FLOSS: FLOW Sensitive Scheduling on Mobile Platforms

Haibo Zhang, Prasanna Venkatesh Rengasamy, Nachiappan Chidambaram Nachiappan,
Shulin Zhao, Anand Sivasubramaniam, Mahmut T. Kandemir, Chita R. Das
The Pennsylvania State University

{haibo,pur128,suz53,anand,kandemir,das}@cse.psu.edu,nachi@alumni.psu.edu

ABSTRACT

Today's mobile platforms have grown in sophistication to run a wide variety of frame-based applications. To deliver better QoS and energy efficiency, these applications utilize multi-flow execution, which exploits hardware-level parallelism across participating accelerators in the SoC. Our study shows that multi-flow execution increases memory pressure, and motivates us to propose a rate-based memory-scheduling scheme, called FLOSS, that considers a flow, individual frames of a flow, and any sharing of IPs across concurrent flows to schedule memory requests. Experimental results indicate that FLOSS provides 12% QoS improvement over baseline FR-FCFS scheme, and outperforms two QoS-aware schemes in multi-flow execution scenarios.

1 INTRODUCTION

The popularity and widespread market adoption of mobile devices is opening doors to an interesting class of applications that are very data intensive, requiring periodic processing of large volumes of data in a highly interactive manner. Such applications span those in gaming, which periodically render frames for display, video and audio streaming for entertainment (e.g., YouTube and Netflix, etc.), or virtual reality (e.g., Oculus and VIVE). These apps¹ have a common underlying paradigm: they need to periodically process "frames" of data, and produce output in real time. Mobile devices incorporate several accelerators to accommodate the periodic *flow* of frames in these apps, including those for encryption/decryption, video codecs, wireless, radios, cameras, and imaging DSP. For instance, consider Yinzcam [26], which streams simultaneous *points-of-view* (PoV) from different locations within a sports arena, with real-time analytics (e.g., player tracking). Another widely popular instance of this multi-video streaming is the VR apps. These apps, with the help of 3D glasses, convert the ordinary display into immersive 3D videos. To achieve this, they stream a separate video for each eye and interactively change the PoVs with I/O inputs from accelerometers. Further, it may not just be one app with multiple flows, but multiple single flows from multiple apps is also becoming a reality as devices start offering multiple windows (e.g., Split View in iOS [2] and Split Screen in Android [22]). Provisioning multiple IPs to take on multiple flows can become cost, resource, real-estate and energy prohibitive. While it may appear that the "under-utilized" IPs of a single-flow chain

would have sufficient bandwidth to support multi-flows, without requiring additional IPs, we show that keeping IPs simultaneously busy (either IPs within a single flow, or shared/isolated IPs across multiple flows), shifts the bottleneck to the memory system. The problem is accentuated as more IPs are concurrently engaged, either within a single flow or across multi-flows.

We proposed a smart memory scheduling mechanism called FLOSS (**FLOW** Sensitive Scheduling), designed to alleviate the memory bottleneck to support multiple flows without requiring additional bandwidth. Throughput-oriented memory schedulers such as FR-FCFS [19], which are extensively used in commercial systems, do *not* work very well in situations where there are explicit Quality-of-Service (QoS) requirements. Recognizing this, recent proposals [10, 25] try to incorporate QoS within memory scheduling. However, they do *not* recognize flows. As we will show, treating each IP's requests individually, without recognizing the sequence, can lead to inefficiencies. Further, an IP can itself serve the frames of different flows over time, and they cannot all be treated the same. FLOSS, on the other hand, not only recognizes memory requests coming from different IPs as belonging to the same frame of a flow, but also captures the fact that an IP can issue memory requests for different flows (if the flows intersect on the IPs that they require).

We implement the proposed FLOSS, and evaluate the delivered performance (QoS) and energy when running flow-oriented Android apps. Specifically, the paper makes the following **contributions**:

- *Quantifying SoC Utilization in Single/Multi-Flow Cases and Identify the Memory Bottleneck*: Enabling Parallelizing-IPs boosts the overall SoC utilization. In the single flow execution, the under-utilization has dropped from 81% to 5%, and in the multi-flow case, the under-utilization has dropped from 48% to 0%. Since more IPs are concurrently busy with the multi-flow and parallelizing IPs, the concurrency demands on the memory system have also accordingly increased, leading to memory bottleneck.
- *Rate-Based Memory Scheduling for Flows*: We analyze the advantages of FLOSS over other memory scheduling schemes, and observe that the two prior QoS-aware memory scheduling mechanisms (DynQoS [10] and DASH [25]), are not as competitive as FLOSS. More specifically, with respect to FR-FCFS, DynQoS, and DASH, FLOSS can provide average QoS improvements of 12%, 10%, 21%, respectively. These improvements are because of (i) understanding and exploiting the QoS needs of each frame in each flow, (ii) recognizing the requests coming from different IPs for the same frame at different times in a flow, and (iii) recognizing requests coming from the same IPs for different flows at different times.

2 MOTIVATION

In this section, we use a typical execution flow of a representative application to characterize the inefficiencies in current mobile apps.

Application Example: We use a video playback scenario to explain how a frame is prepared and displayed on the screen. The high-level

¹In this paper, we use the term "cores" to specifically mean CPU-cores. We use the terms "IP" $\Leftrightarrow$ "accelerator", and "application" $\Leftrightarrow$ "app" interchangeably.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5700-5/18/06...\$15.00

<https://doi.org/10.1145/3195970.3196052>

(a) Video playback code invoking IPs in parallel.

(b) FPS improvement.

(c) Utilization for the Video Playback.

view is given in Fig. 1a, showing the three main components in the app. The app first *extracts* an (*encoded*) frame from the Flash, then *decodes* it frame by frame, and finally *posts* each frame to the Display. The app requires its underlying IP flow to deliver frames periodically at a certain rate – Frames Per Second (FPS) – to meet its QoS. While a video in general requires 30 or 60 FPS, there are more stringent applications such as Slow-Mo capture [1, 7], that may require recording live video at 120 FPS or even 240 FPS.

As described, the frame is required to flow through each of the stages to complete its decoding in a *sequential order*. A straightforward implementation [6] is to use a single thread to manage the whole flow. Since single-thread only utilizes one IP at a time, it cannot attempt to keep all the IPs involved in serving a flow busy. Hence, when a frame blocks at video decoder for a long time, even though the flash controller is idle, the next frames can not utilize it.

Instead, another approach, which is adopted by the modern video players [9], is to parallelize IPs (PIP). This leverages the fact that the function calls for each stage at the application can be invoked *in parallel*. In fact, in order to meet the QoS requirements, the application developers need to utilize the IPs as much as possible. As shown in Fig. 1a, each of the thread manages the corresponding IPs in the SoC, through a set of drivers provided by the Android and the corresponding IP vendors. As we will show, such paradigm (PIP) can efficiently increase FPS as well as the application utilization. On the other hand, since PIP enables multiple memory consumers at the same time, it leads to more pressure on the memory controller.

Application Characterization: We show our characterization results in four settings to better understand the memory and IP utilization: one video playback with and without PIP; two video playbacks with and without PIP. Figs 1c shows the utilization of IP and memory. We find two important observations:

- **FPS and SoC utilization increase:** With PIP, the application can provide significantly better QoS (FPS)² to the user. The FPS of one video and two video executions has increased by 160% and 31.9%, respectively (refer to Fig. 1b). This is because PIP boosts the overall IP utilization substantially. This can be observed not just in the two video case where under-utilization has dropped from 48% to 0%, but even in the single video execution, where the under-utilization has dropped from 81% to 5%.

²FPS, or inversely the number of frames that are not delivered at the specified rate (referred to as *Frame Drops*), is usually used as the main “performance metric” for these applications. Although the baseline FPS in this example is already 60 FPS, a higher FPS would be beneficial for a variety of real-time scenarios (VR, 3D videos), as illustrated by our multi-app workloads evaluated.

- **Memory is bottlenecked:** We can see that with increasing applications from one video playback to two video playbacks and by enabling PIP, more IPs are concurrently busy in parallel, the concurrency demands on the memory system have also accordingly increased. Consequently, memory contention is now much higher, and the IPs are stalling much more (increases between 10% to 40% of the time for instance in video decoder (VD)) for memory requests.

PIP has considerably boosted the utilization of the precious IP resources, for both single flow and multiple flows scenarios. However, this enhancement comes at a price: *increased memory contention*. Thus, we analyze the limitation of using existing memory schedulers in Sec. 3, and then in Sec. 3.2, we propose a *flow-aware* memory system, custom designed to handle the high concurrency demands of IPs without significantly increasing the provisioned hardware.

3 SOLVING MEMORY BOTTLENECK

By parallelizing IP, the IPs begin to access their respective data in parallel. As a result, the overall memory bandwidth required for execution is much higher than the baseline execution case. Note that, provisioning IPs with additional cache/scratchpad memories will help the IPs access the data stored in these memories quickly at the cost of additional SoC chip area [15]. The rest of this paper shows that, instead of going towards this costly hardware, we can gain better QoS and energy by just addressing the inefficiencies in the contemporary memory schedulers.

3.1 Limitations of Current Memory Schedulers

We first show why FR-FCFS is not able to provide QoS, and then examine the limitations of DynQoS [10] and DASH [25], two recently-proposed QoS-aware memory request schedulers.

FR-FCFS Scheduler FR-FCFS is aimed at improving the row-buffer hit rate, and in turn the overall throughput, rather than any application-specific QoS needs. Here, the requests for the opened-row are prioritized over the ones that need a new row to be opened. When no such requests are available, it defaults to FCFS scheduling. Fig. 2a illustrates the limitations of FR-FCFS scheduler. In this example, there are two flows indicated by the labels “1” and “2”, each using one IP, namely, IP-A and IP-B, respectively. IP-A in Flow-1 requires 3 memory requests to complete before starting its “processing” phase. It also requires 4 time units to “process” the data fetched from memory. A similar interpretation for IP-B in Flow-2 indicates that it also requires 3 memory requests before it can start its computation which takes 6 time slots. *For convenience of illustration, all time slots in these examples are normalized to the time taken to process one memory request.* The QoS deadline for Flow-2

Figure 2: Examples showing the advantages and drawbacks of different QoS-aware memory schedulers. Timeline is measured as units of memory requests, and all time slots are *normalized* to the time taken to process one memory request.

occurs after the deadline of Flow-1, and is shown by vertical red and blue lines near the end of the time-line. Within this deadline, flows 1 and 2 are required to complete 4 executions of IP-A and IP-B, respectively. For simplicity, let us assume that the individual requests of IP-A and IP-B alternate in the FCFS order, and this is the order chosen for service by FR-FCFS without any row-buffer locality. As we can see, the strict alternation in FR-FCFS causes Flow-1 to miss its 24 slot deadline (since it could not be scheduled by 20th slot), even if Flow-2 finishes its tasks before its deadline.

DynQoS Scheduler Instead of treating all requests the same (as in FR-FCFS), DynQoS dynamically adjusts the priorities. It was originally proposed [10] to differentiate between CPU and GPU memory requests, and we have adapted it to our context. DynQoS is a “progress-aware” scheduler and tracks two metrics for each IP in each request slot: (i) *actual progress*: Flow-1 has 12 requests to complete 100% of its work; so, completing each request adds 8.3% to the actual progress; and (ii) *expected progress*: Flow-1 should complete 100% of its memory requests within 20 slots; so, the expected progress of Flow-1 is $12/20 = 0.6$ per slot (or 5% per slot). It is important to note that this scheme prioritizes only when an IP is in the urgent zone. DynQoS defines “urgent zone” as to be within the 90% of the QoS deadline. For example, if a frame has a deadline of 16.6 ms, all its requests in the memory queue from the 15th ms are assumed to be in the urgent zone. The scheduler gives *higher* priority to the requests in the urgent zone whose actual progress is *less than* the corresponding expected progress, and *lower* priority for all IPs whose actual progress is *greater than* their expected progress. Thus in Fig. 2a, DynQoS is similar to FR-FCFS, in the beginning. But, by prioritizing Flow-1 in the urgent zone, it is able to meet the deadlines of both flows³.

However, consider now the scenario in Fig. 2b. Each flow again uses one IP: IP-C is used by Flow-1, and IP-D by Flow-2. IP-C requires 1 memory request to complete before beginning its compute phase, which itself takes 4 time slots. IP-D, on the other hand, requires 6 memory requests to complete, before its computation which takes 6 time slots. In this example, even though DynQoS tries to prioritize Flow-1, it comes a bit too late, i.e., the urgent zone arises only for the last memory request. This happens because, Flow-1 has fewer memory requests for the required computation, and there are not too many memory requests in the urgent zone to prioritize and make up for any lost time.

DASH Scheduler As an alternative to priority requests in an urgent zone at the end, we have adapted a version of DASH [25], which distributes priorities at a finer granularity (request level). In this scheme, the scheduler starts tracking the “expected progress” and “actual progress” for each flow, and assigns the priority calculated as in DynQoS for every request (right from the beginning). For the example in Fig. 2b, Flow-2’s progress is 0.24 per slot, while Flow-1 makes 25% progress each time its request get serviced. So, for every four requests of Flow-2 that get serviced, 1 request of Flow-1 gets serviced right from the beginning, enabling both flows to meet their respective deadlines.

However, consider now the scenario depicted in Fig. 2c. Flow-1 uses IP-E which needs 2 memory requests to be serviced before its computation which takes 3 time units. Flow-2 uses IP-F which requires 4 memory requests to be serviced before its processing for 6 time slots. Flow-1 should process $4 \times 2 = 8$ memory requests, while Flow-2 should process $4 \times 4 = 16$, in 25 (0.72 requests per slot) and 22 (0.33 requests per slot) time slots, respectively. Let us assume that, DASH processes Flow-1 in the first slot. Its progress at the end of the slot is 12.5%, which is ahead of the expected progress (7.2%). Hence, it switches to processing Flow-2 for the next 2 time slots, which progresses by 6.3% in each of these slots before switching back to Flow-1 again. Thus in this example, even though the final deadlines are the same, the deadlines to the memory are different due to the differential IP processing times. Consequently, DASH, which does *not* account for IP computation times (and only tries to balance memory progress rates as opposed to memory+computation progress rates), ends up missing the deadline for Flow-2.

3.2 A Flow-Sensitive Scheduler: FLOSS

Two of QoS-aware memory scheduling techniques explained above track the progress of individual IPs and *not* the whole flow. There are three problems with such “IP centric” memory scheduling.

First, each flow in an application consists of a diverse set of IPs (refer Table 1) and has a per-frame QoS requirement. Using a constant/predefined deadline for all IPs involves the risk of becoming too relaxed or too stringent for the frame deadlines. Second, since multiple IPs process different frames in parallel and access the memory concurrently, the memory scheduler should recognize the *actual progress* of various *frames* instead of IPs to satisfy the frame-level QoS. Third, an IP could be shared by different flows and thus, the frame processing time could be affected due to such dependencies. So, even if a light-weight IP finishes well before its deadline, we lose the flexibility to borrow that *slack (idle time)* for better use by the subsequent IPs.

³The details of the calculations used to prioritize requests are omitted in the interest of space, and only the end-results are shown in the figures.

We believe that addressing these three problems is essential in designing a memory scheduler that can guarantee application-level QoS. In this context, we propose a flow-aware scheduler, called “FLOSS” (FLOw Sensitive Scheduling) for mobile platforms. FLOSS embeds the above three design points in scheduling a flow or multiple flows to satisfy the per-frame QoS requirement.

Requirements for Flow-Sensitive Scheduling: The three problems associated with IP centric scheduling imply that we need to be aware of the following parameters: (i) the IPs constituting a flow, (ii) the concurrent flows accessing the memory, and (iii) the progress made by each flow to estimate its bandwidth requirement. The first two parameters are tracked by using the flow details available at the application code level and maintaining them at the OS driver. The third parameter, *progress*, is defined based on the number of memory requests processed. However, as demonstrated in the example shown in Fig. 2c, it is not enough to track the memory requests alone to meet the QoS deadlines. Consequently, we track the actual progress as an estimate of the compute and memory stall time for each flow. We define this as “*Work*”, which is a tuple consisting of the compute time of each IP (t_{IP}) in a flow and the total number of memory requests for that flow. These parameters are made aware at the OS driver (explained in Sec. 3.3) at the beginning of a new flow by the application. To enable these, we do minimal changes to the application code, using a function call to the OS driver. For example, the video playback code is instrumented with the following function call before processing a video:

$\text{RegFlow}(\text{FLOW} = \langle \text{FLASH} \rightarrow \text{VD} \rightarrow \text{DISPLAY} \rangle, \text{WORK} = \langle t_{\text{flash}}, t_{\text{vd}}, t_{\text{display}}, \text{memory_requests_frame} \rangle, \text{QoS_frame})$

This function registers the flow with three IPs in sequence, namely, FLASH, VD, and DC, with the OS driver. Along with the *FLOW*, the function also supplies the necessary parameters (execution time of flow, number of memory requests, and QoS of the flow) for tracking the *progress* of each frame in the flow and calculating the *FlowRate* (described in Sec. 3.2). We also developed a simple library call to get the estimated time taken (t_{IP}) for each IP (e.g., t_{flash} is the processing time for FLASH IP). [14] shows that predicting t_{IP} based on history (initialized from vendor data) is accurate for 95% of the time. To compute this estimate, we initialize each flow with an IP’s execution time obtained from the vendor data sheet and update the estimate with a feedback from the real execution time seen at the hardware after completing each IP call. The number of memory requests is inferred from the frame size, which can be obtained from the frame header at runtime. When an application invokes this function, the flow information is stored in the system, similar to the flow maintenance found in VIP [15].

Bandwidth Allocation Using FlowRate: Once the OS driver is made aware of all the three parameters described earlier for all the concurrent flows in the system, FLOSS uses this information to decide the “bandwidth allocation” for different flows. For example, in the top-left cell of Fig. 3, we present a single non-PIP flow $B \rightarrow C \rightarrow D$ accessing the memory. Since only one flow is in progress, FLOSS allocates the whole memory bandwidth to that flow. On the other hand, when the system registers multiple concurrent flows, it should partition the memory bandwidth among all the concurrent flows judiciously. To estimate the relative memory bandwidth demands of all the requesting concurrent flows, we *normalize* the relative progress of various flows with respect to their

Figure 3: FLOSS: Rate calculation under various scenarios.

Work and *QoS*, and define a metric known as *FlowRate*. As explained above, $Work_{flow}$ is the estimated progress to be made by a flow and *QoS* is the time available to make that progress. Hence, $FlowRate_{flow} = \frac{Work_{flow}}{QoS_{flow}}$ is the rate at which a flow is *expected* to progress, and can be used to quantify the relative demands of two or more flows for accessing the memory.

Concurrent independent flows: We can now compute *FlowRates* for any number of independent concurrent flows using this formula. This is depicted in the 2nd column of Fig. 3. To allocate bandwidth for these concurrent flows, we proportionally divide the available memory bandwidth according to their corresponding *FlowRates*. This is the scenario explained in the example shown in Fig. 2c for FLOSS, where Flow-1 and Flow-2 have their works as $[8 \text{ memory} + 4 \times 3 \text{ compute}] = 20$ slots and $[16 \text{ memory} + 4 \times 6 \text{ compute}] = 40$ slots, respectively. Since each flow has a QoS deadline of 28 slots, we can compute the *FlowRates* of Flow-1 and Flow-2 as $\frac{Work_{flow1}}{QoS_{flow1}} = \frac{20}{28}$

and $\frac{Work_{flow2}}{QoS_{flow2}} = \frac{40}{28}$, respectively. This is an advantage of FLOSS over DASH or DynQoS right from the beginning. This is, instead of starting with equal priorities in the beginning, we use the relative bandwidth requirements between the flows, and prioritize Flow-2 over Flow-1 ($FlowRate_{Flow2} > FlowRate_{Flow1}$). Thus, when we allocate Flow-2 with a higher bandwidth from the beginning based on a fair queue memory scheduler (detailed in Sec. 3.3), we satisfy the QoS requirements of both the flows with FLOSS, whereas DASH fails to do so.

Concurrent dependent flows: Note however that, in cases where IPs are shared among concurrent flows (refer Table 1), this computation should take each shared IP as the *Work*’s boundary. This is because, only one flow can invoke an IP at any given time. The other flows sharing the same IP get blocked till the current invocation to the IP is complete. To account for the time lost in this blocked state in their respective QoS, we *recompute* the *FlowRates* for all the shared flows after any flow completes the shared IP invocation. This is shown in the 3rd column of Fig. 3, where we assume that the top flow ($B \rightarrow C \rightarrow D$) will reach the shared IP “C” first. Note that, the next column (4th) is another instance of this scenario, where all the IPs in two flows are shared. So, in this case, we will recompute the *FlowRates* for each flow after every IP completion.

PIP+FLOSS: As mentioned earlier, PIP will cause all the IPs to be active at the same time. Thus, to cater to individual IPs with a portion of the memory bandwidth and still be flow aware, we assign

App	IP Flows (with relative work distribution among IPs, W_{IP})
Screenshare	DC(6) - VE(24) - I/O
Video Record	CAM(3) - IMG(6.2) - DC(6); CAM(3) - VE(24) - FLASH(1); MIC - AE - FLASH
Video Playback	FLASH(1) - VD(8) - DC(6); AD - SND
Video Conferencing	I/O - VD(8) - DC(6); CAM(3) - VE(24) - I/O; AD - SND; MIC - AE - I/O

Legend: IP Abbreviations and Expansions

IP Abbr.	Expansion	IP Abbr.	Expansion	IP Abbr.	Expansion
VD/VE	Video CODEC	AD/AE	Audio CODEC	I/O	NW, Radio I/O
IMG	Imaging(DSP)	FLASH	Flash Controller	MIC	Microphone
SND	Sound	DC	Display Controller	CAM	Camera

Table 1: List of our applications and their respective IP flows.

Key	App Mix	QoS required	Description
A1	Screenshare	60FPS	Live stream user screen to remote user
A2	Video Record	30FPS	Record a video
A3	Video Playback	60FPS	Watch one video
A4	Video Conference	30FPS	Two-party Video Conference
W1	2 × A3	60FPS each	Simultaneous streaming, e.g. VR, 3D Video Play
W2	A4 + A3	30FPS + 60FPS	Video conference and play video
W3	4 × A3	30FPS each	Simultaneously play four videos, e.g. Yinzcam
W4	A3 + A2	60FPS + 30FPS	Record video while playing e.g. Gear VR
W5	A2 + A4	30FPS + 30FPS	Recording a live video conference

Table 2: Description of our workloads. In the first column, A* represents a single-app workload, and W* represents a multi-apps workload. Note that W* demand a baseline FPS ≥ 60 .

Core	ARM ISA; 4-cores; 32B cache line; 32 KB L1-I/D; 512 KB L2
Memory	LPDDR3; 2 channels; 1 rank per channel; 8 Banks per rank; Vdd = 1.2V; t _{CL} , t _{RP} , t _W = 15, 18, 18 ns
Frame Specs	Audio: 16KB size; Video: 4K (3840x2160), Camera: 2560x1620

Table 3: Platform details.

the FlowRate according to the Work of each IPs in the flow, as illustrated in Fig. 3.

CPU memory requests: FLOSS considers CPU memory requests as being highly latency sensitive and therefore, always prioritizes them. We discuss this further in Sec. 4.2.

3.3 Implementation Details

OS Driver Support: At the Android OS driver side, the Hardware Abstraction Layer (HAL) [8] needs to support the following tasks. When apps invoke RegFlow as discussed above, the function implemented in HAL will take care of storing the flow information, compute the FlowRate as described in Sec.3.2, and update the flow info with its rate to the memory controller. If there are common IPs between two or more flows, the hardware is updated with all their new FlowRates. Whenever an IP starts with a new frame request, HAL updates the active IP of the corresponding flow in the system agent. Due to PIP, the driver updates all the IPs that are active in the whole flow. As the SA is the first-leg of the memory requests sent by the IP, it will be able to tag the flow from which the memory request has originated. This flow-tagged request then reaches the virtual clock based memory scheduling system and gets prioritized as described below.

Hardware Support: The virtual clock based queuing systems have been studied in the network domain [27, 30], and have also been adapted for memory scheduling in [17]. We adapt this system to use our FlowRates as *weights*. This ensures fairness by ensuring a bandwidth allocation proportional to the FlowRates computed for all the concurrent flows as described earlier. After a flow is completed, the flow details are invalidated from the SA and memory scheduler. Finally, the flow information along with the FlowRate are stored in the virtual clock based rate proportional memory scheduler for it to distribute the available memory bandwidth accordingly.

4 EXPERIMENTAL EVALUATION

4.1 Experiment Setup

Target Platform: Our target system is a mobile SoC containing multiple cores, devices, accelerators, SA, and LPDDR-3 memory.

The SA in our platform integrates the interconnect (which is typically a high-bandwidth bus), the request/response queues, and the memory controller (MC). We use a 4-core system in our experiments, and the workloads used in this study are described in Table 2. We consider a set of IPs that are commonly used in mobile systems [21] and a list of all IP-flows used in our workloads is given in Table 1. Some of these apps have multiple flows (up to 4), and tend to exercise multiple cores and IPs concurrently during their executions. For more complex use-cases like video-recording and video conferencing, multiple concurrent flows (2+) are present. Note that, some of the IPs in these flows are *time-shared*. For instance, in W5, the same VE-IP is time-shared between video-recording and video conferencing.

Evaluation Platform: Our evaluation framework builds on top of GemDroid [16], which utilizes Google’s Android Emulator [5] to capture the complete system-level behavior. The emulation platform runs Android 4.4.4, and its associated drivers for emulating various devices and accelerators. We capture IP and CPU traces from the instrumented emulator and feed them to the GemDroid to perform full-system trace-based simulation of various platforms that include cores, memory components, and accelerators. Note that, the invocation order of IPs is replayed as in the original sequence. Our platform also incorporates DRAMSim2 [20] to model LPDDR-3 power and performance. We simulate 4 cores which can support up to 4 applications in parallel, each running for around 80 frames. Further details about the parameters of the modeled system are provided in Table 3.

4.2 Results

We present performance brought on by different memory scheduling schemes with PIP enabled. Note that, in all the discussions below, “baseline” refers to non-PIP execution with the FR-FCFS.

Fig. 4a and Fig. 4b show the FPS and energy efficiency improvement per frame due to different schemes. It can be seen that PIP+FR-FCFS (FR-FCFS) improves the overall FPS by 74%. PIP+FLOSS (FLOSS) posts an additional QoS improvement of 12%. The corresponding overall energy efficiency improvement obtained is 5%.

We now look at the impact of the different schemes, in terms of the memory stall time per frame (see Fig. 4c) for all the applications and workloads. Note that, the memory stall is on the critical path in processing the frames and is crucial for improving FPS. Memory requests in an SoC may originate from two sources, namely, CPUs and IPs. Therefore, the overall memory stall time for a flow consists of CPU memory stalls and IP memory stalls. We show the breakdown of the overall memory stall time in Fig. 4c. FLOSS, compared to other memory schedulers, provides the lowest overall memory stall time per frame across all the applications and workloads.

However, in some cases, we observe that FLOSS has higher IP stall time than the other schemes. This is the case in A1 (0.01 ms higher than FR-FCFS), A3 (0.28 ms higher than DynQoS), W2 (0.88 ms higher than DASH), and W4 (0.84 ms higher than DynQoS). This is because, FLOSS prioritizes CPU memory requests over IP memory requests. Consequently, the CPU memory stall time is reduced in these apps and workloads, and hence, compensates appropriately to save the overall memory stall time. Note that, since FLOSS calculates the appropriate rate at which the IPs are serviced, it keeps the IP memory stall time low.

It is also interesting to note that, in A3, W1, W3 and W4, the memory stall times do not correlate to the FPS improvements in some of the schemes. To explain this phenomenon, we focus on A3 and study the variations in IP utilization by different schemes. Note

Figure 4: Different memory scheduling schemes on FPS and energy efficiency per frame.

that, DynQoS does *not* prioritize among IP requests till the "urgent zone". This in turn balances all the IPs' utilization, as shown in Fig. 4d. Hence, its FPS improvement (138%) is lesser than that of the baseline PIP+FR-FCFS (160%). The same scenario can be observed for DASH, wherein DC being at the end of the flow always receives a greater priority than VD (DC has more urgent deadline compared to VD). This worsens the FPS improvement even further to 123%, though having a very similar memory stall time as FR-FCFS and DynQoS (4.61 ms). FLOSS, on the other hand, considers *both* work and QoS at a flow level, and hence improves the utilization of the bottlenecked IP (FLASH in this case).

5 RELATED WORK

Early memory scheduling schemes targeted only the CPUs [11, 12, 23, 24], and consequently, they cannot be directly extended to SoC systems, which need to accommodate IPs with differing traffic patterns and deadlines. Recent work have focused on optimizing performance/energy on CPU and IPs [4, 18, 28, 29]. Relatively few prior works have investigated memory scheduling techniques in the context of SoCs [3, 10, 13, 25]. Of the ones closest to our work, [13] proposes a memory scheduling policy that guarantees certain latency and bandwidth across the system. [10] investigated a dynamic memory scheduling policy for the CPU and GPU, adjusting their request priorities. [25] proposed the same for a generic heterogeneous SoC system containing multiple accelerators. We have already detailed their shortcomings in Sec. 3 with examples, and compared our approach to them in Fig. 4.

6 CONCLUSION

Current mobile devices support a variety of data-intensive applications that need real-time processing of data in an interactive manner. We show in this paper that the current programming paradigm for IPs used in mobile devices is not very effective in handling such flows since it treats each IP in isolation, without considering the other IPs that belong to the same flow. Since none of the existing memory schedulers employs the concept of a "flow" in scheduling memory requests, we also propose a flow-aware, rate-proportional memory scheduler based on virtual clock, called FLOSS, that considers the QoS demands of a single or multiple flows in scheduling requests. The proposed FLOSS memory scheduling technique is shown to perform better than the FR-FCFS, DynQoS and DASH schedulers, by providing an additional 12% improvement in FPS, and an additional 5% improvement in energy efficiency per frame.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants #1213052, #1302557, #1317560, #1320478, #1409095, #1439021, #1439057, #1626251, #1629129, #1629915, #1714389, #1526750, DARPA/SRC JUMP grant, and Intel.

REFERENCES

- [1] Apple. 2013. iPad Air 2 Technical Specifications. <https://goo.gl/uiP2em> (2013).
- [2] Apple. 2015. iOS 9 - Apple. <http://apple.com/ios9> (2015).
- [3] S. Chu, M. Lo, and H. Yang. 2008. MediaMem: A dynamically adjustable memory subsystem for high-bandwidth required multimedia SoC systems. *APC-SAC*.
- [4] J. Gaur et. al. 2016. Instruction and Logic for Managing Cumulative System Bandwidth through Dynamic Request Partitioning. In *U.S. Patent No. 20160179387*, 14/971,057, 2016.
- [5] Google. 2013. Android SDK - Emulator. <http://developer.android.com/> (2013).
- [6] Google. 2014. Grafika. <https://github.com/google/grafika> (2014).
- [7] Google. 2015. Hummingbird on Nexus6P. <http://goo.gl/42lpoe> (2015).
- [8] Google. 2016. Android HAL. <http://goo.gl/GO2w7W> (2016).
- [9] GStreamer. 2017. GStreamer: open source multimedia framework. In <https://goo.gl/YGhpcx>.
- [10] M. Jeong, M. Erez, C. Sudanthi, and N. Paver. 2012. A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC. In *DAC*.
- [11] Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. 2010. Thread cluster memory scheduling: exploiting differences in memory access behavior. In *MICRO*.
- [12] Jagadish B. Kotra, Narges Shahidi, Zeshan A. Chishti, and Mahmut T. Kandemir. 2017. Hardware-Software Co-design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling. (2017).
- [13] K. Lee and T. Chang. 2006. *Essential Issues in SoC Design Designing - Complex Systems-on-Chip*. Springer, Chapter SoC Memory System Design.
- [14] N.C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M.T. Kandemir, R. Iyer, and C.R. Das. 2015. Domain Knowledge Based Energy Management in Handhelds. In *HPCA*. 150–160.
- [15] N.C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das. 2015. VIP: Virtualizing IP Chains on Handheld Platforms. In *ISCA*. 655–667.
- [16] N.C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. Kandemir, and C. R. Das. 2014. GemDroid: A Framework to Evaluate Mobile Platforms. In *SIGMETRICS*.
- [17] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. 2006. Fair Queuing Memory Systems. In *MICRO*. 208–222.
- [18] P. V. Rengasamy, H. Zhang, N. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. 2017. Characterizing diverse handheld apps for customized hardware acceleration. In *IISWC*.
- [19] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. 2000. Memory Access Scheduling. In *ISCA*. 128–138.
- [20] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. Dramsim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters* (2011), 16–19.
- [21] Samsung. 2014. Samsung Galaxy S5. <http://goo.gl/MCRkcU> (2014).
- [22] Samsung. 2015. Open Apps in Split Screen View on Your Galaxy S7. <http://goo.gl/JFYgfV>.
- [23] X. Tang, M. Kandemir, P. Yedlapalli, and J. Kotra. 2016. Improving bank-level parallelism for irregular applications. In *MICRO*.
- [24] Xulong Tang, Orhan Kislal, Mahmut Kandemir, and Mustafa Karakoy. 2017. Data Movement Aware Computation Partitioning. In *MICRO*.
- [25] H. Usui, L. Subramanian, K. Chang, and O. Mutlu. 2016. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *ACM Trans. Archit. Code Optim.* (2016), 65:1–65:28.
- [26] Yinzcam. 2015. YinzCam, Inc. <http://www.yinzcam.com/> (2015).
- [27] H. Zhang. 1995. Service Disciplines for Guaranteed Performance Service in Packet-Switching Networks. *Proc. IEEE* (1995), 1374–1396.
- [28] H. Zhang, W. Han, F. Li, S. He, Y. Cheng, H. An, and Z. Chen. 2014. A Criticality-Aware DVFS Runtime Utility for Optimizing Power Efficiency of Multithreaded Applications. In *2014 IPDPSW*.
- [29] H. Zhang, P.V. Rengasamy, S. Zhao, N.C. Nachiappan, A. Sivasubramaniam, M. Kandemir, R. Iyer, and C.R. Das. 2017. Race-to-sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds. In *MICRO*.
- [30] L. Zhang. 1990. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. In *SIGCOMM*.