Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.Doi Number

Memory-Aware Fair-Share Scheduling for Improved Performance Isolation in the Linux Kernel

Jungho Kim[1], Philkyue Shin[2], Myungsun Kim[3] and Seongsoo Hong[1,2]
[1]Department of Transdisciplinary Studies, Seoul National University, Seoul 08826, South Korea
[2]Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826, South Korea
[3]Electrical Division of IT Convergence Engineering, Hansung University, Seoul 02864, South Korea

Corresponding author: Seongsoo Hong (sshong@redwood.snu.ac.kr)

This work is supported by the Samsung Electronics' university R&D program.

ABSTRACT Performance interference between QoS and best-effort applications is getting more aggravated as data-intensive applications are rapidly and widely spreading in recently emerging computing systems. While the completely fair scheduler (CFS) of the Linux kernel has been extensively used to support performance isolation in a multitasking environment, it falls short of addressing memory-related interference due to memory access contention and insufficient cache coverage. Though quite a few memory-aware performance isolation mechanisms have been proposed in the literature, many of them rely on hardware-based solutions, inflexible resource management or ineffective execution throttling, which makes it difficult for them to be used in widely deployed operating systems like Linux running on a COTS SoC platform. We propose a memory-aware fair-share scheduling algorithm that can make QoS applications less susceptible to memory-related interference from other co-running applications. Our algorithm carefully separates the genuine memory-related stall from a running task's CPU cycles and compensates the task for the memory-related interference so that the task gets the desired share of CPU before it is too late. The proposed approach is adaptive, effective and efficient in the sense that it does not rely on any static allocation or partitioning of memory hardware resources and improves the performance of QoS applications with only a negligible runtime overhead. Moreover, it is a software-only solution that can be easily integrated into the kernel scheduler with only minimal modification to the kernel. We implement our algorithm into the CFS of Linux and name the end result mCFS. We show the utility and effectiveness of the approach via extensive experiments.

INDEX TERMS Memory-related interference, backend stall cycle, operating system, Linux, CFS

I. INTRODUCTION

Data-intensive applications, most noticeably deep learning-based applications, are rapidly and widely spreading in recently emerging computing systems. Services provided by such applications are often human-perceivable and subject to quality-of-service (QoS) requirements. As such, system developers are tasked with ensuring sufficient computing performance for their applications within limited cost, size, weight and power budgets of the underlying system [1][2].

Between the QoS and best-effort applications, there exists unavoidable performance interference since they share various computing resources in the system. Such interference to QoS applications is malicious since it can indefinitely increase their response time and thus prevent them from meeting the imposed QoS requirements. Diverse performance isolation techniques have been proposed in the literature and then widely used as a viable weapon against QoS applications' performance degradation.

To address the need for performance isolation in extensive use cases of industry, Linux has offered the completely fair scheduler (CFS) since its 2.6.23 kernel release [3][4][5][6]. The CFS has been successfully exploited as a fair-share scheduler in numerous Linux installations ranging from huge datacenter servers to desktops and to small handheld devices such as smartphones. Despite such a significant contribution of CFS over a decade, the Linux kernel has started to show its limitations in dealing with recently emerging application workloads that generate massive memory traffic. Our approach is motivated by such limitations of the Linux kernel.

While the CFS is capable of fairly distributing CPU cycles among running tasks proportionally to the tasks' weights, it cannot take into account interference that the running tasks experience due to memory contention and insufficient cache coverage. This is because CFS assumes that absolute physical performance achieved by a task is proportional to the number of CPU cycles allocated to the task. It thus simply attempts to equalize the weighted performance of runnable tasks in the system. In the presence of memory access contention and cache misses, however, the execution of a task may stall and waste CPU cycles for nothing. Unless the kernel scheduler takes into account such stall cycles, the underlying runtime system cannot provide exact performance isolation for QoS applications.

In this paper, we propose a memory-aware fair-share scheduling algorithm that can make QoS applications less susceptible to memory-related interference from other co-running applications in the system. We also seamlessly integrate the algorithm into CFS with minimal modification to the Linux kernel. We name the end result *memory-aware CFS* (*mCFS*).

Memory-aware performance isolation is a difficult problem to formulate since modern microarchitecture has become too complex to be fully analyzed. Moreover, it is tricky to accurately measure the amount of genuine memory-related stall that a given application experiences. Observe that an application's memory-related CPU stall is ascribed to not only memory-related interference from other applications but also the idiosyncrasy of the application's code itself. For instance, an application demonstrating a sequential data access pattern may incur many cache misses during execution, even without cache contention. In this case, it is fair to say that the performance isolation mechanism should not compensate the application for such intrinsic CPU stall. As such, a performance isolation algorithm must be able to distinguish between the genuine memory-related stall and the intrinsic stall.

To compute the amount of the genuine memory-related stall, mCFS uses a runtime formula we derive via qualitative analysis of the underlying microarchitecture and quantitative analysis of the execution of diverse applications. In this formula, we model the genuine memory-related stall using easily measurable entities such as stall cycles at the backend of the pipeline of the underlying microarchitecture. We duly note that the backend stall is caused by cache misses and memory access contention as well as data dependencies and internal resource contention between micro-operations inside the pipeline. To single out the genuine memory-related stall from the entire backend stall, the formula subtracts the estimated intrinsic stall from the measured backend stall.

To estimate the amount of intrinsic memory-related stall, we introduce the average intrinsic backend stall rate (IBSR) of a given application. The average IBSR is a rate of the backend stall cycles of an application over a long period of time when the application is running alone in an isolated manner. It is a characteristic value that represents the memory access behavior of a given application. The average IBSR can be computed offline on a per-application basis. A backend stall cycle count can be easily measured via a performance monitoring unit commonly provided by modern SoCs.

For a given time interval $[t_1, t_2]$, the formula for computing the genuine memory-related stall cycle count $b_i^m(t_1, t_2)$ is given as follows where $b_i(t_1, t_2)$ is the backend stall cycle count:

$$b_i^m(t_1, t_2) = b_i(t_1, t_2) - average\_IBSR \cdot (t_2 - t_1)$$

We experimentally validate the formula in Sections IV and VI.

The crux of mCFS lies in "*memory-aware virtual runtime calculation*" for the task scheduler. The virtual runtime of a task is defined as the task's cumulative CPU time inversely scaled by its weight. In CFS, tasks at the runqueue of a CPU core compete for the core and eventually the task with the smallest virtual runtime wins it. When computing the virtual runtime of a running task, the original CFS considers the CPU time that the task physically used, without considering the CPU stall time. Since such a notion of virtual runtime cannot capture the memory-related interference that a task receives, we propose to redefine a task's virtual runtime in a memory-aware manner and apply the new notion to the CFS.

As the first step in memory-aware virtual runtime calculation, mCFS performs a computation we name *CPU time actualization*. In this step, the genuine memory-related stall time of a task is deducted from the task's physical CPU time. Since a task is always given actualized CPU time no greater than the original CPU time, the task is made to run more frequently by mCFS until it receives a sufficient amount of actualized CPU time.

In the next step, mCFS scales the actualized CPU time according to the relative performance of the core hosting the task. This step is needed to take into account the dynamic voltage and frequency scaling (DVFS) of modern SoCs [7][8]. Since DVFS changes the operating frequencies of cores at runtime, a task would demonstrate performance variability without the performance scaling of this step.

In the final step, mCFS computes virtual runtime from the actualized scaled CPU time derived in the previous steps. To do so, mCFS divides a task's actualized scaled CPU time by the task's weight.

The benefits of mCFS are three-fold. First, mCFS is *adaptive*. Since it does not rely on static memory resource allocation or partitioning for performance isolation, it can adaptively react to changes in resource demands without wasting valuable resources. Second, mCFS is *effective*. Our experiment demonstrates that mCFS achieves less slowdown or more performance improvement for the YOLO face detection application by up to 67% than the conventional CFS, depending on the mix of the co-running applications. It yielded a similar performance enhancement with benchmark programs as well. Third, mCFS is *efficient* since it does not employ any costly runtime mechanisms such as CPU idling or request throttling. It incurs only a negligible runtime overhead of 0.091%.

We have implemented the proposed approach into Linux kernel 4.9.108 on top of the NVIDIA Jetson AGX Xavier platform. Since the Xavier series SoC is particularly designed for performance-hogging, embedded deep learning-based applications, performance isolation is one of the highly desired features of the underlying runtime system. We show the effectiveness of mCFS through extensive experiments and measurements with SPEC 2017 benchmark suites. We make the source code for the mCFS kernel patch, the workload generators and the launcher command as well as running scripts publicly available so that anybody can evaluate or use mCFS freely [9].

The remainder of the paper is organized as follows. Section II surveys existing approaches to memory-aware performance isolation on multicore architecture and compares the representative techniques with mCFS. Section III provides the readers with the technical background of mCFS to help them in understanding the proposed approach. Section IV defines a measure to estimate the genuine memory-related stall and validates its effectiveness. Section V formally states our problem to solve and then defines the notion of memory-aware fairness for multicore architecture. Section VI explains the proposed solution approach along with its kernel-level implementation. Section VII reports on the experimental evaluation. Finally, Section VIII concludes this paper.

II. RELATED WORK

The performance interference problem in a computing system has been particularly pronounced for memory resources as more and more data-intensive applications are running on a heterogeneous multicore system possessing graphics processing units (GPU) and neural processing units (NPUs) [10][11]. Many studies have been conducted to improve the performance of latency-sensitive applications and/or to reduce their performance variability in the presence of memory-related interference. The existing 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 performance interference among multiple co-running applications by physically partitioning memory hardware resources used by the applications. Application-aware memory channel partitioning in [12] mitigates the problem of memory access interference by mapping to different memory channels the data of applications that are likely to severely interfere with each other. Similarly, the approach in [13] allocates a specific subset of DRAM banks to a core using a bank-level partition mechanism based on page-coloring supported by the underlying operating system. Cache partitioning is one of the most representative techniques for spatial performance isolation. Kpart in [14] is a hybrid cache partitioning and sharing technique that offsets the ineffectiveness of the well-known utility-based way-partitioning.

Clearly, spatial memory resource isolation is a preventive measure to lower memory-related interference in a predictive manner but it tends to incur frequent resource over-provisioning. Some memory resources may not be fully utilized and even wasted if applications are not distributed adaptively and fairly among the resource partitions. Besides, memory resource contention cannot be completely removed if a 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 cache and memory hardware resources. This technique can be further divided into prevention-based and compensation-based, depending on whether to avoid or allow memory-related interference in advance.

In *prevention-based temporal memory resource isolation*, the cores that are expected to incur an unfair amount of memory resource contention get restricted to run tasks. The rate-based approach in [15] throttles down the processing rate of a core if it is running a low-priority task and its execution is interfering with a high priority task through memory resource contention. It considers clock modulation and frequency scaling as a rate throttling mechanism. MemGuard [16] tries to make the average memory access latency of a task no larger than when running on a dedicated memory system that processes memory requests at a given service rate. To do so, it assigns a specific memory access quota to each core in every regulation period and restrains the core having depleted its quota from running for the rest of the period. The approach in [17] estimates the amount of memory access interference that the critical task group is experiencing, by using the number of outstanding requests in the request buffer at the memory controller. Based on such estimation, it throttles down or up memory requests generated by the normal task group.

Like the spatial memory resource isolation, prevention-based temporal memory resource isolation is a preventive technique. Unless the unfairness is measured correctly and updated adaptively, it can restrict tasks unnecessarily. Also, it may deteriorate the overall system performance when memory request throttling is used for the rate control.

In *compensation-based temporal memory resource isolation*, the tasks that failed to make a desired amount of progress get compensated for their tardiness. The source throttling technique in [18] estimates unfairness in the shared memory system using a task's slowdown and throttles down cores causing interference to the most slowed down core by limiting the number of requests they can inject into the system and the frequency at which they do. The fair-progress process scheduling technique in [19] forces the equally weighted tasks to have the same amount of slowdown when they run concurrently. It uses the same measure for a task's slowdown as the source throttling technique [18] to monitor the progress of all tasks at runtime. It allocates more CPU time to the task that experienced a slowdown. The cache-fair algorithm in [20] increases a task's CPU time slice if the task executes fewer instructions per cycle than it would under fair cache allocation. The algorithm claims that two tasks are cache-friendly if they

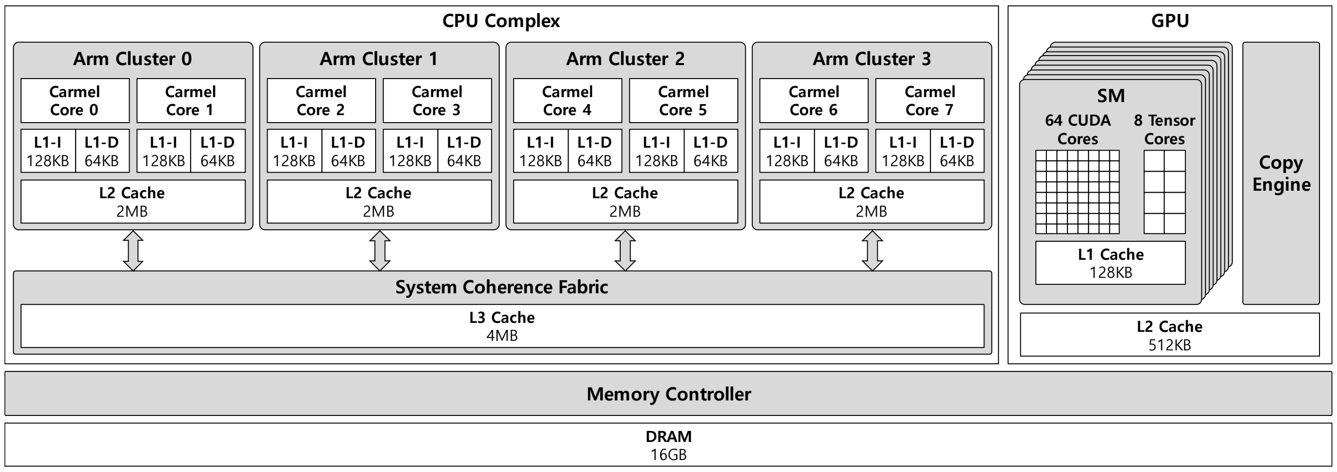


FIGURE 1. CPU complex and GPU of NVIDIA Jetson AGX Xavier platform.

experience similar miss rates when running together. It estimates a task's fair miss rate and derives its fair cache allocation by fitting experimental data into a linear approximation function. Dike [21] is a contention-aware scheduler for heterogeneous multicore systems. It divides execution time into fixed-length quanta. Dike measures the memory access rate of every task during every quantum and then predicts the potential effects of migrating tasks onto different cores. Dike attempts to achieve fairness among the tasks by moving them back and forth from the maximum frequency core to the minimum frequency core.

Our approach belongs to compensation-based temporal memory resource isolation. In designing mCFS, we ruled out the other two techniques due to their drawbacks we analyzed above. The approaches in [20] and [21] come closest to our approach. The cache-fair scheduler [20] compensates a tardy task by extending its time slice but the notion of cache-fairness does not scale up for a system having performance-asymmetric multicore architecture. Dike [21] makes use of task migration between cores to balance tasks' progress whereas mCFS relies on task scheduling. Unfortunately, task migration can be very costly when it incurs cache line refills. Dike addresses the performance asymmetry of a multicore system like mCFS but it works with only two frequency levels.

III. BACKGROUND

The approach proposed in this paper is designed for and implemented into the Linux kernel running on top of the NVIDIA Jetson AGX Xavier platform [22]. We explain the architectural elements and performance monitoring unit of the Xavier series SoC and the CFS of Linux.

TABLE I
SELECTED PMUv3 PER-CORE PMU EVENTS

Event Number	Event mnemonic	Description
0x01	`L1I_Cache_REFILL`	Level-1 instruction cache refill
0x03	`L1D_Cache_REFILL`	Level-1 data cache refill
0x04	`L1D_Cache`	Level-1 data cache access
0x08	`INST_RETIRED`	Instruction architecturally executed
0x11	`CPU_CYCLES`	CPU cycle
0x14	`L1I_Cache`	Level-1 instruction cache access
0x23	`STALL_FRONTEND`	No operation issued due to the frontend
0x24	`STALL_BACKEND`	No operation issued due to the backend

A. NVIDIA JETSON AGX XAVIER PLATFORM

The NVIDIA Xavier series SoC includes a CPU complex and a GPU as shown in Figure 1. The CPU complex and GPU share only the main memory. The Xavier series SoC provides performance monitoring units (PMU), which are classified into per-core PMUs and uncore PMUs. Table I lists the selected per-core PMU events [22][23]. Using them, programmers can effectively measure valuable performance metrics of the underlying SoC platform including the number of instructions per cycle (IPC), the number of cache misses and the frontend and the backend stall cycle count.

Our approach uses only the two events, `CPU_CYCLES` and `STALL_BACKEND` among PMU events in Table I. Thus, it can be easily implemented on any micro architectures including various Arm-based SoCs and Intel processors that provide the same or similar performance counters.

B. CLASSIFICATION OF CPU CYCLES

During the execution of a program, CPU often wastes a nontrivial number of valuable CPU cycles without performing any useful work [24]. Such a CPU cycle is referred to as either a frontend stall cycle or a backend stall cycle. They are named after the frontend and backend of the pipeline in modern superscalar, out-of-order microarchitecture such as the Arm v8.2 Carmel CPU core [22] depicted in Figure 2.

The frontend consists of an instruction fetcher and an instruction decoder. Obviously, the frontend fetches instructions and decodes them into a series of micro-operations to issue them to the backend. The frontend utilizes the branch predictor, instruction cache (I-cache) and instruction TLB (I-TLB) to expedite feeding micro-operations to the backend.

The backend includes a micro-operation scheduler, various execution units, register files, a data cache (D-cache) and a data TLB (D-TLB). The backend schedules micro-operations buffered at the issued micro-operation queue and executes them in an order consistent with data and control

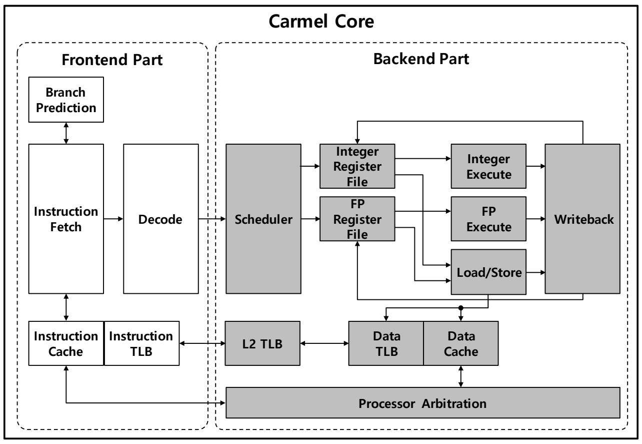


FIGURE 2. **Frontend and backend of the pipeline in the Carmel core.**

dependencies derived from the original code. It finally writes the results back to the register files and D-cache [24].

I-cache misses, I-TLB misses and branch misprediction can make CPU stall at the beginning of the pipeline. This phenomenon is called frontend stall. Similarly, lack of required resources, such as execution units and memory resources inside the pipeline prevents issued micro-operations from retiring. This is referred to as backend stall.

Unlike the frontend and backend stall cycles, a running task is spending useful cycles where micro-operations are retired from the pipeline. Such a cycle is called a retired instruction cycle. The frontend stall counts every CPU cycle on which no micro-operation is issued since there are no micro-operations available to issue from the frontend [23]. The backend stall counts each CPU cycle on which no micro-operation is issued since the backend is unable to retire any micro-operations [23].

C. CFS OF THE LINUX KERNEL

The CFS is the primary task scheduler of the Linux kernel since its 2.6.23 release. CFS distinguishes itself from its predecessors in that its objective is achieving fairness among multiple runnable tasks. In Linux, the fairness of a task scheduler is defined by the property that the physical execution times of tasks should be proportional to their weight values [3][4][25]. As such, the weight of a task is an important task attribute for CFS. In order to allow users to specify a weight value for each task in a way consistent with conventional Linux kernels, CFS makes use of nice values. In conventional Linux, nice values were used to denote task priorities. In CFS, a nice value is mapped to a specific weight value. Nice values range over [-20, 19] where a smaller value corresponds to a larger weight.

As a measure of fairness among tasks, CFS introduces a notion of virtual runtime. The virtual runtime of a task is defined as the task's cumulative runtime inversely scaled by its weight. If virtual runtimes are the same among all the tasks at a given point in time, then the tasks are given the exactly fair amount of CPU time at that time. Clearly, CFS tries to schedule runnable tasks such that they have virtual runtimes only with small differences.

Let w_0 be the weight value of nice value 0 and w_i be the weight value of task τ_i. Suppose $p_i(t)$ denotes the amount of the cumulative physical runtime of task τ_i at time *t*. In CFS, the virtual runtime $v_i(t)$ of task τ_i at time t is defined as bellow:

$$v_i(t) = \frac{w_0}{w_i} \times p_i(t)$$

The smaller a task's virtual runtime is, the more the task needs to be scheduled.

In order to enforce fair-share scheduling at a reasonable run-time cost, CFS also makes use of the notion of a time slice. A time slice is associated with a task and defined as a time interval for which the task is allowed to run without being preempted. In CFS, the length of a task's time slice is proportional to its weight. The time slice s_i of a task τ_i is computed by

$$s_i = \frac{w_0}{\sum_{\tau_j \in R} w_j} \times P$$

where R is the set of runnable tasks, w_i is the weight of τ_i and P is the constant for a given workload. In Linux, P is given as below:

$$P = \begin{cases} sysctl\_sched\_latency & \text{if } n < nr\_latency, \\ min\_granularity \times n & \text{otherwise} \end{cases}$$

where n is the number of tasks and $sysctl\_sched\_latency$, $nr\_latency$ and $min\_granularity$ are system-wide constants whose values are 6, 8 and 0.75, respectively, in the current Linux implementation.

CFS is a symmetric multiprocessor scheduling algorithm with a distributed runqueue structure. The Linux kernel maintains a dedicated runqueue for each core and lets a CFS instance of a core make scheduling decisions independently of each other. A runqueue for a core keeps a list of its runnable tasks. These tasks are sorted according to the non-decreasing order of their virtual runtimes. When the currently running task runs out of it time slice, CFS selects the first task in the runqueue and dispatches it for execution.

IV. DEFINING AND VALIDATING A MEASURE FOR MEMORY-RELATED INTERFERENCE

It is a challenging mission to come up with a memory-aware fair-share scheduling framework that is practically implementable in a kernel. In fact, it is often tricky to quantify an accurate measure for memory-related interference, particularly in a multitasking environment on a complex multicore processor. Even if we have a well-defined measure, it is sometimes infeasible to instrument it due to the limited performance monitoring capabilities of an underlying SoC.

To overcome these hurdles in our approach, we define a measure that is calculable with only existing PMU hardware at a low runtime cost. To come up with the measure, we qualitatively analyze backend stall cycles appearing in the modern CPU architecture, particularly in the Xavier series SoC. We then justify the proposed measure in a quantitative manner via experiments.

A. CLASSIFYING BACKEND STALL CYCLES AND DEFINING THE MEASURE

As described in the previous section, the backend of a pipeline in the modern SoC architecture executes arithmetic micro-operations with either the integer execute unit or the FP execute unit. It also processes memory micro-operations using the load/store unit that exploits the memory hierarchy consisting of the L1, L2, L3 cache and memory. The situation where the backend has to wait for nothing can be divided into four cases depending on the reason:

1) The backend has to wait for the arithmetic unit required by the current micro-operation to be released if the unit is being occupied by another micro-operation. A long latency divide micro-operation tends to cause such serialization. Stall in this case usually lasts only for a couple of CPU cycles [24].
2) It has to wait until operands stored in registers become available in the presence of data dependency. Stall in this case is short in time as well, only for a few CPU cycles [24].
3) It has to wait for D-cache misses to be dealt with even if the task is running alone in the system. Stall in this case lasts for a nontrivial number of CPU cycles due to a huge gap between CPU cycle time and memory access time [24].
4) It has to wait for the handling of additional D-cache misses that arise since the running task is sharing the cache with other co-running tasks in the system. When severe memory access contention among the cores is coupled with excessive D-cache misses, stall in this case can be significantly increased.

Backend stall belonging to the first three cases results from intra-task attributes such as the data dependencies and intrinsic cache behavior of the task's code. We refer to such backend stall as *intrinsic backend stall*. Backend stall belonging to the last case occurs due to memory-related interference among multiple co-running tasks in the system. We call such backend stall *genuine memory-related backend stall*.

By definition, the whole backend stall is the sum of intrinsic and genuine memory-related backend stall. We formally define them as follows:

Definition 1. Let $b_i^*(t_1, t_2)$ and $b_i^m(t_1, t_2)$ be the intrinsic and genuine memory-related backend stall cycle counts of a task τ_i in a time interval $[t_1, t_2]$, respectively. By definition, the total backend stall cycle count of τ_i in $[t_1, t_2]$ is as follows:

$$b_i(t_1, t_2) = b_i^*(t_1, t_2) + b_i^m(t_1, t_2)$$

In our approach, we propose to use $b_i^m(t_1, t_2)$ as a measure of the memory-related interference of τ_i in $[t_1, t_2]$. In what follows, we show that $b_i^m(t_1, t_2)$ serves correctly in our memory-aware fair-share scheduling.

B. EXPERIMENTAL VALIDATION OF THE MEASURE

We validate the effectiveness of the proposed measure through experiments with highly memory-intensive benchmark programs selected from the 43 benchmarks of SPEC CPU2017 [26][27]. They are `619.lbm_s`, `620.omnetpp_s` and `623.xalancbmk_s`. We ran the benchmarks on the NVIDIA Jetson AGX Xavier platform. Table II gives the detailed specification of the hardware and the system software that constitute the experimental platform. During the experiments, we cautiously controlled factors that might affect the memory access patterns. For instance, we fixed the operating frequencies of the CPU, GPU and EMC (external memory controller) throughout the experiments.

The test variable of the experiments was the amount of memory-related interference and the evaluation metric was the genuine memory-related backend stall cycle count during the entire execution of each benchmark. The amount of memory-related interference is affected by the number of cache misses and the memory access time in handling each cache miss. As such, the memory-related interference was generated by the workload consisting of a cache contention generator and a memory access contention generator.

The cache contention generator is a simple C program that is designed to incur cache line refill for each load/store instruction. We controlled the amount of cache contention by stuffing the different number of arithmetic instructions between two consecutive bundles of load/store instructions. We made five test instances: no cache contention generator running and four cache contention generator instances with ratios of load/store instructions and arithmetic instructions being 1:15, 1:10, 1:5 and 1:1, respectively.

The memory access contention generator is a multithreaded CUDA program. Each CUDA thread repeatedly performed memory reads of 8MB and memory writes of 4MB in turn. We varied the number of memory requests by changing the number of CUDA threads.

To create a discrete value space for the test variable that ranges over various amounts of memory-related interference,

TABLE II
SPECIFICATION OF NVIDIA JETSON AGX XAVIER

Classification		Description
HW	CPU	8-core ARM v8.2 Carmel 64-bit CPU, 8MB L2, 4MB L3 cache
	GPU	512-core Volta GPU with Tensor cores
	Memory	16GB 256-Bit LPDDR4x, 137GB/s
	Storage	32GB eMMC 5.1
SW	Kernel	Linux 4.9.108
	OS	Ubuntu 18.04.1 LTS
	SW package	JetPack 4.1.1
	C library	GNU C library 2.27
	GPGPU library	CUDA 10.0
	C compiler	GNU Arm compiler 7.4.0
	GPGPU compiler	CUDA v10.0.117

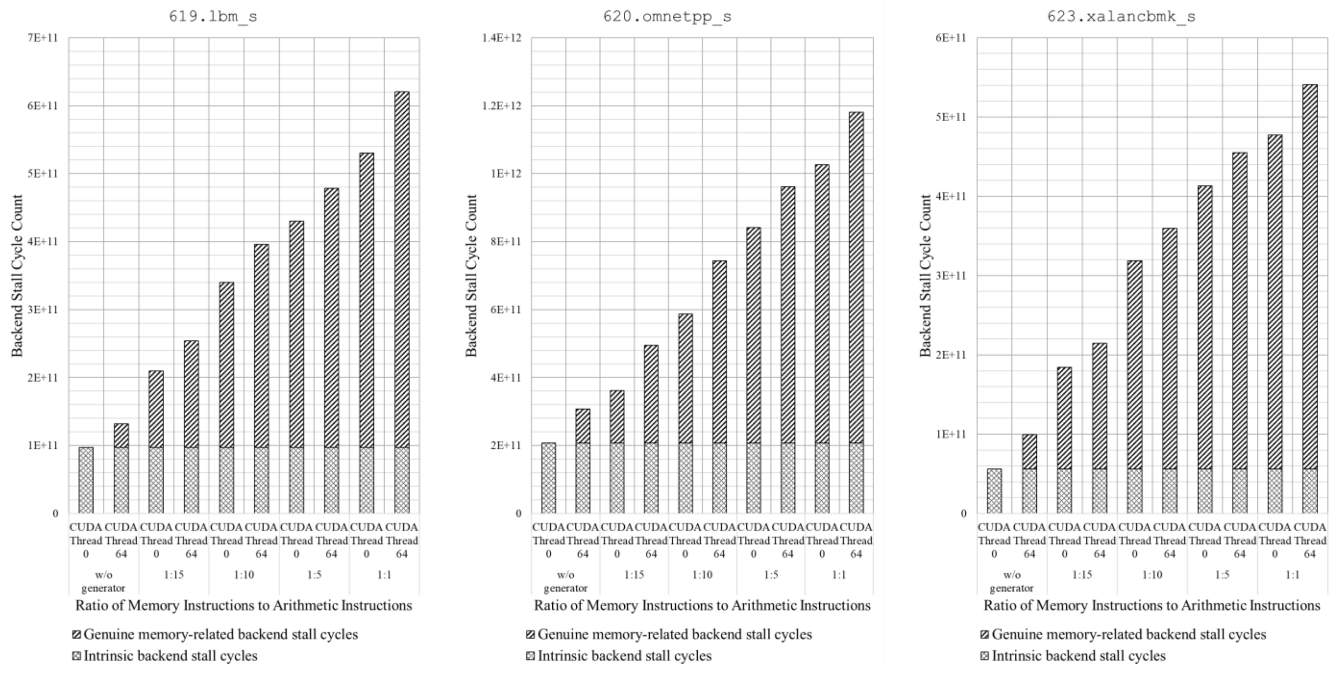


FIGURE 3. **Relationship between memory-related interference and backend stall cycle count.**

we made two memory access contention cases for each cache contention instance: no memory access contention via zero CUDA thread and the maximum memory access contention via 64 CUDA threads. More than 64 CUDA threads did not make a noticeable increase in memory access contention due to memory bandwidth saturation. As a result, our test variable has ten unique combinations.

For each experimental run, we pinned a benchmark program on core 7 and allocated an instance of the cache contention generator onto each core so that they could cover L1, L2 and L3 cache contention altogether. We ran the memory access contention generator on the GPU. Among per-core PMU events shown in Table I, we counted the `STALL_BACKEND` event in each experimental run.

Our experiments were conducted in two steps. We first measured the intrinsic backend stall of each benchmark program by running it alone until termination and then counting the backend stall cycles. By definition, this count was the amount of the intrinsic backend stall of the benchmark. Next, we ran each benchmark to completion while imposing different amounts of memory-related interference as stated above. We then counted backend stall cycles in each experimental run.

Figure 3 shows the experimental results. The horizontal axis denotes the amount of memory interference expressed by the test variable space. The vertical axis is the number of backend stall cycles, which is decomposed into an intrinsic backend stall cycle count below and a genuine memory-related backend stall cycle count above. The experimental results clearly show that the genuine memory-related backend stall cycle count increases with memory-related interference. Also, it is observed that cache contention has a greater impact on memory-related interference than memory access contention, which is consistent with our intuition.

V. PROBLM FORMULATION

In this section, we model our target system and define the notion of memory-aware fairness. We then formulate the problem at hand. We summarize frequently used notations in Table III.

A. TARGET SYSTEM AND TASK MODEL

Our target system is a symmetric multicore processor that consists of a set P of identical cores. The target system employs dynamic voltage and frequency scaling (DVFS) for reducing power consumption. To work under DVFS, mCFS

TABLE III
NOTATIONS

Symbol	Definition
P	Set of cores
F	Set of allowable operating frequency of a core
$r(f)$	Relative performance of frequency $f \in F$
Q	Set of QoS tasks
B	Set of best-effort tasks
τ_i	Task in $Q \cup B$
w_i	Weight of task τ_i
$c_i(t_1, t_2)$	Measured CPU time of task τ_i in $[t_1, t_2]$
$c_i^a(t_1, t_2)$	Actualized CPU time of task τ_i in $[t_1, t_2]$
$c_i^{as}(t_1, t_2)$	Actualized scaled CPU time of task τ_i in $[t_1, t_2]$
$n_i(t_1, t_2)$	Measured CPU cycles of task τ_i in $[t_1, t_2]$
$b_i(t_1, t_2)$	Measured backend stall cycles of task τ_i in $[t_1, t_2]$
$b_i^*(t_1, t_2)$	Intrinsic backend stall cycles of task τ_i in $[t_1, t_2]$
$b_i^m(t_1, t_2)$	Genuine memory-related backend stall cycles of task τ_i in $[t_1, t_2]$
$v_i(t_1, t_2)$	Actualized scaled virtual runtime (ASVR) of task τ_i in $[t_1, t_2]$
$\lvert v_{i,j}(t) \rvert$	ASVR difference between τ_i and τ_j in $[0, t]$
$v_{max}(t)$	The biggest $\lvert v_{i,j}(t) \rvert$ in $[0, t]$
$\gamma_i(f, t_1, t_2)$	IBSR of task τ_i in $[t_1, t_2]$ uner a given operating frequency $f \in F$
$\overline{\gamma_i}(f)$	Average IBSR of task τ_i uner a given operating frequency $f \in F$

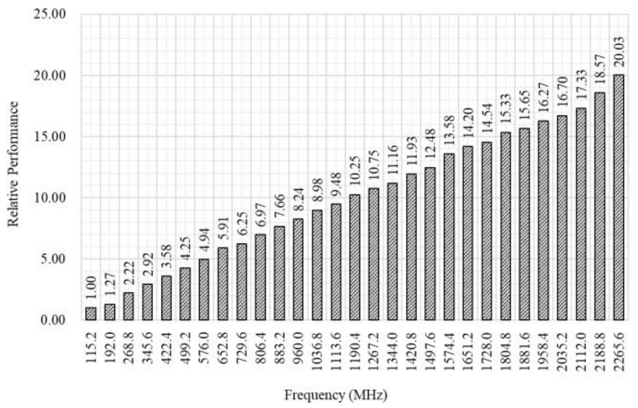


FIGURE 4. Relative performance for each available frequency.

must take into account the changing performance of each core when computing the virtual runtime of a task running on that core. To aid in this process, we use a frequency-to-relative performance mapping $r: F \rightarrow R$ where F is a set of allowable frequencies of the target system and R is a set of relative performance values. The relative performance $r(f)$ is simply the speedup factor against the base performance $r(f_{min})$ where f_{min} is the minimum frequency in F. We statically build the mapping table containing an entry $r(f)$ for each $f \in F$, as explained in [5][6]. Figure 4 shows such a mapping table in graph form.

We present the application model. An application is a multithreaded Linux process. In Linux, a user-level thread becomes a *task* that is a kernel-level entity scheduled by the kernel. An application can be either a QoS application or a best-effort application, based on the programmer's decision. By definition, all threads belonging to a QoS application become QoS tasks. A dedicated launcher process is used to fork a process from a QoS application's executable code, which will be explained in the next section. A best-effort application is an ordinary Linux application and does not require any special treatment in mCFS.

We define the task model. The target system runs a set of n tasks $Q \cup B = \{\tau_1, \tau_2, \ldots, \tau_n\}$ where Q is a set of QoS tasks and B is a set of best-effort tasks. The system developer classifies a task as stated above. QoS tasks are processed by mCFS while best-effort tasks are handled by the conventional CFS. The rationale behind this decision is that QoS tasks must be compensated for genuine memory-related interference at the cost of the degraded performance of best-effort tasks. CPU bandwidth allocation among tasks is a zero-sum game anyway and it is semantically correct that best-effort tasks become victims.

A task τ_i in $Q \cup B$ is associated with a fixed weight value denoted by w_i. Recall that the virtual runtime of a task is defined as the task's cumulative runtime inversely scaled by its weight in CFS. CFS uses virtual runtime as a measure of fairness.

In order to incorporate memory-aware fairness into CFS, we extend our previous work on scaled virtual runtime in [5][6]. Specifically, we define *actualized scaled virtual runtime* (*ASVR*). To do so, we first introduce the notion of actualized CPU time of a task τ_i for a given time interval $[t_1, t_2]$. We then scale the actualized CPU time according to the operating frequency in that time interval. Finally, we compute the virtual runtime by dividing the actualized scaled CPU time by its weight. Using the virtual runtime, we finally define the memory-aware fairness and formulate the problem at hand.

We let $c_i(t_1, t_2)$ and $n_i(t_1, t_2)$ denote the CPU time and the CPU cycle count of τ_i in $[t_1, t_2]$, respectively.

Definition 2. Actualized CPU time of a task τ_i at the end of a time interval $[t_1, t_2]$ is

$$c_i^a(t_1, t_2) = \begin{cases} c_i(t_1, t_2) \cdot \left(1 - \dfrac{b_i^m(t_1, t_2)}{n_i(t_1, t_2)}\right) & if\ \tau_i \in Q \\ c_i(t_1, t_2) & if\ \tau_i \in B \end{cases}$$

The actualized CPU time of a best-effort task is the same as the CPU time since best-effort tasks are not compensated for any memory-related interference. The actualized CPU time of a QoS task is defined as the CPU time deducted by the stall time due to genuine memory-related interference. Recall that $b_i^m(t_1, t_2)$ is the genuine memory-related backend stall cycle count in $[t_1, t_2]$.

We now scale the actualized CPU time according to the operating frequency as follows:

Definition 3. Actualized scaled CPU time of τ_i with an operating frequency f is

$$c_i^{as}(t_1, t_2) = c_i^a(t_1, t_2) \cdot r(f)$$

We in turn define the actualized scaled virtual runtime as follows:

Definition 4. Actualized scaled virtual runtime (ASVR) of τ_i is

$$v_i(t_1, t_2) = \frac{1}{w_i} \cdot c_i^{as}(t_1, t_2)$$

It is trivial that equalizing the ASVRs of tasks in the system achieves perfect memory-aware fairness for the target multicore system.

Definition 5. A memory-aware perfectly fair scheduler for a multicore system is one for which

$$\frac{c_i^{as}(0, t)}{c_j^{as}(0, t)} = \frac{w_i}{w_j}$$

holds for any tasks τ_i and τ_j for time interval $[0, t]$.

B. PROBLEM STATEMENT

The problem we address in this paper is to minimize ASVR difference between any pair of tasks from $Q \cup B$. Let $v_{i,j}(t)$ be the ASVR difference between two tasks τ_i and τ_j for time interval $[0, t]$. We define the maximum ASVR difference $v_{max}(t)$ as below.

$$v_{max}(t) = \max_{\tau_i, \tau_j \in Q \cup B} \left| v_{i,j}(t) \right|$$

Obviously, the objective of our problem is to reduce $v_{max}(t)$.

VI. MEMORY-AWARE COMPLETELY FAIR SCHEDULING

We propose to incorporate the memory-aware fairness into the Linux kernel to protect QoS tasks from the memory-related interference of other co-running tasks. We aim to extend the CFS with minimal modifications possible and with only existing hardware support from the target processor.

A. THE mCFS ARCHITECTURE

We present the kernel-level architecture of mCFS. It consists of the CFS task scheduler, the ASVR updater and two supporting kernel components as depicted in Figure 5. We engineer the CFS task scheduler very carefully so that it remains untouched except the virtual runtime updater. The ASVR updater replaces the original virtual runtime updater. It is invoked at every scheduling tick of the Linux kernel. On each invocation, the five components inside it get executed in tandem and calculate the actualized scaled virtual runtime of the currently running task. In doing so, the ASVR updater refers to the `CPUFreq` governor to obtain the current operating frequency of the core and reads in some PMU counters via the PMU driver.

Among the five components of the ASVR updater, the memory-related interference estimator deserves an in-depth explanation while others do not due to their self-explanatory definitions in the previous section.

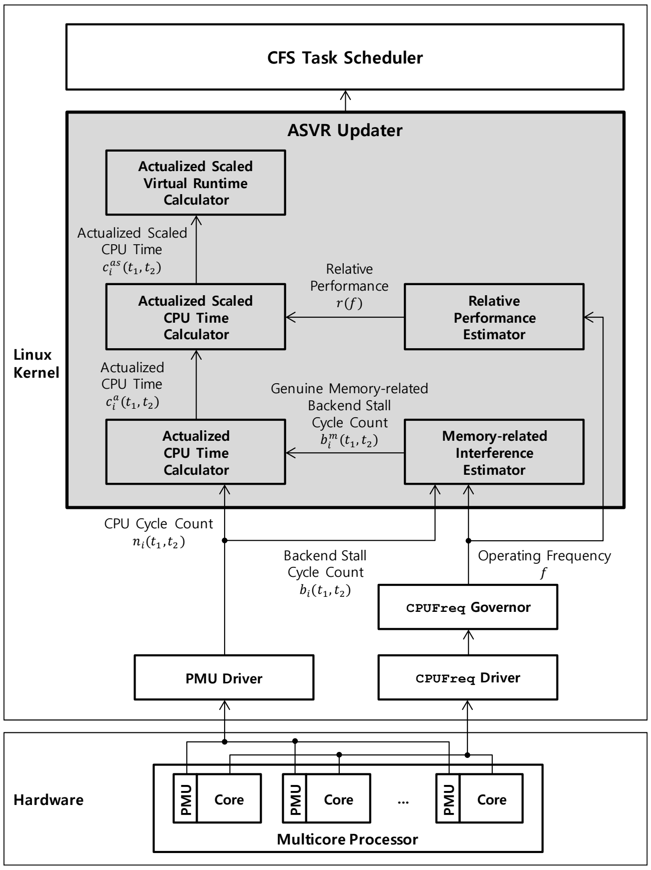

CFS Task Scheduler
ASVR Updater
Actualized Scaled Virtual Runtime Calculator
Actualized Scaled CPU Time $c_i^{as}(t_1,t_2)$
Relative Performance $r(f)$
Actualized Scaled CPU Time Calculator
Relative Performance Estimator
Linux Kernel
Actualized CPU Time $c_i^{a}(t_1,t_2)$
Genuine Memory-related Backend Stall Cycle Count $b_i^{m}(t_1,t_2)$
Actualized CPU Time Calculator
Memory-related Interference Estimator
CPU Cycle Count $n_i(t_1,t_2)$
Backend Stall Cycle Count $b_i(t_1,t_2)$
Operating Frequency f
CPUFreq Governor
PMU Driver
CPUFreq Driver
Hardware
PMU
Core
PMU
Core
...
PMU
Core
Multicore Processor

FIGURE 5. The mCFS architecture.

B. ESTIMATING MEMORY-RELATED INTERFERENCE

Let T_s be the scheduling tick interval size. On the occurrence of a scheduling tick at a time point t, the memory-related interference estimator calculates $b_i^m(t-T_s,t)$ according to Definition 1 re-written below.

$$b_i^m(t-T_s,t) = b_i(t-T_s,t) - b_i^*(t-T_s,t) \qquad (1)$$

To be practically feasible, the memory-related interference estimator must be able to obtain the values of $b_i(t-T_s,t)$ and $b_i^*(t-T_s,t)$ with only existing PMU support. It can easily get $b_i(t-T_s,t)$ by monitoring the `STALL_BACKEND` event of the PMU but cannot immediately obtain $b_i^*(t-T_s,t)$. Thus, we convert b_i^* into a combination of measurable entities.

We start by defining the intrinsic backend stall rate (IBSR) of τ_i in $[t_1,t_2]$ under a given operating frequency $f \in F$. A formal definition is given as follows.

Definition 6. Under an operating frequency $f \in F$, the intrinsic backend stall rate for a running task τ_i in $[t_1,t_2]$ is

$$\gamma_i(f,t_1,t_2) = \frac{b_i^*(t_1,t_2)}{t_2-t_1} \qquad (2)$$

Eq. (2) implies that we can compute $b_i^*(t_1,t_2)$ simply by knowing $\gamma_i(f,t_1,t_2)$. However, it is not feasible to pre-calculate all the values of $\gamma_i(f,t_1,t_2)$ for any arbitrary time intervals $[t_1,t_2]$. We thus propose to use the average IBSR as an approximation. We define the average IBSR as follows.

Definition 7. The average IBSR of τ_i is defined with sufficiently large T as follows.

$$\overline{\gamma_i}(f) = \gamma_i(f,0,T) \qquad (3)$$

If we rewrite Eq. (2) for $b_i^*(t_1,t_2)$ and substitute $\gamma_i(f,t_1,t_2)$ with its approximation $\overline{\gamma_i}(f)$, we have

$$b_i^*(t_1,t_2) \approx \overline{\gamma_i}(f) \cdot (t_2-t_1) \qquad (4)$$

Therefore, the memory-related interference estimator ends up with calculating $b_i^m(t-T_s,t)$ using the following equation at each scheduling tick occurring at time t.

$$b_i^m(t-T_s,t) = b_i(t-T_s,t) - \overline{\gamma_i}(f) \cdot T_s \qquad (5)$$

As a task runs for tens of thousands of scheduling tick intervals, the memory-related interference estimator adds up the intrinsic backend stall cycle count $b_i^*(t-T_s,t)$ of each scheduling tick interval as many times. For k scheduling ticks, the approximate value for the accumulated intrinsic backend stall cycle count simply becomes $k \cdot \overline{\gamma_i}(f) \cdot T_s$. We argue that $k \cdot \overline{\gamma_i}(f) \cdot T_s$ gets sufficiently close to the actual intrinsic backend stall cycle count if k is sufficiently large. We justify this argument.

We first show via an experiment that for a sufficiently large time interval $[t_1,t_2]$, $\overline{\gamma_i}(f)$ gets closer to $\gamma_i(f,t_1,t_2)$ with sufficiently large $T \le t_2 - t_1$ as stated below:

$$\gamma_i(f,t_1,t_2) \approx \overline{\gamma_i}(f) \qquad (6)$$

In our experiment for supporting Eq. (6), we pinned one of our benchmarks, `619.lbm_s` on core 7 and ran it alone without any memory-related interference. In this case, the

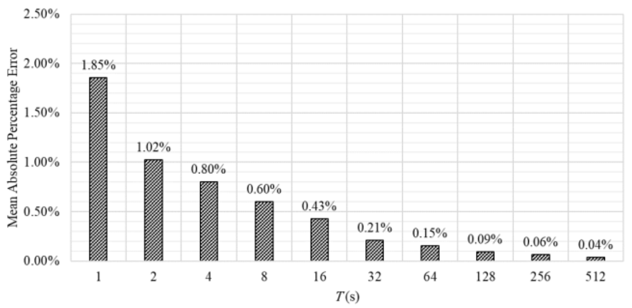


FIGURE 6. **Relationship between *T* and accuracy of average IBSR.**

backend stall becomes the intrinsic backend stall, i.e., $b_i(t_1, t_2) = b_i^*(t_1, t_2)$. We ran the benchmark program for 1,300s. We iteratively measured the backend stall cycle count for every 1ms while it was running. We then repeatedly calculated numerous IBSR values $\gamma_i(f, t_3, t_3 + T)$ by changing t_3 and T. We also computed the IBSR for the entire running time, $\gamma_i(f, 0, 1300)$. To analyze how close $\gamma_i(f, t_3, t_3 + T)$ is to $\gamma_i(f, 0, 1300)$, we compute the mean absolute percentage error between them. Figure 6 shows the result. As T gets closer to the benchmark's entire running time, the mean absolute percentage error gets reduced. When T is above 128s, the mean absolute percentage error becomes sufficiently small, below 0.1%. We observed the same behavior with the other eight benchmarks.

In CFS, a task runs non-preemptively for every scheduling tick interval given to the task. We consider k scheduling tick intervals $[t_j, t_j + T_s]$ for $1 \le j \le k$ for which a task τ_i has been running. If k is sufficiently large such that $k \cdot T_s \ge T$, then the following holds true according to Eq. (4).

$$\sum_{j=1}^{k} b_i^*(t_j, t_j + T_s) = \gamma_i(f, 0, k \cdot T_s) \cdot k \cdot T_s$$
$$\approx \overline{\gamma_i}(f) \cdot k \cdot T_s \qquad (7)$$

Thus, Eq. (7) proves our argument.

We suggest a guideline for selecting T for $\overline{\gamma_i}(f)$ using the mean absolute percentage error. Users are first asked to choose a threshold for the mean absolute percentage error. Then they can choose any T that satisfies the threshold. In our experiment, we chose 1% as the threshold and we selected 128s for T. As a rule of thumb, any value greater than 100s suffices.

C. INTERFACING WITH USERS

Since only QoS tasks are protected from memory-related interference via memory-aware fair-share scheduling, mCFS needs to differentiate QoS applications from best-effort applications. In our approach, we offer a dedicated launcher process that programmers use to let mCFS know about their QoS applications. The pseudo code for the launcher process is given in Figure 7. The launcher enabled us to incorporate mCFS into the Linux kernel without modifying any existing system call interfaces.

The launcher accepts three arguments: the path name of a QoS application's executable file, its parameters and a list of its average IBSR values. The launcher works in two steps. First, it stores the average IBSR values into the file named `/proc/pid/ibsr` via the `write()` system call where "`pid`" is the launcher's process id. We associate with the `write()` system call a callback function that copies the average IBSR values into the `task_struct` instance of the launcher process.

```
int main(int argc, void *argv[]) {
  // Pass the list of IBSRs to kernel
  fd = fopen("/proc/pid/ibsr", "w+");
  fwrite(argv[2], 1, strlen(argv[2]), fd);
  fclose(fd);
  // Construct a command to run the QoS app
  // Run the command
  system(app_cmd);
}
```

FIGURE 7. **Pseudo code for the launcher process.**

Second, the launcher forks and executes the QoS application. We slightly modified the `fork()` system call code so that the average IBSR values stored in the parent process are copied into the `task_struct` instance of the child process. Since a best-effort task has the default value of zero for the average IBSRs, mCFS can easily distinguish between a QoS task and a best-effort task.

D. INTERACTING WITH KERNEL COMPONENTS

As shown in Figure 5, mCFS closely interacts with two kernel components: the PMU driver and the `CPUFreq` governor. We explain such interactions in detail.

mCFS accesses the core's operating frequency that is independently maintained by the `CPUFreq` governor. Among the various governor types supported in Linux, we consider the `schedutil` governor for mCFS since it is the default `CPUFreq` governor that was newly added to Linux v4.7 [8]. Other governor types can be easily integrated into mCFS in a similar manner.

The `schedutil` governor collects a core's utilization statistics periodically at each scheduling tick. Additionally, it gathers the same information upon the occurrences of the sporadic events that can affect CPU utilization, such as task creation and termination.

According to Definition 3, the actualized scaled virtual runtime of a task τ_i is defined over a time interval $[t_1, t_2]$ and computed at the end of that interval. This requires that the frequency of the core hosting τ_i be constant throughout the interval. In the mCFS implementation, $[t_1, t_2]$ exactly corresponds to a scheduling tick interval. The `schedutil` governor assures this requirement since the governor adjusts a core's frequency mostly at tick boundaries. The effect of sporadic adjustments is negligible since they occur very rarely compared to the periodic adjustments.

TABLE IV
MEMORY INTENSIVENESS OF BENCHMARKS

Benchmark	Percentage of Memory Access Cycle Count Compared to Total CPU Cycle Count
619.lbm_s	11.831%
623.xalancbmk_s	5.645%
602.gcc_s	3.303%
600.perlbench_s	0.455%
648.exchange2_s	0.014%

VII. EXPERIMENTAL EVALUATION

In this section, we report on the experiments that we performed to demonstrate the effectiveness of mCFS. We first describe the experimental setup and then show the experimental results along with our analysis.

A. EXPERIMENTAL SETUP

We used the same experimental setup as in Section IV. As QoS applications, we used five benchmark programs from SPEC CPU2017 as well as the YOLO face detection program [28]. The five benchmark programs are `619.lbm_s`, `623.xalancbmk_s`, `602.gcc_s`, `600.perlbench_s` and `648.exchange2_s`. Table IV shows one of the important characteristics of the benchmark programs: the degree of memory intensiveness of each benchmark, with the most memory intensive at the top.

Each benchmark program ran to completion three times with three different configurations, respectively: (1) running with no memory-related interference, (2) running under the conventional CFS with memory-related interference and (3) running under mCFS with memory-related interference. We measured their response times.

We used two test variables for these experiments. First, we varied the ratio of the QoS applications and the best-effort applications in the workloads, from 1:1 to 1:5. Second, we varied the QoS applications while running the same best-effort application.

We used two performance metrics to analyze the gain and the cost incurred by mCFS. We measured the percentile performance improvement of a QoS application on mCFS over on CFS. Similarly, we measured the percentile performance degradation of a best-effort application on mCFS over on CFS.

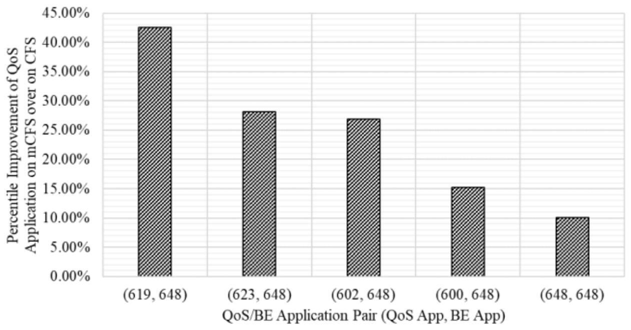


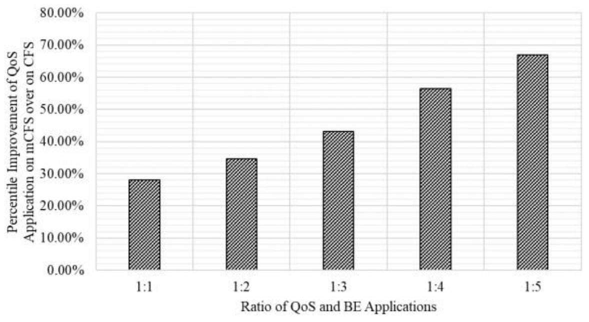
FIGURE 8. Performance improvement of QoS applications on mCFS according to memory intensiveness of QoS applications.

In our experiments, we pinned the QoS applications and best-effort applications on core 7 so that all of them became subject to per-core fair-share scheduling. To generate memory-related interference, we ran the cache contention generators on the remaining cores and the memory contention generator on the GPU.

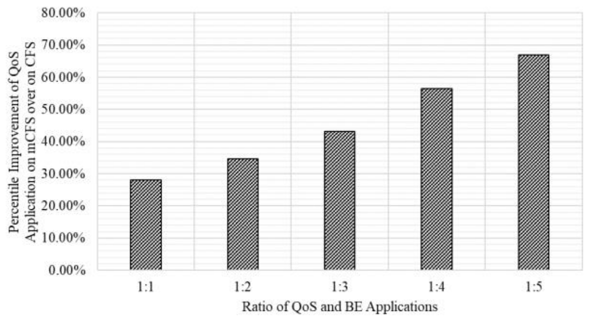
B. EXPERIMENTAL RESULTS

We conducted three experiments to observe and analyze the performance improvement of the QoS applications under mCFS. We first ran the five benchmark programs one by one as a QoS application with the best-effort application commonly being `648.exchange2_s`. The experimental result is given in Figure 8. The performance improvement of mCFS over CFS ranges from 10% to 43% and increases with the memory intensiveness of the QoS applications. This result states that mCFS adaptively improves performance as needed. The more memory traffic, the greater the performance improvement.

In the second experiment, we varied the ratio of the QoS applications and the best-effort applications from 1:1 to 1:5. We used `623.xalancbmk_s` and `648.exchange2_s` as the QoS application and best-effort application, respectively. We ran multiple instances of `648.exchange2_s` to increase the proportion of the best-effort application. Figure 9 (a) shows the result. As the amount of the best-effort workload increases, mCFS yields greater performance improvement for the QoS application. It shows the resilience of mCFS in the sense that mCFS works more aggressively as the best-effort workload increases. We repeated the same experiment with the YOLO face detection



(a) `623.xalancbmk_s` as QoS applications



(b) YOLO face detection as QoS applications

FIGURE 9. Performance improvement of QoS applications on mCFS according to the ratio of QoS applications and best-effort applications.

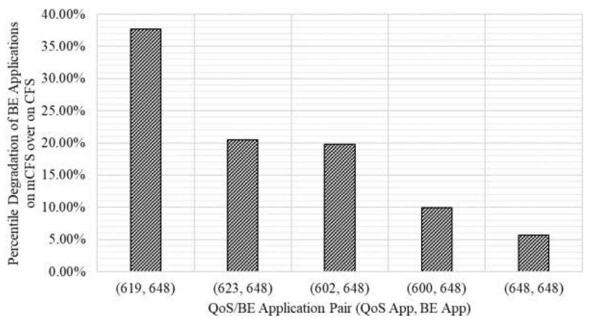


FIGURE 10. **Performance degradation of best-effort applications on mCFS according to memory intensiveness of QoS applications.**

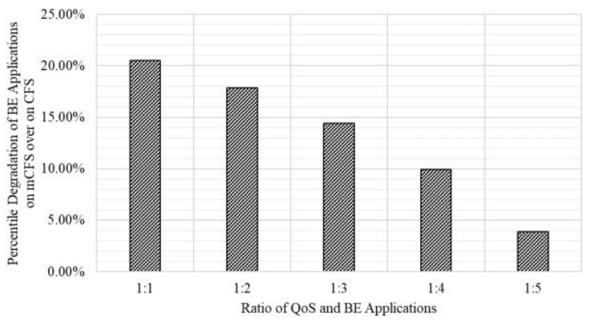


FIGURE 11. **Performance degradation of best-effort applications on mCFS according to the ratio of QoS applications and best-effort applications.**

program. The result given in Figure 9 (b) is consistent with that in Figure 9 (a).

We performed two additional experiments to assess the performance degradation that mCFS caused to the best-effort applications. The first experiment is dual to the experiment of Figure 8. We ran the five benchmark programs one by one as a QoS application while running `648.exchange2_s` as a best-effort application. We measured the performance degradation of the best-effort application. Figure 10 shows that the performance degradation ranges from 5% to 38% and increases with the memory intensiveness of the QoS applications. This result is consistent with that of Figure 8.

The next experiment is dual to the experiment of Figure 9 (a). We varied the ratio of the QoS applications and the best-effort applications from 1:1 to 1:5. We measured the performance degradation of the best-effort application. The result is given in Figure 11. As the amount of the best-effort workload increases, the performance degradation of each application decreases. This is because multiple best-effort applications share the burden.

C. EVALUATING RUN-TIME OVERHEAD

From the architecture of mCFS, it is obvious that only the ASVR updater incurs an extra runtime overhead to the kernel scheduler. We thus measured the execution time of the ASVR updater while running the benchmark `619.lbm_s`. As a result of the measurement, we obtained 3,624ns. Since the ASVR updater is invoked every 4ms by the scheduling tick handler, the extra runtime overhead is only 0.091%.

VIII. CONCLUSION

We presented a memory-aware fair-share scheduling algorithm that makes QoS applications less susceptible to memory-related interference from other co-running applications. Our algorithm dynamically separates the genuine memory-related stall from a running task's backend stall cycles and compensates the task for the memory-related interference so that the task gets the desired share of CPU before it is too late.

To compute the genuine memory-related stall amount of a task, our algorithm first defines the average intrinsic backend stall rate of a task. It estimates the amount of the task's intrinsic backend stall using the IBSR and deducts it from the task's entire backend stall amount. Our algorithm actualizes the CPU time of a task by decreasing the task's physical CPU time according to the estimated memory-related interference. To take into account performance asymmetry among cores caused by inevitable DVFS, our algorithm scales the actualized CPU time according to the relative performance of the core hosting the task. The algorithm finally computes the virtual runtime so that the task becomes schedulable by CFS.

Our algorithm is a compensation-based temporal memory resource isolation technique. As a result, it does not rely on either inflexible resource management, ineffective execution throttling or potentially wasteful execution restriction. Moreover, it seamlessly supports the performance asymmetry of multicore architecture.

Since our algorithm is a software-only solution, we could implement it into the CFS of the Linux kernel, with minimal modifications to the kernel. We named the end result mCFS. We have also conducted extensive experiments to validate the effectiveness of mCFS. The experimental results assert that mCFS is effective in protecting QoS applications from memory-related interference as well as it is adaptive, resilient and efficient. We make the source code for mCFS freely available through the `github` [9].

REFERENCES

[1] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *proc. 23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Pittsburgh, PA, 2017, pp. 353-364.

[2] A. Gupta and R. K. Jha, "A survey of 5G network: Architecture and emerging technologies," *IEEE Access*, vol. 3, pp. 1206-1232, 2015.

[3] S. Huh, J. Yoo and S. Hong, "Improving interactivity via VT-CFS and framework-assisted task characterization for Linux/Android Smartphones," in *proc. IEEE Int. Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Seoul, 2012, pp. 250-259.

[4] S. Huh, J. Yoo and S. Hong, "Cross-layer resource control and scheduling for improving interactivity in Android," *International Journal of Software: Practice and Experience*, vol. 45, issue. 11, pp. 1549-1570, Nov. 2015.

[5] M. Kim, S. Noh, S. Huh and S. Hong, "Fair-share scheduling for performance-asymmetric multicore architecture via scaled virtual runtime," in *proc. IEEE 21st Int. Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Hong Kong, 2015, pp. 60-69.

[6] M. Kim, S. Noh, J. Hyeon and S. Hong, "Fair-share scheduling in single-ISA asymmetric multicore architecture via scaled virtual

runtime and load redistribution," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 111, pp. 174-186, Jan. 2018.

[7] D. Brodowski, "CPUFreq governors," [online] Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt, 2013

[8] N. Brown, "Improvements in CPU frequency management," [online] Available: https://lwn.net/Articles/682391/, 2016.

[9] mCFS implementation code, cache contention generator code, memory access contention code and QoS task launcher code, [online] Available: `https://github.com/yearnotw/mCFS.git`.

[10] R. Cavicchioli, N. Capodieci and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *proc. 22nd IEEE Int. Conf. Emerging Technologies and Factory Automation (ETFA)*, Limassol, 2017, pp. 1-10.

[11] H. Wen and Z. Wei, "Interference evaluation in CPU-GPU heterogeneous computing", in *proc. IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, 2017

[12] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *proc. 44th Annual IEEE/ACM Int. Symposium on Microarchitecture (MICRO)*, Porto Alegre, 2011, pp. 374-385.

[13] Lei Liu, Z. Cui, Mingjie Xing, Y. Bao, M. Chen and Chengyong Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *proc. 21st Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, Minneapolis, MN, 2012, pp. 367-375.

[14] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *proc. IEEE Int. Symposium on High Performance Computer Architecture (HPCA)*, Vienna, 2018, pp. 104-117.

[15] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha and J. Moses, "Rate-based QoS techniques for cache/memory in CMP platforms," in *proc. 23rd ACM Int. Conf.* Supercomputing, Yorktown Heights, NY, 2009, pp. 479-488.

[16] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *proc. IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Philadelphia, PA, 2013, pp. 55-64.

[17] J. Kim, P. Shin, S. Noh, D. Ham and S. Hong, "Reducing memory interference latency of safety-critical applications via memory request throttling and Linux cgroup," in *proc. 31st IEEE Int. System-on-Chip Conference (SOCC)*, Washington DC, 2018, pp. 215-220.

[18] E. Ebrahimi, C. J. Lee, O. Mutlu, Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems", *ACM SIGPLAN Notices*, vol. 45, no. 3, pp. 335-346, Mar. 2010.

[19] D. Xu, C. Wu, P. Yew, J. Li, and Z. Wang, "Providing fairness on shared-memory multiprocessors via process scheduling," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 295-306, Jun. 2012.

[20] A. Fedorova, M. Seltzer and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *proc. 16th Int. Conf. Parallel Architecture and Compilation Techniques (PACT)*, Brasov, 2007, pp. 25-38.

[21] S. Barati and H. Hoffmann, "Providing fairness in heterogeneous multicores with a predictive, adaptive scheduler," in *proc. IEEE Int. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Chicago, IL, 2016, pp. 38-49.

[22] Xavier series SoC technical reference manual, NVIDIA Co., Santa Clara, CA, USA, 2019.

[23] ARMv8 architecture reference manual, ARM Ltd., Cambridge, UK, 2017.

[24] A. Yasin, "A top-down method for performance analysis and counters architecture," in *proc. IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, 2014, pp. 35-44.

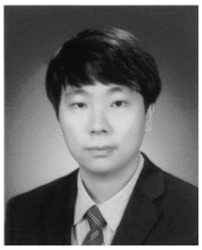
[25] C. S. Pabla, "Completely fair scheduler," Linux Journal, 2009.

[26] SPEC, "SPEC CPU®2017 Utilities," [online] Available: https://www.spec.org/cpu2017/Docs/utility.html.

[27] R. Hebbar S R and A. Milenković, "SPEC CPU2017: Performance, event, and energy characterization on the Core i7-8700K," *in proc. ACM/SPEC Int. Conf. Performance Engineering (ICPE)*, New York, NY, 2019, pp. 111-118.

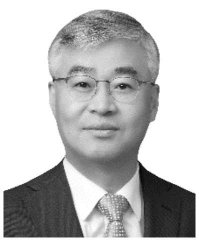
[28] YOLOFace, [online] Available: https://github.com/sthanhng/yoloface

Jungho Kim earned his B.S. degree in the Department of Electronic and Electrical Engineering and the Department of Computer Science and Engineering from Pohang University of Science and Technology, Korea, in 2010. He earned his M.S. degree in the Department of Electrical and Computer Engineering from Seoul National University, Korea, in 2013. He is currently a Ph.D. candidate in the Department of Transdisciplinary Studies, Graduate School of Convergence Science and Technology, Seoul National University, Korea. He is also a member of Real-Time Operating System Laboratory at Seoul National University. His current research interests include software architecture for embedded systems, Linux kernel techniques for resource management.



Philkyue Shin earned his B.S. degrees in the Department of Electrical and Computer Engineering from Seoul National University, Korea, in 2017. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering at Seoul National University, Korea. He is also a member of Real-Time Operating System Laboratory at Seoul National University. His current research interests include Linux kernel techniques for active resource management.

Myungsun Kim is currently an assistant professor of Division of IT Convergence Engineering at Hansung University. He had worked in the R&D center of Samsung Electronics from 2002 to 2018. His fields of research in Samsung R&D were processor architecture for DNN, heterogeneous core computing, Linux kernel internal, Android framework and SoC simulators. He earned his BS and MS degrees in Electrical Engineering from Chung-Ang University, in 1998 and 2000, respectively. He received his PhD degree in the Department of Electrical and Computer Engineering at Seoul National University, in 2016. His current research interests include DNN computing, embedded and real-time systems design, HW/SW cross-layer optimization.



Seongsoo Hong earned his B.S. and M.S. degrees in computer engineering from Seoul National University, Korea, in 1986 and 1988, respectively. He received his Ph.D. degree in computer science from the University of Maryland, College Park, in 1994. He is currently a professor in the Department of Electrical and Computer Engineering at Seoul National University. His current research interests include embedded and real-time systems design, real-time operating systems, software architecture for embedded and real-time systems and cross-layer optimization of complex multi-layered software systems. He is the steering committee of IEEE RTCSA. He served as a general co-chair of IEEE RTCSA 2006 and CASES 2006 and as a program committee co-chair of IEEE RTAS 2005, RTCSA 2003, IEEE ISORC 2002 and ACM LCTES 2001. He has served on numerous program committees, including IEEE RTSS and ACM OOPSLA. He is a member of the National Academy of Engineering of Korea. He is currently a senior member of the IEEE and a senior member of the ACM.