VIP: Virtualizing IP Chains on Handheld Platforms

Nachiappan Chidambaram Nachiappan[1], Haibo Zhang[1], Jihyun Ryoo[1], Niranjan Soundararajan[2], Anand Sivasubramaniam[1], Mahmut T. Kandemir [1], Ravi Iyer[2], and Chita R. Das[1]

[1]The Pennsylvania State University
[2]Intel Corporation
[1]{nachi, haibo, jihyun, anand, kandemir, das}@cse.psu.edu
[2]{niranjan.k.soundararajan, ravishankar.iyer}@intel.com

Abstract

Energy-efficient user-interactive and display-oriented applications on handhelds rely heavily on multiple accelerators (termed IP cores) to meet their periodic frame processing needs. Further, these platforms are starting to host multiple applications concurrently on the multiple CPU cores. Unfortunately, today's hardware exposes an interface that forces the host software (Android drivers) to treat each IP core as an isolated device. Consequently, the host CPU has to get involved in the (i) processing of each frame, (ii) scheduling them to ensure timely progress through the IP cores to meet their QoS needs, and (iii) explicitly having to move data from one IP core to the next, with main memory serving as the common staging area.

We show in this paper through measurements on a Nexus 7 platform that the frequent invocation of the CPU for processing these frames and the involvement of main memory as a data flow conduit, are serious limitations. Instead, we propose a novel IP virtualization framework (VIP), involving three key ideas that allow several IPs to be chained together and made to appear to the software as a single device. First, chaining of IPs avoids data transfer through the memory system, enhancing the throughput of flows through the IPs. Second, by using a burst-mode, the CPU can initiate the processing of several frames through the virtual IP chain, without getting involved (and interrupted) for each frame, thereby allowing better energy saving and utilization opportunities. Removing the CPU from this loop, requires alternate orchestration of frame flows to ensure QoS guarantees for each frame of each application. Our third enhancement in VIP creates several virtual paths, one for each flow, through these IP chains with the hardware scheduling the frames to enforce QoS guarantees despite any contention for resources along the way. Our experimental evaluations demonstrate the effectiveness of VIP on energy consumption and QoS for multiple applications.

ISCA '15, June 13 - 17, 2015, Portland, OR, USA
 978-1-4503-3402-0/15/06 $15.00
http://dx.doi.org/10.1145/2749469.2750382

1. Introduction

Handheld devices are being increasingly used for a wide number of very demanding and interactive applications that work on frames of data within soft real-time bounds. Examples of such frame-based applications include video and audio streaming, video chat and social networking, and interactive games. It has been reported [7] that out of the total time spent by users on these devices, more than 80% is spent in such interactive frame-oriented applications, making this an important class of applications to sustain. These applications are characterized by large volumes of data that need to be periodically processed within soft real-time bounds (specified by a frame rate $\sim$16 ms bound for providing 60 FPS), and delivered to the output devices, e.g., HD displays, wireless interfaces and flash storage. It has long been recognized [29] that general purpose CPU cores are inefficient, especially from the energy-efficiency perspective, to process such applications. Instead, customized accelerators (termed IP cores in this paper) are often employed to avoid the limitation of von Neumann architectures in such applications, to attain the high energy efficiencies [12]. From the main CPU core's perspective (the software running on it), each IP core is considered as a *separate device*, and there is a hand-off mechanism – involving synchronization, data and results communication – to offload the computation to the IPs.

When we examine the interface that exists today between such IPs and the CPU cores, there are two main deficiencies, leading to both performance and power inefficiencies:

- Many of today's applications use more than one IP. For instance, Skype video conferencing involves, besides the CPU, the camera, video encoder and display IPs. They require a pipelined processing of the data flowing from one IP to another. However, today's hardware simply exposes an interface to each IP as if it were an isolated unit, and it is up to the host software (the libraries and device drivers running on the main CPU) to explicitly chain these IPs

The authors would like to confirm that this work is an academic exploration and does not reflect any effort within Intel.

together. As a consequence, this implies (i) unnecessary extra data movement (usually involving memory that is both a bandwidth constrained and power consuming resource) that could have been circumvented by directly making data flow from one IP to the next, and (ii) the host CPU has to get involved in orchestrating the movement of each (or some small sub-group) frame, both taking away the cycles that it could have spent doing other useful work and, more importantly, consuming power due to periodic activities as opposed to going into a low power mode (sleep state) for prolonged periods.

- Today's handhelds run multiple applications, whether on the same host core or on the multiple cores available today. When multiple applications depend on IPs, including possibly some common IPs, it introduces several points of contention – the shared IP(s), the data flow paths, and the memory. Even if not all IPs in the data flow in the applications are shared, the points of contention can severely restrict the concurrency, effectively making the parallelism offered by multiple cores under-utilized. The problem is somewhat similar to message flow in networks, where even if source-destination pairs of different messages may have no commonality, the fact that one or more links in their respective paths intersect, can lead to unnecessary contention and back-pressure as has been well studied [14]. Throwing more hardware (i.e., increasing the number of IPs) to handle the contention is not desirable from the cost and/or energy viewpoint, especially when we are not clear if we are extracting the maximum utilization out of the existing IPs.

In this paper, our approach to tackling these issues in the context of IP-data flows on handhelds uses a similar solution strategy to network routing. Even if flits of a message move hop-by-hop from one switch to the next, they do not necessarily need the source involvement after an end-to-end channel is set up (in wormhole routing and circuit switching strategies), making it a lot more efficient for bulk data transfers (with is the case with these bulky frames). Further, each such message is relatively isolated from another (i.e., blocking of one message is not always holding up network resources that could allow another to proceed) by virtualizing these channels of data flow in the network [16, 15] in the shared resources.

Borrowing these ideas from the network domain, this paper investigates the following questions to improve the performance and energy-efficiency of data flow across IPs of a handheld in these emerging and increasingly popular frame-based applications:

- *Rather than treat each IP as a separate piece of hardware, can we dynamically create/initiate a Virtual IP (VIP) hardware chain of IP cores that can be treated as a single device in software by the host?* Consequently, explicit periodic data movement with CPU and memory involvement can be avoided, for better performance and energy efficiency.
- *Can we instantiate a version of this Virtual IP hardware chain for each application's data flow through its sequence of IPs?* Essentially we are trying to create a virtual channel of data flow across IPs for each application's data flow.
- *What hardware enhancements are needed to chain these IPs to create a VIP chain? What should be the APIs/Interfaces exposed by the hardware to facilitate this?*
- *What software APIs are needed to create/initiate these VIP chains for each application? Can we achieve this without significant alterations to existing application codes?*
- *How do we allocate resources, and schedule the processing and data flows to ensure that each such VIP channel progresses as per the corresponding application's flow rate requirements, without interfering with each other?*

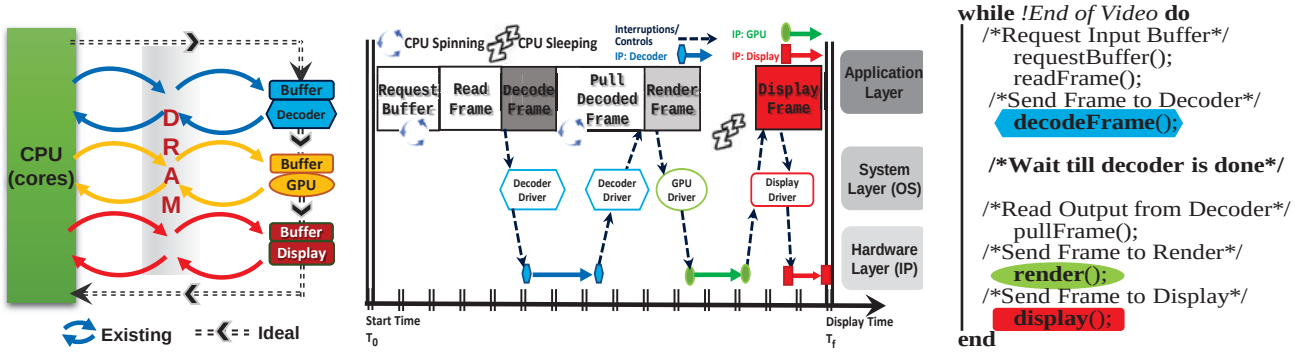
Chaining individual IPs and exposing a smarter interface to this integrated chain, relieves the main CPU core from explicitly moving frames across IPs. Further, by introducing a burst mode of operation, the CPU can send several such frames in one burst through this chain, avoiding the need to take interrupts upon completion of each frame. While these two enhancements alone can enhance the throughput of frame flows and energy efficiencies, we may consequently lose out on the orchestration that the CPU was doing earlier in scheduling individual frames of multiple applications to ensure their QoS needs. As a result, our VIP proposal additionally incorporates enhancements to conduct rate-based flow control/scheduling mechanisms to meet each frame's QoS requirements.

To our knowledge, this is the first paper to propose such an approach to virtualize the flow of data across multiple IPs for co-existing applications that are involved in periodically processing frames. This approach can help boost the utilization of existing IPs, while still allowing individual applications to meet their required frame/flow rates. Such a solution can increase the availability of host cores for other useful work by relieving them from the mundane role of data transfer conduits between IPs. It also achieves this goal without requiring additional IPs to be introduced into the already power and real-estate constrained handheld device.

We first conduct a series of measurements on several handheld platforms to capture the CPU, memory and IP traffic and their timing constraints. Then, extending the GemDroid simulator [9] to support our proposed virtualization framework for multiple applications, we conduct an in-depth evaluation of the proposed designs. Our experimental evaluations with two-application workloads show that the proposed VIP scheme can provide on an average $\sim$22% energy saving, and $\sim$15% improvement in QoS (frame drop rate) compared to just enabling IP-to-IP direct communication.

2. How Flow-based Applications Work?

In this section, we illustrate how a typical display bound flow-based application works on current handhelds, describe how the problem exacerbates with multiple applications, and finally present the two major types of inefficiencies present in the system: *periodic CPU interrupts* and *memory as a bottleneck*.



(a) Video Player's conceptual data flow (present vs ideal)

(b) Time line depicting control-flow in Video Player

(c) High-level algorithm of Video Player

Figure 1: Video playback flow

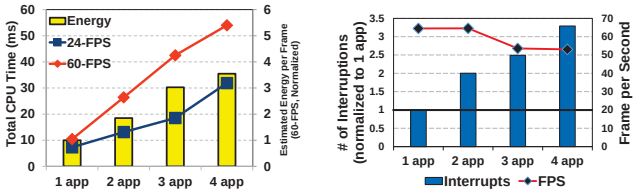
2.1. Single Application Scenario

Currently, most mobile apps follow a sequence of steps through multiple IPs to display frames of content onscreen. Fortunately, Android [26] supports different SDKs (OpenGL ES [24], RenderScript [25]) besides several layers in the Android framework to prevent the need for app developers to directly interact with the IP hardware and in turn avoid code portability issues. Therefore, applications use API calls to talk to the driver which in turn handles the requests to the corresponding IP. The drivers present in the Android OS stack virtualize the underlying hardware components (essentially, a single IP), as applications neither have any specific knowledge about the IP nor do they know about whether other applications are accessing them. To illustrate all the above, we study a typical video-player application (Grafika [23], available as open-source) where we show when the APIs are invoked, and how the drivers are called (shown in Figure 1).

As shown, the CPU does some simple computation and invokes the IPs, through their drivers, by providing the request details, frame size, and pointers to the locations that need to be read from and written back to. Once the IP processes the data, its output data is transferred from its local buffers to main memory and it notifies the CPU with an interrupt. The interrupt is handled by the OS, and a call-back response is sent to the application which then invokes the next IP in pipeline though the corresponding driver. This shuttling of data back-and-forth between the CPU, memory, and IPs is shown in Figure 1. This sequence is repeated for each IP until the frame gets displayed. Also, the above series of events are repeated for each and every frame. An optimal approach would be to bypass the CPU interrupts for every IP, as well as bypass memory by making each IP directly communicate with the next one, as shown in Figure 1 (as Ideal). Although the CPU does not do computationally intensive tasks, it cannot go to deep sleep states because of it being the "task-master" that handles API calls besides memory allocations.

2.2. Multiple Application Scenario

Stock Android OS does not support concurrent multiple application execution currently. In variations of Android, where multiple application execution is supported, IPs are typically not "tied" to an application through its run-time. Multiple applications multiplex the IPs over time. They queue their requests to the driver unaware of other application(s). To understand how a multiple-app system would work, we instrumented the above mentioned open source video-player application to support up to four video playbacks and ran it on a Nexus 7 device. Each concurrently running video player would inject its requests into the video-decoder IP, and we found the following from our experiment: (1) Video-Decoder (VD) accelerator/IP limits the number of requests that can be queued. For example, we found that in Nexus 7, the queue size is seven; beyond this, the CPU/driver is blocked from sending additional requests. (2) In a multi-app scenario, if one application blocks a shared IP by enqueueing multiple requests continuously, the other applications can suffer. Some level of control or coordination at the application/driver/OS level is needed to ensure fairness, and, (3) as the number of video-player instances is increased, video-playback quality decreased visibly[1], as well as the overheads (CPU activity) observed at the cores increased substantially. In the next section, we explore reasons for these inefficiencies.



(a) Profiled CPU Active Time and Energy Consumption

(b) Increase in number of interrupts handled by CPU for VD

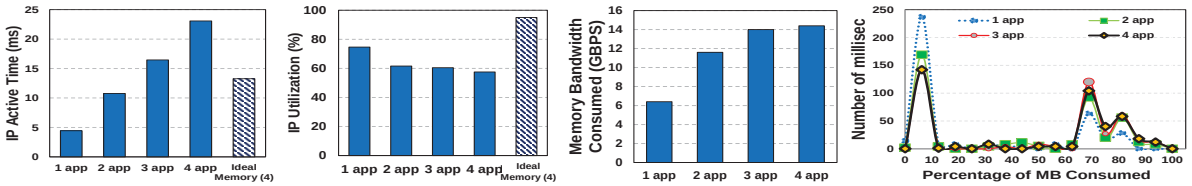
Figure 2: Energy inefficiency of cores in handling interrupts and in scheduling each frame.

3. Motivation: Inefficiencies in Current Systems

Android is built on a traditional software-based OS model which virtualizes the underlying hardware[2]. Also it does not

[1] A Nexus tablet was able to run four concurrent lower-quality videos but four HD videos were not runnable. Another tablet from Asus (MemoPad 8) ran four HD videos, but at a low FPS.

[2] By virtualizing the hardware, the OS hides the hardware details and provides an illusion of exclusive access to each application.



(a) Total IP active time (b) IP-Utilization (c) Avg. Mem. BW (d) Time Distribution of Mem. BW

Figure 3: Inefficiency due to memory

distinguish these frame-based IPs such as graphics, video and imaging from other traditional devices incurring fewer interrupts. It fails to recognize the fact that (1) these IPs need to process frames of data within a limited amount of time, (2) CPUs are invoked at regular intervals to do a specific task per frame, (3) it is not just one driver that runs on the CPU – it is a sequence of multiple drivers invoking different IPs that need to work in unison to produce the final frame, and (4) drivers and the OS work at a much higher level that they do not take into account the hardware inefficiencies such as stalls due to memory traffic while scheduling requests. Therefore, the conventional virtualization model in the Android stack does not scale well in mobile devices.

3.1. Inefficiencies of CPU Interrupts and Per-frame Processing

When running multiple applications, multiple cores need to be active to setup IP calls and handle their interrupts. Consequently, the cores are kept busy, increasing the overall energy consumption. Moreover, even if the IPs are fast enough, such a dependency on the CPU to setup each IP for every frame, will impact the overall system throughput.

To quantify the impact of CPU's active time, on a real platform, we instrumented Grafika's source code to profile the time taken by the CPU to request input/output frame buffers, set up decoders for frame input, read the next frame, and render it to the display. This includes the time when the CPU is stalled when an IP is busy. We have also instrumented the application to support decoding multiple video streams in parallel, and render all of them on screen. Figure 2 shows the *summation* of active CPU time (across all cores) to display 24-FPS video and a 60-FPS video when multiple copies of the same video is played. It can be observed from this plot that the total CPU active time to display a frame for each video stream increases with increase in number of applications. To meet the 16-ms deadline for 60-FPS HD videos, multiple cores are run in parallel to set up each frame. Also, with the higher frame rate requirement, the overall activity across all the cores increases [3]. In Figure 2(b), we show the number of times the CPU cores were interrupted. With 4 applications, we find a 3X increase in the number of interrupts when compared to running a single application.

[3] As per component energy drain cannot be captured on a real device, we use the simulator to obtain an estimate of the energy consumed by the cores for the same application for a period of 1 second.

3.2. Performance Inefficiencies due to shared resources

To complete the analysis, we also look at the other factor that impacts the overall efficiency – amount of time that an IP is doing useful work when it is active (processing one frame). As mobile devices start to support multiple applications, increased contention for shared resources like system-agent, memory controllers and DRAM leads to drop in utilization of the IPs. The results for a video-decoder IP running *n*-copies of video-player is shown in Figure 3. We see in Figure 3 (a) the IP processing time increases with the number of requests as expected. In Figure 3 (b), we see the overall utilization of an IP decreases with increase in number of applications. Note that, with an ideal zero-latency memory (denoted as `Ideal` in figure), utilization is close to 100%. Due to "over-subscription" of IPs, when 4 applications are run concurrently, it takes 23ms to process and display a frame which exceeds the 60 FPS deadline (16-ms per frame) by a large margin. Comparatively, with an ideal memory, even when handling 4 applications, the IP is able to process the frame well within the 16ms deadline.

The low-utilization observed in Figure 3 (b), and the failure to meet the deadline when running 4 applications can be directly attributed to the memory system. Figures 3 (c) and (d) demonstrate that the average memory bandwidth increases with the number of applications and the percentage of time when the memory is close to its peak bandwidth (>80%) is high.

Summary: To summarize, as mobile devices start to support more applications simultaneously, current systems do not seem to scale. This is attributed to two main reasons – too frequent CPU interrupts and memory stalls affecting IP throughput. *Even with multiple applications, we see that IPs are not fully utilized, hence adding more IPs will not resolve the problem.*

4. Virtualizing IP Flows

4.1. Overview of Our Proposed Solutions

Figure 4 illustrates the overall idea with the incremental addition of each technique proposed. Part (a) shows the baseline system where the CPU invokes IPs and orchestrates control flow for each frame. On the other hand, (b) depicts the scenario when IPs communicate between themselves *without using memory as an intermediary.* In this technique, core sends a "super-request" that flows through a sequence of IPs.

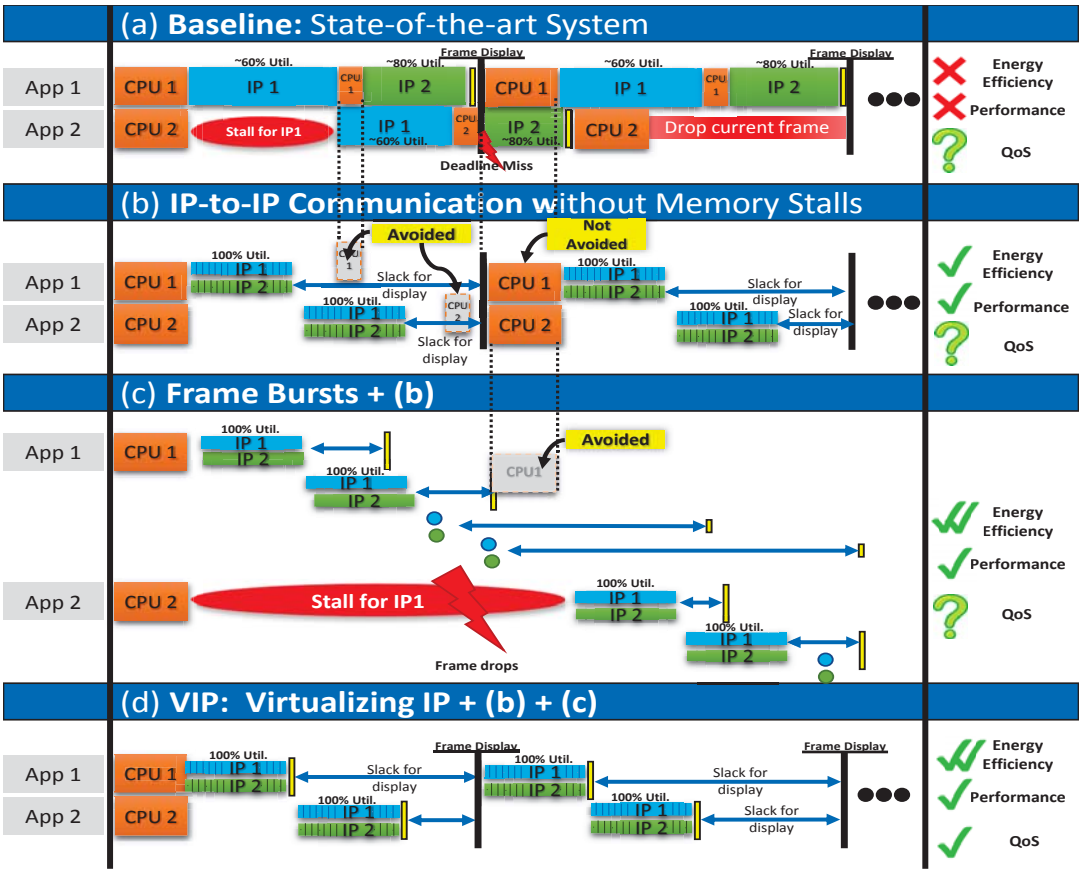


Figure 4: Detailed Overview Of Virtualization at Multiple Levels

This technique eliminates memory stalls, and as a result, we expect improvement in per-frame processing time of IPs. In (c), we propose *Frame-Bursts*, where the CPU sends a burst of these super-requests at one instant. Through this frame burst, the CPU schedules a set of frames at once instead of orchestrating IPs for each frame. This reduces CPU active time and interrupts substantially. Finally, in (d), we propose applications reserving *virtual data channels across multiple IPs* - our VIP scheme. Such a methodology ensures that each application can meet its QoS requirement. We propose to achieve this via fine-grained *hardware level virtualization of the IPs*, where they can context switch between multiple requests when needed. This not only increases the throughput, but also limits the impact of *head-of-the-line blocking problem* if an IP is heavily used by one application.

In the subsections below, we discuss each of the above-mentioned parts in greater detail.

4.2. IP-to-IP Communication

As mentioned in Section 3.2, memory stalls play a major role in curtailing IP utilization to sub-optimal numbers. To overcome this problem, a few recent works have proposed direct *IP-to-IP communication* [5, 56, 51], where IPs avoid explicit reading or writing of frames to DRAM memory, and directly forward their output to the next IP through caches or flow buffers. Clearly, this enables more effective communication between the producer and consumer IPs leading to reduced memory stalls. Speed mismatch between the IPs is accounted for by careful buffer-sizing and fine-grain synchronization between the IPs. In this paper, we simply adopt this solution to overcome the memory problem.

This IP-to-IP communication enables looking at a sequence of multiple IPs as a single unified resource as each IP autonomously communicates with the next when needed. This eliminates the CPU from needing to setup each IP in a chain of IPs for a frame. Instead data flows through a sequence of IPs avoiding intermediate memory operations.

4.3. Frame Bursts

While IP-to-IP communication relieves the CPU from intra-frame inefficiencies, it still needs to do the task of setting up data frames, pointers, and has to trigger the IP-flow for each frame. (See *Not Avoided* arrow in Figure 4.)

To further reduce such inefficiencies *across* frames, we propose "frame bursts" – where CPUs send aggregated requests to the IPs instead of sending them one by one for each frame. The frame burst request sent to the first IP contains a header packet with information about the processing requirements (such as FPS, frame sizes and IPs in sequence). The IPs can work on them continuously without interrupting the CPU core,

thereby allowing them to go into longer and deeper sleep state, hence saving energy. This solution will need the CPU to intervene only once every *n*-frames, where *n* is chosen in correspondence with application requirements. In this work, we consider three major classes of commonly used applications, where we apply frame bursts: (i) video playback which include any video playing or streaming apps, (ii) video encoding which includes video recording, photo-capture, Skype, Google Hangout and other similar 'recording' apps, and (iii) gaming based applications – any touch or flick/swipe based games. Note that the flows considered in these three applications are representative of display bound Android applications.

Video playback and Video Encoding Applications: These applications employ common video formats like VP8/H264 and are apt for frame bursts. Every uncompressed full-frame (known as independent-frame) in the video is followed by *n* predicted frames. Typically, the distance between the independent frames (called as GOP size) is less than 20 frames to keep the quality of video high[3]. During video playback, some videos have variable GOP sizes. Each set of frames in between independent frames can be directly scheduled using a single frame burst. In video encoding apps, this GOP size can be varied, and is usually determined by the user. Burst size can be the same as chosen GOP size or a few frame-bursts can capture the GOP. Due to this, in encoding apps, there is better control and uniformity when choosing the burst sizes. Having a larger burst size will improve energy savings and performance. Further details regarding how these applications can be modified to take advantage of frame bursts is discussed in Section 5.

Game Applications: These applications need careful attention as the frame burst sizing can affect game-play and user-interactivity. A large burst of frames means the graphics and display pipeline will be occupied and the CPU can avoid polling for each frame. Therefore, when a user touches or swipes, the system might not be as responsive as before since the core needs to wake up (or context switch from a different job), leading to reduced interactivity and feedback. In lieu of this, to tailor this technique for gaming apps, we considered two open-source versions of popular games, Flappy Bird[13] (touch-based) and Fruit Ninja[20] (flick/swipe-based). We instrumented both the games to capture user-touch behavior. With the help of 20 users (of varying degrees of player efficiency), each playing both the games for at least 10 minutes, we captured a typical game play behavior. In Figure 5, we plot the average time taken between user-touches for Flappy-Bird. We note that rapid successive clicks will be at least 0.15 sec apart, with most touches (>60%) above 0.5 seconds. Note that 0.15 seconds translate to a leeway of around 10 frames (for a 60 FPS game). With a frame burst of size 5, two frame bursts can comfortably fit in this duration. In Fruit Ninja, we captured the duration of flick and the time interval between

*Open-source version of FlappyBirds and FruitNinja are used.

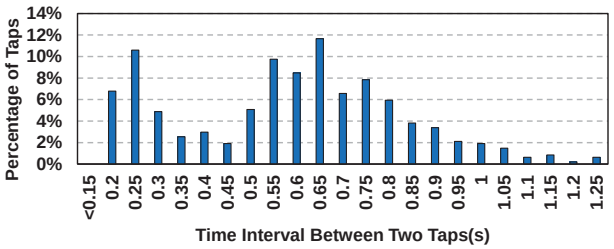
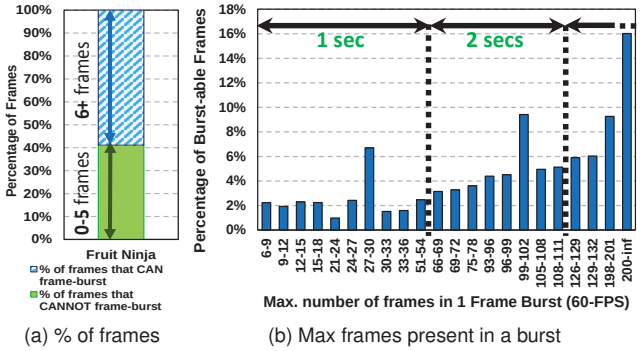


Figure 5: Distribution of percentage of frames in between two taps in `FlappyBird`*



(a) % of frames (b) Max frames present in a burst

Figure 6: Number of frame between flicks in `FruitNinja`*

each flick. In Figure 6(a), we plot the percentage of frames that could be in the frame burst in between flicks. Note that 40% of the frames cannot be burst as the user is flicking and the rest 60% of the frames could be in the burst. In Figure 6(b), we plot the distribution of the 60% of the frames that can be burst as a function of the maximum burst size. Theoretically, the maximum burst size could be more than 180 frames if the flick duration is longer than 3 seconds. For example, there could be 27 to 30 frames in a burst for about 7% of the burstable time. Based on this user behavior, we propose a hybrid approach to design the frame bursts for gaming applications. When a user is not flicking the screen, the frame bursts can be longer, and while flicking, the technique will be disabled for maximum responsiveness.

During the non-flick phases, we limit the frame burst to <10 frames (<160 milli-sec) to ensure that the system will be responsive even if user touches in between the burst. With smaller bursts, the maximum duration between the touch and the responsiveness would be imperceptible to human eye[37].

Note that, in all above applications, frame bursts will *always* improve frame time and FPS. We observed that for gaming apps, with 10 frames per burst, system responsiveness remains unaffected. Further details regarding how game applications can implement frame bursts are given in Section5.

Consequences of Fusing Frame-Bursts and IP-to-IP Communication: While the above two solutions address parts of the larger problem at hand, together they introduce a new problem. With the IP-to-IP communication, IPs are configured to communicate directly with the next IP. In essence, all IPs together form a chain to produce the frames. With frame-bursts, CPU schedules a number of frames and does not intervene for

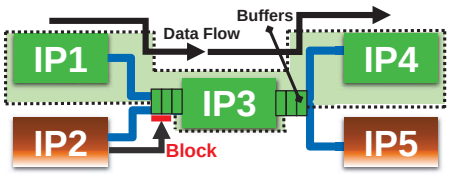


Figure 7: Combining Frame-Bursts and IP-to-IP communication leads to Head-Of-Line blocking by shared IPs.

every frame, expecting all tasks to be complete. But, consider a multiple applications scenario with an IP being shared across their respective data flows. Here, one application's frame-burst could occupy a chain of IPs, and thus, the shared IP could be blocked for considerable periods (for the duration of a full frame-burst, e.g. 80ms for a 60-FPS 5-frame burst). Such a scenario will allow only one of the applications to progress at one point (shown in Figure 7), with the other being blocked resulting in QoS violations. Earlier the CPU (Android) was involved in each frame and could thus avoid such violations. By removing the CPU out of the equation, some other component (the hardware) has to take on this role to ensure QoS, which serves as the motivation for our solution in the next section.

4.4. VIP: Virtualization of IP Chains

The aim of VIP is to enable IP-to-IP communication, circumventing the source (CPU) involvement, as well as avoiding the head-of-line (HOL) blocking phenomena caused in the multiple application execution scenario, as explained above. One of the commonly used networking techniques to avoid HOL blocking in routers is to *virtualize the channels* such that even if one path is blocked, messages in other paths can still pass through. It is the process of *time sharing* the data path by multiple messages that prevents blocking. Inspired by *virtual-channels*, we propose to virtualize the IP data paths by concurrent requests with the goal of preventing HOL blocking, and enabling each application to independently progress to meet its QoS.

Virtualizing IP flows has three major advantages:

• First, when the IP-chain is instantiated by the CPU, akin to network messages, source intervention is avoided. Data flows from the source to destination routed appropriately through the IPs present in the flow. Further, as the data "flows" from one to another, the detour through memory and the CPUs is avoided. This reduces memory energy, CPU energy, and data transfer latency across IPs by minimizing the data movement across the system.

• Second, it satisfies individual application QoS requirements of each co-existing data flow. With multiple applications, each may have its own frame rate requirements. If one application has a slow frame rate, with frame bursts enabled, it might occupy one IP or a chain of IPs for a considerable amount of time, thereby preventing other applications from progressing. By allowing concurrent data flows, VIP enables multiple applications to progress individually.

• Third, due to virtualization, IPs maintain independent contexts for each request, and thus, it gives them complete control to manage their resources in the most efficient way. Traditionally, they are controlled by the CPU (their software drivers), which is unaware of the memory stalls or hardware level bottlenecks faced by the IPs. With fine grained control over which request to prioritize at every instant, along with enabling IP-to-IP communication, overall IP utilizations are maximized.

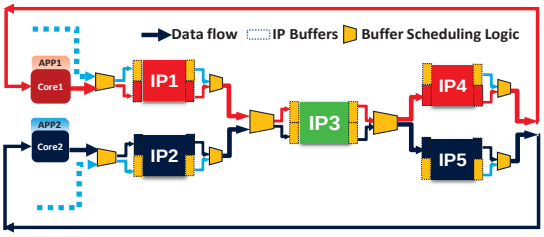


Figure 8: VIP Solution with Multiple Applications

An illustration of multiple apps sharing IP3 is shown in Figure 8. IP3 has two sets of input and output registers, along with contexts for storing requests from both core 1 and core 2. Note that, maintaining the contexts of applications barely require a handful of registers. As shown, IP3 partitions both the application's data flow through its virtual data channels. The scheduling logic controls the rate at which both applications progress and determines who gets the available compute/IP resources at any point in time. It also makes sure that both requests progress at the rate determined by the applications and meet their respective deadlines. In this work, we implement a simple EDF (earliest deadline first) scheduling policy. In the next section, we discuss implementation of VIP in detail.

5. Implementation

To enable VIP, support at the application layer, OS/driver and system layer, and hardware layer is necessary. First, at the application level, the programmer specifies the flow of IPs, frame properties and QoS requirements of each application through an API. This API communicates these to the software drivers, which in turn, passes the data to the hardware IP controllers. We discuss how each one is implemented, and how they work in cohesion.

5.1. Application Level Changes

As discussed before, applications currently are designed to produce frames of data at regular intervals. To enable scheduling a "burst of frames", two modifications are proposed. First, the application programmer needs to specify the *chain of IPs* used and the frame properties of the application. Second, there needs to be an *interface* through which the programmer can initiate a frame burst request. To accomplish these, we propose simple extensions to the existing APIs (application programming interfaces). Using the API, at the beginning of the application, programmer specifies the sequence of IPs that will be needed for each frame using the `open()` calls (as shown in Figures 9, 10, and 11. The API interacts with the software IP driver to instantiate the chain of IPs in the

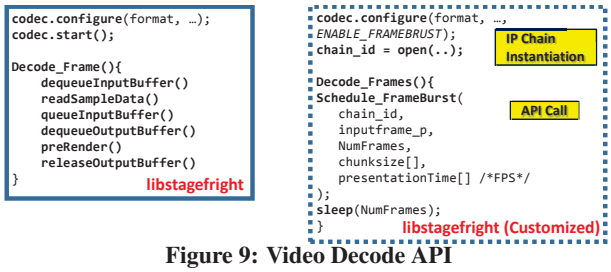


Figure 9: Video Decode API

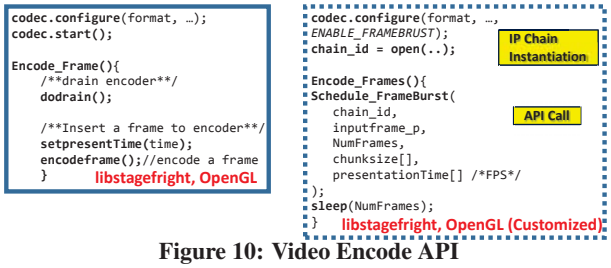


Figure 10: Video Encode API

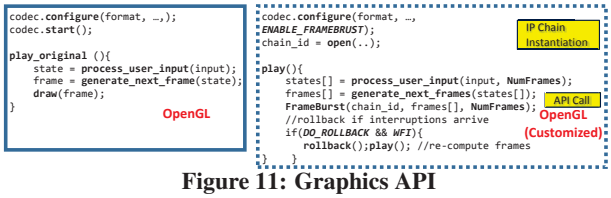


Figure 11: Graphics API

hardware. The driver provides an identifier for the chain of IPs, which the programmer can use when he is scheduling the frame burst requests. Currently, in Android, the programmers need to instantiate all IPs individually. With VIP, we can create/initiate a virtual IP chain that can be treated as a single device through the rest of the program.

In this work, we studied the open-source version of three categories of applications and carefully identified the precise changes needed for each application. The changes are generic and we do not envision them to be limited to just these set of applications. We show the simplified version of the original code and the frame burst API proposed for video-playback, video-encoding and graphics in Figure 9,10, and 11. These APIs need number of frames to be issued, the memory addresses of input frames, and the size of each frame, all of which are a part of the original Android API. Apart from what the original API has, the modified version needs the frame burst size, and the chain identifier provided by the driver. This API calls the software drivers which in turn passes it to the CPU to initiate the frame bursts.

5.2. OS Driver and System Layer

Only minor modifications to the original media library are needed to support frame bursts. Currently, stock Android audio codecs *already* handle frame bursts for external devices like the audio and network devices. As audio plays at a very high frequency (at least 48000 Hz) and network device brings in bursts of packets, drivers of these devices tune the system to handle multiple frames/packets. Such a design needs to be extended for video encode/decode/graphics IPs.

5.3. Changes to IP scheduling

Conventionally, software drivers in the Android kernel provide the physical addresses where the data can be read from and written to, and the deadlines (QoS), based on application requirements. Drivers use the deadlines to pick which request to schedule next from among the set of requests in their queue. For VIP, since the CPU is out of the loop for many of the frames, the scheduling tasks have to be undertaken in *hardware.* Consequently, we have implemented a simple EDF (*earliest deadline first*) scheduler to pick the next task to run on an IP when it is supporting multiple flows. While EDF may not be suitable for ensuring fairness, it is still a reasonable option for our purposes given its simplicity in hardware implementation especially for the small number of channels that we support. The information on the deadlines and next IPs in the flow are passed on in the header packet that is sent to each IP (from the producer).

5.4. Header Packet

The header packet contains the context information to support the different flows and this information is passed on between the different IPs. The information is shown in Figure 12.

Packet Field	Field Size
IPs in flow <IP_1, IP_2,..IP_n>	32 bits, 4 bits/IP
Frame size in KB	16 bits
Frame rate (deadline)	4 bits
Frame Burst size	4 bits
Frame Src Mem address	32 bits
Frame Dest Mem address	32 bits
Frame context 1	1KB for IP_1
Frame context 2	1KB for IP_2
…	…
Frame context n	1KB for IP_n

Figure 12: Description of a header packet.

The size the header packet is variable as it depends on the number of IPs in the flow. It is important to note that, most of the information is related to frame context information that needs to be maintained for every IP. The frame context includes details like the pixel format, encoding/decoding formats, frame height and width. Given the fact that these details fit in a few 32-bit registers in current IPs, we do not expect them to be more that 1KB per IP. The longest app flow has about 4 IPs (Table 1) and therefore, we expect the header packet to be about 4KB. This is relatively small (since a header packet is incurred only once every frame burst) compared to the several MBs of data flowing through the SA (system-agent). When accounted, we found negligible impact due to header packets in our experiments.

5.5. Hardware Support for VIP

Virtual IP data flow can be integrated into existing IPs by replacing their single stack of input/output buffers with multi-lane buffers. If the IP needs to support more concurrent flows, then the number of lanes in the buffers needs to be increased as well. Each lane also has a corresponding set of registers

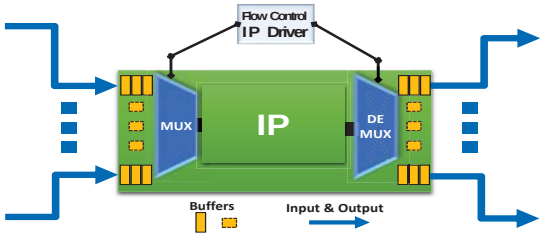
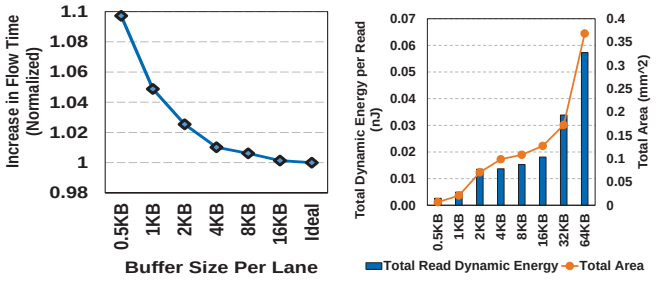


Figure 13: Overview of Virtualized IP architecture.

that holds the context of the requests (containing application and request specific details and QoS requirements). To enable switching across multiple lanes (maximum 4 in our case to support up to 4 applications), a multiplexer (and demultiplexer) is necessary before (and after) the scheduling logic, as shown in Figure 13. To support IPs operating at different rates, we integrate flow control mechanisms to allow the producer and consumer IPs to indicate the availability of buffers. These signals are low bandwidth (flag to indicate buffer full/not-full) and are communicated through the system-agent through which the flow data is also transferred. We wish to reiterate that the arrows indicating data flow between the IPs, as in Figure 8, is only to show the logical data flow, and *the physical realization of such flows is realized through the normal wires via the System Agent*, which is the centralized interconnect and controller on these handhelds. Also, note that, when IPs are not chained together, each IP can still operate in isolation (as before), with explicit programmability/interactions from the host driver.



(a) Variation in flow time with change in buffer size

(b) Energy, area consumed for different buffers sizes

Figure 14: Impact of buffer size at each IP on its energy consumption and area.

Buffer Sizing: IP-to-IP communication happens through the flow buffers at the producer and consumer, with a *sub-frame*. We divide a frame into smaller units, called sub-frames, similar to the flit concept in packet processing. Scheduling at the IPs happen at the subframe granularity. If the buffer slots in the producer's lane (corresponding to a flow) are full, the producer IP gets stalled until it gets enough buffer space to write its output. Therefore, if different IPs use different subframe sizes, then the minimum buffer capacity needs to be at least the size of the biggest subframe. However, not all IPs communicate with one another. For example, an audio-decoder will never forward its output to the camera IP and vice-versa. Hence, buffer size per lane can be limited to the maximum subframe size among the possible producers it has. From our experiments, for the IP configurations chosen, few tens of cache lines per lane was found to be sufficient for all IPs to avoid pipeline stalls significantly and not add much to the energy costs. As seen in Figure 14 (a), when the size of the buffer reduces, the end to end time for a frame increases due to stalls. Impact of buffer size on their energy consumption and area occupied (using CACTI[54]) is shown in Figure 14 (b). Based on this data, we choose 32 cache-lines (2KB) as the buffer size per IP in our experiments with support up to 2 lanes.

If there are fewer buffer lanes in the IPs than the number of flows that need to be simultaneously handled, then there are two possible techniques to handle this scenario: (i) either block the sender IP till the data can flow again, or (ii) forward the data to main memory so that the consumer can pick it up later. The latter method necessitates more complex communication protocols between IPs, with additional logic to figure out where the data resides. We, instead, use the former method, and simply stall the sender IP till buffer space is available and the consumer can take this data. In our implementation, we use a buffer size of 2 KB (32 cache-lines) per IP and sub-frame size of 1 KB.

6. Evaluation and Results

6.1. Applications and Workloads Used

Table 1 lists the frame-based applications studied in our evaluations. Since we focus on efficiently supporting multiple applications in our work, we consider practical combinations (commonly encountered scenarios) of the listed applications. These are listed in Table 2 along with the reasoning behind choosing such a combination. All the applications produce display frames, and involve multiple IPs to produce each frame (refer to [9] for IP-abbreviations).

App	App Name	IP Flows
A1	Game-1	GPU - DC; AD - SND
A2	AR-Game	GPU - DC; CPU - VE - NW; AD - SND; MIC - AE - NW
A3	Audio-Play	CPU - AD - SND; CPU - DC
A4	Skype	CPU - VD - DC; CAM - VE - NW; AD - SND; MIC - AE - NW
A5	Video Player	CPU - VD - DC; AD - SND
A6	Video Record	CAM - IMG - DC; CAM - VE - MMC; MIC - AE - MMC
A7	Youtube	CPU - VD - DC; AD - SND

Table 1: Applications and their IP flows.

Wkld	Application Combinations	Use-case
W1	2 Video-Play	Concurrent multiple Video Playback from disk
W2	1 HD-Video + 2-video Playback	Concurrent multiple Video Playback
W3	Video-Play + YouTube	Youtube video played with video on disk
W4	Skype + Video-Play	Watching video while teleconferencing
W5	Game-1 + Skype	Online multi-player gaming
W6	AR-Game + Audio-Play	Music playback from disk while gaming
W7	Video-Play + Video-record	Recording while playing another video
W8	Video-Play + AR Game	Multiplayer gaming with video-streaming

Table 2: Multiple Applications Workloads.

Evaluation Platform: For the initial part of the work, we used real systems including Nexus 7, Asus Memo Pad8, Sam-

Processor	ARM ISA; 4-core processor; In-order 1-issue
Caches	64KB cache line; 32 KB L1-I; 32KB L1-D; 512 KB L2
Memory	LPDDR3; 4 channel; 1 rank; 8 Banks Vdd = 1.2V; t_{CL},t_{RP},t_{RCD} = 12, 12, 12 ns
IP Parameters	Aud.Frame: 16KB frame; Vid.Frame: 4K (3840x2160) Camera Frame: 2560x1620 Required FPS: 60 (16.66ms)

Table 3: Platform details.

sung S4 and S5 to understand application and system behavior. In all the systems, as noted before, we use instrumented version of the the Grafika application suite released by Google for video playback, encoding, and recording. We use `ftrace`[42] to understand interrupt and kernel behavior running stock Android 4.4.2. Instrumented applications along with their ftraces were used in determining the time between successive user taps (or flicks) to figure out the optimal frame burst sizes for each application. Since implementing IP-to-IP communication and our proposed VIP scheme require hardware modifications, we implemented them on a simulation framework. Our evaluation framework builds on top of the GemDroid framework [9], which uses Android open-source emulator to capture complete system-level behavior. GemDroid performs trace based simulation of the full platform. Further details about the platform including each component's parameters is in Table 3.

6.2. Results

While motivating the need for virtualizing IP chains in Section 4, we showed in Figure 4 (d) that VIP should provide benefits in all three aspects – energy, performance and QoS. Below, we present these benefits obtained using VIP, which combines all the enhancements proposed in this paper. There are 4 possible systems that we compare VIP with : the *Baseline* system available today, *Frame Burst* (which just uses Frame Burst on top of the Baseline without IP-to-IP communication support), *IP-to-IP* (which has IP chaining but no Frame Burst Support), and *IP-to-IP with Frame Burst* (but no virtualization and hardware scheduling).

Energy Efficiency: First, we plot the energy benefits of the evaluated schemes in Figure 15 normalized with respect the baseline. The energy benefits are primarily due to 3 reasons – (i) reduced CPU energy, (ii) reduced data movement, and (ii) reduced IP stalls (causing IPs to complete their work faster). Of these, the last two effects are more visible in Figure 15 when IP-to-IP direct transfer is enabled (the last 3 bars for each workload).

To understand the impact of frame bursts, we separately study the energy consumption of the CPU in Figure 16. On an average, the CPU consumes ∼25% lower energy when frame

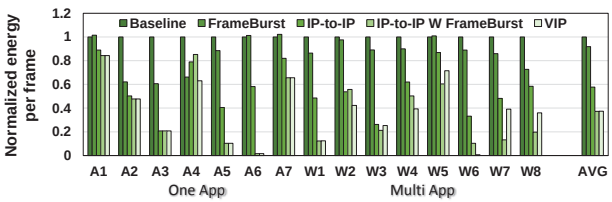
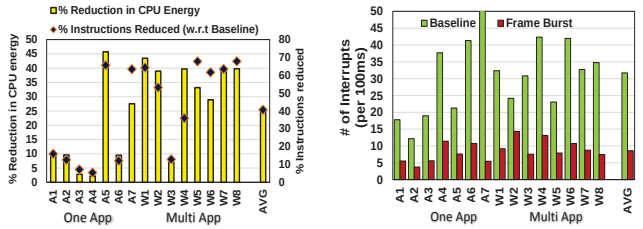


Figure 15: Energy Efficiency of VIP.



(a) Reduction in CPU energy and executed instructions

(b) Reduction in the number of interrupts processed by the CPU

Figure 16: Improved energy efficiency of CPUs handling interrupts and scheduling frames with Frame Burst.

bursts are enabled. This is due to the reduction in the number of instructions executed (shown in the figure). With CPU not needing to handle each frame, it executes fewer instructions as it does not need to save the driver/app context every time. Further, since it has longer gaps before processing (5 frames of gap), it goes to deeper sleep states. These CPU energy savings due to frame bursts, lead to ∼10% savings in the system level energy as shown in Figure 15.

The IP-to-IP communication has a substantial impact on energy savings as is clearly visible in the last three bars. This is because of the decreased number of memory bandwidth/accesses, and the lower stall time for the IPs (reducing their static power). With VIP, in some cases, as the IPs context switch across concurrent requests, IPs producing data for the next IP do not get blocked resulting in energy savings. Overall, with VIP, we achieve extra energy savings of ∼ 22% over a system with IP-to-IP communication.

Performance: Frame bursts with IP-to-IP communication provides two benefits. First, as we can see in Figure 16, the number of interrupts (shown per 100ms) reduces substantially, primarily because each IP directly communicates with the next IP instead of interrupting the core. Second, as IPs communicate directly with each other bypassing the memory, core utilization reduces as the average percentage of instructions reduces by 40% due to reduction in number of interrupts. Although not shown here due to space constraints, the proposed virtualization support reduces core and memory utilization, while improving IP utilization. As an overall performance metric, we plot the reduction in flow time per frame for the three different techniques in Figure 17. For example, let us consider the video player application. If memory is used to transfer data from one IP to the next, approximately 12-14

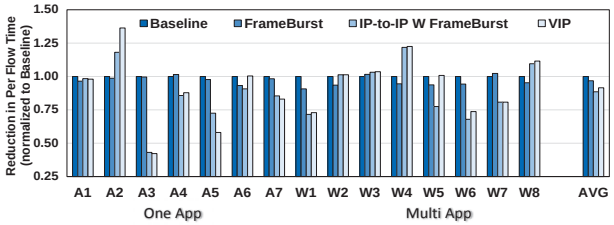


Figure 17: Normalized flow time per frame with VIP.

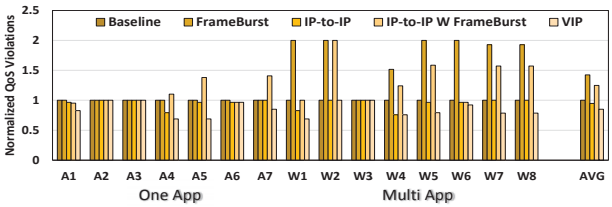


Figure 18: VIP enabling meeting QoS deadlines with IP-to-IP communication and Frame bursts.

MB of data needs to be read+written to DRAM per 1080p frame. This translates to 700MB-800MB data movement per second. For a 4K frame, this is 3-4 GBPS. With IP-to-IP communication, we avoid this memory traffic, thereby reducing IP stalls, reducing memory utilization and flow processing time. With Frame Bursts, we observe an improvement of about 20% in frame processing time. With VIP, when there are multiple requests contending for a resource, due to context switching overheads, and because the locality at memory gets disturbed, we see a slight loss in performance compared to the burst mode. However, as we will see next, the burst mode faces significant QoS violations compared to VIP in the multiple application scenario, thereby demonstrating the overall advantages of VIP.

Meeting Quality-Of-Service Deadlines: Figure 18 depicts the QoS benefits of the proposed scheme in terms of normalized frame drop rate with respect to the base line design. Though we see performance improvements and energy reduction in Frame Bursts and IP-to-IP with Frame Bursts, they cause serious degradation in QoS for all eight workloads. This is because, when one application progresses, the other application is blocked, resulting in increase in overall frame drop rate. VIP mitigates this problem by enabling time division multiplexing of the IP between both applications. With EDF scheduling implemented in the IP's hardware, requests closer to their deadline get prioritized leading to fewer frame drops. Overall, while IP-to-IP communication reduces frame drops by 5%, with frame bursts and EDF in place, VIP helps reduce $\sim$15% frame drops compared to the baseline.

7. Related Work

Recently there has been a growing interest in exploring different avenues to enhance the performance and energy efficiency of handheld devices. This primarily includes performance and energy centric optimizations [59, 39, 4, 8, 11, 17, 19, 18, 1, 55], memory centric optimization [38, 2, 41, 34, 56, 40], IP related optimizations [21, 30, 36, 48, 50, 31, 22, 53, 27] and platform development for analyzing these systems [28, 47, 9, 32]. However, to the best of our knowledge, this is probably the first effort to bring IP scheduling from software to hardware and combining it with memory bypassing for optimizing energy, performance and QoS in handhelds.

Memory Specific Optimization in SoCs: Recently, few works have proposed data communication optimizations for inter-accelerator communication [56, 43, 51, 33]. Particularly, [56] analyzes the impact of performance enhancement techniques by having a shared buffer between two IPs. These solutions have been proposed for single application execution, while VIP is targeted for multiple application execution.

Application Specific Optimization Many works [60, 59, 58, 57, 6] have proposed mobile web browser optimizations to improve browsing performance and energy efficiency. For frame-base application optimization [10, 45, 1] focused on power management for video decoder to achieve higher energy efficiency.

Virtualization Virtualization has a long history along with the development of computation devices[49], and has been widely applied on CPU, network and accelerators. In the accelerator domain, [52, 46, 44, 35] propose ways to share GPU resources concurrently. In networking domain, while there have been multiple works, most of them are based on virtualizing network resources to improve throughput (using virtual channels[16, 15]). Unlike the works above, our work focus on virtualizing IP cores on handhelds to support running multiple applications.

8. Conclusion

Current IP interfaces are grossly insufficient for the emerging class of frame based applications that stream frames of data through several IP cores. The problem gets exacerbated with multiple such applications running on a platform. The main CPU cores that are continuously engaged in the processing of each frame, and the memory system serving as the conduit for data flow, are some of the points of contention and inefficiencies on such systems. Despite running multiple applications, many of which may use the same IP cores, the utilizations of these accelerators is not very high, suggesting that throwing more hardware to serve these multiple applications is not necessarily the best option. Instead, this paper proposes a new paradigm for creating Virtual IP chains (VIP) to address the throughput and energy inefficiencies of current designs. VIP employs three complementary innovations to achieve this goal. First, we enable multiple applications to use IP-to-IP chaining for direct data transfer avoiding the memory system overhead. Second, we propose a burst-mode transfer, where a CPU can initiate the processing of several frames through the virtual IP chain without being involved/interrupted for each frame. Third, akin to the virtual channel flow control in the networking domain, we provide a virtual path for each flow to enforce proportional sharing of IPs for satisfying the QoS guarantees. Our experimental evaluations with two-application workloads provides 22% energy saving, 10% improvement in frame processing time and 10% improvement in frame drop rate compared to just enabling IP-to-IP communication.

9. Acknowledgments

This research is supported in part by NSF grants #1205618, #1213052, #1302225, #1302557, #1317560, #1320478, #1409095, #1439021, #1439057 and Intel.

References

[1] "Dynamic voltage scaling techniques for power efficient video decoding," *Journal of Systems Architecture*, vol. 51, no. 10–11, 2005.

[2] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable sdram memory controller," in *CODES+ISSS*, 2007.

[3] Apple. (2014) Final Cut Pro 7 User Manual. Available: https://documentation.apple.com/en/finalcutpro/usermanual/index.html#chapter=C%26section=12%26tasks=true

[4] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: A measurement study and implications for network applications," in *IMC*, 2009.

[5] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: Design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, ser. CODES '02, 2002.

[6] V. Bhatt, N. Goulding-Hotta, J. S. Q. Zheng, S. Swanson, and M. B. Taylor, "Sichrome: Mobile web browsing in hardware to save energy," in *DaSi: First Dark Silicon Workshop*.

[7] D. Bosomworth. Mobile Marketing Statistics 2014. Available: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics

[8] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *USENIXATC*, 2010.

[9] N. Chidambaram Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. Kandemir, and C. R. Das, "GemDroid: A Framework to Evaluate Mobile Platforms," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2014.

[10] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram, "Frame-based dynamic voltage and frequency scaling for a mpeg decoder," in *Proceedings of the International Conference on Computer-aided Design*, ser. ICCAD, 2002.

[11] Y.-F. Chung, C.-Y. Lin, and C.-T. King, "Aneprof: Energy profiling for android java virtual machine and applications," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, Dec 2011.

[12] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, "Accelerator-rich architectures: Opportunities and progresses," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14, 2014.

[13] cubei. (2014) cubei/FlappyCow. Available: https://github.com/cubei/FlappyCow

[14] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[15] W. Dally, "Virtual-channel flow control," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 3, no. 2, Mar 1992.

[16] W. Dally and C. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *Computers, IEEE Transactions on*, vol. C-36, no. 5, May 1987.

[17] Q. Deng, D. Meisner, A. Bhattacharjee, T. Wenisch, and R. Bianchini, "Coscale: Coordinating cpu and memory system dvfs in server systems," in *International Symposium on Microarchitecture (MICRO)*, Dec 2012.

[18] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "Multiscale: Memory system dvfs with multiple memory controllers," in *Proceedings of the International Symposium on Low Power Electronics and Design*, ser. ISLPED '12, 2012.

[19] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, "Memscale: Active low-power modes for main memory," in *ASPLOS*, 2011.

[20] emmaguy. (2014) emmaguy/FruitNinja. Available: https://github.com/emmaguy/FruitNinja

[21] S. Fenney, "Texture compression using low-frequency signal modulation," in *HWWS*, 2003.

[22] C. Gao, A. Gutierrez, R. Dreslinski, T. Mudge, K. Flautner, and G. Blake, "A study of thread level parallelism on mobile devices," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, March 2014.

[23] Google. Google/Grafika. Available: https://github.com/google/grafika

[24] Google. OpenGL ES - Android Developers. Available: http://developer.android.com/guide/topics/graphics/opengl.html

[25] Google. RenderScript - Android Developers. Available: http://developer.android.com/guide/topics/renderscript/compute.html

[26] Google. (2014) Android Developers. Available: http://developer.android.com/

[27] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," *Micro, IEEE*, vol. 31, no. 2, March 2011.

[28] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *IISWC*, 2011.

[29] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the Annual International Symposium on Computer Architecture*, ser. ISCA, 2010.

[30] K. Han, A. Min, N. Jeganathan, and P. Diefenbaugh, "A hybrid display frame buffer architecture for energy efficient display subsystems," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013.

[31] J. Hauswald, T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge, "A hybrid approach to offloading mobile image classification," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2014.

[32] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A mobile benchmark suite for architectural simulators," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014.

[33] K. Irick and N. Chandramoorthy, "Achieving high-performance video analytics with lightweight cores and a sea of hardware accelerators," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2014.

[34] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *DAC*, 2012.

[35] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs," in *PACT*, 2013.

[36] H. b. T. Khan and M. K. Anwar, "Quality-aware Frame Skipping for MPEG-2 Video Based on Inter-frame Similarity," Malardalen University, Tech. Rep.

[37] R. J. Kosinski. A Literature Review on Reaction Time. Available: http://biae.clemson.edu/bpc/bp/lab/110/reaction.htm

[38] K.-B. Lee and T.-S. Chang, *Essential Issues in SoC Design Designing - Complex Systems-on-Chip*. Springer, 2006, ch. SoC Memory System Design.

[39] K.-B. Lee, T.-C. Lin, and C.-W. Jen, "An efficient quality-aware memory controller for multimedia platform soc," *Circuits and Systems for Video Technology, IEEE Transactions on*, 2005.

[40] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing soc accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013.

[41] Y.-J. Lin, C.-L. Yang, T.-J. Lin, J.-W. Huang, and N. Chang, "Hierarchical memory scheduling for multimedia mpsocs," in *ICCAD*, 2010.

[42] Linux. ftrace - Function Tracer. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[43] M. Loghi and M. Poncino, "Exploring energy/performance tradeoffs in shared memory mpsocs: snoop-based cache coherence vs. software solutions," in *Design, Automation and Test in Europe*, 2005.

[44] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014.

[45] S. Mohapatra, R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian, "Integrated power management for video streaming to mobile handheld devices," in *Proceedings of the Eleventh ACM International Conference on Multimedia*, ser. MULTIMEDIA '03, 2003.

[46] Nvidia. Shared Virtual GPU (vGPU) Technology | NVIDIA. Available: {http://www.nvidia.com/object/virtual-gpus.html}

[47] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite : Mobilebench," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[48] K. Patel, E. Macii, and M. Poncino, "Frame buffer energy optimization by pixel prediction," in *ICCD*, 2005.

[49] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, Jul. 1974.

[50] H. Shim, N. Chang, and M. Pedram, "A compressed frame buffer to reduce display power consumption in mobile systems," in *ASP-DAC*, 2004.
[51] P. Sungho, A. Ahmed, M. Kevin, C. Aarti, C. Matthew, C. Nandhini, D. Michael, and N. Vijaykrishnan, "System-on-chip for biologically inspired vision applications," *IPSJ transactions on system LSI design methodology*, 2012.
[52] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *International Symposium on Computer Architecture (ISCA)*, June 2014.
[53] Y. Wang, B. Krishnamachari, Q. Zhao, and M. Annavaram, "Markov-optimal sensing policy for user state estimation in mobile devices," in *Proceedings of the International Conference on Information Processing in Sensor Networks*. ACM, 2010.
[54] S. Wilton and N. Jouppi, "CACTI: An Enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, May 1996.
[55] Y. Xiao, R. S. Kalyanaraman, and A. Yla-Jaaski, "Energy consumption of mobile youtube: Quantitative measurement and analysis," in *NGMAST*, 2008.
[56] P. Yedlapalli, N. C. Nachiappan, N. Soundararajan, M. Kandemir, A. Sivasubramaniam, and C. R. Das, "Short-Circuiting Memory Traffic in Handheld Platforms," *MICRO*, 2014.
[57] V. J. R. "Yuhao Zhu, Matthew Halpern, *"Event-based Scheduling for Energy-Efficient QoS (eQoS) in Interactive Mobile Web Applications"*, 2015.
[58] Y. Zhu, A. Srikanth, J. Leng, and V. Reddi, "Exploiting webpage characteristics for energy-efficient mobile web browsing," *Computer Architecture Letters*, vol. 13, no. 1, Jan 2014.
[59] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA '13, 2013.
[60] Y. Zhu and V. J. Reddi, "Webcore: Architectural support for mobileweb browsing," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, 2014.