Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms

Waqar Ali
University of Kansas
Lawrence, USA
wali@ku.edu

Heechul Yun
University of Kansas
Lawrence, USA
heechul.yun@ku.edu

Abstract

Integrated CPU-GPU architecture provides excellent acceleration capabilities for data parallel applications on embedded platforms while meeting the size, weight and power (SWaP) requirements. However, sharing of main memory between CPU applications and GPU kernels can severely affect the execution of GPU kernels and diminish the performance gain provided by GPU. For example, in the NVIDIA Jetson TX2 platform, an integrated CPU-GPU architecture, we observed that, in the worst case, the GPU kernels can suffer as much as 3X slowdown in the presence of co-running memory intensive CPU applications. In this paper, we propose a software mechanism, which we call BWLOCK++, to protect the performance of GPU kernels from co-scheduled memory intensive CPU applications.

1 Introduction

Graphic Processing Units (GPUs) are increasingly important computing resources to accelerate a growing number of data parallel applications. In recent years, GPUs have become a key requirement for intelligent and timely processing of large amount of sensor data in many robotics applications, such as UAVs and autonomous cars. These intelligent robots are, however, resource constrained real-time embedded systems that not only require high computing performance but also must satisfy a variety of constraints such as size, weight, power consumption (SWaP) and cost. This makes integrated CPU-GPU architecture based computing platforms, which integrate CPU and GPU in a single chip (e.g., NVIDIA's Jetson [5] series), an appealing solution for such robotics applications because of their high performance and efficiency [15].
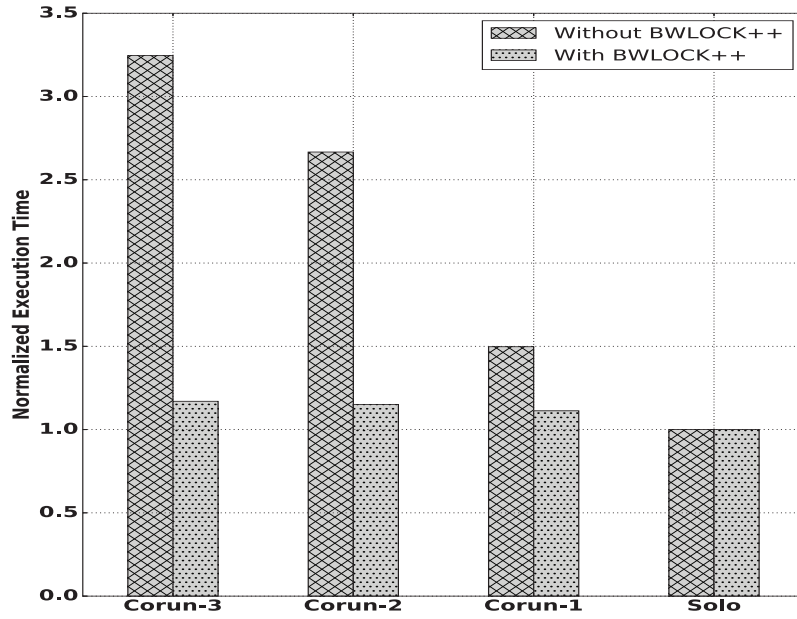
Figure 1 Performance of *histo* benchmark on NVIDIA Jetson TX2 with CPU corunners.

Designing critical real-time applications on integrated CPU-GPU architectures is, however, challenging because contention in the shared hardware resources (e.g., memory bandwidth) can significantly alter the applications' timing characteristics. On an integrated CPU-GPU platform, such as NVIDIA Jetson TX2, the CPU cores and the GPU typically share a single main memory subsystem. This allows memory intensive batch jobs running on the CPU cores to significantly interfere with the execution of critical real-time GPU tasks (e.g., vision based navigation and obstacle detection) running in parallel due to memory bandwidth contention.

To illustrate the significance of the problem stated above, we evaluate the effect of co-scheduling memory bandwidth intensive synthetic CPU benchmarks on the performance of a GPU benchmark *histo* from the parboil benchmark suite [18] on a NVIDIA Jetson TX2 platform (See Table 3 in Section 6 for the detailed time breakdown of *histo*.) We first run the benchmark alone and record the solo execution statistics. We then repeat the experiment with an increasing number of interfering memory intensive benchmarks on the idle CPU cores to observe their impact on the performance of the *histo* benchmark with and without the BWLOCK++ framework, which we propose in this paper. Figure 1 shows the results of this experiment. As can be seen in *'Without BWLOCK++'*, co-scheduling the memory intensive tasks on the idle CPU cores significantly increase the execution time of the GPU benchmark—a 3.3X increase—despite the fact that the benchmark has exclusive access to the GPU. The main cause of the problem is that, in the Jetson TX2 platform, both CPU and GPU share the main memory and its limited memory bandwidth becomes a bottleneck. As a result, even though the platform offers plenty of raw performance, no real-time execution guarantees can be provided if the system is left unmanaged. In *'With BWLOCK++'*, on the other hand, performance of the GPU benchmark remains close to its solo performance measured in isolation.

BWLOCK++ is a software framework designed to mitigate the memory bandwidth contention problem in integrated CPU-GPU architectures. More specifically, we focus on protecting real-time GPU tasks from the interference of non-critical but memory intensive CPU tasks. BWLOCK++ dynamically instruments GPU tasks at run-time and inserts a

memory bandwidth lock while critical GPU kernels are being executed on the GPU. When the bandwidth lock is being held by the GPU, the OS throttles the maximum memory bandwidth usage of the CPU cores to a certain threshold value to protect the GPU kernels. The threshold value is determined on a per GPU task basis and may vary depending on the GPU task's sensitivity to memory bandwidth contention. Throttling CPU cores inevitably negatively affects the CPU throughput. To minimize the throughput impact, we propose a throttling-aware CPU scheduling algorithm, which we call Throttle Fair Scheduler (TFS). TFS favors CPU intensive tasks over memory intensive ones while the GPU is busy executing critical GPU tasks in order to minimize CPU throttling. Our evaluation shows that BWLOCK++ can provide good performance isolation for bandwidth intensive GPU tasks in the presence of memory intensive CPU tasks. Furthermore, the TFS scheduling algorithm reduces the CPU throughput loss by up to 75%. Finally, we show how BLWOCK++ can be incorporated in existing CPU focused real-time analysis frameworks to analyze schedulability of real-time tasksets, utilizing both CPU and GPU.

In this paper, we make the following contributions:

- We apply memory bandwidth throttling to the problem of protecting GPU accelerated real-time tasks from memory intensive CPU tasks on integrated CPU-GPU architecture
- We identify a negative feedback effect of memory bandwidth throttling when used with Linux's CFS [13] scheduler. We propose a throttling-aware CPU scheduling algorithm, which we call Throttle Fair Scheduler (TFS), to mitigate the problem
- We introduce an automatic GPU kernel instrumentation method that eliminates the need of manual programmer intervention to protect GPU kernels
- We implement the proposed framework, which we call BWLOCK++, on a real platform, NVIDIA Jetson TX2, and present detailed evaluation results showing practical benefits of the framework [1]
- We show how the proposed framework can be integrated into the existing CPU focused real-time schedulability analysis framework

The remainder of this paper is organized as follows. We present necessary background and discuss related work in Section 2. In Section 3, we present our system model. Section 4 describes the design of our software framework BWLOCK++ and Section 5 presents implementation details. In Section 6, we describe our evaluation platform and present evaluation results using a set of GPU benchmarks. In Section 7, we present the analysis framework of BWLOCK++ based real-time systems. We discuss limitations of our approach in Section 8 and conclude in Section 9.

2   Background and Related Work

In this section, we provide necessary background and discuss related work.

GPU is an accelerator that executes some specific functions requested by a master CPU program. Requests to the GPU can be made by using GPU programming frameworks such as CUDA that offer standard APIs. A request to GPU is typically composed of the following four predictable steps:

- Copy data from host memory to device (GPU) memory
- Launch the function—called *kernel*—to be executed on the GPU
- Wait until the kernel finishes
- Copy the output from device memory to host memory

---

[1] The source code of BWLOCK++ is publicly available at: `https://github.com/wali-ku/BWLOCK-GPU`

In the real-time systems community, GPUs have been studied actively in recent years because of their potential benefits in accelerating demanding data-parallel real-time applications [11]. As observed in [1], GPU kernels typically demand high memory bandwidth to achieve high data parallelism and, if the memory bandwidth required by GPU kernels is not satisfied, it can result in significant performance reduction. For discrete GPUs, which have dedicated graphic memories, researchers have focused on addressing interference among the co-scheduled GPU tasks. Many real-time GPU resource management frameworks adopt scheduling based approaches, similar to real-time CPU scheduling, that provide priority or server based scheduling of GPU tasks [9, 10, 23]. Elliot et al., formulate the GPU resource management problem as a synchronization problem and propose the GPUSync framework that uses real-time locking protocols to deterministically handle GPU access requests [6]. Here, at any given time, one GPU kernel is allowed to utilize the GPU to eliminate the unpredictability caused by co-scheduled GPU kernels. In [12], instead of using a real-time locking protocol that suffers from busy-waiting at the CPU side, the authors propose a GPU server mechanism which centralizes access to the GPU and allows CPU suspension (thus eliminating the CPU busy-waiting). All the aforementioned frameworks primarily work for discrete GPUs, which have dedicated graphic memory, but they do not guarantee predictable GPU timing on integrated CPU-GPU based platforms because they do not consider the problem of the shared memory bandwidth contention between the CPU and the GPU.

Integrated GPU based platforms have recently gained much attention in the real-time systems community. In [15, 14], the authors investigate the suitability of NVIDIA's Tegra X1 platform for use in safety critical real-time systems. With careful reverse engineering, they have identified undisclosed scheduling policies that determine how concurrent GPU kernels are scheduled on the platform. In SiGAMMA [4], the authors present a novel mechanism to preempt the GPU kernel using a high-priority spinning GPU kernel to protect critical real-time CPU applications. Their work is orthogonal to ours as it solves the problem of protecting CPU tasks from GPU tasks while our work solves the problem of protecting GPU tasks from CPU tasks.

More recently, GPUGuard [7] provides a mechanism for deterministically arbitrating memory access requests between CPU cores and GPU in heterogeneous platforms containing integrated GPUs. They extend the PREM execution model [16], in which a (CPU) task is assumed to have distinct computation and memory phases, to model GPU tasks. GPUGuard provides deterministic memory access by ensuring that only a single PREM memory phase is in execution at any given time. Although GPUGuard can provide strong isolation guarantees, the drawback is that it may require significant restructuring of application source code to be compatible with the PREM model.

In this paper, we favor a less intrusive approach that requires minimal or no programmer intervention. Our approach is rooted on a kernel-level memory bandwidth throttling mechanism called MemGuard [22], which utilizes hardware performance counters of the CPU cores to limit memory bandwidth consumption of the individual cores for a fixed time interval on homogeneous multicore architectures. MemGuard enables a system designer—not individual application programmers—to partition memory bandwidth among the CPU cores. However, MemGuard suffers from system-level throughput reduction due to its coarse-grain bandwidth control (per-core-level control). In contrast, [21] is also based on a memory bandwidth throttling mechanism on homogeneous multicore architectures but it requires a certain degree of programmer intervention for fine-grain bandwidth control by exposing a simple lock-like API to applications. The API can enable/disable memory bandwidth control in a fine-grain manner within the application source code. However, this means that the application source code must be modified to leverage the feature.

Our work is based on memory bandwidth throttling, but, unlike prior throttling based approaches, focuses on the problem of protecting GPU accelerated real-time tasks on integrated CPU-GPU architectures and does not require any programmer intervention. Furthermore, we identify a previously unknown negative side-effect of memory bandwidth throttling when used with Linux's CFS scheduler, which we mitigate in this work. In the following, we start by defining the system model, followed by detailed design and implementation of the proposed system.

3   System Model

We assume an integrated CPU-GPU architecture based platform, which is composed of multiple CPU cores and a single GPU that share the same main memory subsystem. We consider independent periodic real-time tasks with implicit deadlines and best-effort tasks with no real-time constraints.

Task Model.   Each task is composed of at least one CPU execution segment and zero or more GPU execution segments. We assume that *GPU execution is non-preemptible* and we do not allow concurrent execution of multiple GPU kernels from different tasks at the same time. Simultaneously co-scheduling multiple kernels is called GPU co-scheduling, which has been avoided in most prior real-time GPU management approaches [10, 6, 12] as well due to unpredictable timing. According to [15], preventing GPU co-scheduling does not necessarily hurt—if not improve—performance because concurrent GPU kernels from different tasks are executed in a time-multiplexed manner rather than being executed in parallel. [2]

Executing GPU kernels typically requires copying considerable amount of data between the CPU and the GPU. In particular, synchronous copy directly contributes to the task's execution time, while asynchronous copy can overlap with GPU kernel execution. Therefore, we model synchronous copy separately. Lastly, we assume that a task is single-threaded with respect to the CPU. Then, we can model a real-time task as follows:

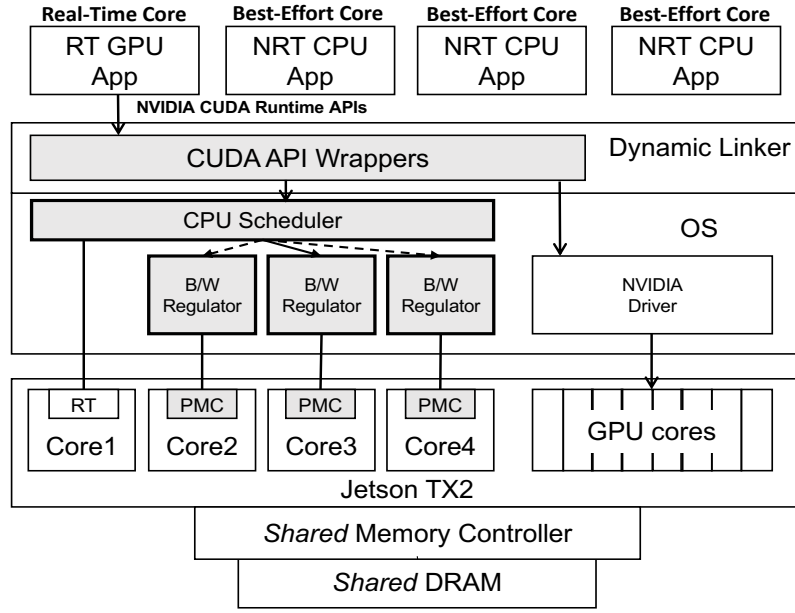$$\tau_i := (C_i, G_i^m, G_i^e, P_i)$$

where:
- C_i is the cumulative WCET of CPU-only execution
- G_i^m is the cumulative WCET of synchronous memory operations between CPU and GPU
- G_i^e is the cumulative WCET of GPU kernels
- P_i is the period

Note that the goal of BWLOCK++ is to reduce G_i^m and G_i^e under the presence of memory intensive best-effort tasks running in parallel.

CPU Scheduling.   We assume a fixed-priority preemptive real-time scheduler is used for scheduling real-time tasks and a virtual run-time based fair sharing scheduler (e.g., Linux's Completely Fair Scheduler [13]) is used for best-effort tasks. For simplicity, we assume a single dedicated real-time core schedules all real-time tasks, while any core can schedule best-effort tasks. Because GPU kernels are executed serially on the GPU, as mentioned

---

[2] Another recent study [2] finds that GPU kernels can only be executed in parallel if they are submitted from a single address space. In this work, we assume that a task has its own address space, whose GPU kernels are thus time-multiplexed with other tasks' GPU kernels at the GPU-level.

Figure 2 BWLOCK++ System Architecture.

above, for GPU intensive real-time tasks, which we focus on in this work, this assumption does not significantly under-utilize the system, especially when there are enough co-scheduled best-effort tasks, while it enables simpler analysis.
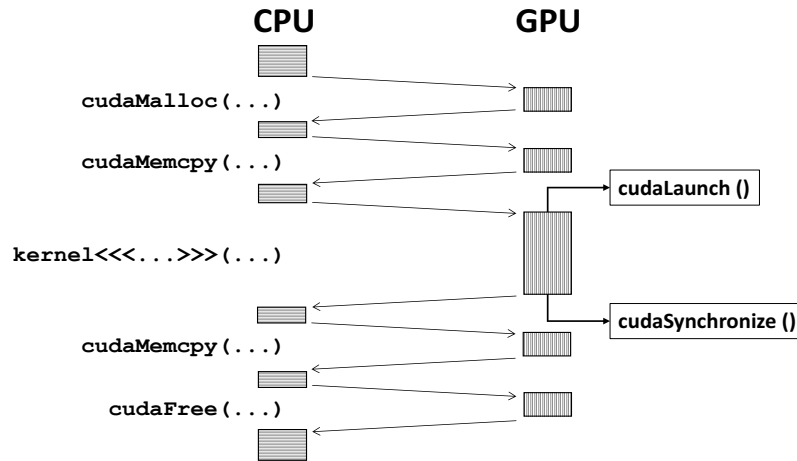
4 BWLOCK++

In this section, we provide an overview of BWLOCK++ and discuss its design details.

4.1 Overview

BWLOCK++ is a software framework to protect GPU applications on integrated CPU-GPU architecture based SoC platforms. We focus on the problem of the shared memory bandwidth contention between GPU kernels and CPU tasks in integrated CPU-GPU architectures. More specifically, we focus on protecting GPU execution intervals of real-time GPU tasks from the interference of non-critical but memory intensive CPU tasks.

In BWLOCK++, we exploit the fact that each GPU kernel is executed via explicit programming interfaces from a corresponding host CPU program. In other words, we can precisely determine when the GPU kernel starts and finishes by instrumenting these functions.

To avoid memory bandwidth contention from the CPU, we notify the OS before a GPU application launches a GPU kernel and after the kernel completes with the help of a system call. Apart from acquiring the bandwidth lock on the task's behalf, this system call also implements the priority ceiling protocol [17] to prevent preemption of the GPU using task. While the bandwidth lock is being held by the GPU task, the OS regulates memory bandwidth consumption of the best-effort CPU cores to minimize bandwidth contention with the GPU kernel. Concretely, each best-effort core is periodically given a certain amount of memory bandwidth budget. If the core uses up its given budget for the specified period, the (non-RT) CPU tasks running on that core are throttled. In this way, the GPU kernel suffers minimal memory bandwidth interference from the best-effort CPU cores. However,

Figure 3 Phases of GPU Application under CUDA Runtime.

throttling CPU cores can significantly lower the overall system throughput. To minimize the negative throughput impact, we propose a new CPU scheduling algorithm, which we call the Throttle Fair Scheduler (TFS), to minimize the duration of CPU throttling without affecting memory bandwidth guarantees for real-time GPU applications.

Figure 2 shows the overall architecture of the BWLOCK++ framework on an integrated CPU-GPU architecture (NVIDIA Jetson TX2 platform). BWLOCK++ is comprised of three major components: (1) Dynamic run-time library for instrumenting GPU applications; (2) the Throttle Fair Scheduler; (3) Per-core B/W regulator. Working together, they protect real-time GPU kernels and minimize CPU throughput reduction. We will explain each component in the following sub-sections.

4.2   Automatic Instrumentation of GPU Applications

To eliminate manual programming efforts, we automatically instrument the program binary at the dynamic linker level. We exploit the fact that the execution of a GPU application using a GPU runtime library such as NVIDIA CUDA typically follows fairly predictable patterns. Figure 3 shows the execution timeline of a typical synchronous GPU application that uses the CUDA API.

In order to protect the runtime performance of a GPU application from co-running memory intensive CPU applications, we need to ensure that the GPU application automatically holds the memory bandwidth lock while a GPU kernel is executing on the GPU or performing a memory copy operation between CPU and GPU. Upon the completion of the execution of the kernel or memory copy operation, the GPU application again shall automatically release the bandwidth lock. This is done by instrumenting a small subset of CUDA API functions that are invoked when launching or synchronizing with a GPU kernel or while performing a memory copy operation. These APIs are documented in Table 1. More specifically, we write wrappers for these functions of interest which request/release bandwidth lock on behalf of the GPU application before calling the actual CUDA library functions. We compile these functions as a shared library and use Linux' `LD_PRELOAD` mechanism [8] to force the GPU application to use those wrapper functions whenever the CUDA functions are called. In this way, we automatically throttle CPU cores' bandwidth usage whenever real-time GPU kernels are being executed so that the GPU kernels' memory bandwidth can be guaranteed.

■ **Table 1** CUDA APIs instrumented via `LD_PRELOAD` for BWLOCK++.

API	Action	Description
cudaConfigureCall	Update active streams	Specify the launch parameters for the CUDA kernel
cudaMemcpy	Acquire BWLOCK++ (*Before*) Release BWLOCK++ (*After*)	Perform synchronous memory copy between CPU and GPU
cudaMemcpyAsync	Acquire BWLOCK++ and update active streams	Perform asynchronous memory copy between CPU and GPU
cudaLaunch	Acquire BWLOCK++	Launch a GPU kernel
cudaDeviceSynchronize	Release BWLOCK++ and clear active streams	Block the calling CPU thread until all the previously requested tasks in a specific GPU device have completed
cudaThreadSynchronize	Release BWLOCK++ and clear active streams	Deprecated version of cudaDeviceSynchronize
cudaStreamSynchronize	Update active streams and release BWLOCK++ if there are no active streams	Block the calling CPU thread until all the previously requested tasks in a specific CUDA stream have completed

A complication to the automatic GPU kernel instrumentation arises when the application uses CUDA streams to launch multiple GPU kernels in succession in multiple streams and then waits for those kernels to complete. In this case, the bandwidth lock acquired by a GPU kernel launched in one stream can potentially be released when synchronizing with a kernel launched in another stream. In our framework, this situation is averted by keeping track of active streams and associating bandwidth lock with individual streams instead of the entire application whenever stream based CUDA APIs are invoked. A stream is considered active if:

- A kernel or memory copy operation is launched in that stream
- The stream has not been explicitly (using `cudaStreamSynchronize`) or implicitly (using `cudaDeviceSynchronize` or `cudaThreadSynchronize`) synchronized with

Our framework ensures that a GPU application continues holding the bandwidth lock as long as it has one or more active streams.

The obvious drawback of throttling CPU cores is that the CPU throughput may be affected especially if some of the tasks on the CPU cores are memory bandwidth intensive. In the following sub-section, we discuss the impact of throttling on CPU throughput and present a new CPU scheduling algorithm that minimizes throughput reduction.

4.3   Throttle Fair CPU Scheduler

As described earlier in this section, BWLOCK++ uses a throttling based approach to enforce memory bandwidth limit of CPU cores at a regular interval. Although effective in protecting critical GPU applications in the presence of memory intensive CPU applications, this approach runs into the risk of severely under-utilizing the system's CPU capacity; especially in cases when there are multiple best-effort CPU applications with different memory characteristics running on the best-effort CPU cores. In the throttling based design, once a core exceeds its memory bandwidth quota and gets throttled, that core cannot be used for the remainder of

the period. Let us denote the regulation period as T (i.e., $T = 1ms$) and the time instant at which an offending core exceeds its bandwidth budget as t. Then the wasted time due to throttling can be described as $\delta = T - t$ and the smaller the value of t (i.e., throttled earlier in the period) the larger the penalty to the overall system throughput. The value of t depends on the rate at which a core consumes its allocated memory budget and that in turn depends on the memory characteristics of the application executing on that core. To maximize the overall system throughput, the value of δ should be minimized—that is if throttling never occurs, $t \geq T \Rightarrow \delta = 0$, or occurs late in the period, throughput reduction will be less.

4.3.1 Negative Feedback Effect of Throttling on CFS

One way to reduce CPU throttling is to schedule less memory bandwidth demanding tasks on the best-effort CPU cores while the GPU is holding the bandwidth lock. Assuming that each best-effort CPU core has a mix of memory bandwidth intensive and CPU intensive tasks, then scheduling the CPU intensive tasks while the GPU is holding the lock would reduce CPU throttling or at least delay the instant at which throttling occurs, which in turn would improve CPU throughput. Unfortunately, Linux's default scheduler CFS [13] actually aggravates the possibility of early and frequent throttling when used with BWLOCK++'s throttling mechanism.

The CFS algorithm tries to allocate fair amount of CPU time among tasks by using each task's weighted virtual runtime (i.e., weighted execution time) as the scheduling metric. Concretely, a task τ_i's virtual runtime V_i is defined as

$$V_i = \frac{E_i}{W_i} \tag{1}$$

where E_i is the actual runtime and W_i is the weight of the task. The CFS scheduler simply picks the task with the smallest virtual runtime.

The problem with memory bandwidth throttling under CFS arises because the virtual run-time of a memory intensive task, which gets frequently throttled, increases more slowly than the virtual run-time of a compute intensive task which does not get throttled. Due to this, the virtual runtime based arbitration of CFS tends to schedule the memory intensive tasks more than the CPU intensive tasks while bandwidth regulation is in place.

4.3.2 TFS Approach

In order to reduce the throttling overhead while keeping the undesirable scheduling of memory intensive tasks quantifiable, TFS modifies the throttled task's virtual runtime to take the task's throttled duration into account. Specifically, at each regulation period, if there exists a throttled task, we scale the throttled duration of the task by a factor, which we call TFS punishment factor, and add it to its virtual runtime.

Under TFS, a throttled task τ_i's virtual runtime V_i^{new} at the end of jth regulation period is expressed as:

$$V_i^{new} = V_i^{old} + \delta_i^j \times \rho \tag{2}$$

where δ_i^j is the throttled duration of τ_i in the jth sampling period, and ρ is the TFS punishment factor.

The more memory intensive a task is, the more likely the task get throttled in each regulation period for a longer duration of time (i.e., higher δ_i). By adding the throttled time back to the task's virtual runtime, we make sure that the memory intensive tasks are not

■ **Table 2** Taskset for Example.

Task	Compute Time (C)	Period (P)	Description
τ_{RT}	4	15	Real-time task
τ_{MEM}	4	N/A	Memory intensive best-effort task
τ_{CPU}	4	N/A	CPU intensive best-effort task

favored by the scheduler. Furthermore, by adjusting the TFS punishment factor ρ, we can further penalize memory intensive tasks in favor of CPU intensive ones. This in turn reduces the amount of throttled time and improves overall CPU utilization. On the other hand, the memory intensive tasks will still be scheduled (albeit less frequently so) according to the adjusted virtual runtime. Thus, no tasks will suffer starvation.

Scheduling of tasks under TFS is fair with respect to the adjusted virtual runtime metric but it can be considered unfair with respect to the CFS's original virtual runtime metric. A task τ_i's "lost" virtual runtime Δ_i^{TFS} (due to TFS's inflation) over J regulation periods can be quantified as follows:

$$\Delta_i^{TFS} = \sum_{j=0}^{J} \delta_i^j \times \rho. \tag{3}$$
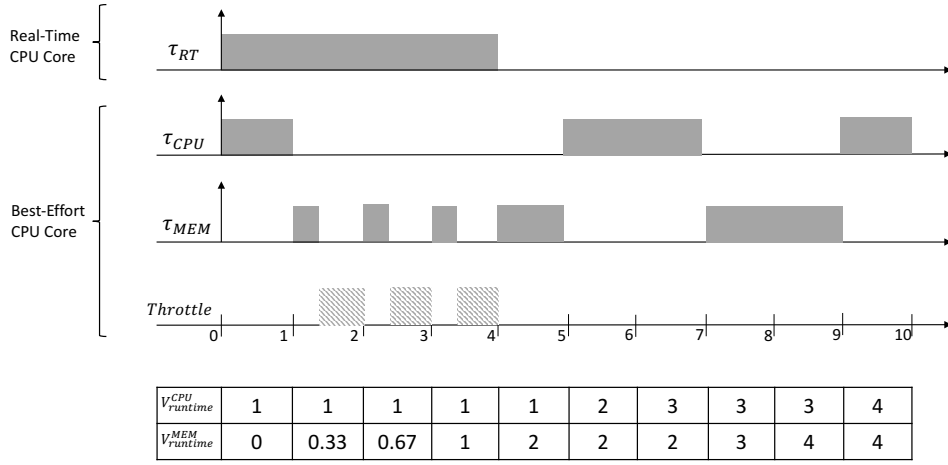
4.3.3 Illustrative Example

We elaborate the problem of CFS and the benefit of our TFS extension with a concrete illustrative example.

Let us consider a small integrated CPU-GPU system, which consists of two CPU cores and a GPU. We further assume, following our system model, that Core-1 is a real-time core, which may use the GPU, and Core-2 is a best-effort core, which doesn't use the GPU.
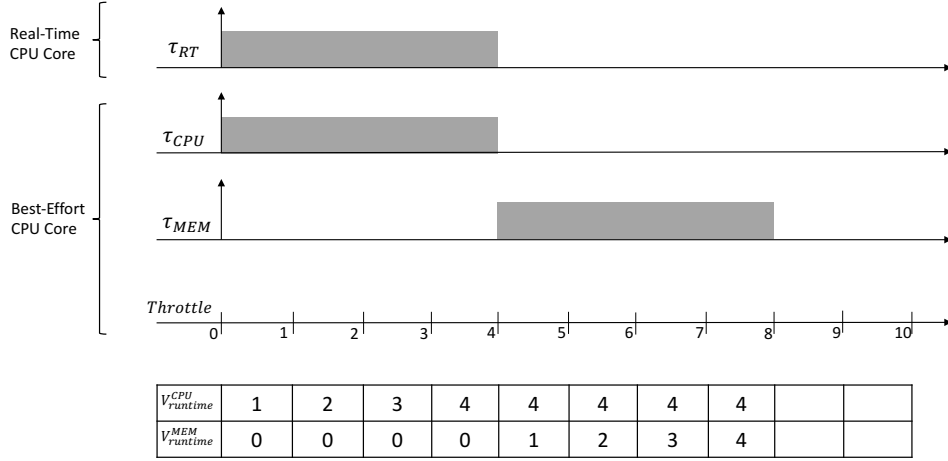
Table 2 shows a taskset to be scheduled on the system. The taskset is composed of a GPU using real-time task, which needs to be protected by our framework for the entire duration of its execution; and two best-effort tasks (of equal CFS priority), one of which is CPU intensive and the other is memory intensive.

Figure 4a shows how the scheduling would work when CFS is used to schedule best-effort tasks τ_{CPU} and τ_{MEM} on the best-effort core with its memory bandwidth is throttled by our kernel-level bandwidth regulator. Note that in this example, both OS scheduler tick timer interval and the bandwidth regulator interval are assumed to be 1ms. At time 0, τ_{CPU} is first scheduled. Because τ_{CPU} is CPU bound, it doesn't suffer throttling. At time 1, the CFS schedules τ_{MEM} as its virtual runtime 0 is smaller than τ_{CPU}'s virtual runtime 1. Shortly after the τ_{MEM} is scheduled, however, it gets throttled at time 1.33 as it has used the best-effort core's allowed memory bandwidth budget for the regulation interval. When the budget is replenished at time 2, at the beginning of the new regulation interval, the τ_{MEM}'s virtual runtime is 0.33 while τ_{CPU} is 1. So, the CFS picks the τ_{MEM} (smaller of the two) again, which gets throttled again. This pattern continues until the τ_{MEM}'s virtual runtime finally catches up with τ_{CPU} at time 4 by which point the best-effort core has been throttled 66% of time between time 1 and 4. As can be seen in this example, CFS favors memory intensive tasks as their virtual runtimes increase more slowly than CPU intensive ones when memory bandwidth throttling is used.
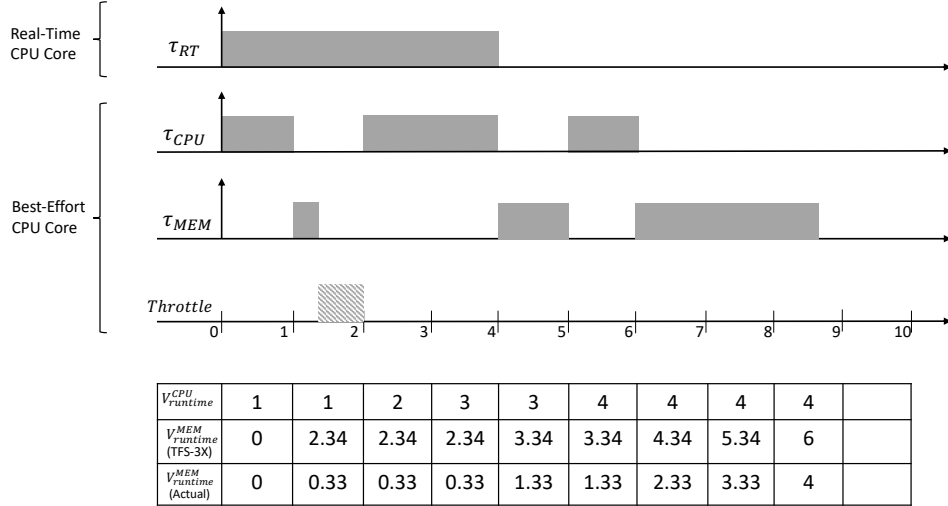
Figure 4b shows a hypothetical schedule in which the execution of τ_{MEM} is delayed in favor of the τ_{CPU} while τ_{RT} is running (thus, memory bandwidth regulation is in place.)

(a) Example schedule under CFS with 1-msec scheduling tick.

$V_{runtime}^{CPU}$	1	1	1	1	1	2	3	3	3	4
$V_{runtime}^{MEM}$	0	0.33	0.67	1	2	2	2	3	4	4

(b) Example schedule with zero throttling.

$V_{runtime}^{CPU}$	1	2	3	4	4	4	4	4		
$V_{runtime}^{MEM}$	0	0	0	0	1	2	3	4		

(c) Example schedule under TFS with $\rho = 3$.

$V_{runtime}^{CPU}$	1	1	2	3	3	4	4	4	4	
$V_{runtime}^{MEM}$ (TFS-3X)	0	2.34	2.34	2.34	3.34	3.34	4.34	5.34	6	
$V_{runtime}^{MEM}$ (Actual)	0	0.33	0.33	0.33	1.33	1.33	2.33	3.33	4	

Figure 4 Example schedules under different scheduling schemes.

(a) CFS.   **(b)** TFS.   **(c)** TFS-3X.

■ **Figure 5** Virtual runtime progress of the two synthetic tasks. One is cpu-intensive and the other is memory-intensive.

In this case, because τ_{CPU} never exhausts the memory bandwidth budget, it never gets throttled. As a result, the best-effort core never experiences throttling and thus is able to achieve high throughput. While this is ideal behavior from the perspective of throughput, it may not be ideal for the τ_{MEM} as it can suffer starvation.

Figure 4c shows the schedule under the TFS (with a TFS punishment factor $\rho = 3$). The TFS works identical to CFS until at time 2, when the BWLOCK++'s periodic timer is called. At this point, the τ_{MEM}'s virtual runtime (V^{MEM}) is 0.33ms. However, because it has been throttled for 0.67ms during the regulation period ($\delta = 0.67$), according to Equation 2, TFS increases the task's virtual runtime to 2.34 ($V^{MEM} + \delta \times \rho = 0.33 + 0.67 \times 3 = 2.34$). Because of the increased virtual runtime, the TFS scheduler then picks τ_{CPU} as its virtual runtime is now smaller than that of τ_{MEM} ($1 < 2.34$). Later, when the τ_{CPU}'s virtual runtime becomes 3 at time 4, the TFS scheduler can finally re-schedule the τ_{MEM}. In this manner, TFS favors CPU intensive tasks over memory-intensive ones, while preventing starvation of the latter. Note that TFS works at each regulation period (i.e., 1ms) independently and thus automatically adapts to the task's changing behavior. For example, if a task is memory intensive only for a brief period of time, the task will be throttled only for the memory intensive duration, and the throttled time will be added back to the task's virtual runtime at each 1ms regulation period. Furthermore, even for a period when a task is throttled, the task always makes small progress as allowed by the memory bandwidth budget for the period. Therefore, no task suffers complete starvation for an extended period of time.

4.3.4   Effects of TFS using Synthetic Tasks

We experimentally validate the effect of TFS in scheduling best-effort tasks on a real system. In this experiment, we use two synthetic tasks: one is CPU intensive and the other is memory-intensive. We use *Bandwidth* benchmark for both of these tasks. In order to make *Bandwidth* memory intensive, we configure its working-set size to be twice the size of LLC on our platform. Similarly, to make *Bandwidth* compute (CPU) intensive, we make its working set size one half the size of L1 data cache in our platform. We assign these two best-effort tasks on the same best-effort core, which is bandwidth regulated with a 100 MB/s memory bandwidth budget.
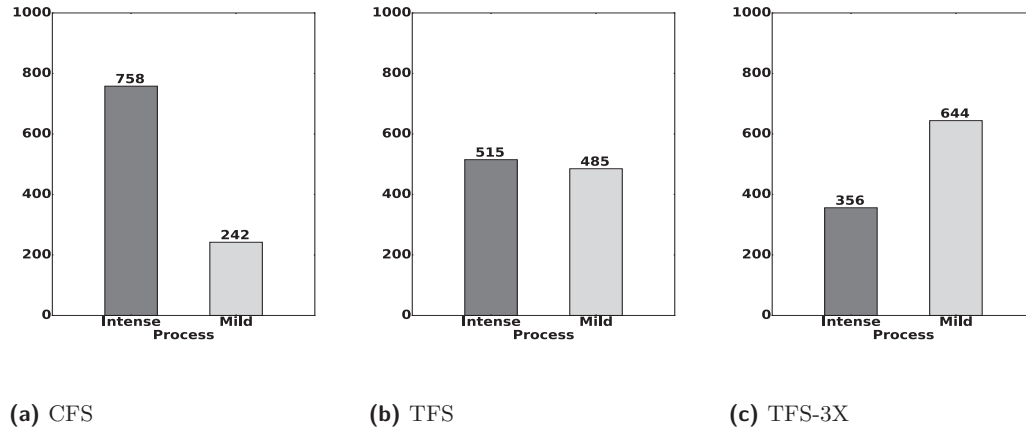
(a) CFS   **(b)** TFS   **(c)** TFS-3X

Figure 6 The number of periods during which the two tasks are scheduled. 'Intense' refers to the memory-intensive task. 'Mild' refers to the CPU-intensive task.

Figure 5 shows the virtual runtime progression over 1000 sampling periods of the two tasks under three scheduler configurations: CFS, TFS ($\rho = 1$), and TFS-3X ($\rho = 3$). In CFS, the memory intensive process gets preferred by the CFS scheduler at each scheduling instance, because its virtual run-time progresses more slowly. In TFS and TFS-3X, however, as memory-intensive task's virtual runtime is increased, CPU-intensive task is scheduled more frequently.

This can be seen more clearly in Figure 6, which shows the number of periods utilized by each task on the CPU core, over the course of one thousand sampling periods. Under CFS, out of all the sampling periods, 75% are utilized by the memory intensive process and only 25% are utilized by the compute intensive process. With TFS, the two tasks get to run in roughly the same number of sampling periods whereas in TFS-3x, the CPU intensive task gets to run more than the memory intensive task.

5 Implementation

In this section, we describe the implementation details of BWLOCK++.

5.1 BWLOCK++ System Call

We add a new system call `sys_bwlock` in Linux kernel 4.4.38. The system call serves two purposes. 1) It acquires or releases the memory bandwidth lock on behalf of the currently running task on the real-time core; and 2) it implements a priority-ceiling protocol, which boosts the calling task's priority to the system's ceiling priority, to prevent preemption. We introduce two new integer fields, `bwlock_val`, `bw_old_priority`, in the task control block: `bwlock_val` stores the current status of the memory bandwidth lock and `bw_old_priority` keeps track of the original real-time priority of the task while it is holding the bandwidth lock.

Algorithm 1 shows the implementation of the system call. To acquire the memory bandwidth lock, the system call must be invoked from the real-time system core and the task currently scheduled on the real-time core must have a real-time priority (*line 2*). At the time of acquisition of bandwidth lock, the priority of the calling task, which is tracked by

---

Algorithm 1: BWLOCK++ System Call.

1 syscall sys_bwlock(bw_val**)**

2   **if** $smp\_processor\_id$ $() == $ RT_CORE_ID $\wedge$ rt_task $(current)$ **then**

3   rt_core_data := get_rt_core_data ()

4   rt_core_data $\rightarrow$ current_task := current

5   **if** $bw\_val \geq 1$ **then**

6   current $\rightarrow$ bwlock_val := 1

7   current $\rightarrow$ bw_old_priority := current $\rightarrow$ rt_priority

8   current $\rightarrow$ rt_priority := MAX_USER_RT_PRIO - 1

9   **else**

10   current $\rightarrow$ bwlock_val := 0

11   current $\rightarrow$ rt_priority := current $\rightarrow$ bw_old_priority

12   **end**

13   **end**

14 return;

---

the globally accessible `current` pointer in Linux kernel, is raised to the maximum allowed real-time priority value (the ceiling priority) for any user-space task to prevent preemption (*line 7*). The real-time priority value of the the task is restored to its original priority value when the bandwidth lock is released (*line 10*). In this manner, the system call updates the state of the currently scheduled real-time task on the real-time system core, which is then used by the memory bandwidth regulator on best-effort cores to enforce memory usage thresholds, as explained in the following subsection.

5.2   Per-Core Memory Bandwidth Regulator

The per-core memory bandwidth regulator is composed of a periodic timer interrupt handler and a performance monitoring counter (PMC) overflow interrupt handler. Algorithm 2 shows the implementation of the memory bandwidth regulator.

The periodic timer interrupt handler is invoked at a periodic interval (currently every 1 msec) using a high resolution timer in each best-effort core. The timer handler begins a new bandwidth lock regulation period and performs the following operations:

- Unthrottle the core if it was throttled in the last regulation period (*line 3*)

- Scale the virtual runtime of the task currently scheduled on the core based on the throttling time in the last period and the TFS punishment factor (*line 4-5*)

- Determine the new memory usage budget based on the bandwidth lock status of the task currently scheduled on the real-time system core (*line 7-12*)

- Program the performance monitoring counter on the core based on the new memory usage budget for the current regulation period (*line 13*). We use the *L2D_CACHE_REFILL* event for measuring the memory bandwidth traffic in ARM Cortex-A57 processor core

The PMC overflow interrupt occurs when the core at hand exceeds its memory usage budget in the current regulation period. The interrupt handler prevents further memory transactions from this core by scheduling a high priority idle kernel thread on it for the remainder of the regulation period (*line 17*).

Algorithm 2: Memory Bandwidth Regulator.

1 procedure `periodic_interrupt_handler`(*core_data*)
2 | **if** *core_is_throttled (core_data → core_id) == TRUE* **then**
3 | | unthrottle_core (core_data → core_id)
4 | | record_throttling_end_time (core_data → current_task)
5 | | scale_virtual_runtime (core_data → current_task)
6 | **end**
7 | rt_core_data := get_rt_core_data ()
8 | **if** *rt_core_data → current_task → bwlock_val == 1* **then**
9 | | core_data → new_budget := rt_core_data → throttle_budget
10 | **else**
11 | | core_data → new_budget := `MAX_BANDWIDTH_BUDGET`
12 | **end**
13 | program_pmc (core_data → new_budget)
14 return;

15 procedure `pmc_overflow_handler`(*core_data*)
16 | record_throttling_start_time (core_data → current_task)
17 | throttle_core (core_data → core_id)
18 return;
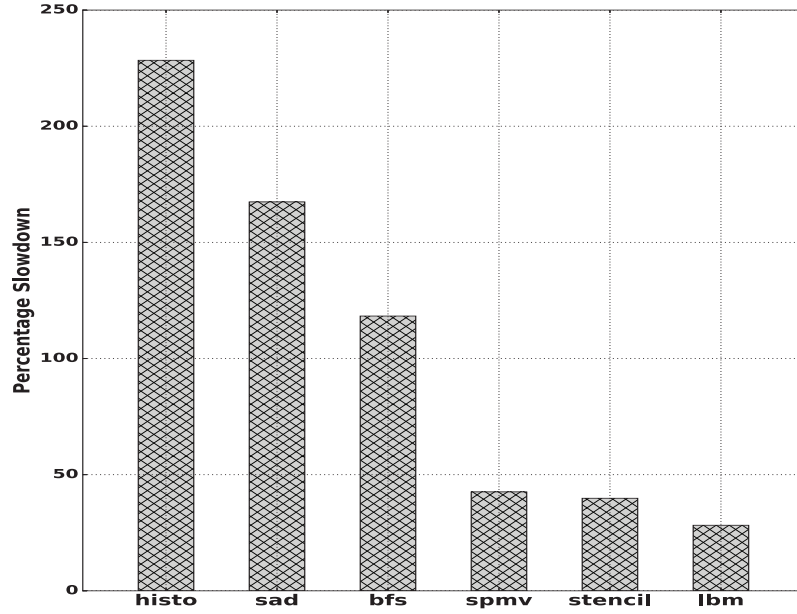
---

6   Evaluation

In this section, we present the experimental evaluation results of BWLOCK++.

6.1   Setup

We evaluate BWLOCK++ on NVIDIA Jetson TX2 platform. We use the Linux kernel version 4.4.38, which is patched with the changes required to support BWLOCK++. The CUDA runtime library version installed on the platform is 8.0, which is the latest version available for Jetson TX2 at the time of writing. In all our experiments, we place the platform in maximum performance mode by maximizing GPU and memory clock frequencies and disabling the dynamic frequency scaling of CPU cores. We also shutdown the graphical user interface and disable the network manager to avoid run to run variation in the experiments. As per our system model, we designate the Core-0 in our system as real-time core. The remaining cores execute best-effort tasks only. All the tasks are statically assigned to their respective cores during the experiment. While NVIDIA Jetson TX2 platform contains two CPU islands, a quad-core Cortex-A57 and a dual-core Denver, we only use the Cortex-A57 island for our evaluation and leave the Denver island off because we were unable to find publicly available documentation regarding the Denver cores' hardware performance counters, which is needed to implement throttling. In order to evaluate BWLOCK++, we use six benchmarks from parboil suite which are listed as memory bandwidth sensitive in [18].

Table 3 GPU execution time breakdown of selected benchmarks.

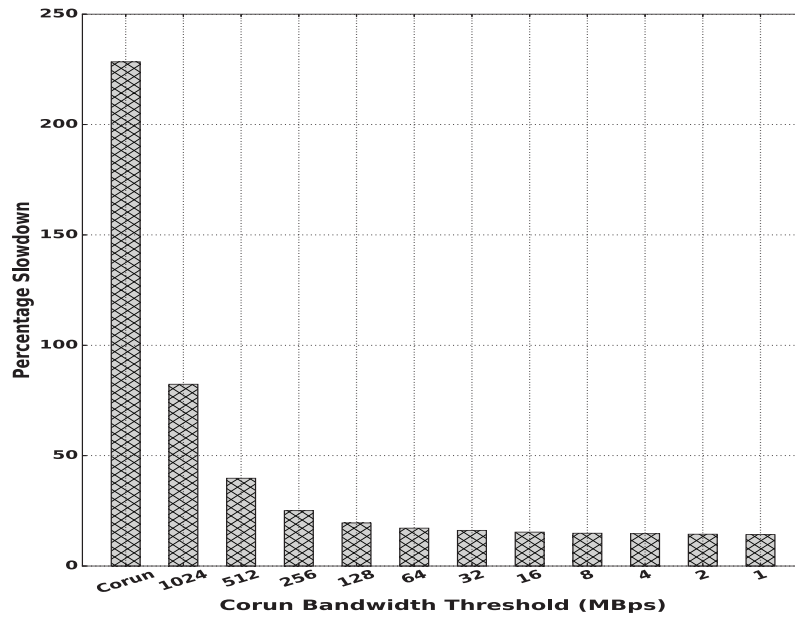| Benchmark	Dataset	Copy (KBytes)	Timing Breakdown (msec)			
			Kernel (G^e)	Copy (G^m)	Compute (C)	Total (E)
histo	Large	5226	83409	18	0	83428
sad	Large	709655	152	654	53	861
bfs	1M	62453	174	72	0	246
spmv	Large	30138	69	51	10	131
stencil	Default	196608	749	129	9	888
lbm	Long	379200	43717	358	2004	46080

Figure 7 Slowdown of the total execution time of GPU benchmarks due to three *Bandwidth* corunners.

6.2 Effect of Memory Bandwidth Contention

In this experiment, we investigate the effect of memory bandwidth contention due to co-scheduled memory intensive CPU applications on the evaluated GPU kernels.

First, we measure the execution time of each GPU benchmark in isolation. From this experiment, we record the GPU kernel execution time (G^e), memory copy time for GPU kernels (G^m) and CPU compute time (C) for each benchmark. The data collected is shown in Table 3. We then repeat the experiment after co-scheduling three instances of a memory intensive CPU application as co-runners. We use the *Bandwidth* benchmark from the IsolBench suite [19] as the memory intensive CPU benchmark, which updates a big 1-D array sequentially. The sequential write access pattern of the benchmark is known to cause worst-case interference on several multicore platforms [20].

The results of this experiment are shown in Figure 7 and they demonstrate how much the total execution time of GPU benchmarks ($E = G^e + G^m + C$) suffers from memory bandwidth contention due to the co-scheduled CPU applications.

Figure 8 Effect of corun bandwidth threshold on the execution time of histo benchmark.

From Figure 7, it can be seen that the worst case slowdown, in case of *histo* benchmark, is more than 250%. Similarly, for SAD benchmark, the worst case slowdown is more than 150%. For all other benchmarks, the slowdown is non-zero and can be significant in affecting the real-time performance. These results clearly show the danger of uncontrolled memory bandwidth sharing in an integrated CPU-GPU architecture as GPU kernels may potentially suffer severe interference from co-scheduled CPU applications. In the following experiment, we investigate how this problem can be addressed by using BWLOCK++.

6.3 Determining Memory Bandwidth Threshold

In order to apply BWLOCK++, we first need to determine safe memory budget that can be given to the best-effort CPU cores in the presence of GPU applications. However, an appropriate threshold value may vary depending on the characteristics of individual GPU applications. If the threshold value is set too high, then it may not be able to protect the performance of the GPU application. On the other hand, if the threshold value is set too low, then the CPU applications will be throttled more often and that would result in significant CPU capacity loss.

We calculate the safe memory budget for best-effort CPU cores by observing the trend of slowdown of the total execution time of GPU application as the allowed memory usage threshold of CPU co-runners is varied. We start with a threshold value of 1-GB/s for each best-effort CPU core. We then continue reducing the threshold value for best-effort cores by half and measure the impact of this reduction on the slowdown of execution time (E) of the benchmark.

Figure 8 shows this trend for the execution time of *histo* benchmark from the parboil suite. From the figure, it can be seen that after `64-MBps` threshold value for best-effort CPU cores, further reduction of threshold value does not yield significant improvement in reducing the slowdown of benchmark. Hence, for *histo* benchmark, we select `64-MBps` as the threshold value for the best-effort CPU cores. In a similar fashion, we plot this trend for all the selected benchmarks and determine the value of corun threshold for the best-effort CPU cores.
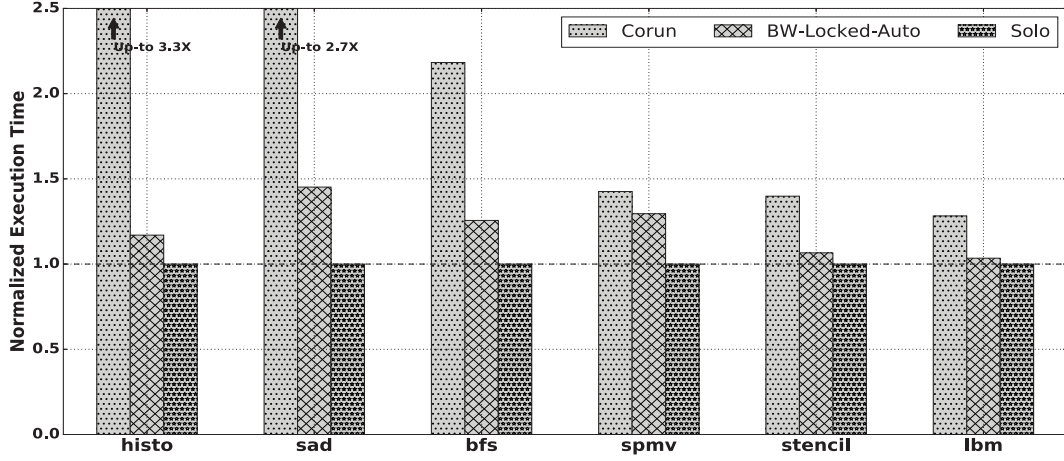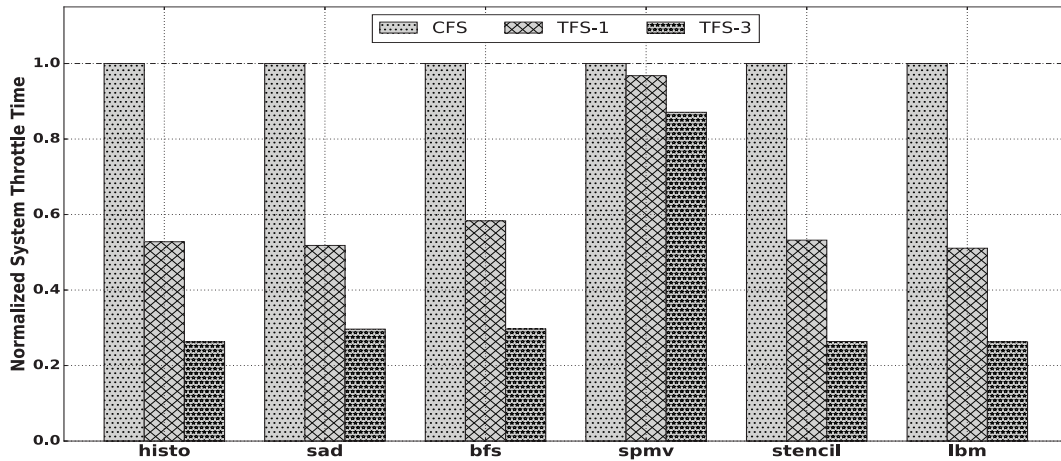
Figure 9 BWLOCK++ Evaluation Results.

6.4   Effect of BWLOCK++

In this experiment, we evaluate the performance of BWLOCK++. Specifically, we record the corun execution of GPU benchmarks with the automatic instrumentation of BWLOCK++. We call this scenario *BW-Locked-Auto*. We compare the performance under *BW-Locked-Auto* against the *Solo* and *Corun* execution of the GPU benchmarks which represent the measured execution times in isolation and together with three co-scheduled memory intensive CPU applications, respectively.

To get the data-points for *BW-Locked-Auto*, we configure BWLOCK++ according to the allowed memory usage threshold of the benchmark at hand and use our dynamic GPU kernel instrumentation mechanism to launch the benchmark in the presence of three *Bandwidth* benchmark instances (write memory access pattern) as CPU co-runners. The results of this experiment are plotted in Figure 9. In Figure 9, we plot the total execution time of each benchmark for the above mentioned scenarios. All the time values are normalized with respect to the total execution time ($E_{solo} = C_{solo} + G^e_{solo} + G^m_{solo}$) of the benchmark in isolation. As can be seen from this figure, execution under *BW-Locked-Auto* incurs significantly less slowdown of the total execution time of GPU benchmarks due to reduction of both GPU kernel execution time and memory copy operation time.

6.5   Throughput improvement with TFS

As explained in Section 4.3, throttling under CFS results in significant system throughput reduction. In order to illustrate this, we conduct an experiment in which the GPU benchmarks are executed with six CPU co-runners. Each CPU core, apart from the one executing the GPU benchmark, has a memory intensive application and a compute intensive application scheduled on it. For both of these applications, we use the *Bandwidth* benchmark with different working set sizes. In order to make *Bandwidth* memory intensive, we configure its working set size to be twice the size of LLC on our evaluation platform. Similarly for compute intensive case, we configure the working set size of *Bandwidth* to be half of the L1-data cache size. We record the total system throttle time statistics with BWLOCK++ for all the GPU benchmarks. The total system throttle time is the sum of throttle time across all system cores. We then repeat the experiment with our Throttle Fair Scheduling scheme. In *TFS-1*, we configure the TFS punishment factor as one for the memory intensive threads

Figure 10 Comparison of total system throttle time under different scheduling schemes.

and in *TFS-3*, we set this factor to three. We plot the normalized total system throttle time
for all the scheduling schemes and present them in Figure 10. It can be seen that TFS results
in significantly less system throttling (On average, 39% with *TFS-1* and 62% with *TFS-3*)
as compared to CFS.

6.6 Overhead due to BWLOCK++

The overhead incurred by real-time GPU applications due to BWLOCK++ comes from the
following sources:

- `LD_PRELOAD` overhead for CUDA API instrumentation
- Overhead due to BWLOCK++ system call

The overhead due to `LD_PRELOAD` is negligible since we cache CUDA API symbols for
all the instrumented functions inside our shared library; after searching for them only once
through the dynamic linker. We calculate the overhead incurred due to BWLOCK++ system
call by executing the system call one million times and taking the average value. In NVIDIA
Jetson TX2, the average overhead due to each BWLOCK++ system call is $1.84 usec$. Finally,
we experimentally determine the overhead value for all the evaluated benchmarks by running
the benchmark in isolation with and without BWLOCK++. Our experiment shows that for
all the evaluated benchmarks, the total overhead due to BWLOCK++ is less than 1% of the
total solo execution time of the benchmark.

7 Schedulability Analysis

As we limit the scheduling of real-time tasks on a single real-time core, our system can
be analyzed using the classical unicore based response time analysis for preemptive fixed
priority scheduling with blocking [3], because we model each GPU execution segment as
a critical section, which is protected by acquiring and releasing the bandwidth lock. The
bandwidth lock serializes GPU execution and regulates memory bandwidth consumption of
co-scheduled best-effort CPU tasks. The bandwidth lock implements the standard priority
ceiling protocol [17], which boosts the priority of the lock holding task (i.e., the task executing
a GPU kernel) to the ceiling priority of the lock, which is the highest real-time priority of

the system, so as to prevent preemption. With this constraint, a real-time task τ_i's response time is expressed as:

$$R_i^{n+1} = E_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{P_j} \right\rceil E_j \tag{4}$$

where $hp(i)$ represents the set of higher priority tasks than τ_i and B_i is the longest GPU kernel or copy duration—protected by the memory bandwidth lock—of one of the lower priority tasks.

The benefit of BWLOCK++ lies in the reduction of worst-case GPU kernel execution or GPU memory copy interval of real-time tasks (which would in turn reduce E_i and B_i terms in Equation 4). As shown in Section 6.2, without BWLOCK++, GPU execution of a task can suffer severe slowdown (up to 230% slowdown in our evaluation), which would result in pessimistic WCET estimation for GPU kernel and copy execution times, hampering schedulability of the system. BWLOCK++ helps reduce pessimism of GPU execution time estimation and thus improves schedulability.

8 Discussion

Our approach has following limitations. First, we assume that all real-time tasks are scheduled on a single dedicated real-time core while the rest of the cores only schedule best-effort tasks. In addition, we assume only real-time tasks can utilize the GPU while best-effort tasks cannot. While restrictive, recall that scheduling multiple GPU using real-time tasks on a single dedicated real-time core does not necessarily reduce GPU utilization because multiple GPU kernels from different tasks (processes) are serialized at the GPU hardware anyway [15] as we already discussed in Section 3. Also, due to the capacity limitation of embedded GPUs, it is expected that a few GPU using real-time task can easily achieve high GPU utilization in practice. We claim that our approach is practically useful for situations where a small number of GPU accelerated tasks are critical, for example, a vision-based automatic braking system.

Second, we assume that GPU applications are given a priori and they can be profiled in advance so that we can determine proper memory bandwidth threshold values. If this assumption cannot be satisfied, an alternative solution is to use a single threshold value for all GPU applications, which eliminates the need of profiling. But the downside is that it may lower the CPU throughput because the memory bandwidth threshold must be conservatively set to cover all types of GPU applications.

9 Conclusion

In this paper, we presented BWLOCK++, a software based mechanism for protecting the performance of GPU kernels on platforms with integrated CPU-GPU architectures.

BWLOCK++ automatically instruments GPU applications at run-time and inserts a memory bandwidth lock, which throttles memory bandwidth usage of the CPU cores to protect performance of GPU kernels. We identified a side effect of memory bandwidth throttling on the performance of Linux default scheduler CFS, which results in the reduction of overall system throughput. In order to solve the problem, we proposed a modification to CFS, which we call Throttle Fair Scheduling (TFS) algorithm. Our evaluation results have shown that BWLOCK++ effectively protects the performance of GPU kernels from memory intensive CPU co-runners. Also, the results showed that TFS improves system throughput,

compared to CFS, while protecting critical GPU kernels. In the future, we plan to evaluate BWLOCK++ on other integrated CPU-GPU architecture based platforms. We also plan to extend BWLOCK++ not only to protect critical GPU tasks but also to protect critical CPU tasks.

References

1 Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page placement strategies for gpus within heterogeneous memory systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

2 Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

3 N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

4 Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. Sigamma: Server based gpu arbitration mechanism for memory accesses. In *International Conference on Real-Time Networks and Systems (RTNS)*, 2017.

5 NVIDIA Corp. Nvidia jetson platforms. `https://developer.nvidia.com/embedded-computing`.

6 Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. Gpusync: A framework for real-time gpu management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

7 Björn Forsberg, Andrea Marongiu, and Luca Benini. Gpuguard: Towards supporting a predictable execution model for heterogeneous soc. In *Design, Automation & Test in Europe (DATE)*, 2017.

8 Greg Kroah Hartman. Modifying a dynamic library without changing the source code | linux journal. `http://www.linuxjournal.com/article/7795`.

9 Shinpei Kato, Karthik Lakshmanan, Ragunathan (Raj) Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference (ATC)*, 2011.

10 Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Brandt Scott. Gdev: First-class gpu resource management in the operating system. In *USENIX Annual Technical Conference (ATC)*, 2012.

11 Shinpei Kato, Eijiro Takeuchi, Yoshiki Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6):60–68, 2015.

12 Hyoseung Kim, Pratyush Patel, Shige Wang, and Ragunathan (Raj) Rajkumar. A server based approach for predictable gpu access control. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017.

13 Ingo Molnar. Modular scheduler core and completely fair scheduler. `https://lwn.net/Articles/230501`.

14 Nathan Otterness, Ming Yang, Sarah Rust, and Eunbyun Park. Inferring the scheduling policies of an embedded cuda gpu. In *Workshop on Operating Systems Platforms for Embedded Real Time Systems Applications (OSPERT)*, 2017.

15 Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alexander C. Berg, and Shige Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

16   Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

17   Lui Sha, Ragunathan (Raj) Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

18   John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois at Urbana-Champaign, 2012.

19   Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

20   Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Addressing isolation challenges of non-blocking caches for multicore real-time systems. *Real-Time Systems*, 53(5):673–708, 2017.

21   Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. *IEEE Transactions on Computers (TC)*, PP(99):1–1, 2016.

22   Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

23   Husheng Zhou, Guangmo Tong, and Cong Liu. Gpes: a preemptive execution system for gpgpu computing. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.