section III introduces the system and reliability models. We experimentally explore how task mapping impacts execution times in Section IV. Based on the experiments, we extend the real-time task model in Section V. Section VI formulates the problem and provides an overview of our framework. Section VII describes HyFRO in detail. Sections VIII and IX describe our experimental setup and results, respectively. Section X concludes this article.

II. Related Work

MPSoCs with integrated CPUs and GPUs provide both massively parallel and general-purpose computing capabilities. For such MPSoCs, several papers have discussed how to achieve a lower power consumption and higher performance [28]–[31]. Since the integrated CPU and GPU share memory, Jeong *et al.* proposed to adapt the priority of CPU and GPU memory requests to improve GPU Pathania *et al.* [29] and Prakash *et al.* [30] proposed to maximize MPSoC performance by scaling CPU and GPU core frequency. By considering the specific thermal features of integrated CPUs and GPUs, Wang *et al.* [31] developed a framework to partition and map concurrent applications to maximize system performance under a temperature constraint. While all the above papers consider the specific features of CPUs and GPUs, none focus on reliability.

Several researches have worked to increase CPU soft-error reliability [6]–[11] and/or lifetime reliability [12]–[15]. For soft-error reliability, Zhao *et al.* [7], [9] and Ma *et al.* [11] proposed multiple methods to allocate recoveries to failed tasks. Zhao *et al.* [6] and Fan *et al.* [8] proposed to reduce soft-error rate by increasing core frequencies. In order to improve lifetime reliability, Huang *et al.* [12] mapped and scheduled tasks to guard against aging effects. Das *et al.* [15] proposed a machine learning based algorithm to reduce temperature and mitigate thermal cycling (TC). Since both soft errors and permanent errors may cause system failure, lifetime reliability, and soft-error reliability have been jointly studied [10], [32], [33]. In order to improve system availability, Das *et al.* [10] scaled core frequencies while Zhou *et al.* [32] proposed to allocate replications of tasks and determine core frequencies statically. Ma *et al.* [33] focused on "big–little" type MPSoCs and improved their soft-error reliability under lifetime reliability constraint. All above efforts are effective in improving CPU reliability but ignore the GPU reliability.

In contrast to CPU reliability, there has been little research on GPU reliability [19], [21]–[24]. Tan *et al.* [19] developed a framework to estimate the soft-error vulnerability of general-purpose GPU (GPGPU) and proposed to leverage resistive memory to improve soft-error reliability and reduce energy consumption [21]. To improve soft-error reliability, Lee *et al.* [22] developed a compilation and instruction scheduling method. To improve GPU lifetime reliability, Namaki-Shoushtari *et al.* [23] proposed to balance the wear states of GPU register files. To minimize GPU aging, Rahimi *et al*. [24] developed an aging-aware instruction assignment scheme to evenly distribute the stress of instructions. Although these methods improve soft-error reliability or lifetime reliability, not all GPUs support them. In addition, they ignore CPU reliability.

In this article, we consider the reliability of both CPU and GPU, and propose to maximize reliability by mapping tasks and scaling core frequencies.

III. Preliminaries

This section introduces our hardware and reliability models.

A. Hardware Model

We focus on MPSoCs composed of one GPU (ρ_G) and m homogeneous CPU cores ($\{\rho_1, \ldots, \rho_m\}$). Although Nvidia's GPUs have multiple cores, we cannot explicitly assign tasks

[2]Although Nvidia's GPU has multiple cores, we cannot explicitly schedule specific cores to execute tasks.

to specific cores. Hence, we abstract the GPU as a single processor. The GPU is idle only when no operations execute on any of the GPU cores [34].

We assume both the CPU and GPU support voltage and frequency scaling. A higher voltage and frequency generally produce a higher temperature. We define the utilization of a CPU core or the GPU in a given time interval $|\Delta t|$ as $U = [t_a/|\Delta t|]$, where t_a is the amount of time that the core executes operations. Clearly, a lower core frequency leads to a higher core utilization. The core utilization is commonly used to estimate soft-error reliability and guarantee deadline constraints. The operating temperature of both CPU and GPU cores can be estimated by using an RC thermal modeling tool (e.g., HotSpot [35]) or measured by thermal sensors. Generally, a higher operating temperature leads to a lower lifetime reliability. In order to avoid unexpected thermal throttling, we also require the operating temperature be lower than a threshold.

B. Soft-Error and Lifetime Reliability Model

We consider both soft-error reliability due to transient faults and lifetime reliability due to permanent faults. Since CPU and GPU are identical at the device level, the device-level soft-error reliability, and lifetime reliability models are applicable to both CPU and GPU. The soft-error reliability in a time interval is the probability that no soft errors occur in that time interval [32], i.e.,

$$r = e^{-\lambda(f)\times U\times|\Delta t|} \tag{1}$$

where f is the core frequency, $|\Delta t|$ is the length of the time interval, and U is the core's utilization in this time interval. $\lambda(f)$ is the average fault rate depending on f [32], i.e.,

$$\lambda(f) = \lambda_0 \times 10^{\frac{d(f_{\max}-f)}{f_{\max}-f_{\min}}} \tag{2}$$

where λ_0 is the average fault rate at the maximum core frequency. $f_{\min}$ and $f_{\max}$ are the minimum and maximum core frequency and d ($d > 0$) is a hardware specific constant indicating the sensitivity of fault rates to frequency scaling. The system-level soft-error reliability of an MPSoC is

$$R = r_G \times \prod_{i=1}^{m} r_i \tag{3}$$

where r_G and r_i ($i = 1, \ldots, m$) are the soft-error reliability of ρ_G and ρ_i, respectively [33].

We consider four main integrated circuit (IC) failure mechanisms in this article: 1) electromigration (EM); 2) time dependent dielectric breakdown (TDDB); 3) stress migration (SM); and 4) TC [36]. EM is the dislocation of metal atoms and TDDB is the deterioration of the gate oxide layer. SM is caused by directionally biased motion of atoms in metal wires. Wear due to EM, SM, and TDDB are exponentially dependent on operating temperature. Wear due to TC depends on the amplitude, period, and peak temperature of each cycle. Generally, a lower operating temperature, a smaller amplitude, and a larger period result in a higher lifetime reliability. In this article, we use an existing tool [33] to check whether the lifetime reliability resulting from a thermal profile exceeds a lifetime reliability constraint. Note that this article is independent of the lifetime reliability modeling tool used. The goal of this article is to improve soft-error reliability under peak temperature, real-time, and lifetime reliability constraints. Before developing a framework to address this problem, we first discuss our observations on the relationship between task assignment and execution time.

TABLE I
CUDA TASKS USED TO MEASURE ADDITIONAL EXECUTION TIMES

Name	Description	Source
VectorAdd	Vector addition	CUDA Samples [37]
SimpleTexture	Texture use	
MatrixMul	Matrix multiplication	
Gaussian	Gaussian elimination	Rodinia [38]
BFS	Breadth-first search	
Backprop	Back propagation	

IV. EMPIRICAL STUDY: EFFECTS OF MAPPING ON TASK EXECUTION TIMES

In this section, we discuss one of our major contributions. We focus on CUDA tasks, which use GPU resources through the driver in the OS and some I/O services. One open question is whether a CUDA task's execution time varies if the OS[3] and/or related I/O services are executed on different CPU cores from this CUDA task. We determine the answer by conducting experiments on different hardware platforms.

We performed experiments on Nvidia's TK1 [26] chip (with CUDA 6.5) to measure the CPU times of a CUDA task. We use the default settings: CPU frequency is 2.1 GHz and GPU frequency is 72 MHz. Note that we use the CUDA API cudaEventRecord to record the time stamps before and after the GPU execution, and the elapsed time between two time stamps is GPU time. So, if running CUDA tasks at a high frequency, the elapsed time is about to zero. Hence, in this measurement, we run tasks at a low GPU frequency and repeat multiple times to get the average GPU time. However, for systems having soft real-time requirements, a high GPU frequency is necessary to guarantee the tasks that can complete before their deadlines. To obtain general conclusions, we execute 6 CUDA tasks from different benchmark suites (see Table I). Each task's increases in CPU times are shown in Fig. 1 and the averages of additional GPU times are shown in Table II. The additional CPU times can be significant and vary with different inputs. Although the additional CPU times increase or decrease with different inputs, they can be predicted if tasks' inputs are given. We will use the maximum additional CPU time when designing our framework. In contrast to the additional CPU time, the additional GPU time is negligible: the additional GPU times of all measured CUDA tasks are less than 1% of the tasks' execution times. This increase can be ignored in most of the soft real-time applications.

We also consider a category of tasks which rely on I/O services to complete. CUDA tasks YOLO [39] and

[3]Although it is possible that kernel threads in the OS are allowed to execute on multiple cores, not all of OSs support it. In this article, we assume all kernel threads run on the same core.

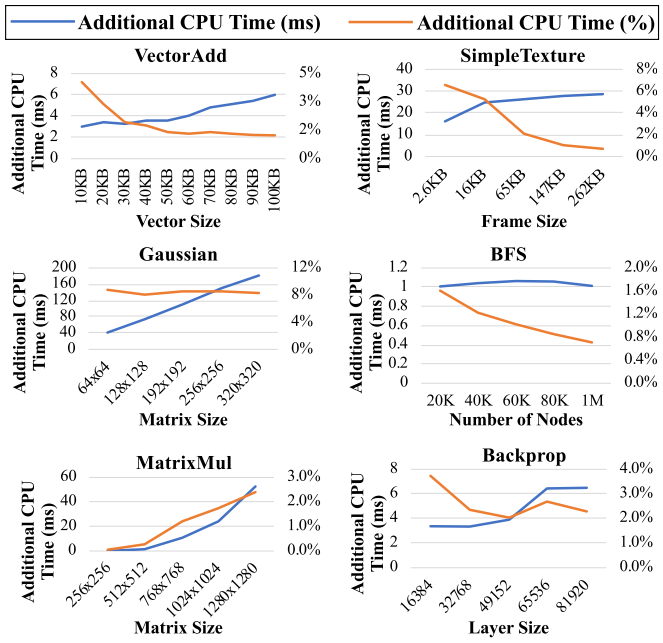


Fig. 1. Measured additional CPU times on TK1 if a CUDA task executes on a different CPU core than the OS.

TABLE II
OBSERVED ADDITIONAL GPU TIME ON TK1

Tasks	Additional GPU Time	
	In Millisecond	In Percentage (%)
VectorAdd	0.38	0.01
SimpleTexture	0.09	0.00
MatrixMul	0.22	0.11
Gaussian	0.38	0.00
BFS	0.003	0.20
Backprop	0.31	0.68

ThunderStruck [40] fall into this category. Both of them rely on xorg, an I/O service in Linux's display system. We determined whether their CPU times and GPU times increase if executed on different cores than related OS services. The additional CPU times of YOLO and ThunderStruck are shown in Fig. 2. Similar as Fig. 1, the additional CPU time can be significant and must be considered. We also determine their additional GPU times. For YOLO and ThunderStruck, the additional GPU times are only 0.02 ms and 0.44 ms, respectively. Since the additional GPU times are less than 0.1% of the tasks' execution times, they can be ignored in most applications.

To determine whether our observation is platform independent, we have also measured the execution times of these tasks on Nvida's TX2 [27] chip (with CUDA 8.0), which consists of 4 ARM cores and 2 Denver cores. Since the OS must execute on the primary core (an ARM core) and a task's execution time is different if executing on an ARM core or a Denver core, we only execute tasks on ARM cores and power off all Denver cores. We set the frequency of the ARM cores to 2.0 GHz and the GPU frequency to 115 MHz. The additional CPU times of CUDA tasks are illustrated in Figs. 3 and 4. The experimental results are similar to those for TK1. Based on the above experiments, we again observe that executing a CUDA

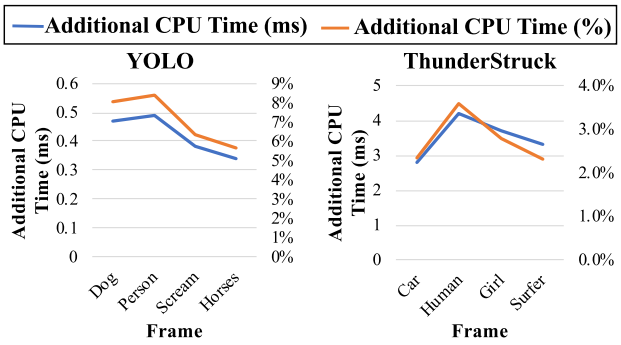


Fig. 2. Measured additional CPU times on TK1 if a CUDA task executes on a different CPU core from the I/O service xorg.

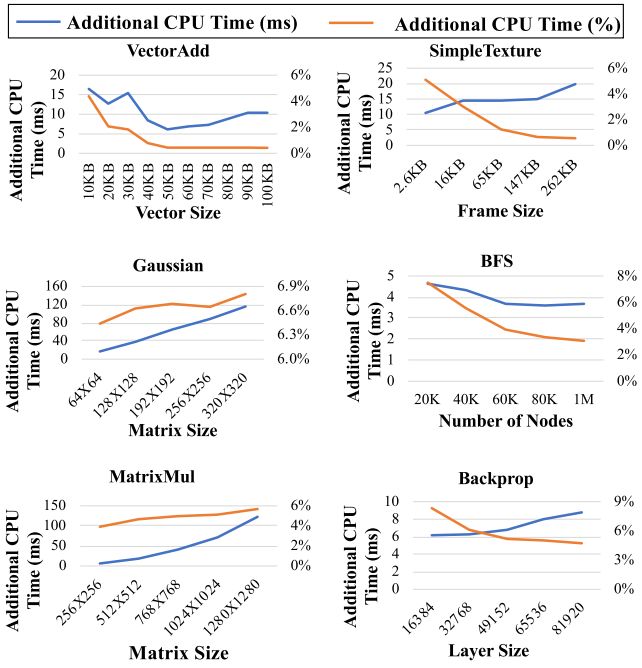


Fig. 3. Measured additional CPU times on TX2 if a CUDA task executing on a different CPU core from the OS.

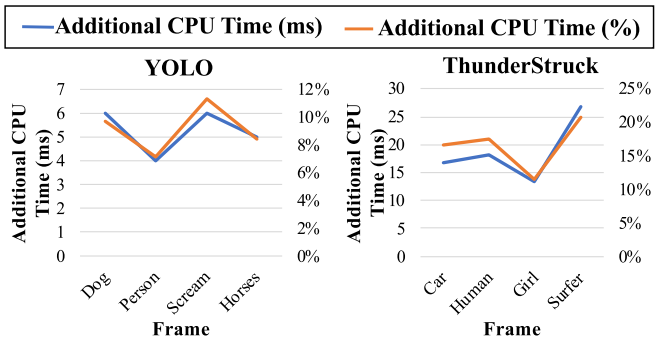
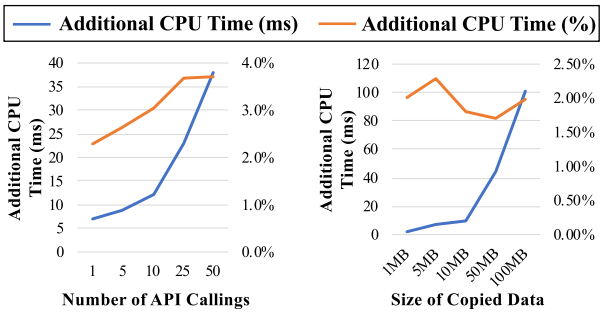


Fig. 4. Measured additional CPU times on TX2 if a CUDA task executing on a different CPU core from the I/O service xorg.

task on a different core from the OS and/or I/O services can significantly increase CPU time but not GPU time.

In order to discover the sources of the additional CPU time, we measured the CPU time of each used CUDA function by



(a) (b)

Fig. 5. Measured additional CPU times on TK1 if (a) calling the CUDA API, cudaMemcpy, a different number of times and (b) copying different size of data between CPU's and GPU's memory space.

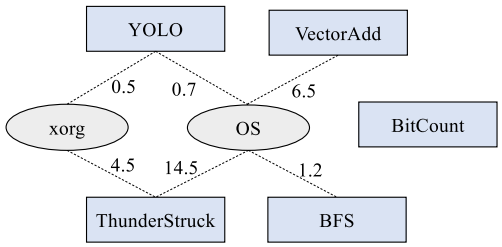


Fig. 6. Example of our task model showing tasks rely on the OS and I/O services to complete.

using nvprof, Nvidia's profiling tool [41]. This reveals that CPU time due to calling synchronized CUDA functions, such as cudaMemcpy, significantly increases if the CUDA task executes on a different core from the OS. Fig. 5(a) illustrates how the additional CPU time depends on the number times of cudaMemcpy is called. Generally, a task that frequently calls synchronized CUDA functions suffers a larger additional CPU time. The data copy between CPU and GPU memory spaces is the another reason for additional CPU time[4] [see Fig. 5(b)]. Hence, we measured the additional CPU time when calling the memory copy CUDA function, cudaMemcpy, only once but copying different amounts of data between CPU and GPU memory space. Fig. 5(b) illustrates how the size of the copied data impacts the additional CPU time. Although the additional CPU time increases with more data, the percentage is a constant. The above experimental results lead to a guideline: to reduce the additional CPU time, a CUDA task should transfer more data per copy to reduce the number of data transfers, and thus the number of CUDA API calls.

Based on the above experiments, we conclude that a GPU task's CPU time increases if executing on a different core from the OS and/or I/O services, but its GPU time does not change. Although all intracore communications may increase the execution times of tasks, our experiments show that the additional CPU times of GPU tasks are much more significant than those of CPU tasks. Hence, we extend the real-time task model to account for the additional CPU times of GPU tasks. In addition, since soft-error reliability is increased if a system has a lighter workload [in (1)], we develop a mapping policy to minimize task CPU times.

V. Integrated Task, OS, and I/O Services Model

We extend the real-time task model to account for the fact that a task depends on the OS and I/O services to complete. The real-time tasks considered in this article are independent, periodic, and have soft deadlines. Task τ_i is associated with a tuple $\{e_i, d_i, A_i\}$ where e_i is the execution time, d_i is the deadline, and A_i captures the dependencies of τ_i on the OS and I/O services. The execution time of a CUDA task consists of two parts: 1) the execution time on CPU, e_i^C and 2) the execution time on GPU, e_i^G. Since a CUDA task may call multiple synchronous and asynchronous CUDA functions, the relationships among e_i^C, e_i^G, and e_i are complicated. However, for CUDA tasks executing on Nvidia's TK1 or TX2, the CPU busy waits during GPU operation [42], i.e., e_i^C including the time executing code on CPU. Hence, it is safe to assume that $e_i^C = e_i$ and a longer (shorter) e_i^G leads to a longer (shorter) e_i^C and e_i.

For task τ_i, we use A_i to describe its dependencies on the OS and I/O services. A_i is a hash table where the key is the name of an I/O service or the OS, and the value is the additional CPU time. This table can be built off-line through profiling. For example, for the task YOLO (in Section IV), entry A = {OS:0.7, xorg:0.5} indicates that YOLO depends on OS and xorg, an I/O service. The additional CPU times are 0.7 and 0.5 ms if executing YOLO on different cores from the OS and xorg, respectively. Although the additional CPU time varies with input, we choose the worst-case additional CPU time for each dependency to guarantee the real-time constraint.

We can use an undirected graph to describe how tasks depend the OS and I/O services to complete. Tasks, the OS, and I/O services are represented by nodes, and edges represent their dependencies. The weight of each edge is the additional CPU time, if the task and the OS or I/O service execute on different cores. For example, Fig. 6 illustrates that the tasks YOLO and ThunderStruck rely on both xorg and the OS, but VectorAdd and BFS only depend on the OS. The BitCount task from the MiBench Benchmark Suite [43] only uses CPU resources. It is independent of the OS and I/O services. Since this graph shows all dependencies, we can use it to develop a mapping method to minimize additional CPU times of tasks and reduce the overall workload of cores and hence improve soft-error reliability.

VI. Problem Formulation and Framework Overview

In this section, we first formulate the problem addressed in this article and give an overview of our solution HyFRO.

A. Problem Formulation

The problem in this article is motivated by applications such as in-vehicle infotainment system. Such system has soft deadline, temperature, lifetime reliability, and soft-error reliability requirements [44]. Before formulating the problem, we

[4]For TK1 and TX2, although their CPU and GPU share main memory, memory copy functions, such as cudaMemcpy, still copy data between CPU and GPU memory spaces.

first introduce two concepts: 1) sampling window (W) and 2) profiling window. A sampling window is defined as a time interval in which the temperature can be treated as constant. Task migration is not allowed inside a sampling window [33]. A profiling window is composed of multiple equal-length sampling windows and is used to estimate lifetime reliability. Note that our proposal can be easily applied to profiling windows of arbitrary length.

Let us assume that a profiling window is composed of n sampling windows and the MPSoC has m CPU cores and one GPU. Our objective is to maximize the system-level soft-error reliability in each profiling window and satisfy the design constraints in each sampling window (represented by the jth sampling window W_j), which is formulated as

$$\max\left\{\prod_{j=1}^{n}\left(r_{G,j}\times\prod_{i=1}^{m}r_{i,j}\right)\right\} \tag{4}$$

$$\text{s.t.}\begin{cases}\max\{T(\rho_i,W_j)\}\le T_{th}^{C},\forall\rho_i,\forall W_j & (5)\\ T(\rho_G,W_j)\le T_{th}^{G},\forall\rho_i,\forall W_j & (6)\\ U(\rho_i,W_j)\le U_{th},\forall\rho_i,\forall W_j & (7)\\ \min\{LTR(\rho_i),LTR(\rho_G)\}>LTR_{th},\forall\rho_i & (8)\end{cases}$$

$r_{G,j}$ and $r_{i,j}$ are the soft-error reliability of ρ_G and ρ_i at W_j, respectively. For any CPU core ρ_i and the GPU core ρ_G, the constraints in (5)–(8) should be satisfied. The first two constraints require the temperature of both CPU cores and the GPU to be less than some thresholds T_{th}^{C} and T_{th}^{G} in each sampling window. For soft real-time systems, temporarily violating the deadline and the lifetime reliability constraints is acceptable, but the temperature constraint must be satisfied to avoid thermal throttling. The third constraint captures the real-time requirement, where $U(\rho_i, W_j)$ is the utilization of ρ_i at W_j and U_{th} is the upper bound on utilization to satisfy schedulability. The last constraint requires the MTTF resulting from the runtime temperature to be larger than a threshold LTR_{th}. In this article, the CPU and GPU are integrated on a single chip and permanent faults from either the CPU or GPU can cause system failure. Hence, we require that the minimum lifetime reliability of CPU and GPU cores be larger than the lifetime reliability threshold [45].

Since soft-error reliability is related to core utilization, we first design an off-line heuristic to map tasks, the OS, and I/O services to reduce the total utilization of cores. Then, we dynamically migrate tasks among CPU cores to achieve a higher lifetime reliability and scale frequencies of CPU and GPU cores in each sampling window to improve the soft-error reliability. We integrate these two efforts into a hybrid framework to solve our problem efficiently.

B. Overview of HyFRO

In this article, we develop a hybrid off-line/on-line framework, HyFRO, to solve the problem defined in (4)–(8) (see Fig. 7). There are three major challenges to address: 1) the impact of task mapping on task execution times; 2) unbalanced wear states among cores, which reduces the lifetime reliability; and 3) workload and runtime environment variations. In order to address these challenges, the basic idea of our framework is to: 1) map tasks, I/O services, and the OS to appropriate CPU cores statically to minimize additional CPU time; 2) dynamically migrate tasks to balance the wear states among cores; and 3) scale frequencies of CPU and GPU cores to maximize soft-error reliability under lifetime reliability and operating temperature constraints.

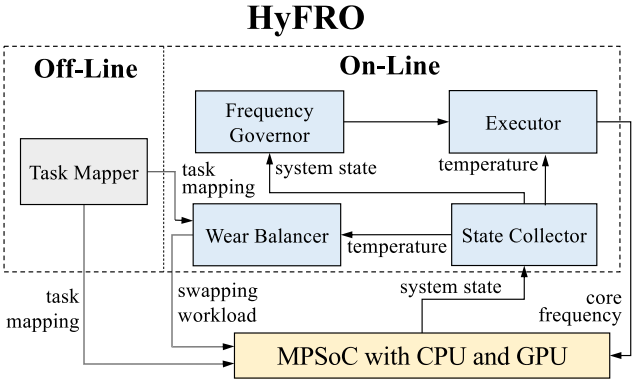


Fig. 7. High-level overview of HyFRO.

The off-line component in HyFRO maps tasks, I/O services, and the OS to appropriate CPU cores (the left part of Fig. 7). Based on the soft-error reliability model in (1), soft-error reliability is improved with shorter task execution times. We develop a task mapper that allows CUDA tasks, the OS, and related I/O services to execute on the same core. This mapping minimizes additional CPU times and reduces task execution times, which improves soft-error reliability. However, it reduces lifetime reliability since cores' workloads and wear states are uneven [45]. We improve lifetime reliability by dynamically migrating tasks and balancing the wear states among CPU cores in the on-line component of HyFRO.

The on-line component in HyFRO: 1) balances the wear states among cores by migrating tasks and 2) improves soft-error reliability by dynamically increasing CPU and GPU core frequencies. This component consists of four main parts: 1) a wear balancer 2) a frequency governor; 3) an executor; and 4) a state collector (the right part of Fig. 7). The wear balancer and frequency governor are triggered at the beginning of each profiling window, and the executor and state collector are triggered in each sampling window.

In order to compensate for the impact of the off-line task mapping on lifetime reliability, the wear balancer swaps the workload among CPU cores at each profiling window to balance the wear states among cores and improve lifetime reliability [45]. To improve the soft-error reliability, the state collector, frequency governor, and executor work together to dynamically increase CPU and GPU core frequencies. In each sampling window, the state collector collects and saves the system states, including each core's temperature, utilization, and frequency. Based on the system states, at the beginning of each profiling window, the frequency governor determines the cores' frequencies for both CPU and GPU for all sampling windows in the next profiling window. This decision is based on the past system states. However, since the workload may change at run time, the decision made by the frequency

Algorithm 1 Task Mapping

1: $U(s_i)$: utilization of the service s_i, which can be measured offline
2: $U(\rho_i)$: utilization of the core ρ_i
3: $\delta_{i,j}$: the additional CPU time if s_i and τ_j are on different cores
4: Δ: $\{\delta_{i,j}$, for all dependencies among tasks and services$\}$
5: **procedure** MAPPING
6: Set $U(\rho_i) = 0$ for all cores
7: Sort Δ in deceasing order
8: **for** each element in the sorted Δ **do**
9: Assume the current element in Δ is $\delta_{i,j}$
10: **if** Both s_i and τ_j have not been mapped **then**
11: ρ_l: the core with lowest utilization
12: **if** $U(\rho_l) + U(s_i) + \frac{e_j}{d_j} \leq U_{th}$ **then**
13: Map s_i and τ_j to ρ_l
14: **else**
15: Map s_i, τ_j to two cores with lowest utilizations
16: **end if**
17: **end if**
18: **if** τ_j (or s_i) is already mapped **then**
19: ρ_l: the core τ_j (or s_i) executes on
20: **if** $U(\rho_l) + U(s_i) \leq U_{th}$ (or $U(\rho_l) + \frac{e_j}{d_j} \leq U_{th}$) **then**
21: Map s_i (or τ_j) to ρ_l
22: **else**
23: Map s_i (or τ_j) on the core with lowest utilization
24: **end if**
25: **end if**
26: **end for**
27: Map CPU tasks to balance the workload among cores
28: **end procedure**

governor may not meet all the constraints in (5)–(8). In order to always meet the hard constraint, i.e., the temperature constraint in (5) and (6), in each sampling window, the executor may modify that the decision to adapt to runtime variations. We elaborate on the details of HyFRO in the next section.

VII. Framework Details

In this section, we provide the details of our framework to improve soft-error reliability under temperature, real-time, and lifetime reliability constraints.

A. Task Mapper

The task mapper maps tasks, I/O services, and the OS to appropriate CPU cores to reduce total execution times, which helps to improve soft-error reliability. Since the assignment-dependent CPU times of GPU tasks is large, we first try to map GPU tasks, I/O services, and the OS. Then, we map normal CPU tasks to spatially balance the workload among cores. In order to reduce computational complexity, we minimize the additional CPU time, thus improving soft-error reliability (see Algorithm 1). The general idea of this algorithm is to map tasks to the same core running the OS and related I/O services. In this algorithm, we represent OS with s_{os} and I/O service as s_i. $\delta_{i,j}$ ($\delta_{os,j}$) represents the additional CPU time if the I/O service s_i (the OS s_{os}) and task τ_j are on different cores. We first sort the dependencies and iteratively select the largest additional CPU time (supposing it is $\delta_{i,j}$), and map s_i and τ_j to appropriate cores (lines 7–26). If both s_i and τ_j have not been mapped, we attempt to map them to a single core if doing so would not violate the real-time requirement. Otherwise, we map s_i and τ_j to two cores with low utilizations (lines 10–17). If s_i (τ_j) is already mapped, we map τ_j (s_i) to the same core running s_i (τ_j) if allowed, and map τ_j (s_i) to a core with lowest utilization otherwise (lines 18–25). After mapping all GPU related tasks, we attempt to map normal CPU tasks and spatially balance workload among CPU cores by using an existing approach [45] (line 27). Since Algorithm 1 checks every dependency, its complexity is linear in the number of dependencies among tasks, I/O services, and the OS.

Algorithm 2 Balance Workload Among CPU Cores

1: **procedure** BALANCER
2: Sort cores with temperature: $T(\rho_i) > T(\rho_{i+1})$
3: **for** $i = \{1, \ldots, m-1\}$ **and** $j = \{m, m-1, \ldots, i+1\}$ **do**
4: **if** $T(\rho_i) - T(\rho_j) < T_{cyc}$ **and** ρ_i and ρ_j not swapped **then**
5: Swap the workload between ρ_i and ρ_j
6: **break**
7: **end if**
8: **end for**
9: **end procedure**

B. Wear Balancer

Although the task mapper in HyFRO can avoid additional CPU time and reduce the total utilization of cores, it leads to the unbalanced core workloads and wear states. For homogeneous cores having same fault-to-failure rate, balancing the wear states among cores is optimal to maximize lifetime reliability, but still suboptimal in other cases. Hence, to achieve higher lifetime reliability, it is necessary to migrate tasks and balance the wear states among cores at run time. In order to not sacrifice the soft-error reliability, we design a heuristic to balance the wear states by swapping the workload among cores.

The basic idea of the wear balancer is to swap the entire workload on a core having a higher temperature with that on another core having a lower temperature (see Algorithm 2). Swapping workload is efficient to balance the wear states, but it may cause TC which reduces lifetime reliability. Hence, at the beginning of each profiling window, we first sort cores by their temperatures (in line 2) and only swap the workloads between cores if the difference in the cores' temperatures is smaller than a prespecified threshold, T_{cyc} (lines 3–8). T_{cyc} is used to ensure that the TC will not become the dominant reason for permanent faults [45]. We can determine the value of T_{cyc} by using an existing lifetime reliability modeling tool [46] at design time. Specifically, for a given thermal profile, this tool reports the effect of both temperature and TC on lifetime reliability. T_{cyc} is the largest value where the effect of TC is smaller than the effect of temperature. The complexity of Algorithm 2 is $O(m)$ where m is the number of CPU cores.

C. Frequency Governor

To improve the soft-error reliability, the frequency governor is triggered at the beginning of each profiling window to determine core frequencies for the next profiling window based on the past system states. The resulting, schedule, for

Algorithm 3 Determine Core Frequencies

1: $W_{p,q}$: the p^{th} sampling window in the q^{th} profiling window
2: $L(\rho_i, W_{p,q})$: core frequency level of ρ_i at $W_{p,q}$
3: $U(\rho_i, W_{p,q})$: utilization of ρ_i at $W_{p,q}$
4: $T(\rho_i, W_{p,q})$: operating temperature of ρ_i at $W_{p,q}$
5: **procedure** GENERATOR($Sched_q$, $State_q$)
6: **if** LTR(GPU) $< LTR_{th}^G$ **then**
7: Sort W: $T(\rho_G, W_{p,q}, L) > T(\rho_G, W_{p+1,q}, L)$
8: **for** $i = \{1, 2, 3, \ldots, n\}$ **do**
9: **if** $L(\rho_G, W_{i,q}) - 1$ not violating Eq. (7) **then**
10: $L(\rho_G, W_{i,q+1}) = L(\rho_G, W_{i,q}) - 1$
11: **Break**
12: **end if**
13: **end for**
14: **else**
15: **while** $T(\rho_G, W_{p,q}, L+1) \leq T_{th}^G$ **do**
16: $L(\rho_i, W_{p,q+1}) = L(\rho_i, W_{p,q}) + 1$
17: **end while**
18: **end if**
19: **if** LTR(CPU) $< LTR_{th}^C$ **then**
20: $W_{h,q}$: the CPU has highest average temperature at $W_{h,q}$
21: **for** $i = \{1, 2, \ldots, m\}$ **do**
22: **if** $L(\rho_i, W_{h,q}) - 1$ not violating Eq. (7) **then**
23: $L(\rho_i, W_{h,q+1}) = L(\rho_i, W_{h,q}) - 1$
24: **end if**
25: **end for**
26: **else**
27: $W_{l,q}$: the CPU has lowest average temperature at $W_{l,q}$
28: **for** $i = \{1, 2, \ldots, m\}$ **do**
29: **if** $T(\rho_i, W_{l,q}, L+1) \leq T_{th}^C$ **then**
30: $L(\rho_i, W_{l,q+1}) = L(\rho_i, W_{l,q}) + 1$
31: **end if**
32: **end for**
33: **end if**
34: Return the schedule $Sched_{q+1}$ with updated core frequencies
35: **end procedure**

the $(q+1)$th profiling window, Sched_{q+1}, is generated by tuning the schedule Sched_q, along with the system states State_q. The initial schedule used in the first profiling window sets the lowest frequencies of both CPU and GPU cores to honor the temperature constraint.

The main procedure of the schedule generator is given in Algorithm 3. If a GPU schedule violates the lifetime reliability constraint, we sort the sampling windows by temperatures, then iteratively find sampling windows in which the GPU has the highest temperature. The core frequencies in these windows are reduced if doing so does not violate the utilization constraint (lines 6–14). If the current schedule satisfies the lifetime reliability constraint for the GPU, we increase the GPU frequency in each sampling window to maximize soft-error reliability if doing so does not violate the temperature constraint (lines 15–17). We also adjust CPU core frequencies in a similar manner. The core's frequencies in the sampling window with highest (lowest) temperature are reduced (increased) if the previous schedule cannot (can) adhere to the lifetime reliability constraint (lines 19–33). After tuning the schedule in the previous profiling window, a new schedule for the next profiling window is generated (line 34). The complexity of Algorithm 3 is $O(m \times n + n)$, where m is the number of CPU cores and n is the number of sampling windows in a profiling window.

D. Executor

The executor sets the frequencies of CPU and GPU cores at the beginning of each sampling window. A straightforward approach is to simply follow the schedule generated by the frequency governor. However, this schedule is generated based on the past system state and the workload of tasks may vary at the run time. Hence, this schedule may actually violate some or all of the constraints. For soft real-time systems, it is acceptable to temporarily violate constraints (7) and (8), as this can be compensated for in the next profiling window. However, violating the temperature constraint in (5) and (6) may cause thermal throttling. Therefore, the executor should be designed properly to handle this case. We statically establish a safe initial temperature for every core frequency [33]. At the beginning of each sampling window, the executor only needs to determine whether the initial temperature exceeds the constrain. If so, the temperature of this sampling window may be larger than the thermal threshold, so we need to reduce the core frequency in this sampling window [33]. We can statically establish the safe initial temperature, so the time complexity of executor is $O(1)$.

VIII. Experimental Setup

We conducted the experiments comparing `HyFRO` with existing approaches. In this section, we present the existing approaches, experimental platforms, and workloads in the experiments.

A. Comparison Targets

We compared our framework with two representative approaches: 1) the dynamic reliability improvement framework (`DRI`) [33] and 2) multiobjective optimization of reliability (`MOO`) [10]. Similar to `HyFRO`, `DRI` dynamically scales core frequencies to improve soft-error reliability under temperature, real-time, and lifetime reliability constraints. However, the additional CPU times of GPU tasks are ignored and task migration is not allowed at run time. `MOO` maximizes the minimum of soft-error reliability and lifetime reliability by statically determining the frequencies of CPU cores [10]. Since both `DRI` and `MOO` ignore GPU reliability requirements, they use the default strategies deployed in the OS to scale the GPU frequency.

Three metrics are considered in the comparison. The probability of failures (PoFs) due to soft errors quantifies the soft-error reliability. The PoF is defined as $1 - R$, where R is the system-level soft-error reliability defined in (3). Approaches achieving lower PoFs achieve higher soft-error reliabilities. The percentage of feasible solutions under the real-time constraints (FS-RT) is used to describe the probability of satisfying real-time constraints. Based on the concept of job, i.e., a task instance, the percentage of FS-RT is quantified as the ratio of the number of jobs meeting their deadlines over the total number of jobs. Similarly, the percentage of feasible solutions for lifetime reliability constraint (FS-LTR) describes the capability of satisfying lifetime reliability requirements. FS-LTR is the ratio of the number of profiling windows achieving a higher lifetime reliability than the lifetime reliability constraint over the total number of profiling windows.

Meanwhile, we set one profiling window to contain 50 sampling windows. The length of each sampling window is 1 s. Similar to our previous work [47], a thermal profile with 50 temperature points (sampling one temperature point per sampling window) is long enough to analyze the effect of TC.

B. Experimental Platforms

The experiments were conducted on two boards containing Nvidia's TK1 [26] and TX2 [27] chips. The TX2 contains two Denver [48] cores and four ARM Cortex-A57 cores. We implemented HyFRO on user space and deployed it on one Denver core. Tasks, OS, and I/O services are executed on ARM cores. TX2 also contains a Pascal architecture-based GPU that is highly power efficient and supports most modern graphics APIs. As a low-power chip, the CPU supports multiple frequencies from 0.35 GHz to 2.04 GHz, and the GPU frequency ranges from 0.83 GHz to 1.30 GHz. Thermal sensors are used to sample temperatures of the CPU, GPU, and other components. Since the default interface only provides one CPU temperature for all CPU cores, we assume all CPU cores have the same temperature. TX2 is shipped with an OS based on Ubuntu and is capable of executing some widely used benchmarks.

The TK1 provides four homogeneous high-performance CPU cores[5] and contains a Kepler architecture based GPU. These four CPU cores must share the same voltage and frequency. We use all CPU cores to execute tasks as well as HyFRO. TK1's CPU supports multiple frequencies from 1.32 GHz to 2.32 GHz, and its GPU can work from 0.25 GHz to 0.54 GHz. Similar to TX2, we obtain CPU and GPU operating temperatures by reading thermal sensors and assuming all CPU cores have the same temperature.

C. Workloads

We now discuss the task set used in the experiments. We selected six tasks from multiple benchmark suites: MiBench [43], CUDA samples [37], and Rodinia [38] (see Table III).[6] Tasks VectorAdd, MatrixMul, Backprop, and Gaussian are CUDA tasks and rely on the OS. CRC and Dijkstra only use the CPU. We measured the CPU and GPU times of these tasks on TX2's ARM core at 2.04 GHz and GPU at 1.30 GHz. The measured execution time and additional CPU time will be used by the task mapper in Algorithm 1. In the experiments, the jobs of each task are periodically released, and a job missing its deadline is immediately terminated. Considering the different real-time requirements, we designed two groups of task setups. In the first group, tasks are frame-based and share the same period and deadline. We evaluated HyFRO when the deadlines are 0.50, 0.75, 1.00, and 1.25 s. In the second group, a task's deadline and period are randomly selected from the following ranges: 0.50–0.75 s, 0.75–1.00 s, and 1.00–1.25 s.

[5]TK1 also has a low-power CPU core, but it cannot run simultaneously with the high-performance cores. Hence, in the experiments, the low-power CPU core is powered off.

[6]Note that we do not use all cores to execute more tasks. One reason is for the memory limit. The other reason is to avoid the overhead of GPU whose resources are shared by all tasks.

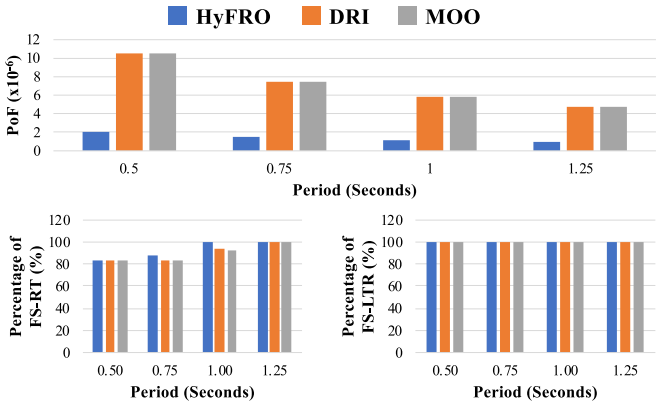


Fig. 8. Probabilities of failures due to soft errors and percentages of feasible solutions for frame-based tasks running on TX2.

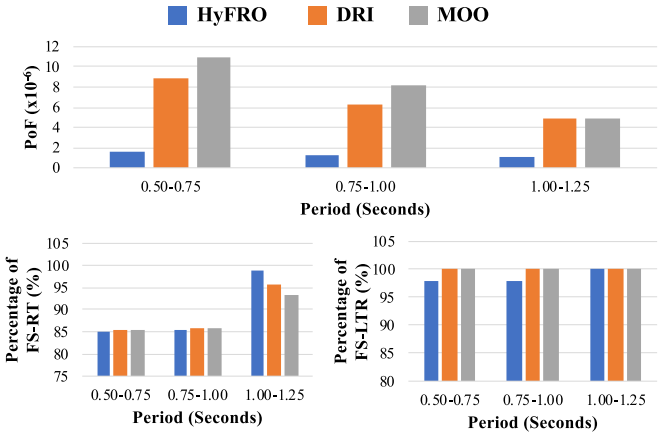


Fig. 9. Probabilities of failures due to soft errors and percentages of feasible solutions for general periodic tasks running on TX2.

IX. Experimental Results

In this section, we compare HyFRO with DRI and MOO.

A. Experiments on Nvidia's TX2 Chip

We first validated our approach on Nvidia's TX2 chip whose GPU can work at a high frequency. We compared HyFRO with MOO and DRI to determine whether HyFRO can improve soft-error reliability without violating temperature, real-time, and lifetime reliability constraints.

Fig. 8 illustrates the experimental results when tasks are frame-based. As can be seen, HyFRO achieves lower PoFs than both DRI and MOO for all considered periods. The PoFs of HyFRO are 2×10^{-6}, 1.6×10^{-6}, 1.1×10^{-6}, and 9.4×10^{-7} when the periods are 0.50, 0.75, 1.00, and 1.25 s. The achieved PoFs indicate that the system can work without soft errors about 5.8 days, 7.4 days, 10.2 days, and 12.3 days. In contrast with DRI, HyFRO dynamically migrates the tasks among CPU cores and scales GPU core frequencies, which represents CPU and GPU executing at high frequencies and reducing soft error rates. Our experimental results show that the PoFs of HyFRO is 19.10%, 21.14%, 19.48%, and 20.08% of DRI. With DRI, the time that the system can run without soft errors is only 1.1 days, 1.6 days, 2.0 days, and 2.5 days when the period is 0.50 s, 0.75 s, 1.00 s, and 1.25 s, respectively. Similarly, the

TABLE III
TASKS IN EXPERIMENTS

Task	GPU Time	CPU Time	Additional CPU Time
VectorAdd [37]	3.9 ms	356 ms	4 ms
MatrixMul [37]	23.0 ms	120 ms	6 ms
Gaussian [38]	16.6 ms	196 ms	12 ms
Backprop [38]	6.1 ms	125 ms	7 ms
CRC [43]	0 ms	65 ms	0 ms
Dijkstra [43]	0 ms	60 ms	0 ms

PoF of HyFRO is only 19.11%, 21.17%, 19.45%, and 19.89% of MOO. It means that HyFRO can make the system can run 4.7 days, 5.8 days, 8.2 days, and 9.9 days longer than MOO. In addition, since all these approaches consider real-time and lifetime reliability constraints, the percentages of FS-RT and FS-LTR are HyFRO, DRI, and MOO are close.

We also evaluated HyFRO on implicit deadline periodic tasks with randomly generated periods (see Fig. 9). The PoF of HyFRO is 1.6×10^{-6}, 1.2×10^{-6}, and 1.0×10^{-6} when periods of 0.50–0.75 s, 0.75–1.00 s, and 1.00–1.25 s. This indicates that the system can work without soft errors about 7.0 days, 9.6 days, and 11.4 days. The average PoF of HyFRO is 19% of DRI, which translates to HyFRO allowing the system to successfully work for 7.5 days longer on average, and 5.7 days longer, in the worst case. Similarly, the average PoF of HyFRO is 16% of MOO, and HyFRO allows the system to successfully work for 7.8 days longer on average, and 6.0 days longer, in the worst case. In contrast to HyFRO, both DRI and MOO use Linux's default power management strategy to control the core frequency of the GPU. With this strategy, the GPU frequency increases only when the workload is heavy. However, in our experiments, the GPU workload is not heavy, so the GPU frequency is generally low. On the contrary, HyFRO increases core frequency but still honor the peak temperature constraint. Compared to DRI and MOO, HyFRO achieves a similar percentage of FS-RT when the workload is light and a higher one when the workload is heavy. Although the FS-LTR for HyFRO is 2% lower than DRI and MOO when the workload is light, the impact on lifetime reliability can be ignored since it is compensated for in the next profiling window.

We also measured the time and power consumption of HyFRO. HyFRO generally scales core frequencies once per second, and this imposes less than 1 ms of overhead. Hence, we claim that the time overhead and power consumption of HyFRO can be ignored.

B. Experiments on Nvidia's TK1 Chip

We conducted the experiments on TK1 chip to evaluate the performance of HyFRO. The GPU in TK1 has a different microarchitecture from the TX2 and runs CUDA 6.5. This experiment is used to determine whether the effectiveness of HyFRO is independent of hardware platform.

Similar to the experiments on TX2, we tested HyFRO for 1) frame-based tasks (see Fig. 10) and 2) general periodic tasks (see Fig. 11). For the frame-based task set, the PoF of HyFRO is 4.8×10^{-6}, 3.5×10^{-6}, 2.9×10^{-6}, and 1.9×10^{-6} for periods of 0.50 , 0.75 , 1.00 , and 1.25 s. This indicates that the system can work without soft errors for 2.4 days, 3.3 days, 4.1 days, and 6.1 days. Compared to DRI, the PoF of HyFRO is 0.15%, 0.16%, 0.18%, and 0.15% of DRI when the period

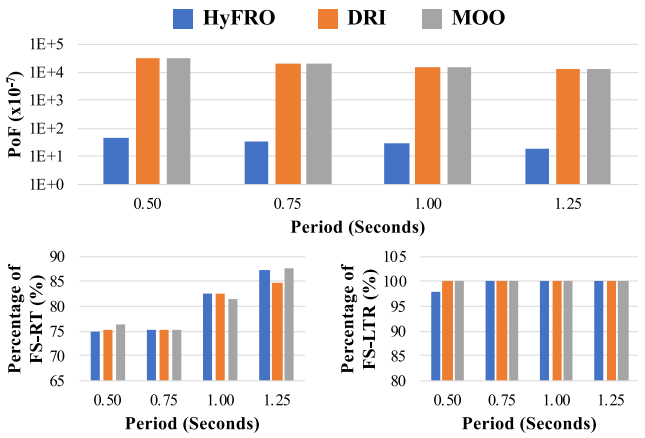


Fig. 10. Probabilities of failures due to soft errors and percentages of feasible solutions for frame-based tasks running on TK1.

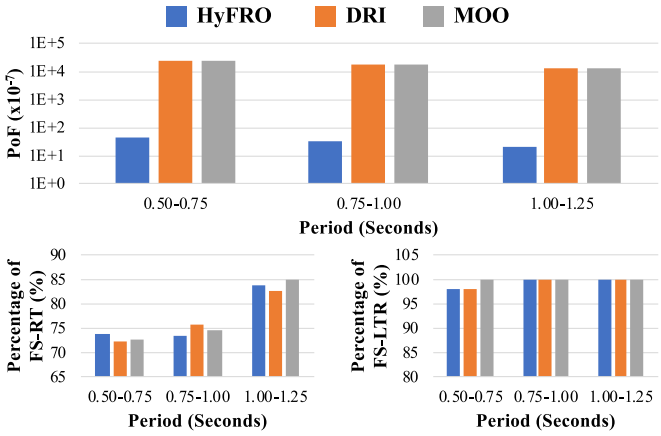


Fig. 11. Probabilities of failures due to soft errors and percentages of feasible solutions for general periodic tasks running on TK1.

is 0.50 , 0.75 , 1.00 , and 1.25 s. In other words, DRI can only make the system successfully run 0.08 , 0.13 , 0.17 , and 0.22 h. Compared to Fig. 8, the soft-error reliability improvement of HyFRO is much longer. The TK1 has a larger range of GPU core frequencies with the highest being double to the lowest. In this situation, HyFRO reduces the soft-error rate more. Compared to MOO, the PoF of HyFRO is 0.18% of MOO at most, and 0.16% on average. This leads to a system that can run 3.9 days longer than MOO on average and up to 6.1 days. In terms of satisfying the real-time and lifetime reliability constraints, HyFRO achieves similar percentages of FS-RT and FS-LTR to both DRI and MOO.

Fig. 11 shows the performance of HyFRO when running periodic tasks with randomly generated periods. The PoF of HyFRO is about 0.16%, 0.18%, and 0.17% of DRI for task periods of ranges 0.50–0.75 s, 0.75–1.00 s, and 1.00–1.25 s. This indicates that HyFRO allows the system to function 4.0 days more than DRI on average, and up to 5.4 days. Meanwhile, the soft-error reliability improvement of HyFRO over MOO is similar to that over DRI. The time overhead and power consumption of HyFRO on TK1 are also too small to be observed. In summary, the above experiments confirm that HyFRO is better of improving the soft-error reliability in all considered cases.

X. Conclusion

In this article, we aimed to improve the reliability of real-time embedded systems on integrated CPU and GPU platforms. We observed that the tasks CPU execution times may vary with task-to-core mappings. Based on this observation, we first extended the real-time task model by considering the dependencies among tasks, the OS, and I/O services. We then described a hybrid soft-error reliability improvement framework that considers temperature, real-time, and lifetime reliability constraints. We described an off-line mapping policy to reduce the total utilization of cores and improve soft-error reliability. We also described an on-line component that dynamically migrates tasks to achieve a higher lifetime reliability and adjusts the frequencies of CPU and GPU cores to improve soft-error reliability. The experimental results show that our approach increases the soft-error reliability without violating temperature, real-time, and lifetime reliability constraints.

XI. Future Works

In the future, we plan to extend our measurements in Section IV by experimenting with additional GPU frequencies and hardware platforms. We also plan to establish a concrete method to model the impact of assign decisions on CPU and GPU times.

References

[1] Z. Li, Y. Wang, T. Zhi, and T. Chen, "A survey of neural network accelerators," *Front. Comput. Sci.*, vol. 11, no. 5, pp. 746–761, Jan. 2017.

[2] *Nvidia Metropolis—The Foundation for Smart Cities*, Nvidia, Santa Clara, CA, USA, 2018. Accessed: Oct. 2018. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/intelligent-video-analytics-platform/

[3] *CUDA DRIVE Software*, Nvidia, Santa Clara, CA, USA, 2019. Accessed: Apr. 2019. [Online]. Available: https://developer.nvidia.com/drive/drive-software

[4] J. McLeish. (2018). *Autonomous Vehicles Electronics: Reliability Challenges and Solutions*. Accessed: Oct. 2018. [Online]. Available: https://www.dfrsolutions.com/autonomous

[5] B. Zhao, H. Aydin, and D. Zhu, "On maximizing reliability of real-time embedded applications under hard energy constraint," *IEEE Trans. Ind. Informat.*, vol. 6, no. 3, pp. 316–328, May 2010.

[6] B. Zhao, H. Aydin, and D. Zhu, "Reliability-aware dynamic voltage scaling for energy-constrained real-time embedded systems," in *Proc. Int. Conf. Comput. Design*, Oct. 2008, pp. 633–639.

[7] B. Zhao, H. Aydin, and D. Zhu, "Enhanced reliability-aware power management through shared recovery technology," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2009, pp. 63–70.

[8] M. Fan, Q. Han, S. Liu, and G. Quan, "On-line reliability-aware dynamic power management for real-time systems," in *Proc. Int. Symp. Qual. Electron. Design*, Santa Clara, CA, USA, Mar. 2015, pp. 361–365.

[9] B. Zhao, H. Aydin, and D. Zhu, "Generalized reliability-oriented energy management for real-time embedded applications," in *Proc. Design Autom. Conf.*, Jun. 2011, pp. 381–386.

[10] A. Das, A. Kumar, B. Veeravalli, C. Bolchini, and A. Miele, "Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs," in *Proc. Design Autom. Test Europe*, Dresden, Germany, Mar. 2014, pp. 1–6.

[11] Y. Ma, T. Chantem, R. P. Dick, and X. S. Hu, "Improving reliability for real-time systems through dynamic recovery," in *Proc. Design Autom. Test Europe*, Dresden, Germany, Mar. 2018, pp. 515–520.

[12] L. Huang, F. Yuan, and Q. Xu, "Lifetime reliability-aware task allocation and scheduling on MPSoC platform," in *Proc. Design Autom. Test Europe*, Nice, France, Mar. 2009, pp. 51–56.

[13] C. Bolchini, M. Carminati, A. Miele, A. Das, A. Kumar, and B. Veeravalli, "Run-time mapping for reliable many-cores based on energy/performance trade-offs," in *Proc. Design Autom. Conf.*, Jun. 2013, pp. 58–64.

[14] A. Das, A. Kumar, and B. Veeravalli, "Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia MPSoCs," in *Proc. Design Autom. Test Europe*, Mar. 2014, pp. 1–6.

[15] A. Das, R. Shafik, G. V. Merrett, B. Al-Hashimi, A. Kumar, and B. Veeravalli, "Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems," in *Proc. Design Autom. Conf.*, San Francisco, CA, USA, Jun. 2014, pp. 1–6.

[16] J. Zhou *et al.*, "Resource management for improving soft-error and lifetime reliability of real-time MPSoCs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published. doi: 10.1109/TCAD.2018.2883993.

[17] Y. Ma, J. Zhou, T. Chantem, R. P. Dick, S. Wang, and X. S. Hu, "On-line resource management for improving reliability of real-time systems on 'Big–little' type MPSoCs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published. doi: 10.1109/TCAD.2018.2883990.

[18] J. Zhou, X. S. Hu, Y. Ma, J. Sun, T. Wei, and S. Hu, "Improving availability of multicore real-time systems suffering both permanent and transient faults," *IEEE Trans. Comput.*, to be published. doi: 10.1109/TC.2019.2935042.

[19] J. Tan, N. Goswami, T. Li, and F. Xu, "Analyzing soft-error vulnerability on GPGPU microarchitecture," in *Proc. Int. Symp. Workload Characterization*, Austin, TX, USA, Nov. 2009, pp. 226–235.

[20] D. J. Palframan, N. Kim, and M. H. Lipasti, "Precision-aware soft error protection for GPUs," in *Proc. Int. Symp. High Perform. Comput. Archit.*, Orlando, FL, USA, Feb. 2014, pp. 49–59.

[21] J. Tan, Z. Li, and X. Fu, "Soft-error reliability and power co-optimization for GPGPUs register file using resistive memory," in *Proc. Design Autom. Test Europe*, Grenoble, France, Mar. 2015, pp. 369–374.

[22] H. Lee, H. Chen, and M. A. Al Faruque, "PAIS: Parallelization aware instruction scheduling for improving soft-error reliability of GPU-based systems," in *Proc. Design Autom. Test Europe*, Dresden, Germany, Mar. 2016, pp. 1568–1573.

[23] M. Namaki-Shoushtari, A. Rahimi, N. Dutt, P. Gupta, and R. Gupta, "ARGO: Aging-aware GPGPU register file allocation," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Montreal, QC, Canada, Oct. 2013, pp. 1–9.

[24] A. Rahimi, L. Benini, and R. K. Gupta, "Aging-aware compiler-directed VLIW assignment for GPGPU architectures," in *Proc. Design Autom. Conf.*, Austin, TX, USA, May 2013, pp. 16–21.

[25] H. Lee, M. Shafique, and M. A. Faruque, "Low-overhead aging-aware resource management on embedded GPUs," in *Proc. Design Autom. Conf.*, Jun. 2017, pp. 1–6.

[26] *Jetson Tegra K1*, Nvidia, Santa Clara, CA, USA, 2018. Accessed: Oct. 2018. [Online]. Available: https://developer.nvidia.com/embedded/develop/hardware

[27] *Jetson Tegra X2*, Nvidia, Santa Clara, CA, USA, 2018. Accessed: Oct. 2018. [Online]. Available: https://developer.nvidia.com/embedded/buy/jetson-tx2

[28] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in *Proc. Design Autom. Conf.*, Jun. 2012, pp. 1–6.

[29] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, "Integrated CPU–GPU power management for 3D mobile games," in *Proc. Design Autom. Conf.*, San Francisco, CA, USA, Jun. 2014, pp. 1–6.

[30] A. Prakash, H. Amrouch, M. Shafique, and J. Henkel, "Improving mobile gaming performance through cooperative CPU–GPU thermal management," in *Proc. Design Autom. Conf.*, Jun. 2016, pp. 1–6.

[31] S. Wang, G. Ananthanarayanan, and T. Mitra, "OPTiC: Optimizing collaborative CPU–GPU computing on mobile devices with thermal constraints," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 3, pp. 393–406, Mar. 2019.

[32] J. Zhou, X. S. Hu, Y. Ma, and T. Wei, "Balancing lifetime and soft-error reliability to improve system availability," in *Proc. Asia South Pac. Design Autom. Conf.*, Macau, China, Jan. 2016, pp. 685–690.

[33] Y. Ma, T. Chantem, R. P. Dick, S. Wang, and X. S. Hu, "An on-line framework for improving reliability of real-time systems on 'big-little' type MPSoCs," in *Proc. Design Autom. Test Europe*, Mar. 2017, pp. 1–6.

[34] P. Mercati *et al.*, "Multi-variable dynamic power management for the GPU subsystem," in *Proc. Design Autom. Conf.*, Jun. 2017, pp. 1–6.

[35] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, pp. 94–125, Mar. 2004.

[36] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," in *Proc. Int. Symp. Comp. Archit.*, Jun. 2004, pp. 276–287.

[37] *CUDA Samples*, Nvidia, Santa Clara, CA, USA, 2018. Accessed: Oct. 2018. [Online]. Available: http://docs.nvidia.com/cuda/cuda-samples/index.html
[38] *Rodinia: Accelerating Compute-Intensive Applications With Accelerators*, Univ. Virginia, Charlottesville, VA, USA, 2018. Accessed: Oct. 2018. [Online]. Available: https://rodinia.cs.virginia.edu
[39] J. Redmon, D. Santosh, G. Ross, and F. Ali, "You only look once: Unified, real-time object detection," in *Proc. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 779–788.
[40] S. Hare *et al.*, "Struck: Structured output tracking with kernels," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 10, pp. 550–561, Oct. 2016.
[41] *Profiler—CUDA Toolkit Documentation*, Nvidia, Santa Clara, CA, USA, 2018. Accessed: Oct. 2018. [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/index.html
[42] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar, "A server-based approach for predictable GPU access control," in *Proc. Int. Conf. Real Time Comput. Syst. Appl.*, Aug. 2017, pp. 1–10.
[43] *Mibench Version 1.0*, Elect. Eng. Comput. Sci. Dept., Univ. Michigan, Ann Arbor, MI, USA, 2018. Accessed: Oct. 2018. [Online]. Available: http://vhosts.eecs.umich.edu/mibench
[44] *Nvidia Automotive Driving Innovation*, Nvidia, Santa Clara, CA, USA, 2018. Accessed: Oct. 2018. http://www.nvidia.com/content/tegra/automotive/pdf/automotive-brochure-web.pdf
[45] T. Chantem, Y. Xiang, X. S. Hu, and R. P. Dick, "Enhancing multicore reliability through wear compensation in online assignment and scheduling," in *Proc. Design Autom. Test Europe*, Mar. 2013, pp. 1373–1378.
[46] Y. Xiang, T. Chantem, R. P. Dick, X. S. Hu, and L. Shang, "System-level reliability modeling for MPSoCs," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Oct. 2010, pp. 297–306.
[47] Y. Ma, T. Chantem, R. P. Dick, and X. S. Hu, "Improving system-level lifetime reliability of multicore soft real-time systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 6, pp. 1895–1905, Jun. 2017.
[48] *Nvidia Jetson TX2 Delivers Twice the Intelligence to the Edge*, Nvidia Develop. Blog, Santa Clara, CA, USA, 2018. Accessed: Oct. 2018. [Online]. Available: https://devblogs.nvidia.com/parallelforall/jetson-tx2-delivers-twice-intelligence-edge/

Yue Ma (S'16) received the B.S. degree from the Chengdu University of Technology, Chengdu, China, and the M.S. degree from the University of Electronic Science and Technology of China, Chengdu. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA.

His current research interests include real-time embedded systems, reliable system design, power efficiency, and temperature-aware resource management.

Junlong Zhou (S'15–M'17) received the Ph.D. degree in computer science from East China Normal University, Shanghai, China, in 2017.

He was a Visiting Scholar with the University of Notre Dame, Notre Dame, IN, USA, from 2014 to 2015. He is currently an Assistant Professor with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China. His current research interests include embedded systems and cyber physical systems.

Dr. Zhou has been an Associate Editor of the *Journal of Circuits, Systems, and Computers*, and serves as a Guest Editor for several special issues of *ACM Transactions on Cyber-Physical Systems*, *IET Cyber-Physical Systems: Theory & Applications*, and the *Journal of Systems Architecture: Embedded Software Design* (Elsevier).

Thidapat Chantem (S'05–M'11–SM'18) received the bachelor's degree from Iowa State University, Ames, IA, USA, in 2005, and the master's and Ph.D. degrees from the University of Notre Dame, Notre Dame, IN, USA, in 2011.

She is an Assistant Professor of electrical and computer engineering with Virginia Tech, Blacksburg, VA, USA. Her current research interests include real-time embedded systems, energy-aware and thermal-aware system-level design, cyber-physical system design, and intelligent transportation systems.

Robert P. Dick (S'95–M'02) received the B.S. degree from Clarkson University, Potsdam, NY, USA, in 1996 and the Ph.D. degree from Princeton University, Princeton, NJ, USA, in 2002.

He is an Associate Professor of electrical engineering and computer science with the University of Michigan, Ann Arbor, MI, USA. He was a Visiting Professor with the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2002; a Visiting Researcher with NEC Labs America, Irving, TX, USA, in 1999; and was on the faculty with Northwestern University, Evanston, IL, USA, from 2003 to 2008. He was also the CEO of the Stryd, Inc., Boulder, CO, USA, which produces wearable electronics for athletes.

Dr. Dick was a recipient of the NSF CAREER Award and his department's Best Teacher of the Year Award in 2004. In 2007, his technology won a Computerworld Horizon Award and his paper was selected as one of the 30 in a special collection of DATE papers appearing during the past 10 years. His 2010 research has won a Best Paper Award at DATE. He served as the Technical Program Committee Co-Chair of the 2011 International Conference on Hardware/Software Codesign and System Synthesis, as an Associate Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, and as a Guest Editor for the *ACM Transactions on Embedded Computing Systems* and IEEE DESIGN & TEST OF COMPUTERS.

Shige Wang (S'02–M'05–SM'11) received the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, MI, USA, in 2004.

He is a Staff Research Scientist with General Motors Research and Development, Warren, MI, USA. His current research interests include system modeling and analysis, software architecture for parallel processing in automated driving systems, and embedded real-time control systems.

Xiaobo Sharon Hu (S'85–M'89–SM'02–F'16) received the B.S. degree from Tianjin University, Tianjin, China, the M.S. degree from the Polytechnic Institute of New York, New York, NY, USA, and the Ph.D. degree from Purdue University, West Lafayette, IN, USA.

She is Professor with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA. Her current research interests include real-time embedded systems, low-power system design, and computing with emerging technologies. She has published over 250 papers in the above areas.

Prof. Hu was a recipient of the NSF CAREER Award in 1997, the Best Paper Award from Design Automation Conference, in 2001, and the IEEE Symposium on Nanoscale Architectures, in 2009. She served as an Associate Editor for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, the *ACM Transactions on Design Automation of Electronic Systems*, and the *ACM Transactions on Embedded Computing*. She is the Program Chair of 2016 Design Automation Conference (DAC) and the TPC Co-Chair of 2014 and 2015 DAC.