BandWatch: A System-Wide Memory Bandwidth Regulation System for Heterogeneous Multicore

Eric Seals
University of Kansas
erjseals@gmail.com

Michael Bechtel
University of Kansas
mbechtel@ku.edu

Heechul Yun
University of Kansas
heechul.yun@ku.edu

***Abstract*—Unpredictable variation in execution times due to contention in shared hardware resources in integrated CPU-GPU multicore platforms remains a major challenge for safety-critical real-time systems. We present BandWatch, a system-wide memory bandwidth regulation system, which dynamically regulates both CPU cores and the GPU to protect the performance of critical real-time tasks while minimizing the negative throughput impact to the throttled non real-time tasks. BandWatch is implemented on Linux as a kernel module on a NVIDIA Jetson Nano platform. The extensive evaluation results using both real-world and synthetic workloads show the effectiveness of BandWatch.**

I. Introduction

The demand for embedded hardware capable of executing complex applications has led to an increased adoption of heterogeneous multicore platforms that integrate multiple CPU cores and a GPU on a single chip. These platforms can deliver higher throughput while minimizing the size, weight, and power (SWaP) requirements of the system. However, it is challenging to provide predictable timing guarantees needed to design safety-critical real-time systems on such heterogeneous hardware platforms.

One major cause of performance variation stems from the existence of many shared hardware resources, such as shared cache and DRAM banks, which can be contended when multiple tasks try to access them simultaneously. The effect of such contention can be severe – orders of magnitude slowdowns in some extreme cases [8], [10], [16], [29] – which is especially problematic for safety-critical systems, such as cars and airplanes, that often require strong isolation for certification [13], [14].

While mitigation of shared resource contention in multicore has been extensively studied in the real-time systems community [3], [8], [19], [24], [33], [35], by either partitioning the resources or rate throttling, relatively few works have focused on heterogeneous multicore platforms [3], [28] and, to our knowledge, none can support holistic memory bandwidth throttling of both CPU cores and the integrated GPU (iGPU) on a commercial off-the-shelf (COTS) heterogeneous multicore.

In this paper, we focus on mitigating the shared resource contention problem in heterogeneous multicore platforms, particularly contention in shared main memory, and implement a holistic system-wide memory bandwidth regulation system, which we call BandWatch, on a popular COTS heterogeneous multicore platform, namely Jetson Nano [22]. This system leverages available hardware features in the Nano's Tegra X1 SoC to monitor the memory controller's utilization and to selectively throttle the memory bandwidth from the integrated GPU and other accelerators [22]. In addition, BandWatch leverages and extends an existing software-based memory throttling mechanism called MemGuard [35] to monitor and control the memory bandwidth at a per core level within the CPU complex. By combining these two mechanisms, BandWatch can provide strong isolation guarantees to a critical real-time (RT) task while still allowing simultaneous accesses to the shared resources from non real-time (NRT) tasks running on different CPU cores or the GPU with adaptive bandwidth throttling of the NRT tasks.

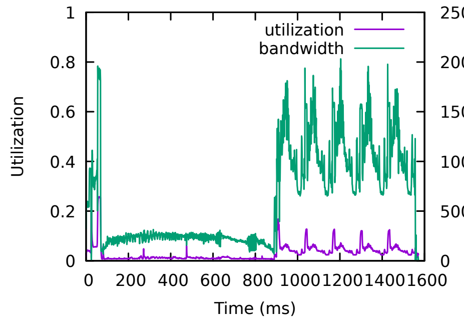


Fig. 1: Memory usage trace of SD-VBS [30] *mser* benchmark

BandWatch's adaptive regulation scheme is motivated by the observation that memory bandwidth throttling of NRT tasks is not always needed to protect the RT task. For example, Figure 1 shows the memory utilization (memory controller busy cycles over a sampling period) and the memory bandwidth (in MB/s) of the SD-VBS [30] benchmark *mser* over the duration of its execution time. Note that, for a significant duration of the runtime, there is limited memory traffic, followed by a section of periodically peaking requests. A static regulation approach to this task would either weaken *mser*'s own peak bandwidth potential or need to reduce the memory bandwidth of the other applications on the system.

Our adaptive throttling approach tries to minimize unnecessary throttling by carefully monitoring the memory usage of both CPU cores and the GPU holistically. There are two major challenges to realize adaptive throttling: the first is the lack of mechanisms to throttle the memory traffic of the integrated

GPU. Fortunately, there exists detailed documentation for the Tegra X1 SoC, which include descriptions on its hardware-based throttling capability in the memory controller. From the documentation, we were able to implement a Linux kernel driver that directly controls the memory controller's registers to realize the GPU throttling capability. Second, unlike homogeneous multicore processors, a heterogeneous multicore may implement a request prioritization scheme that favors CPU cores over GPU in accessing memory, due to different latency and bandwidth sensitivities between CPU and GPU workloads. Therefore, a single metric based adaptive strategy designed for homogeneous multicore (e.g., [24]) may not work well on a heterogeneous multicore SoC.

In summary, this paper makes the following **contributions**:

- We propose BandWatch, a holistic system-wide memory bandwidth regulation system for heterogeneous multicore platforms.
- We implement hardware- and software-based GPU and CPU throttling capabilities, respectively, and an adaptive throttling strategy to protect critical RT tasks with minimal throughput impact to NRT tasks.
- We show our approach achieves good isolation for RT tasks while significantly improving throughput of NRT tasks through extensive experiments on a real platform.

To our knowledge, BandWatch is the first software system that implements an adaptive memory bandwidth regulation of both CPU and iGPU on a COTS heterogeneous multicore platform.

II. Background

In this section, we provide necessary background on NVIDIA Tegra X1 SoC, which supports hardware-based memory bandwidth throttling of the GPU, and MemGuard, which provides software-based memory throttling of the individual CPU cores.

A. NVIDIA Tegra X1 SoC

In this work, we use the the NVIDIA Jetson Nano hardware platform, which is equipped with NVIDIA's Tegra X1 MPSoC chip. The X1 chip features a quad-core ARM Cortex-A57 CPU and a 128-core GPU based on Maxwell architecture. Importantly, X1 provides memory utilization monitoring and memory access throttling capabilities, as explained in the following.

Memory Controller: X1's memory controller is responsible for handling all memory requests to the main memory. Figure 2 shows how various system blocks are multiplexed through multiple layers (rings) of Priority Tier Snap Arbiters (PTSA). These arbiters implement a proprietary Round-Robin scheduling policy [21]. The default hierarchy implementation within the Tegra X1 memory controller places the CPU read/write into the top ring; the display, video engines, image processors, and audio engines into the second ring; and everything else, including the GPU (GM20B), into the third ring.

The PTSAs provide a hardware throttling mechanism (*Throttling Logic* in Figure 2) to selectively throttle the memory requests into the rings. This enables the system to meet

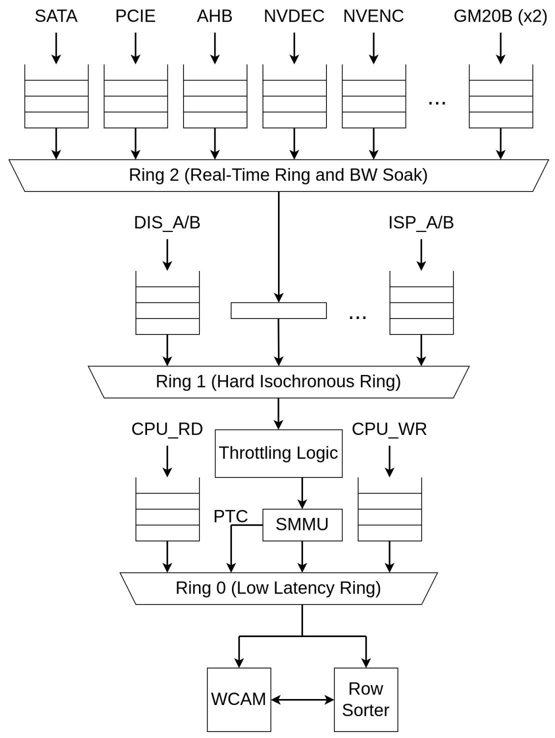


Fig. 2: The Tegra Snap Arbiter, adapted from [21]

latency requirements by keeping a DRAM-limited system from being overburdened with outstanding memory requests. With this mechanism, in combination with the hierarchy of the memory controller, the default Tegra X1 memory subsystem improves CPU latency by limiting the number of back-pressuring requests when the DRAMs are saturated. If enabled, the throttling is triggered when the number of outstanding requests exceeds a programmable threshold. In Section IV-C, we evaluate this mechanism in detail.

Activity Monitors: The Tegra X1 includes activity monitors to dynamically measure the utilization of various units in the system, which are originally designed to determine which power management policy should be enacted. An activity monitor block contains a 32-bit counter for memory controller clock cycles, which is incremented when any memory controller access event is detected. In addition, the block contains a dedicated counter for a running average of the past 128 samples as well as a programmable watermark level which can be used to generate interrupts when the utilization exceeds the value. The sampling period is a configurable value between 1μs and 1ms. Importantly, the Tegra X1 [21] contains two separate memory controller activity monitors: *MC-ALL* which reports the total clock cycles from all memory events and *MC-CPU* which reports only the memory events sourced from the CPU complex. These monitors can be used in conjunction to derive an approximate memory controller utilization from the GPU, which we use to implement BandWatch.

The system's total memory utilization U_{all} can be calculated as $U_{all} = \frac{B_{all}}{L} * 100$ where L is the sampling period in cycles and B_{all} is the busy cycles reported by *MC-ALL*. For this work, the sampling period of the activity monitors is set to 10 μs and as we lock the memory controller clock speed at 1,600MHz (i.e., $L = 1,600 \times 10^6/10^5 = 16,000$). The CPU cluster's memory utilization can be similarly calculated as $U_{cpu} = \frac{B_{cpu}}{L} * 100$, where B_{cpu} is the busy cycles reported by the *MC-CPU*. Lastly, the GPU's memory utilization U_{gpu} can be approximated as $U_{gpu} = U_{all} - U_{cpu}$.

For the GPU utilization, as seen in Figure 2, GPU memory events are not distinguished between other non-CPU memory events; we assume in this paper limited memory traffic outside the CPU and GPU. Also note that activity monitoring is coarse-grained; it can only be applied to either the entire system or the entire CPU complex. It is not possible to monitor each core's memory utilization. Likewise, the throttling capability can be applied to the CPU, GPU, and other clients into the memory controller—but not individual CPU cores.

B. MemGuard

MemGuard [35] is a loadable Linux kernel module which can throttle each CPU core's memory requests at a set rate using the core's hardware performance counters. Specifically, it periodically (e.g., 1ms) monitors each core's LLC miss counter and programs the counter to generate an interrupt once the set budget is reached within the regulation period. Once the interrupt is generated, the memory bandwidth budget is exhausted and the core remains idle until the next period begins, at which point the core's budget is replenished. In this way, MemGuard provides a per-core memory bandwidth throttling capability. In this work, we employ both X1's hardware throttling – mainly for GPU memory traffic – and MemGuard for throttling individual CPU cores. The combined use of these two techniques enables efficient system-wide memory bandwidth management, as explained in the subsequent section.

III. BandWatch

In this section, we describe the design of BandWatch.

A. Assumptions and Objectives

We consider a heterogeneous multicore processor that comprises multiple CPU cores and an integrated GPU, all accessing a single shared main memory (DRAM) subsystem. We assume that the CPU cores and the integrated GPU are partitioned such that some are assigned to execute real-time (RT) tasks while the rest are assigned to execute non-real-time (NRT) tasks. This paper focuses on a specific partitioning scheme where one CPU core is reserved for RT tasks, with the remaining cores assigned to NRT tasks. The same partitioning scheme has been used in many prior works (e.g., [24], [25], [34]) due to its simplicity and practicality. Note, however, our work can support any disjoint partitioning schemes. For instance, it should be possible to assign the integrated GPU or multiple CPU cores for RT tasks (see Section V). In this setting, the primary objective for BandWatch is to isolate RT tasks from interference produced by the NRT tasks while minimizing performance loss of the NRT tasks.

B. Design Overview

BandWatch is built upon three core mechanisms. First, a monitoring mechanism is used to determine the system's memory utilization (*Activity Monitor*; see Section IV-B) and bandwidth usage. Second, a hardware throttling mechanism (see Section IV-C) is employed to regulate the memory traffic from the integrated GPU. Third, MemGuard [35], a software throttling mechanism, is used to regulate the memory traffic from the CPU cores. Using these mechanisms, BandWatch dynamically monitors the system's memory usage and throttles the memory requests from the NRT tasks (from the NRT cores and the GPU) if the RT task is expected to suffer slowdown. Figure 3 shows a high-level system architecture of BandWatch.

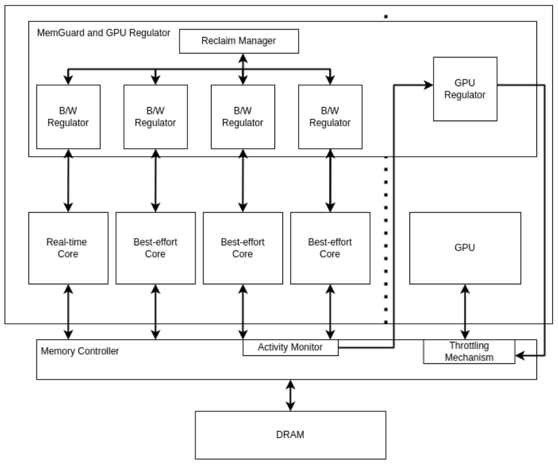


Fig. 3: BandWatch system architecture

C. Runtime Regulation

BandWatch's basic regulation strategy is as follows. It first checks if the RT core is producing significant memory traffic. If not, no NRT CPU cores or the GPU are throttled. If the RT core is sufficiently memory intensive, then BandWatch throttles both NRT CPU cores and the GPU (if the RT task does not use the GPU) in order to protect the performance of the RT core(s). The degree of throttling levels vary depending on the memory utilizations of the system.

Figure 4 shows BandWatch's core regulation algorithm, which is built on top of MemGuard's [35] implementation. As described in II-B, MemGuard replenishes the memory budgets of the CPU cores to prepare the next iteration of monitoring at a regular interval (currently 1ms). The function *periodic_timer_handler* is called after memory bandwidth statistics for the RT core, over the previous regulation period, become available. The RT core's memory bandwidth usage B_{rt} is obtained from MemGuard (Line 3). Next, the RT core statistics over the last regulation period are used to determine if the RT task is actively producing memory requests (Line 4). The T_{cpu} is set to the value of 75MB/s, which was empirically determined. If the RT CPU is memory-active, BandWatch throttles the NRT CPU cores to 75MB/s (Line 6), while the GPU is throttled to a level proportional to the CPU memory

```
function periodic_timer_handler ;
begin
    $B_{rt} \leftarrow$ RT core's memory usage ;
    if $B_{rt} > T_{cpu}$ then
        foreach NRT core $c_i$ do
            program $c_i$ to throttle at $T_{cpu}$;
        $U_{cpu} \leftarrow$ CPU's memory utilization ;
        $TL_{gpu} = \frac{U_{cpu}}{U_{cpu}^{max}} * TL_{gpu}^{max}$;
        program MC to throttle GPU at $TL_{gpu}$ ;
    else
        foreach NRT core $c_i$ do
            unthrottle $c_i$ ;
        unthrottle GPU ;
```

Fig. 4: BandWatch Implementation

utilization (line 7). The rationale behind the dramatic throttling of the CPU, and progressive throttling of the GPU, stems from the inherent priority hierarchies built into the Tegra X1 memory controller. This design enables an interfering CPU core to cause significantly more slowdown for the RT CPU core than a GPU kernel. The amount of throttling on the GPU, TL_{gpu}, is set proportional to the relative CPU utilization in the range [0, T_{gpu}^{max}] (these values are explained in Sections IV-B and IV-C). The $\frac{U_{cpu}}{U_{cpu}^{max}}$ is adequate to use for this calculation as this occurs while the NRT cores have been throttled—making the utilization primarily from the RT core. Lastly, if the RT core is not using memory, the NRT CPU cores and GPU are allowed to freely use as much memory bandwidth as needed.

IV. Evaluation

In this section, we evaluate the effectiveness of BandWatch on a real platform using synthetic and real-world workloads.

A. Hardware and Software Setup

For hardware, we use the NVIDIA Jetson Nano platform, which is based on a Tegra X1 SoC. The X1 SoC has a quad-core ARM Cortex-A57 CPU and a 128-core Maxwell GPU. The CPU has 48K private L1-I and 32K L1-D caches and a 2MB shared L2 cache. Both CPU and GPU share a 4GB LPDDR4 main memory (25.6GB/s peak bandwidth), which is accessed though a common memory controller. For software, we use NVIDIA's Linux for Tegra (L4T) version 32.6.1, Ubuntu 18.04.6, Linux Kernel 4.9.253 and the CUDA version 10.2.300. Throughout the evaluation, we use the SD-VBS [30] benchmark suite as real-world real-time (RT) workloads, while using IsolBench [29] and HeSoC-mark [11] benchmark suites as non-real-time (NRT) synthetic CPU and GPU workloads, respectively. We configured the hardware to run at the maximum performance mode and disabled DVFS and graphical user interface to improve repeatability.

B. Memory Utilization Monitoring

We first measure the memory utilization and bandwidth usage of each RT and NRT workloads in isolation. We use the memory controller's activity monitor, described in Section II-A, to measure the memory utilization while using the CPU core's hardware performance counter (L2 cache miss) to measure the memory bandwidth. For the RT workloads, we ran them to completion and recorded their execution times as well. For the synthetic NRT workloads, we ran them for five seconds and collected the data.

TABLE I: Characteristics of SD-VBS (RT)

Benchmark	Time (s)	Utilization	Bandwidth (MB/s)
disparity	5.6	.06	793
sift	5.7	.02	239
mser	1.5	.03	360
tracking	1.5	.01	129
texture_synthesis	41.8	0	1.9

TABLE II: Characteristics of synthetic workloads (NRT).

Benchmark	Utilization	Bandwidth (MB/s)
bandwidth(read)	.17	4280
bandwidth(write)	.26	3259
cudainterf(memset)	.81	8116
cudainterf(memcpy)	.82	3980

Table I presents the results of the SD-VBS benchmarks, which show varied memory utilization and bandwidth characteristics. The most memory intensive one is *disparity* with 6% memory utilization and a 793MB/s memory bandwidth. On the other hand, *texture_synthesis* is the least memory intensive. Note that memory utilization and bandwidth are strongly correlated though not perfectly proportional as the relationship between the two metrics varies depending on the memory access patterns and whether the memory requests are processed efficiently (or inefficiently) at the memory controller.

Table II shows the results of the synthetic NRT workloads of which the first two are read and write modes of *bandwidth* from the IsolBench suite [29] and the last two are the memset and memcpy modes of *cudainterf* from the HeSoC benchmark suite [11]. Both *bandwidth* and *cudainterf* are designed to generate maximum possible memory traffic—from the CPU and the GPU, respectively. For example, bandwidth runs on a CPU core and repeatedly reads or writes a large array at a cache-line granularity. The array size is configured as twice the L2 cache size so that the memory operations generate a lot of L2 cache misses. On the other hand, cudainterf launches a CUDA kernel on the GPU, which performs large amounts of memset or memcpy operations in parallel from the GPU cores. Note that the GPU based *cudainterf* workloads achieve much higher memory utilization and bandwidth compared to the CPU based *bandwidth* workloads, which suggest that the GPU can generate more memory traffic than the CPU. Recall that the CPU's memory requests have shorter latency allowance (i.e., the number of cycles a memory request can stay outstanding in the memory controller's queue; see Section II-A) than GPU's memory requests. This helps CPU memory requests achieve lower latency but at the cost of lower bandwidth. On the other hand, GPU requests can achieve higher bandwidth at the cost of relatively higher latency.

Note that BandWatch's algorithm (Figure 4) requires the knowledge of maximum memory utilization U_{cpu}^{max} from a single (RT) CPU core. As bandwidth(read) achieves the highest CPU memory bandwidth, we select its observed memory utilization 0.17 as the value in the rest of the experiments that use BandWatch.

C. GPU Throttling

BandWatch uses the memory controller's throttling mechanism, described in Section II-A, to regulate the GPU memory traffic. In this experiment, we evaluate the effect of the throttling mechanism. The experiment setup is that we launch a memory intensive synthetic GPU benchmark from the HeSoc benchmark suite and measure its reported memory bandwidth as we vary the throttling level of the memory controller using a custom kernel module. The memory controller supports 32 discrete throttling levels [21].

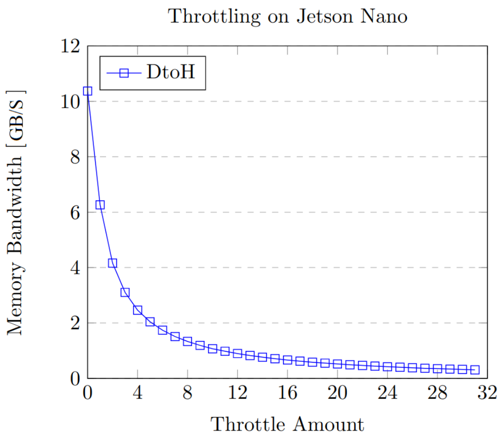
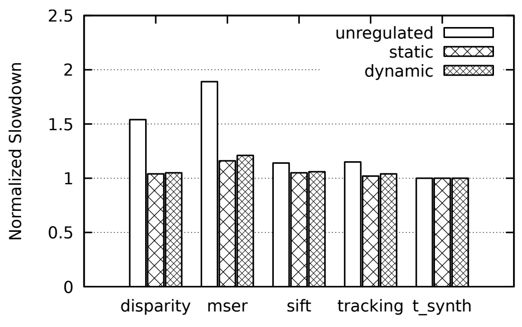


Fig. 5: MC throttling effect on GPU memory bandwidth.

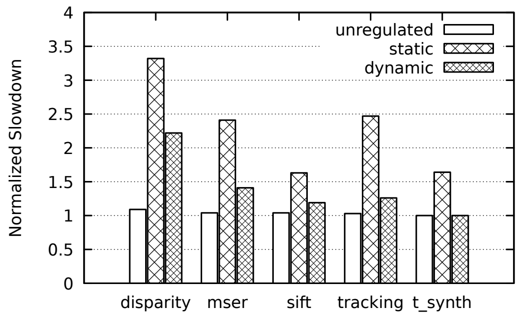
Figure 5 shows the results. As can be seen, as the throttling level increases the measured memory bandwidth decreases inverse proportionally. Given that the memory controller's throttling mechanism applies to not just the GPU but all other peripherals, including display and disk I/O devices, which are connected to the Ring 1 or above (see Figure 2 in Section II-A), we limit the maximum throttle level, T_{gpu}^{max}, of BandWatch to 16 to minimize potentially negative performance impact of throttling to the real-time tasks, which may still need to load data from disk, for example. In our experiments, memory traffic generated by non-GPU I/O devices are small and do not impact the execution of tasks even at the maximum throttling level. I/O intensive workloads is beyond the scope of the paper.

D. Dynamic Regulation

In this subsection, we evaluate BandWatch's effectiveness in providing isolation to the real-time tasks to provide inter-core isolation as well as minimizing the performance degradation from bandwidth throttling.



(a) SD-VBS on RT CPU



(b) cudainterf(memset) on NRT GPU

Fig. 6: Slowdowns suffered by (a) SD-VBS on the RT CPU core and (b) cudarinterp(memset) on the GPU. In both subgraphs, the X-axis shows the RT task and the Y-axis shows the normalized slowdown compared to the performance obtained in isolation.

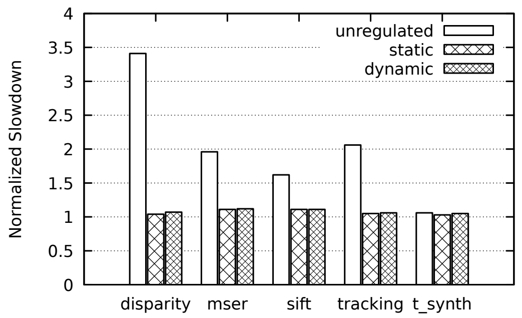
1) Against GPU interference: We first evaluate BandWatch's effectiveness in providing isolation in the presence of memory intensive NRT GPU kernels. The basic setup is to run a RT task from the SD-VBS benchmark suite on a dedicated RT core (core 0) while co-scheduling synthetic NRT tasks on the GPU. In this experiment, we use one instance of cudainterf (memset), which executes a memset CUDA kernel on the integrated GPU. Because CPU and GPU share the LPDDR4 main memory, the RT task and the co-scheduled NRT task may suffer slowdowns due to contention in the main memory. We compare *unregulated* baseline, *static* throttling and *dynamic* throttling, which implements BandWatch's adaptive algorithm (Figure 4). For *static* throttling, we apply a constant level of GPU throttling for the entire duration of the RT task's execution. The static throttling level is determined by exhaustively testing all possible GPU throttling levels and choosing the least throttled level that achieves less than 10% or less slowdown of the RT task. For *dynamic*, the throttling level varies dynamically in response to the changes in memory utilization and the RT task's memory bandwidth usage over time. The algorithm's hyper parameters B_{rt} and T_{cpu} are experimentally determined to be both at 75MB/s.

Figure 6 shows the normalized slowdowns of the five SD-VBS benchmarks (RT) and the corresponding *cudainterf memset* instances (NRT). Note first that in *unregulated*, *disparity* and *mser* suffer significant slowdowns—1.54X and 1.89X—while *sift* and *tracking* suffer modest slowdowns—1.14X and 1.15X respectively. Lastly, *text_synthesis* suffers almost no interference. The observed slowdowns are due to contention in the main memory, which is the only major shared resource between the CPU and the iGPU, and thus is roughly proportional to each RT task's memory intensity in Table I. With static throttling, which statically assigns the GPU throttling level at 3, the slowdowns suffered by the RT task markedly reduced as the interfering GPU's memory traffic is throttled by the memory controller's throttling mechanism. However, the static throttling dramatically reduce the performance of the throttled NRT tasks. For instance, the *cudainterf* instance, which is co-scheduled with *disparity*, suffers 3.3X slowdown.

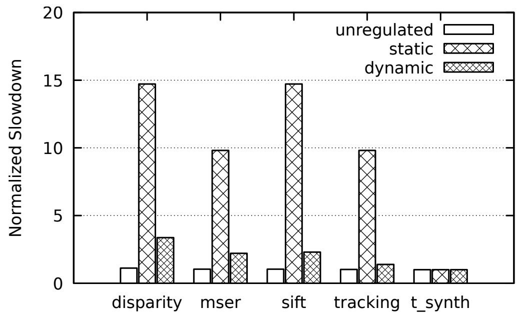
With dynamic throttling, in contrast, BandWatch is able to mitigate the slowdown incurred by the co-running GPU kernel while at the same time minimizing slowdown suffered by the throttled NRT task by dynamically adjusting the GPU throttling level based on the RT task's memory intensity changes over time. Concretely, for the similar level of slowdowns experienced by disparity, BandWatch is able to reduce NRT task's slowdown from 3.3X to 2.2X. Similarily, it reduces the slowdown of the NRT task co-scheduled with mser from 2.4X to 1.6X. Overall, BandWatch's dynamic throttling enables significantly better performance for the throttled NRT GPU kernel, cudarinterf(memset), while still providing similar levels of performance isolation to the RT tasks. We also repeated the experiment but using cudainterf(memcpy) as the NRT task and got similar results.

2) Against CPU interference: In this experiment, we evaluate BandWatch's effectiveness in providing isolation in the presence of memory intensive NRT CPU tasks. The basic setup is similar to the previous experiment in that we run a RT task from the SD-VBS benchmark suite on a dedicated RT core (core 0) while co-scheduling synthetic NRT tasks, but differs in that three instances of bandwidth(write), which run on the rest of CPU cores (core 1-3), are used as the NRT task. Because the NRT tasks are also running from the CPU, we cannot utilize the memory controller level throttling to regulate the NRT tasks. Instead we use MemGuard [35]'s OS-level throttling mechanism, which utilizes each CPU core's performance counters and interrupt handlers to periodically regulate the memory traffic based on the user-defined memory bandwidth budget. We determine the memory bandwidth budget for static throttling by sweeping the allowed memory budget for the NRT tasks in increments of 25 MB starting from 0 up until the execution time slowdown of the protected RT task is acceptable.

Figure 7 shows the results. Note first that the RT tasks suffer significantly more from memory traffic generated by the NRT CPU cores. For instance, the *disparity* suffers 3.4X slowdowns from the bandwidth(write) co-runners compared to 1.5X slowdown from cudainterp(memset) despite the fact that the latter generates higher memory bandwidth. This is because the memory controller prioritizes CPU's memory traffic over GPU's by assigning lower latency allowance cycles to the CPU



(a) SD-VBS on RT CPU



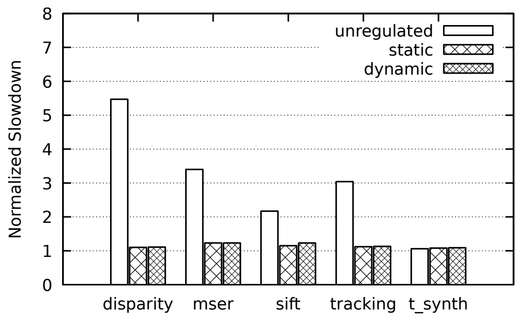
(b) bandwidth(write) on NRT CPU core

Fig. 7: Slowdowns suffered by (a) SD-VBS on the RT CPU core, and (b) bandwidth(write) on NRT CPU cores

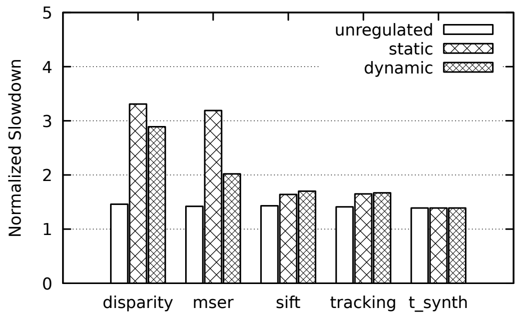
traffic. Unfortunately, however, the memory controller cannot differentiate between the CPU cores thus equally prioritizing both RT and NRT CPU memory traffic, hence the higher slowdown for the RT task. To protect the RT task, static throttling has to throttle the NRT CPU cores, which resulted in up to 14.7X slowdown, as observed in Figure 7b. In contrast, BandWatch's dynamic throttling can drastically reduce the NRT core slowdowns while providing a similar degree of isolation benefits to the RT tasks.

3) Against CPU and GPU interference: Lastly, we evaluate BandWatch's effectiveness in providing isolation in the presence of memory intensive NRT GPU and NRT CPU tasks. The setup is identical as before except that we use both the cudainterp(memset) and bandwidth(write) as NRT GPU and CPU tasks, respectively, and both are scheduled simultaneously for each RT task from SD-VBS. Note that in this experiment, both the GPU and the NRT CPU cores may be throttled to protect the RT task. For static throttling, we again exhaustively search all possible throttling settings and used the best possible settings. For BandWatch's dynamic throttling, no manual intervention is needed as it can dynamically throttle both NRT CPU cores as well as the GPU.

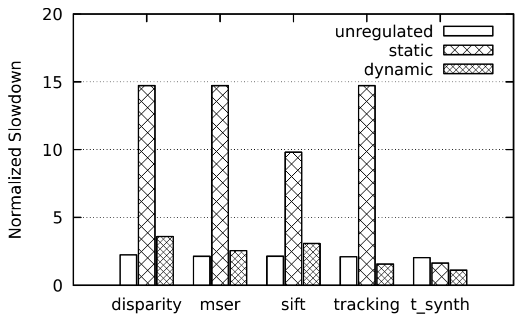
Figure 8 shows the results. As in the previous two experiments, static throttling does protect the RT tasks but at a very high cost to the throughput of the throttled NRT tasks, especially those that run from the NRT CPU cores. In contrast, BandWatch's dynamic throttling provides a same degree of protection at a much lower cost to the NRT tasks.



(a) SD-VBS on RT CPU



(b) cudainterf(memset) on NRT GPU



(c) bandwidth(write) on NRT CPU cores

Fig. 8: Slowdowns suffered by (a) SD-VBS on the RT CPU core and (b) cudainterp(memset) on NRT GPU, and (c) bandwidth(write) on NRT CPU cores

V. Discussion

BandWatch relies on a hardware-level throttling feature in a specific memory controller found in Tegra X1 SoC for iGPU throttling. It also utilizes the memory utilization monitoring capability in the X1 SoC. As such, it may not be directly applicable on other SoCs. Note, however, that many COTS memory controllers do support some forms of Quality-of-Service (QoS) features [26], including throttling. Memory utilization monitoring capabilities are also available in other memory controllers [24], [28]. Moreover, ARM's recent Memory Partitioning and Monitoring (MPAM) specification [6] provides Instruction-Set Architecture (ISA) level support for throttling and monitoring capabilities on shared resources in the memory hierarchy, including the memory controllers, which will be supported in many future hardware designs. As such, we believe our approach will be applicable in future hardware platforms with minor modifications.

Our current implementation focuses on a particular system model where a single CPU core is reserved for RT tasks while the rest of the CPU cores and the iGPU are assigned for NRT tasks. Assigning multiple CPU cores for RT tasks was explored in prior work on real-time gang scheduling [2], [4] and can be incorporated in BandWatch. Likewise, using the iGPU for RT tasks was also previously explored [3] and can be supported in BandWatch by disabling iGPU throttling and dedicating the iGPU for RT tasks.

VI. Related Work

Mitigation of shared resource contention in multicore has been extensively studied in the real-time systems community [3], [8], [15], [17], [18], [20], [24], [32], [33], [35]. Two main approaches have been partitioning and bandwidth throttling.

The partitioning approach tries to improve isolation by reserving dedicated hardware resources. LLC space and DRAM bank partitioning techniques have been extensively studied and shown to be effective [17], [19], [20], [33]. However, partitioning techniques often require knowledge of address to resource mapping information, which is often difficult to obtain, or special hardware support. They also often do not provide sufficient isolation as there could be many other resources that are still being shared.

Memory bandwidth throttling is another popular approach, which can be implemented in software in most modern multicore processors [3], [8], [24], [35]. Those software-based throttling techniques utilize per-core hardware performance counters and enforce a user-defined bandwidth budget periodically, usually at a granularity of milliseconds due to software handling overhead. For example, Saeed et al., recently proposed a memory utilization based feedback control system that dynamically throttles the memory bandwidth usage of non-real-time CPU cores when the system-wide memory utilization reaches a certain threshold.

The use of heterogeneous computing platforms for real-time applications has received significant attention in recent years [5], [7], [11], [12], [23], [31]. The integrated GPUs and other accelerators in heterogeneous platforms are challenging to provide real-time guarantees because they often share the main memory and other resources with the CPU [9]. Memory bandwidth throttling is also explored in heterogeneous platforms in several recent works. Ali et al. and Homa et al., proposed to throttle CPU cores to protect real-time GPU kernels [1], [3]. Serrano-Cases et al. and Zini et al. explored the use of various hardware-level QoS features, including the ARM QoS feature for system bus-level throttling, in Xilinx MPSoCs to regulate the contention [26], [37]. EWarP [28] utilized both the ARM QoS feature and the software based CPU memory bandwidth throttling to holistically regulate memory bandwidth based on the memory utilization on a Xilinx MPSoC. In contrast, our work leverages the hardware throttling capability in the memory controller of Tegra X1

SoC to throttle iGPU, and we propose a simple but effective adaptive throttling policy that holistically regulates both CPU and GPU bandwidth to ensure predictable execution of real-time tasks.

Recently, Intel and ARM began to support some hardware assisted partitioning and bandwidth throttling capabilities, namely Intel RDT [27] and ARM MPAM [36], which provided ISA-level support for low-overhead shared resource management. However, they are currently available only on some high-end server processors and have their own technical limitations [27], [36] in providing isolation needed for use in embedded and real-time systems. In this work, we use both hardware and software based throttling mechanisms. However, we differ from most prior works in that we support system-wide throttling on a COTS heterogeneous SoC that integrates both the CPU and GPU.

VII. Conclusion

In this paper, we have presented BandWatch, a system-wide memory bandwidth regulation system. BandWatch is designed to provide memory performance isolation for RT tasks against interference from CPU cores as well as GPU executing non-real-time workloads. By combining hardware-level GPU throttling capability in NVIDIA's Tegra X1 and well-known software-based CPU core throttling mechanism, BandWatch implemented an adaptive throttle policy that can throttle both CPU cores and the GPU to protect the performance of critical RT tasks, while minimizing negative performance impact to the throttled NRT tasks.

Acknowledgements

This research was supported in part by NSF grant CNS-1815959, CPS-2038923 and NSA Science of Security initiative contract no. H98230-18-D-0009.

References

[1] H. Aghilinasab, W. Ali, H. Yun, and R. Pellizzoni. Dynamic memory bandwidth allocation for real-time gpu-based soc platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39(11):3348–3360, 2020.

[2] W. Ali, R. Pellizzoni, and H. Yun. Virtual gang scheduling of parallel real-time tasks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 270–275. IEEE, 2021.

[3] W. Ali and H. Yun. Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[4] W. Ali and H. Yun. RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[5] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[6] ARM. ARM Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A. https://developer.arm.com/documentation/ddi0598/latest.

[7] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu. Co-Optimizing Performance and Memory Footprint Via Integrated CPU/GPU Memory Management, an Implementation on Autonomous Driving Platform. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[8] M. G. Bechtel and H. Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[9] M. G. Bechtel and H. Yun. Denial-of-Service Attacks on Shared Resources in Intel's Integrated CPU-GPU Platforms. In *IEEE International Symposium On Real-Time Distributed Computing (ISORC)*, 2022.

[10] M. G. Bechtel and H. Yun. Cache Bank-Aware Denial-of-Service Attacks on Multicore ARM Processors. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023.

[11] N. Capodieci, R. Cavicchioli, I. S. Olmedo, M. Solieri, and M. Bertogna. Contending Memory in Heterogeneous SoCs: Evolution in NVIDIA Tegra Embedded Platforms. In *IEEE Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2020.

[12] R. Cavicchioli, N. Capodieci, and M. Bertogna. Memory Interference Characterization Between CPU Cores and Integrated GPUs in Mixed-Criticality Platforms. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017.

[13] Certification Authorities Software Team. CAST-32A: Multi-core Processors, 2016.

[14] EASA. AMC 20-193 Use of multi-core processors, 2022.

[15] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo. Designing mixed criticality applications on modern heterogeneous mpsoc platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[16] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson. Slow and Steady: Measuring and Tuning Multicore Interference. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[17] H. Kim, A. Kandhalu, and R. Rajkumar. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[18] N. Kim, B. C. Ward, M. Chisholm, J. H. Anderson, and F. D. Smith. Attacking the One-Out-of-M Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. *Real-Time Systems*, 2017.

[19] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[20] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[21] NVIDIA. *Tegra X1 Mobile Processor technical reference manual (revision 1.3p)*, 4 2019. Version 1.3p.

[22] NVIDIA. *Jetson Nano System-on-Module*, 4 2020. Version 1.0p.

[23] 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 *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[24] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann. Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

[25] A. Saeed, D. Hoornaert, D. Dasari, D. Ziegenbein, D. Mueller-Gritschneder, U. Schlichtmann, A. Gerstlauer, and R. Mancuso. Memory latency distribution-driven regulation for temporal isolation in mpsocs. In *Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[26] A. Serrano-Cases, J. M. Reina, J. Abella, E. Mezzetti, and F. J. Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2021.

[27] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger. A closer look at intel resource director technology (rdt). In *International Conference on Real-Time Networks and Systems (RTNS)*, 2022.

[28] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-WarP: a System-wide Framework for Memory Bandwidth Profiling and Management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[29] P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[30] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. SD-VBS: The San Diego Vision Benchmark

Suite. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[31] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith. Avoiding Pitfalls When Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[32] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a Dynamic Cache Partitioning System Using Page Coloring. In *Parallel Architecture and Compilation Techniques (PACT)*, 2014.

[33] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[34] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Euromicro Conference on Real-Time Systems*, pages 299–308. IEEE, 2012.

[35] 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 *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[36] M. Zini, D. Casini, and A. Biondi. Analyzing Arm's MPAM From the Perspective of Time Predictability. *Transactions on Computers*, 2022.

[37] M. Zini, G. Cicero, D. Casini, and A. Biondi. Profiling and Controlling I/O-Related Memory Contention in COTS Heterogeneous Platforms. *Software: Practice and Experience*, 2021.