Reducing Memory Interference Latency of Safety-Critical Applications via Memory Request Throttling and Linux `cgroup`

Jungho Kim[a], Philkyue Shin[b], Soonhyun Noh[b], Daesik Ham[a] and Seongsoo Hong[a,b]

[a]*Department of Transdisciplinary Studies*
Seoul National University
Seoul, Republic of Korea
{jhkim, dsham, sshong}@redwood.snu.ac.kr

[b]*Department of Electrical and Computer Engineering*
Seoul National University
Seoul, Republic of Korea
{pkshin, shnoh, sshong}@redwood.snu.ac.kr

Abstract— **With the advent of high-performance multicore processors that operate under a limited power budget, dedicated low-end microprocessors with different levels of criticality are rapidly consolidated into a mixed-criticality system. One of the major challenges in designing such a mixed-criticality system is to tightly control the amount of resource contention for a critical application by effectively limiting its performance interference incurred due to sharing resources with non-critical tasks. In this paper, we propose application-aware dynamic memory request throttling to reduce the memory interference latency of a critical application in a dual criticality system. Our approach carefully differentiates critical task instances from normal task instances and groups them into the critical and normal `cgroup`, respectively. It then predicts the occurrence of excessive memory contention under critical task execution and then throttles memory requests generated by the normal `cgroup` via the `CPUFreq` governor when necessary. We have implemented the approach on the NVIDIA Jetson TX2 with Linux kernel 4.4.38. Experimental results show that the proposed approach reduces the end-to-end latency of a critical application up to 9.49% while incurring only negligible overhead.**

Keywords—memory interference, throttling, cgroup, criticality

I. Introduction

Modern multicore processors are being rapidly adopted in a wide range of mission-critical embedded systems as their advanced architecture enables them to accomplish high performance within limited power budgets [1]. In such a mission-critical embedded system, many of dedicated low-end microcontrollers, each of which used to perform a few specific tasks, are consolidated into a small number of high performance multicore processors. Such consolidated mission-critical embedded system may have multiple applications with different levels of criticality running concurrently while sharing diverse computing resources in the system.

In a mission-critical system, criticality is often referred to as functional safety though it can be in any form of dependability such as availability, reliability, and resiliency [2]. Safety-critical applications are usually subject to stringent timing constraints and thus require real-time guarantees. As a result, system developers must ensure that their safety-critical applications exhibit bounded end-to-end latency; or at least, their timing faults are monitored and reported at runtime for triggering required counteractions.

One of the representative examples of an embedded system having applications with different levels of criticality is the InfoADAS. InfoADAS consists of both an advanced driver assistance system (ADAS) and an infotainment system [3]. ADAS is a safety-critical system that actively engages in human driving to improve the safety of drivers and passengers while driving. In contrast, the infotainment system is simply a convenience device that provides information and entertainment for passengers.

One of the major challenges that system developers must address when designing such a mixed-criticality system is to effectively reduce the performance interference that occurs when applications with different level of criticality compete for shared computing resources. This problem is particularly pronounced for memory resources when data intensive applications such as deep learning are running on multicore embedded systems. This is because the memory access contention tends to get intensified between CPU cores that perform data intensive workload.

Memory access contention results from applications' simultaneous requests for memory hardware resources such as the system bus, a memory controller and memory banks. When memory access contention occurs excessively, without appropriate counter measures, applications with a higher level of criticality could be delayed resulting in their end-to-end latencies increasing unpredictably. From here on out we refer to such increase in end-to-end latency as memory interference latency.

Memory access contention already existed in a conventional system with a single CPU and a DMA controller, but as mentioned above, it became a more serious problem in multicore architecture with multiple CPU cores. The problem is particularly noticeable in a CPU-GPU heterogeneous architecture where CPU and GPU work together on a single chip sharing memory [4, 5]. Many studies have been proposed to solve the problem of delayed application execution due to memory access contention [6, 7, 8, 9, 10, 11, 12, 13, 14]. The common idea behind these approaches is to isolate shared memory hardware resources among competing applications. These approaches can be classified into spatial memory resource isolation and temporal memory resource isolation, depending on how memory resources are isolated.

Spatial memory resource isolation is a technique to prevent interference by physically partitioning memory hardware resources used by applications [6, 7, 8]. Clearly, in this technique, some memory resources may not be fully utilized and waste of resources may occur if applications are not distributed equally between the memory partitions. In addition, memory access contention cannot be completely removed if one memory partition is assigned to multiple applications that have to run concurrently.

Temporal memory resource isolation is a technique to prevent interference by distinguishing the time when applications use memory hardware resources [9, 10, 11, 12, 13, 14]. These schemes can be further divided into memory request scheduling and memory request throttling, depending on how the memory resource usage time is adjusted. Memory request scheduling determines the processing order of memory requests in an application-aware manner in a memory controller [9, 10, 11]. However, this method requires a hardware modification of the memory controller, so it cannot be applied to a commercial-off-the-shelf (COTS) system. On the other hand, memory request throttling dynamically varies the frequency of memory requests in an application-aware manner in a device that generates memory requests [12, 13, 14].

In this paper, we propose an operating system mechanism to dynamically reduce the memory interference latency of critical applications in a dual-criticality system such as the infoADAS. For our approach, we choose temporal memory resource isolation to avoid wasting utilization of memory hardware resources and, specifically, a memory request throttling technique to avoid hardware modification. We call our approach "application-aware dynamic memory request throttling." Our approach is capable of dynamically identifying and grouping the execution of critical tasks using the extended `cgroup` mechanism of the Linux kernel and predict the occurrence of excessive memory contention via counting the number of outstanding memory requests in a memory controller.

Application-aware dynamic memory request throttling consists of four steps. The first step is to dynamically identify critical task instances and to group such tasks into the critical `cgroup`. The second step is to estimate the amount of memory interference that the critical `cgroup` is experiencing by using the number of outstanding requests. The third step is to throttle down the amount of memory requests generated by the normal `cgroup` based on these estimations. The fourth step is to throttle back up the memory requests generated by the normal `cgroup`.

We seamlessly extend the `cgroup` of the Linux kernel for our dynamic memory request throttling mechanism. The `cgroup` is Linux's mechanism commonly used for organizing tasks to distribute system resources among them in a controlled manner [15]. Although memory is part of the system resources that can be controlled by the `cgroup`, the original `cgroup` cannot serve for our purpose since it can only allow users to control the amount of memory to be allocated for each task group. Thus we extend the `cgroup` core of the Linux kernel and added what we call the memory request rate controller to the `cgroup` controller.

We have implemented the proposed mechanism on the NVIDIA Jetson TX2 with Linux kernel 4.4.38 customized by Nvidia. We measured the end-to-end latency of a critical application for evaluating our approach. We also measured the run-time overhead of the mechanism. The experimental results showed that our approach reduced the end-to-end latency of a critical application up to 9.49% while incurring a run-time overhead of only 1.95%.

The remainder of this paper is organized as follows. Section II formally states our problem to solve and gives a solution overview. Section III technically treats our solution approach. Section IV reports on the experimental evaluation. Section V concludes the paper.

II. Problem Description and Solution Overview

In this section, we first describe the system model and define the problem at hand. We then present an overview of our solution approach.

A. Target Memory Architecture Model

From the perspective of memory architecture, our target system consists of four types components as shown in Fig. 1: (1) memory requesters, (2) a memory controller, (3) an external memory controller and (4) memory. A memory requester is a hardware device that generates memory requests, such as a CPU, a GPU and a DMA device. The memory controller schedules memory requests and has a request buffer for storing outstanding memory requests. The external memory controller directly reads data from or writes data into memory. Such memory architecture is commonly employed in SoCs for ordinary embedded systems [16, 17, 18, 19, 20].

In this memory architecture, a memory access is carried out in three stages. First, a memory requester sends a memory request to the memory controller and inserts it into the request buffer. Second, the memory controller reads data from or writes data into memory through the external memory controller.

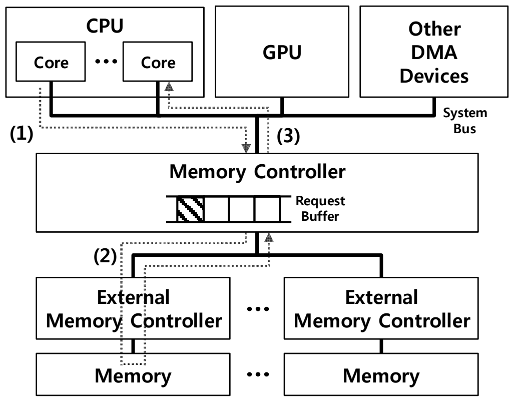

Fig. 1. Memory access on target memory architecture.

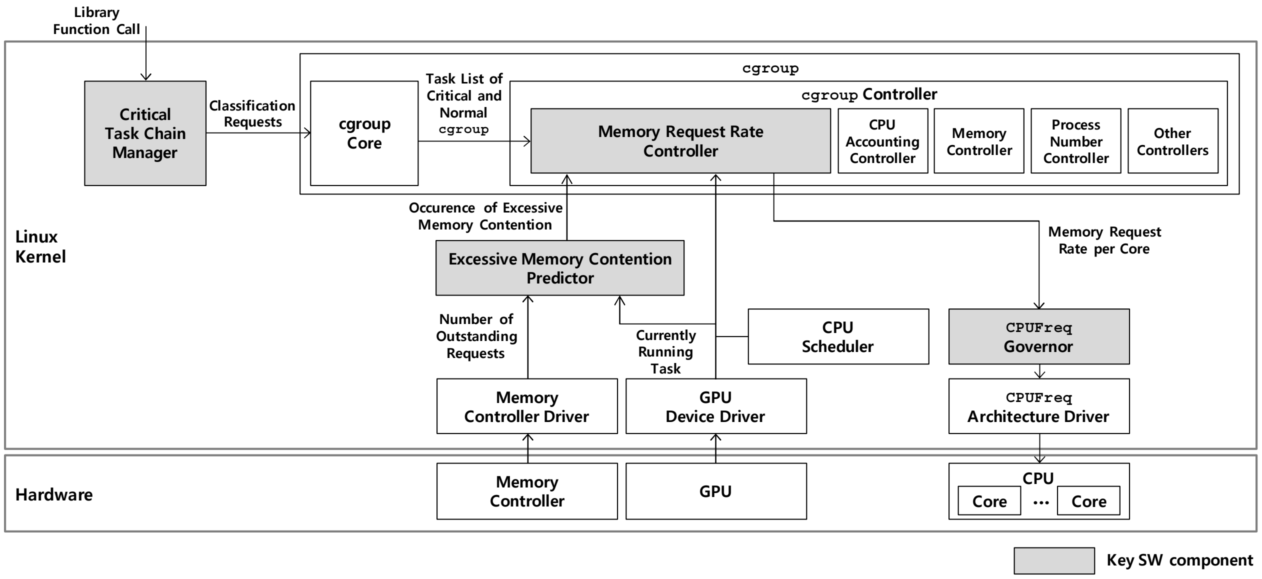
Library Function Call
Critical Task Chain Manager
Classification Requests
cgroup
cgroup Core
Task List of Critical and Normal cgroup
cgroup Controller
Memory Request Rate Controller
CPU Accounting Controller
Memory Controller
Process Number Controller
Other Controllers
Occurence of Excessive Memory Contention
Memory Request Rate per Core
Linux Kernel
Excessive Memory Contention Predictor
Number of Outstanding Requests
Currently Running Task
CPU Scheduler
CPUFreq Governor
Memory Controller Driver
GPU Device Driver
CPUFreq Architecture Driver
Hardware
Memory Controller
GPU
CPU
Core
Core
Key SW component

Fig. 2. Kernel architecture of the proposed solution approach.

Third, the memory controller removes the request from the request buffer when the memory request is completed.

B. Problem Definition

In this paper, we consider a dual-criticality system where applications with high criticality run together with applications with no or low criticality. In order to dynamically differentiate critical task instances from normal task instances, we first define the notion of a critical task chain. We base this notion on the Linux task model where an application process consists of one or more tasks. We also make use of our own two library functions which we will elaborate upon in Section IV.

Definition 1. A critical task chain is a sequence of task executions that begins with a task calling the `start-of-critical-execution` function and ends with a task calling the `end-of-critical-execution` function.

Definition 2. The end-to-end latency of a critical application is the time from when the `start-of-critical-execution` is called to when the `end-of-critical-execution` is called.

Our approach dynamically classifies all the tasks in the system into two groups, a critical task group and a normal task group, using the notion of the critical task chain. These groups are defined below.

Definition 3. Critical task group C is a set of tasks that appear on a critical task chain.

Definition 4. Normal task group N is a set of tasks that do not belong to any critical task chain in the system.

In the proposed approach, we intend to reduce the memory interference latency of each of memory requests from critical tasks to shorten the end-to-end latency of a critical application. To elaborate on our problem formulation, we define the memory interference latency of a memory request from a critical task.

Definition 5. For a given memory request r from a task $\tau \in C$, the memory interference latency (MIL) of r is defined as the difference between the memory access time of r when τ is running with other tasks in $C \cup N$ and the memory access time of r when τ is running with other tasks only in C. The MIL of r is denoted by i_r.

The goal of our problem is to reduce i_r.

C. Solution Overview

To solve the problem described above, we propose application-aware dynamic memory request throttling. The key idea behind this approach is to reduce i_r by lowering the memory request rate of normal tasks when critical tasks are executing. Clearly, we should cautiously throttle the memory request rate of normal tasks to avoid excessive performance degradation of the normal tasks. To do so, we use the number of outstanding memory requests in the request buffer as a metric to predict excessive memory contention.

Fig. 2 shows the kernel architecture of the proposed approach that contains the four actively involved components (the shaded rectangles) along with other kernel components: They are (1) the critical task chain manger, (2) the excessive memory contention predictor, (3) the memory request rate controller and (4) the `CPUFreq` governor.

The critical task chain manager keeps track of a critical task chain and generates classification requests for tasks on a critical task chain. The critical task chain manager gets activated by the `start-of-critical-execution` and `end-of-critical-execution` function calls. It is also activated by IPC and task creation system calls. On invocation, the critical task chain manager transfers classification requests to the `cgroup` core that actually manages the critical and normal `cgroup`.

We introduce the excessive memory contention predictor to the kernel. It periodically predicts the occurrence of excessive memory contention under critical task execution by identifying tasks running on both CPU and GPU and counting the number of outstanding memory requests in a memory controller.

We also add the memory request rate controller to the `cgroup` controller that is responsible for distributing a specific type of system resource among the `cgroups` in the system [15]. The memory request rate controller is activated by the periodic excessive memory contention predictor. It invokes the `CPUFreq` governor to scale up or down the frequency of each core according to its decision.

III. Application-Aware Dynamic Memory Request Throttling

The approach proposed in this paper consists of four specific mechanisms: (1) application-aware task grouping, (2) excessive memory contention prediction, (3) decreasing the memory request rate and (4) increasing the memory request rate. In this section, we give the technical details of these mechanisms.

A. *Application-aware Task Grouping*

This mechanim identifies critical tasks in the system in response to the occurrences of the four types of events that cause a change of task criticality. These events are: (1) the start of a critical task chain, (2) the extension of a critical task chain via an interprocess communication (IPC) call, (3) the extension of a critical task chain via creating a task and (4) the end of a critical task chain. These events are derived from the definition of the critical task chain of **Definition 1**. Fig. 3 pictorially shows the four events. When any of these events occurs, related library and system call functions forward it to the critical task chain manager so as to convert it into a classification request and transfer the request to the `cgroup` core. Fig. 3 also depicts interactions among those functions, the critical task chain manger and the `cgroup` core.

We first design two library functions. The `start-of-critical-execution` function begins a new critical task chain. This function activates the critical task chain manager via the `write()` system call. The critical task chain manager creates a new critical task chain inside a kernel data structure, assigns it a unique ID, and put the calling task into the critical `cgroup`. Please note that all tasks created outside a critical task chain belong to the normal `cgroup` by default.

The `end-of-critical-execution` function ends the critical task chain. This function also activates the critical task chain manager. The critical task chain manager removes the current critical task chain from the kernel data structure and checks all the tasks on the chain to see if it can move them from the critical `cgroup` to the normal `cgroup`. A task is actually moved only if the task does not appear on any critical task chain in the system.

We modify two types of system call functions, IPC and task creation. Both system calls cause the current critical task chain to branch out. We modify these functions so that the receiver task of an IPC call or the task being created can inherit the criticality of the calling task. Like the abovementioned library functions, these system call functions invoke the critical task chain manager.

B. *Excessive Memory Contention Prediction*

The excessive memory contention predictor is activated if the system has one or more critical task chains; otherwise, it is deactivated. Once activated, the excessive memory contention predictor periodically estimates MIL. We denote the estimation period by T_{MIL} and explain how T_{MIL} is empirically selected in Section IV.

The excessive memory contention predictor works in two steps on each iteration. In the first step, the excessive memory contention predictor checks if all the tasks currently running on CPU and GPU are critical or normal. For CPU tasks, we change the Linux kernel to record the criticality of a running task during context switching and let the excessive memory contention predictor read the criticality value. For GPU tasks, we make use of a technique that conservatively estimates the criticality of the task currently executing on GPU. Specifically, the excessive memory contention predictor reads in the current GPU context via accessing the GPU status register to find a list of tasks that share the GPU context. When any of the aforementioned tasks is critical, we safely assume that the current GPU task is critical as well. This technique will be explained in a greater detail in Section IV.

Second, the excessive memory contention predictor estimates MIL if critical tasks are running with normal tasks in the system. We consider that excessive memory contention occurs if the number of outstanding requests exceeds the threshold o_{thr}. We explain o_{thr} more in our experiments.

C. *Decreasing Memory Request Rate*

When the excessive memory contention predictor sees the number of outstanding requests exceed the threshold, it activates the memory request rate controller. Upon activation, the memory request rate controller determines the list of all the CPU cores that run normal task and sends the `CPUFreq` governor the list of such cores to throttle down their frequency.

D. *Increasing Memory Request Rate*

There are two cases where the frequency of a core gets throttled back up to its original frequency. First, when the excessive memory contention predictor sees the number of outstanding requests go lower than the threshold, it activates the memory request rate controller, which in turn invokes the `CPUFreq` governor to increase the core's frequency. Second, when a critical task gets dispatched to a core whose frequency

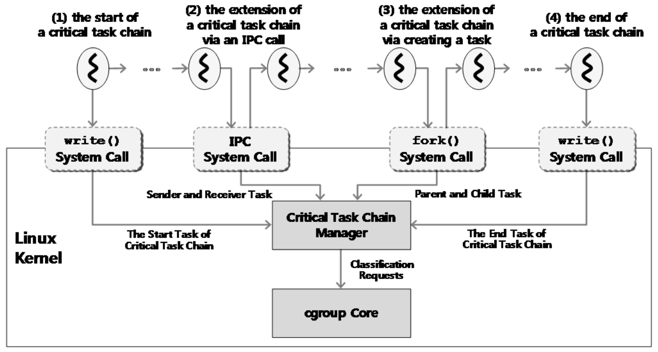
(1) the start of a critical task chain
(2) the extension of a critical task chain via an IPC call
(3) the extension of a critical task chain via creating a task
(4) the end of a critical task chain
write() System Call
IPC System Call
fork() System Call
write() System Call
Sender and Receiver Task
Parent and Child Task
Critical Task Chain Manager
The Start Task of Critical Task Chain
The End Task of Critical Task Chain
Linux Kernel
Classification Requests
cgroup Core

Fig. 3. Four events and incurred interactions in application-aware task grouping.

TABLE I. HARDWARE AND SOFTWARE CONFIGURATION OF THE TARGET SYSTEM.

HW	***CPU***	Denver CPU cluster (dual cores), A57 CPU cluster (quad cores)
	GPU	NVIDIA Pascal 256 cores
	Memory	8GB LPDDR4
SW	***GPGPU Library***	CUDA 8.0
	Framework	Ubuntu 16.04
	Kernel	Linux Kernel 4.4.38 (customized by Nvidia)

was throttled down, the Linux kernel activates the memory request rate controller to restore the core's original frequency.

IV. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

In this section, we first explain the implementation of the proposed approach. We then describe the experimental setup and show the experimental results.

A. *Implementation*

The proposed approach was implemented on Nvidia Jetson TX2. We summarize the hardware and software configuration of the target system in TABLE I. Among the key components of our approach, the excessive memory contention predictor and the memory request rate controller deserve further technical explanation since they have implementational issues.

When we implemented the excessive memory contention predictor, we had to figure out the criticality of a task that is currently running on the GPU. Unfortunately, the Nvidia GPU does not offer a means for one to single out the task currently occupying the GPU. To overcome this limitation, we made the excessive memory contention predictor identify a group of tasks that shared the current GPU context. Specifically, our approach first reads in the current GPU context ID using the function `gr_fecs_current_ctx_r()`. It then identifies the task group sharing the current GPU context by accessing the mapping table between GPU context IDs and task groups. It accesses the mapping table using the function `gk20a_fecs_trace_find_pid()`. The Nvidia GPU driver provides these functions.

When implementing the memory request rate controller, we considered other hardware limitations of TX2 as well. In TX2, the CPU frequency cannot be adjusted in individual core units but can only be in cluster units. Therefore, we were able to throttle the memory request rate of the cluster only when normal tasks executed on all cores within the cluster.

Our approach relies on the assumption that critical tasks are rarely preempted by normal tasks. We need this assumption to avoid an unwanted delay that can occur when a critical task gets preempted by a normal task that will be in turn throttled shortly. To realize this assumption, we assigned critical tasks much higher weights than normal tasks by taking advantage of the behavioral pattern of the weight-based scheduling in Linux's completely fair scheduler [22, 23, 24].

B. *Experimental Setup*

We implemented the proposed approach in the hardware and software configuration mentioned in the previous section. The tunable parameters T_{MIL} and o_{thr} were set to 10 milliseconds and 60, respectively. We empirically selected the value for T_{MIL} since the run-time overhead had rapidly increased when T_{MIL} was less than 10 milliseconds. We also empirically selected the value for o_{thr} since the performance degradation of a normal application had rapidly increased when o_{thr} was less than 60.

We have conducted two different experiments. In the first experiment, we measured the end-to-end latency of a critical application for evaluating our approach. In the second experiment, we measured the run-time overhead of the proposed approach. In both experiments, we used the same workload consisting of a critical application and a normal application. As a critical application, we selected an object detection application running YOLO. This application has been widely used in academia as a research benchmark [21]. As a normal application, we used a synthetic application which generates memory requests based on θ where θ is the relative memory request rate whose value is between 0 to 1.

To clearly demonstrate the effectiveness of the proposed approach, we used the `performance` governor that runs the system at the top operating frequency whenever possible.

C. *Experimental Results*

In our first experiment, we simultaneously ran a critical application and a normal application and measured the end-to-end latency of the critical application. As mentioned above, we selected an object detection application running YOLO as a critical application. We performed experiments on different YOLO models: Tiny YOLO, YOLOv2 608x608 and YOLOv3-320. These YOLO models have different neural network configurations such as the number of layers and the number of neurons in a single layer. We also varied θ of the synthetic application from 0.2 to 1.0.

TABLE II shows the results of our first experiment. As expected, the end-to-end latency of the critical application is always reduced with our approach. In addition, we observed that the speedup of a critical application increased as θ increased.

In our second experiment, we measured the run-time overhead with the same workload used in the first experiment. In this experiment, θ was set to 1.0 to obtain the maximum

TABLE II. COMPARISON OF END-TO-END LATENCY OF CRITICAL APPLICATION WITHOUT AND WITH THE PROPOSED APPROACH.

	Relative Memory Request Rate (θ)	**Critical Application**								
		Tiny YOLO			***YOLOv2 608x608***			***YOLOv3-320***		
		Original (ms)	***Modified (ms)***	***Speedup (%)***	***Original (ms)***	***Modified (ms)***	***Speedup (%)***	***Original (ms)***	***Modified (ms)***	***Speedup (%)***
End-to-end Latency of Critical Application	0.2	55.39	55.31	0.15	142.18	141.38	0.57	312.17	309.60	0.83
	0.4	58.73	58.04	1.19	149.85	145.56	2.95	325.38	317.12	2.60
	0.6	60.13	59.04	1.84	153.61	148.59	3.38	332.59	322.93	2.99
	0.8	62.89	59.38	5.91	160.26	148.81	7.69	344.43	322.93	6.66
	1.0	64.29	59.86	7.42	163.67	149.48	9.49	350.88	325.73	7.72

TABLE III. INCURRED RUN-TIME OVERHEAD BY THE PROPOSED APPROACH.

	Critical Application		
	Tiny YOLO	***YOLOv2 608x608***	***YOLOv3-320***
Runtime Overhead (%)	1.95	1.31	1.33

runtime overhead under excessive memory contention. We measured the time spent to execute the added code implementing the proposed approach. TABLE III shows the measurement results. The proposed approach incurred the maximum run-time overhead by only 1.95%.

V. CONCLUSION

In this paper, we proposed application-aware dynamic memory request throttling to reduce MIL of a critical application in a dual criticality system. It differentiates critical task instances from normal task instances and groups them into the critical and normal `cgroup`, respectively. It then predicts the occurrence of excessive memory contention under critical task execution and then throttles memory requests generated by the normal `cgroup` via the `CPUFreq` governor when necessary.

We implemented the proposed mechanism in Nvidia Jetson TX2 with customized Linux kernel 4.4.38. The experimental results showed that our mechanism reduced the end-to-end latency of a critical application up to 9.49%. In addition, our approach incurred a run-time overhead of only 1.95%.

There are future research directions along which the proposed mechanism can be extended. We aim to refine our memory request throttling policy such that the mechanism can allow for an incremental change in the CPU frequency to achieve the finer manipulation of memory request rates. This will clearly lead to further reduction of the end-to-end latency of a critical application. We also plan to extend our approach to throttle memory requests from GPUs, which is in fact one of the hard challenges in terms of resource isolation due to the nonpreemptive nature of GPU architecture.

ACKNOWLEDGEMENT

This research was supported by the MSIT (Ministry of Science and ICT) Korea under the ITRC (Information Technology Research Center) support program (IITP-2018-2013-1-00717) supervised by the IITP (Institute for Information & communications Technology Promotion), System Industrial Strategic Technology Development Program (10079961, "Development of a deterministic DCU platform with less than 1μs synchronization for autonomous driving system control) funded by the Ministry of Trade, Industry & Energy (MOTIE, Korea), and Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. R7117-16-0164, Development of wide area driving environment awareness and cooperative driving technology which are based on V2X wireless communication).

REFERENCES

[1] Brock Bishop, and Karthick Rajamani, "Dynamic Power Management for Embedded Systems [SOC Design]", Proceedings of the IEEE International System on Chip Conference, pp. 416-419, 2003

[2] Ernst Rolf, and Marco Di Natale, "Mixed Criticality Systems—A History of Misconceptions?", IEEE Design & Test 33.5, pp65-74, 2016

[3] Texas Instruments's Cyril Clocher, "You ask for more, TI's Automotive Processors team delivers", 2015

[4] Cavicchioli Roberto, Nicola Capodieci, and Marko Bertogna, "Memory Interference Characterization between CPU cores and integrated GPUs in Mixed-Criticality Platforms", 22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA), 2017

[5] Wen Hao, and Wei Zhang, "Interference Evaluation In CPU-GPU Heterogeneous Computing", IEEE High Performance Extreme Computing Conference (HPEC), 2017

[6] Muralidhara Sai Prashanth, et al., "Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning", Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011

[7] Liu Lei, et al., "A Software Memory Partition Approach for Eliminating Bank-Level Interference in Multicore Systems", IEEE International Conference on Parallel Architectures and Compilation (PACT), 2012

[8] Lee Donghyuk, et al., "Decoupled direct memory access: Isolating CPU and IO traffic by leveraging a dual-data-port DRAM", IEEE International Conference on Parallel Architecture and Compilation (PACT), 2015

[9] Mutlu Onur and Thomas Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors", Proceedings of Annual IEEE/ACM international Symposium on Microarchitecture (MICRO), 2007

[10] Usui Hiroyuki, et al., "DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators", Transactions on Architecture and Code Optimization, 2016

[11] Rai Siddharth, et al., "Using criticality of GPU accesses in memory management for CPU-GPU heterogeneous multi-core processors", ACM Transactions on Embedded Computing Systems (TECS), 2017

[12] Herdrich Andrew, et al., "Rate-based QoS techniques for cache/memory in CMP platforms", Proceedings of the 23th ACM International Conference on Supercomputing (ISC), 2009

[13] Yun Heechul, et al., "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms", Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013

[14] Yun Heechul, et al., "BWLOCK: A dynamic memory access control framework for soft real-time applications on multicore platforms", IEEE Transactions on Computers 66.7, 2017

[15] Heo Tejun, "Control group v2", 2015

[16] Intel datasheet, "Intel Atom® Processor E3900 and A3900 Series", 2017

[17] Kyriazis George, "Heterogeneous system architecture: A technical review", AMD Fusion Developer Summit, 2012

[18] Nvidia technical reference manual, "NVIDIA Parker Series SoC", 2017

[19] ARM technical reference manual, "ARM® CoreLink™ DMC-620 Dynamic Memory Controller, Revision: r0p0", 2016

[20] Codrescu Lucian, et al., "Hexagon DSP: An architecture optimized for mobile multimedia and communications", Proceedings of the IEEE International Symposium on Microarchitecture (MICRO), 2014

[21] Redmon Joseph, et al., "You Only Look Once: Unified, Real-Time Object Detection", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016

[22] Sungju Huh, Jonghun Yoo, Myungsun Kim and Seongsoo Hong, "Providing Fair Share Scheduling on Multicore Cloud Servers via Virtual Runtime-based Task Migration Algorithm", Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 606-614, Jun 2012

[23] Sungju Huh, Jonghun Yoo and Seongsoo Hong, "Improving Interactivity via VT-CFS and Framework-assisted Task Characterization for Linux/Android Smartphones", Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pp. 250-259, Aug 2012

[24] Sungju Huh, Jonghun Yoo and Seongsoo Hong, "Cross-layer Resource Control and Scheduling for Improving Interactivity in Android", International Journal of Software: Practice and Experience, 2015