The Pennsylvania State University
The Graduate School

PUSHING THE POWER AND PERFORMANCE ENVELOPE OF NEXT-GENERATION HANDHELD PLATFORMS

A Dissertation in
Computer Science and Engineering
by
Nachiappan Chidambaram Nachiappan

© 2015 Nachiappan Chidambaram Nachiappan

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2015

The dissertation of Nachiappan Chidambaram Nachiappan was reviewed and approved* by the following:

Mahmut T. Kandemir
Professor of Computer Science and Engineering
Dissertation Co-Adviser, Committee Co-Chair

Chita R. Das
Distinguished Professor of Computer Science and Engineering
Dissertation Co-Adviser, Committee Co-Chair

Anand Sivasubramaniam
Professor of Computer Science and Engineering

W. Kenneth Jenkins
Professor of Electrical Engineering Department

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Today's handhelds have grown in sophistication to run demanding applications. With the number of wearables and IoTs expanding, the tablets and smartphones are proposed to be used as their compute hubs. This places high compute demand and significant energy drain on these handhelds. In such a landscape, the consumers expectations are *antithetical* – needing the highest performance delivered all the time along with a long battery runtime all through a modest Li-ion battery on device!

To provide higher performance and lower energy consumption, vendors have resorted to the use of hardware accelerators. In current generation handhelds, many applications (especially multimedia apps) rely heavily on *multiple accelerators*. In spite of these efforts from the vendors, full fledged support for multiple concurrent applications have been futile as they are unable to meet the consumer's power-performance expectations. The **main motivation** of this dissertation is to propose techniques to **push the performance boundaries** by alleviating the bottlenecks and to **efficiently make use of the power envelope** when multiple accelerators are involved. It consists of four main components.

The first part of the dissertation presents *GemDroid*, a comprehensive simulation infrastructure to study SoC architectures. Currently, this is one of the first publicly available tool to conduct a holistic evaluation of mobile platforms consisting of cores, IPs and system software.

As the second part, the dissertation analyzes a spectrum of applications with GemDroid, and observes that the memory subsystem is a vital cog in the mobile platform because, it needs to handle both core and IP traffic, which have very different characteristics. Consequently, a *heterogeneous memory controller* (HMC) design is presented, where the memory is physically divided into two address regions, where the first region with one memory controller (MC) handles core-specific application data and the second region with another MC handles all IP related data.

In the third part, the dissertation focuses on improving system throughput by short-circuiting the memory traffic and enabling multiple-applications to run concurrently by virtualizing the data paths. Through measurements on a current generation tablet, it shows that the frequent invocation of the CPU for processing applications frames and the involvement of main memory as a data flow conduit, are serious limitations. Instead, the dissertation proposes 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. Firstly, chaining of IPs avoids data transfer through the memory system, enhancing the throughput of flows through the IPs. Secondly, 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. Third, the dissertation also makes a case for supporting multiple applications through the creation on several virtual paths – one for each flow, and hardware scheduling is used to enforce QoS guarantees despite any contention for resources along the way.

As the final part, the dissertation strives to address the most critical and a daunting task - *efficient* energy management in handheld platforms. With the growing number of accelerators, memory demands are increasing and high computing capacities are required to support applications with stringent QoS needs. Current DVFS techniques that modulate power states of a single hardware component, or even recent proposals that manage multiple components, can lose out opportunities for attaining high energy efficiencies that may be possible by leveraging application domain knowledge. Thus, this dissertation proposes a *coordinated multi-component energy optimization* mechanism for handheld devices, where the energy profile of different components such as CPU, memory, GPU and IP cores are considered in unison to trigger the appropriate DVFS state by exploiting the application domain knowledge. Specifically, it shows that for the important class of frame-based applications, the domain knowledge - frame processing rates, component utilization and available slack - can be used to decide effective DVFS states for each component from among the numerous choices.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am deeply thankful to all the people who have provided intellectual contributions and motivational support to this dissertation. I have no words to describe all their help and support. Hence, a simple acknowledgement – but a sincere thanks from the *bottom of my heart*.

> "*The learned teacher makes you enjoy learning;*
> *On leaving, makes you keep thinking of his teachings.*"
>
> —Thirukkural – 394

Each and everyone acknowledged below has taught me something meaningful.

Advisors, Co-advisors and Teachers
Chita R. Das
Mahmut Kandemir
Anand Sivasubramaniam
Vijaykrishnan Narayanan
Venkateswaran Nagarajan
Kamakoti Veezhinathan

Seniors and Collaborators
Niranjan Soundararajan
Asit K. Mishra
Emre Kultursay
Sai Prashanth Muralidhara
Praveen Yedlapalli
Hayawardh Vijayakumar
Bikash Sharma
Adwait Jog
Onur Kayiran

Jayaram Bobba
Soumya Eachempati
Nandhini Chandramoorthy
Ravindhiran Mukundarajan
Myoungsoo Jung
Karthik Swaminathan

Lab Mates
Prashanth Thinakaran
Jashwant Raj Gunasekaran
Ashutosh Pattnaik
Haibo Zhang
Tulika Parija
Jihyun Ryoo
Jagadish Kotra
Mahshid Sedhgi
Amin Jadidi
Anup Sarma
Xulong Tang

Family and Friends
Nachiappan. C (father)
Karpagambal. RM (mother)
Kiruthika. M (wife)
Sivagami. A. N. (sister)
Grandparents
Cousins
Vivek Narayanan
Prem Anand
Rathan Vignesh
Aswin Sridharan
Bharath Ramasamy
Bharat Rengarajan
Amey Farde
Akshay Virdhe
Deepika Vaidyanathan
Priyanka Gomatam
Sharath Reddy

.. and some more special people (who I have missed to mention here, but) who have made my PhD a happy journey!

Chapter 1

Introduction

There is an exploding demand for mobile systems, which include smartphones, tablets, and wearable devices. Gartner research projects that 2 billion of these units will be sold in 2013 [1] and there will be over 10 billion mobile devices by the end of 2017 [2]. Moreover, it is projected that global mobile data will increase 13-fold between 2012 and 2017 reaching 11 Exabytes per month and two-thirds of this data is projected to be video data [2]. These numbers clearly indicate the importance of designing feature-rich mobile devices to cope up with the market demand. The ITRS roadmap for designing System-on-Chip architectures (SoCs) over the next decade projects that a mobile device could have up to 50 processing cores with about 300 TFLOPS computing capability and more than 400 IP blocks for enabling such feature-rich mobile platforms [3]. Thus, major companies like AMD, ARM, Apple, Intel, NVidia, Qualcomm, and Samsung have already ventured into this growing market targeting devices ranging from wearable wrist watches, glasses, to hand-held smartphones, phablets and tablets. Design and analysis of these devices with required QoS provisioning, power budgets and evolving technology artifacts is a daunting task that computer architects have to deal with in the coming years.

Handheld systems are being transformed from simple, single-core architectures to

complex multicore platforms with heterogeneous cores (with different power/performance characteristics), many different types of IPs, and multiple memory controllers. Currently, most web-pages and applications are customized for smartphones to make them "lighter" in terms of performance demanded and power consumed. Even with such use-cases, current generation smartphones typically last a day at the maximum under normal usage scenarios when fitted with a 2000 mAh battery. Furthermore, on the application side, one can observe a trend towards running multiple, independent applications at the same time on the same device, putting tremendous pressure on all on-chip resources. Clearly, designing a handheld system that can meet all this pressure – demand for *high-performance* capable of supporting desktop applications, *minimal energy consumption* for doing the same task, and conserving energy to *last few days on a single charge*, is not a trivial task.

The Problem:

(I) From Evaluation Perspective

To evaluate existing handheld platforms, there are no tools that can simulate the software and hardware parts to characterize the performance and energy impacts of various applications. Surprisingly, despite the significant amount of research conducted over the last decade targeting handhelds, we are still missing a comprehensive and extensible simulation infrastructure in the public domain. Unfortunately, current tools and simulation/emulation platforms do not enable full platform simulation of handheld systems. More specifically, multicore/manycore simulators such as GEMS [27] and Gem5 [20] model a uniform parallel system with cache hierarchies. There is no concept of IP blocks or heterogeneity in those simulators, which is a must for modeling mobile platforms. On the other hand, existing IP simulators/emulators generally operate in isolation (i.e., model only the target IP), whereas in a handheld platform both IPs (which can be of different types) and parallel cores should be accurately modeled and simulated.

(II) From Performance Perspective

The combined requests from multiple cores and IPs with different characteristics share the memory subsystem. This makes memory one of the major and a critical bottleneck for handheld platforms. The memory access patterns of IPs exhibit high levels of regularity (e.g., sequential data accesses by frame-buffers), as opposed to the memory access patterns of, say, well-known SPEC benchmarks [4]. Similarly, the memory bandwidth demanded by cores and IP when running a video on YouTube are very different from each other. Specifically, while the core's bandwidth demand is more or less constant (requiring $<$ 0.2GBPS for the studied workloads), display IP's needs are very bursty in nature (needing around 0.8GBPS), and the total bandwidth demand for a video recorder can be much higher than that of the memory system. Moreover, these requests not only differ in terms on bandwidth demands, but also their latency demands are also very different from each other. While IPs have strict latency deadlines that *need* to be met, cores do not have deadlines. On the other hand, IPs have time window before which they can be served without affecting the user-experience, but, each cores memory request directly affects the performance of the system.

(III) From System Design Perspective

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. First, many of today's applications use more than one IP, where one feeds data to another. For this data movement, they use memory as a conduit, as well as, needs the host CPU to orchestrate the movement. Second, when multiple applications run each depending on multiple IPs, including possibly some common IPs, it introduces several points of contention – the shared IP(s), the data flow paths, and the memory. 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.

(IV) From Energy Perspective

Despite considerable research in the past two decades leading to multi-fold improvements in energy efficiencies, the problem has become all the more interesting and challenging in current and future handhelds for the following two reasons: (1) Users are increasingly running sophisticated applications that require considerable computing, memory, storage and networking capabilities, all under the constraints of real-time interactivity/responsiveness, e.g., YouTube, video chat, interactive games, etc., and (2) To accommodate these needs, handhelds are starting to look like servers, with multiple components – multiple cores, large memories and storage, specialized accelerators and GPUs – which can all dissipate high power.

Energy efficient operation of handhelds has to thus (a) ensure that any power state setting of components does not lead to violations in application real-time requirements (for, say, processing a frame), (b) take into account that several components are involved in processing such a frame and there could be cascading effects as a result of slowing down a component when saving power, and (c) recognize that it may not suffice to only target the main CPU for power savings since other components could also contribute significantly. Without a holistic/coordinated power management strategy, spanning different components, which understands the needs of these frame-based applications, we may not be able to make the right trade-offs between energy savings and application performance needs (SLAs for frames).

In this dissertation, we target to address the above 4 problems of

- incapability of evaluating current handheld platforms for ineffciencies,
- performance ineffeciency caused due to memory bottleneck,
- system-wide power and performance inefficency observed while running multiple applications.
- power inefficency caused due to unoptimal frequency setting of multiple components

co-running at an instant,

The interactivity in many of the Android applications involves a steady stream of data that needs to be periodically processed energy-efficiently within a time window. We broadly refer to this category of important applications as "frame-based applications" in this dissertation. These "frame-based applications" are the target workloads through-out this dissertation.

Thesis: This dissertation adopts a four pronged approach in addressing the above mentioned inefficiencies.

(I) The first dimension of this dissertation presents an infrastructure built to simulate and evaluate handheld platforms. We propose a comprehensive simulation infrastructure, called **GemDroid**, which incorporates the GEM5 architecture simulator [5], Attila graphics simulator [6] and internal models for the other IPs, and, is capable of running the Android mobile OS for facilitating mobile platform design and optimization research. GemDroid is comprised of two primary layers. The first layer provides emulation of Android OS by the Google Android Emulator [7] and allows us to capture system-level interaction between multiple IPs and I/O devices, including OS activities. What the emulator cannot provide is the timing information of different IP activities and therefore, as the second layer, the timing piece is integrated/built using the existing simulation platforms or model them analytically as needed for different IPs. The framework is flexible for integrating models of varying complexities for the cores and IPs.

GemDroid

A small step for simulation and evaluation platforms,

enabling ideas for giant leaps in design of handheld platforms

(II) The second dimension of this dissertation presents a novel heterogeneous memory controller (HMC) design for SoCs, where one MC is dedicated for latency critical core requests and the second MC is optimized to enhance the bank-level parallelism of the

memory requests it serves. The two memory controllers are still responsible for two distinct (non-intersecting) address ranges. Evaluation of this new MC design shows that it is better than state-of-the-art in terms performance *and* user experience.

(III) The third dimension of this dissertation proposes two important software-hardware codesign ideas - first, to short-circuit memory traffic by allowing IPs to communicate among themselves directly without using memory as the conduit ans second, to virtualize the flow of data across multiple IPs for co-existing applications that are involved in periodically processing frames. In these approaches, rather than treat each IP as a separate piece of hardware, we dynamically create/initiate a hardware chain of IP cores that can be treated as a single device in software by the host. We instantiate a IP hardware chain for each application's data flow through its sequence of IPs. In the second part of this dimension, we virtualize these hardware chains thereby creating a virtual channel of data flow across IPs for each application's data flow. We enable such an approach through minimal hardware and software stack modifications. The combination of these approaches can help boost the utilization of existing IPs, while still allowing individual applications to meet their required frame/flow rates. Such a solution can even boost the utilization of the host cores for other useful work by relieving them from the mundane role of data transfer conduits between the IPs. It can achieve this goal without requiring additional IPs to be introduced into the already power and real-estate constrained handheld device.

(IV) The fourth dimension of this dissertation identifies that even if we have complete application level (Oracle-based) slack/tolerance information, a strategy that can only control the power states of the main CPU cores has the potential to reduce overall system energy by only 8% to adhere to the frame processing guarantees. From a more practical perspective, today's common power governors (like `OnDemand` in Linux/Android), which have neither such Oracle knowledge nor do they manage power states beyond the main CPU cores and memory bus, fall short of even this target. Thus, a multi-component power mode control mechanism can further improve the energy savings. For instance, one

could choose the most energy-efficient DVFS setting for each system component. But such a mechanism, though more energy efficient than `OnDemand`, will loose substantially in terms of performance as it is not aware of frame deadlines. These motivate the need for recognizing performance slack in the applications when performing such DVFS for the different components. Towards that, this dissertation proposes a *frame-aware* coordinated multi-component DVFS framework that leverage frame-level slack and utilization information of components predicted at frame boundaries. In this context, we propose two mechanisms, called *Greedy policy* and *Kaldor-Hicks* [8] *compensation policy* (K&H). The *Greedy policy* in turn examines three different options: Maximum Energy First (MEF), Most to Gain First (MGF) and Slope ($\triangle$E/$\triangle$T) First (SDF). The K&H policy can even choose energy inefficient options for one component, while offsetting this with higher energy savings in another.

Chapter 2

Background and Related Work

In this section, we first provide a brief overview of current SoC (system-on-chip) platforms showing how the OS, core and the IPs interact from a software and hardware perspective. Next, we describe the properties of the applications that are used in this work. We also summarize prior work related to this dissertation.

2.1 Background

2.1.1 Overview of SoC Platforms

As shown in Figure 2.1, handhelds available in the market have multiple cores and other specialized IPs. The IPs in these platforms can be broadly classified into two categories – *accelerators* and *devices*. Devices interact directly with the user or external world and include cameras, touch screen, speaker and wireless. Accelerators are the on-chip hardware components which specialize in certain activities. They are the workhorses of the SoC as they provide maximum performance and power efficiency, e.g. video encoders/decoders, graphics, imaging and audio engines. In other words, devices are usually uni-directional from a memory's perspective, i.e., they either write data to memory (example, touch-screen,

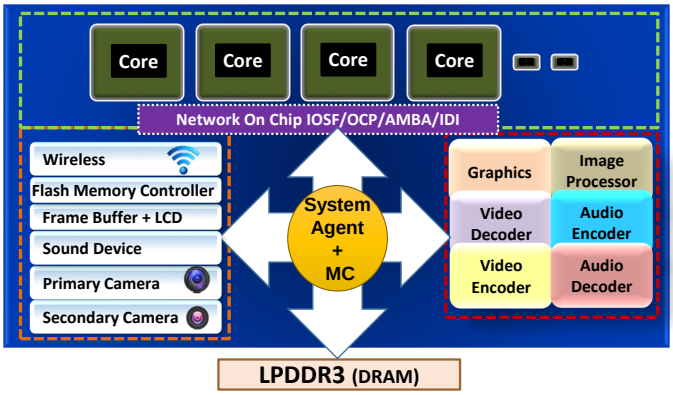


Figure 2.1: Target SoC platform with a high-level view of different functional blocks in the system.

mic), or read data from memory for a specific task (eg., display, speaker, etc,,). Accelerators are two-way, that is, they read some data from memory, process it and write the output back to memory.

Interactions between Core, IPs and Operating System: SoC applications are highly interactive and involve multiple accelerators and devices to enhance user experience. Using API calls, application requirements get transformed to accelerator requirements through different layers of the OS. Typically, the calls happen through software device drivers in the kernel portion of the OS. These calls decide if, when and for how long the different accelerators get used. A certain level of transparency is needed for security, fairness of resource usage and portability of application code. Usually, applications do not directly communicate with the IPs. They are accessed through the software drivers present as a part of the operating system for the sake of providing security and fairness to all applications. Operating systems such as Android, Windows and iOS, include various drivers that interact closely with the accelerators. The device drivers, which are optimized by the IP vendors, control the functionality and the power states of the accelerators. Once an accelerator needs to be invoked, its device driver is notified with request and associated physical address of input data. The device driver sets up the different activities that the accelerator needs to do, including writing appropriate registers

with pointers to the memory region where the data should be fetched and written back. These drivers not only act as links between applications running on the OS and underlying hardware, but also control power states of IPs. The accelerator reads the data from main memory through DMA. Input data fetching and processing are pipelined and the fetching granularity depends on how the local buffer is designed. The input data is buffered till the accelerator starts processing. Once data is processed, it is written back to the local buffers and eventually to the main memory at the address region specified by the driver. As most accelerators work faster than main memory, there is a need for input and output buffers.

The System Agent (SA): Also known as the Northbridge, is a controller that receives commands from the core and passes them on to the IPs. Their design varies with different architectures. Some designs add more intelligence to the SA to prioritize and reorder requests to meet QoS deadlines and to improve DRAM hits. SA usually incorporates the memory controller (MC) as well. Apart from re-ordering requests across components to meet QoS guarantees, even fine-grained re-ordering among IP's requests can be done to maximize DRAM bandwidth and bus-utilization. With increasing user demands from handhelds the number of accelerators and their speeds keep increasing [9–11]. We envision that the number of accelerators and their processing speeds on handhelds will keep increasing as user demand increases [9–11]. These trends will place a very high demand on DRAM traffic. Consequently, unless we design a sophisticated SA that can handle the increased amount of traffic, the improvement in accelerators' performance will not end in improved user experience. As systems become more complex in future, the need for a smarter SA is only becoming more pronounced. A smart SA that is not only aware of platform power and performance requirements, but also intelligently decides if a piece of code has to be run on an accelerator or core along with multiple accelerators.

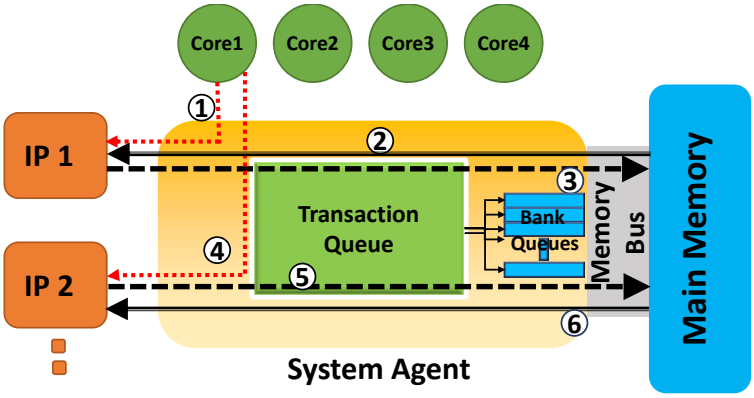


Figure 2.2: Overview of data flow in SoC architectures.

2.1.2 Data movement in SoCs

Figure 2.2 depicts the high-level view of the data flow in SoC architectures. Once a core issues a request to an IP through he SA (shown as (1), the IP starts its work by injecting a memory request into SA. First, the request traverses through an interconnect which is typically a bus or cross-bar, and is enqueued in a memory transaction queue. Here, requests can be reordered by the SA according to individual IP priorities to help requests meet their deadlines. Subsequently, requests are placed in the bank-queues of the memory controller, where requests from IPs are re-arranged to maximize the bus utilization (and in turn, the DRAM bandwidth). Following that, an off-chip DRAM access is made. The response is returned to the IP through the response network in the SA (shown as (2)). IP-1 writes its output data to memory (shown in (3)) till it completes processing the whole frame. After IP-1 completes processing, IP-2 is invoked by the core (shown as 4), and data flow similar to what IP-1 had is followed, as captured by (5) and (6) in Figure 2.2. The unit of data processing in media and gaming IPs (including audio, video and graphics) is a **frame**, which carries information about the image or pixels or audio delivered to the user. *Typically a high frame drop rate corresponds to a deterioration in user-experience.*

One main observation from this figure is that, The total request access latency includes the network latency, the queuing latencies at the transaction queue and bank queue, DRAM

service time, and the response network latency. This latency, as expected, is not constant and varies based on the system dynamics (including DRAM row-buffer hits/misses). When running a particular application, the OS maps the data frames of different IPs to different physical memory regions. Note that these regions get reused during the application run for writing different frames (over time). In a data flow involving multiple IPs that process the frames, one after another, the OS (through device drivers) synchronizes the IPs such that the producer IP writes another frame of data onto the same memory region after the consumer IP had consumed the earlier frame.

Secondly, the total end-to-end latency depends on requests from other accelerators, devices and cores, with each request having it's own priorities and deadlines to meet. Higher the priority of a concurrent request, greater will be the latency suffered. Third, if requests are contending for the same DRAM bank, bank-level parallelism in DRAM will be affected and will result in additional delays. Finally, the total latency of a memory request will depend on whether the access was a row buffer hit or not.

This synchronization between producers and consumers is taken care by the OS. In the figure, even though it seems like an IP is reading or writing to a same location repeatedly, they are to *different* frames of data. The reuse mentioned here are not to be confused with data reuse or locality, as across frames each cache line has new data and is considered "dirty" from a memory perspective.

2.1.3 Decomposing an Application Execution into Flows

Applications cannot directly access the hardware for I/O or acceleration purposes. In Android, for example, application requests get translated by intermediate libraries and device drivers into commands that drive the accelerators and devices. While an application can use multiple devices and accelerators, usually the sequence in which these IPs are called are quite regular and there is significant amount of overlap in the data that one IP writes

and an another IP reads, that is, *there is significant data reuse across different IPs*. This translation results in hardware processing the data, moving it between multiple IPs and finally writing to storage or displaying or sending it over the network. Let us consider for example a video player application. The flash controller reads a chunk of video file from memory, gets processed by the core, and two separate requests are sent to video-decoder and audio-decoder. They read their data from the memory and, once an audio/video frame is decoded, it is sent to the display through memory. In this thesis, we term such a regular stream of data movement from one IP to another as a **flow**. All our target applications have such flows, as shown in Table 2.2. Table 2.1 gives the expansions for IP abbreviations. We classify two IPs as a part of a flow *only when* there is some data being shared between them. It is to be noted that an application can have one or more flows. In multiple flows, each flow could be invoked at a different point in time, or multiple independent flows can be active concurrently *without* sharing any common IP or data across them.

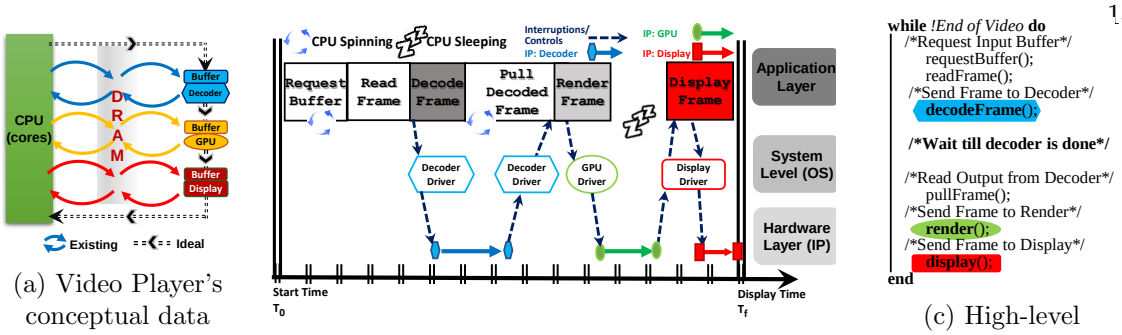
IP Abbr.	**Expansion**	**IP Abbr.**	**Expansion**
VD	Video Decoder	AD	Audio Decoder
DC	Display Controller	VE	Video Encoder
MMC	Flash Controller	MIC	Microphone
AE	Audio Encoder	CAM	Camera
IMG	Imaging	SND	Sound

Table 2.1: Expansions for IP abbreviations used in this thesis.

Id	**Application**	**IP Flows**
A1	Angry Birds	AD - SND; GPU - DC
A2	Sound Record	MIC - AE - MMC
A3	Audio Playback (MP3)	MMC - AD - SND
A4	Photos Gallery	MMC - IMG - DC
A5	Photo Capture (Cam Pic)	CAM - IMG - DC; CAM - IMG - MMC
A6	Skype	CAM - VE; VD - DC; AD - SND; MIC - AE
A7	Video Record	CAM - VE - MMC; MIC - AE - MMC
A8	Youtube	VD - DC; AD - SND

Table 2.2: IP flows in the applications.

Single Application Scenario. As described above, most mobile apps follow a sequence of steps through multiple IPs to display frames of content onscreen. Fortunately, Android [12] supports different SDKs (OpenGL ES [13], RenderScript [14]) 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



(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 2.3: Video playback flow

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 in detail, we study a typical video-player application (Grafika [15], available as open-source) where we show when the APIs are invoked, and how the drivers are called(shown in Figure 2.3.

As shown, CPU does some simple computation, and using API calls provided by the IP drivers, the application invokes the IP, 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. These details are passed on to the driver of the first IP that is to be invoked. Driver sets up data frame pointers to fetch the data from and write it back to in the IP registers and triggers the IP for execution. 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. The application then decides its next course of action based on its current state, sets up data for the next IP, and invokes the next IP in pipeline. This shuttling of data back-and-forth between CPU, memory, and IPs is shown in Figure 2.3. 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 CPU interrupts for every IP, as well as bypass memory by making each IP directly communicate with the next one, as shown in Figure 2.3 (as Ideal). Note that, as IPs typically have a queue of request buffers, CPU submit a series of requests to the drivers. Without stalling for the response, CPU proceed with its other tasks. In most cases, for graphics/frame or display bound applications, 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.

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 the other application. 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. We observed in general that the IPs queue multiple requests from the CPU, and serve them in a specific order. 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 number of video-player instances is increased, video-playback quality decreased visibly [1], as well as the overheads (CPU activity) observed at the cores is very high.

[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

2.2 Related Work

There has been a growing interest in enhancing performance and energy efficiency of handheld devices. This primarily includes memory-centric optimizations [16–21], designing efficient accelerators [22–31], system-level energy-centric optimizations [32–41], and platform development for analyzing SoC systems and applications [42–46]. There has also been considerable work on power modeling/optimization in smartphones including proposals for a system-call-based power model [47], consumption of network devices/protocols [34] and various IPs [41, 48, 49], and a network-based power reduction technique [50].

Simulation Infrastructure and Characterization Handheld Platforms &Applications: Gutierrez et al. [42] analyze the micro-architectural characteristics of smartphone applications without focusing on IP behavior. Another recent work [51] proposes an infrastructure to simulate smartphone cores, by integrating GEM5 and the OS, to study emerging smartphone workloads. Again this study is only core centric and lacks IP analysis. Similarly, the simulator in [52] does not model an SoC system with multiple accelerators, and lacks Android support. With GemDroid [44], we have developed an infrastructure that can simulate multiple IPs as well as cores with Android OS, which can be easily extended to include more IP models. Several works have investigated the power consumption of different applications [53], and different IPs [35] in smartphones, and have proposed infrastructure to simulate mobile networks [54, 55].

Techniques targetting DRAMs and Memory Controllers: Several works have investigated memory scheduling techniques in the context of SoCs and smartphones. Lee and Change [16] describe the essential issues in memory system design for SoCs. Lee et al. [33] propose a memory scheduling mechanism that provides latency and bandwidth guarantees for memory accesses, and Akesson et al. [17] discuss a memory scheduling technique that provides a guaranteed minimum bandwidth and maximum latency bound

to IPs. Lin et al. [18] employ a hierarchical memory scheduler that improves system throughput. Jeong et al. [19] provide QoS guarantees to frames by balancing requests at the memory controller. In the context of CMPs and uni-processor systems, several works have proposed low-power memory designs [38, 56] that can be applied in smartphones for better energy efficiency. Recently, optimizations for inter-accelerator communication have been proposed [57–59]. Note that these solutions primarily target single application executions, and more importantly, focus exclusively on the memory performance.

Accelerator Design: Ozer et al. [60] describe the steps involved in the design and verification of ARM IPs. Saleh et al. [61] discuss reusability, integrity, and scalability of IPs used in SoCs. Along with IP design and analysis, several works have proposed IP-specific optimizations [22–27].

Energy Characterization and Power Management in SoCs: There is a large body of work in this domain for smartphones. Pathak et al. [47] explore the limitations of utilization-based power models, and propose a system-call-based power model. Balasubramanian et al. [34] study the power consumption of network devices and protocols in smartphones. Falaki et. al [50] propose a network-based power reduction technique. Several prior studies have investigated the power consumption of various IPs and applications in smartphones [32, 41, 49]. Many works have proposed optimizations specific to mobile web browser applications [32, 62–65] to improve browsing performance and energy efficiency. For frame-based applications, existing optimizations [40, 66–68] typically focus on power management of video decoders.

Virtualization: Virtualization [69] has found widespread adoption at complete system (server) level, as well as for individual devices including CPU, network [70, 71] and storage. In the accelerator domain, [72–76] 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 [70, 71]).

Chapter 3

GemDroid: A Simulation Infrastructure For Handheld Systems

3.1 An Extendable Simulation Framework

The goal of GemDroid is to serve as a framework that aids in evaluating mobile architecture designs. Given the diversity in the types of phones and tablets that get built and used, the goal of the proposed framework is to provide flexibility for evaluating these designs with multiple cores and IPs. The framework needs to be agnostic to the details in the IP model, which can be a simple analytical model or a complex cycle-accurate model of the IP's micro-architecture.

Currently very limited infrastructure exists for enabling platform level studies across multiple IPs running realistic and/or relevant workloads. GEM5 framework is the closest, that the authors are aware of, which can simulate an ARM or x86 cycle-accurate core with Android/Linux Kernel running on top of it [5, 42]. Currently, GEM5 can simulate only a limited set of IPs (core and only display panel). This limited support and drastic simulation slow down severely restricts the number of apps that can be run. Further, it is

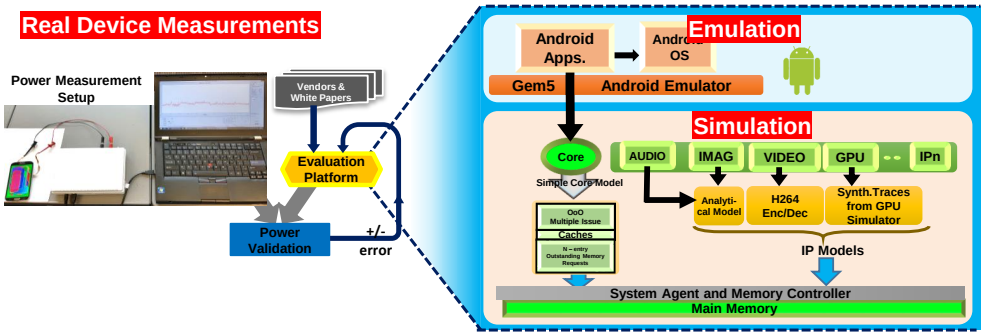


Figure 3.1: Detailed Infrastructure Diagram.

not possible to do IP-centric evaluations. While incorporating GEM5 in its infrastructure, GemDroid proposes expands on the number of IPs modeled to get close to a complete device (see Figure 3.1). Further, GemDroid proposes makes it flexible in terms of the modeling technique adopted for the IPs for which cycle-accurate models are hard to build mainly due to unavailable public information.

The proposed GemDroid framework consisting of two main components – Android emulator and GEM5 – is shown in Figure 3.1. These sub-components are besides the individual IP models GemDroid relies heavily on the Google Android's open-source emulator and it has been enhanced for this dissertation's needs. Android emulator meets two essential goals – booting an operating system and running commonly used applications on top. The emulator runs the latest version of Android compiled for ARMv7 ISA with Neon instructions. The core of the emulator, based on the Qemu tool [7], translates each ARM instruction to a set of native machine instructions and executes them on the host. During this translation, instruction level traces are captured. The proposed framework will also emulate other IPs such as the imaging (handles the images captured using the camera), display, network, audio and there are hooks available to emulate sensors such as accelerometer, gyrometer, etc. [1]

However, the emulator misses out on the crucial part needed for performance studies:

[1]Note, we propose to instrument the code base of the applications to collect traces when those IPs that are not tracked by the emulator are invoked.

it does not incorporate the simulation time for any of the IPs. The emulator's goal is to enable application development for Android and hence has a different set of goals than this dissertation's. GemDroid integrates existing performance models - GEM5 for the core and memory subsystem, Attila for graphics [6] - and analytical models for the other IPs missing a model (like Video, Network and others). This work *does not* claim to have developed performance/power models for all IPs, but the proposed framework is extensible, and will allow for other users in the community to incorporate their models as needed. In particular, it can be augmented with additional cores (CPUs) as well as different types of IPs as they (their models) become available.

Trace-based simulation

Unlike server workloads or widely available benchmarks like SPEC, PARSEC [77], etc., mobile applications are more user interactive. Providing user inputs and studying the system is not an easy task due to the associated non-determinism; for that, one possible method is to capture user inputs that are sent to the OS, replay it exactly while evaluating the system [43, 51]. In the infrastructure, Android emulator is used as the front end, where one can install almost all applications available on Google Play and provide inputs like the way it is done on SoCs. The emulator has been instrumented to capture the ARM instructions and IP calls along with their interactions with the memory in a trace file. Using such a trace, provides determinism in evaluating such applications with user inputs.

3.2 Model characteristics

While a cycle-accurate full system simulation meets the accuracy goals for micro-architectural and system level studies, they cannot simulate considerable durations of target mobile workloads due to complexity associated with handling multiple IPs. On the other hand, while development boards can meet the speed requirements they fail to

provide control for exploring the system by changing the underlying parameters. On the other hand having synthetic models for all IPs, though will drastically reduce simulation time, may not be very helpful in exposing performance trends in architectural design space or in providing detailed insights into application performance. As described in [5], GEM5 can cycle-accurately simulate 200K instructions per second, potentially leading to 800X slowdown for a processor-core based system. If the simulator is augmented with accurate models of GPU, audio/video encoder/decoder, and imaging IPs, the simulation times would become unreasonable. Hence GemDroid looks to keep the infrastructure flexible for integrating models with differing levels of complexity and allowing them to interact. Depending on the IP of interest, users can integrate accurate models for specific IPs and integrate *less accurate models* for the rest of the system. The less accuracy is with respective to not modeling the micro-architecture details of IPs, but having enough information to capture timing associated with different activities. In this work, for studying system-level memory characteristics across IPs, an alternate simplified core model was developed in the infrastructure which assumes a 1-IPC model. This does not affect the timing accuracy of the execution significantly as many frequently used ARM ISA instructions are single-cycle instructions. This is similar to what has been done in [78, 79]. Such a system had only a 180X slowdown compared to real hardware. Users though have the flexibility to switch between the highly accurate GEM5 core model or the simplified core model based on their requirements. Note that when the system is extended with cycle-accurate core model, significant slowdowns were observed resulting in only a short duration of execution time being simulated. Such a simulation is unsuitable for IP based system studies as not many IP calls are seen in such short duration.

For the graphics IP, we used the Attila graphics simulator [6], which handles the openGL calls issued by applications. These OpenGL calls that are used for rendering different images to the screen, are captured in the trace. For video IP, we used the open-source H264 RTL model [80] to capture the timing associated with decoding. For audio, and imaging (applications that use camera for capturing pictures or recording video), we use

the emulator to capture the calls to audio and imaging IP. These calls provide us with the sizes of the frames, and the arrival rate of the frame requests (based on the number of instructions between frames). Currently, the integrated framework has IP models that closely reproduces the memory access behavior for GPUs rendering OpenGl games, display panel, audio/video encoders/decoders, network interface, and imaging.

For power validation of this framework, we use an iterative method to determine the realistic minimum and maximum power drawn by accelerators across multiple use-cases. We compared the average power consumed on the simulation framework with the average power consumed on a Samsung S4 device. To measure device power, we connected Monsoon Power Monitor [81] to the battery terminals of a Samsung S4 device, and measured total system power (at 0.2 ms resolution) for the apps that predominantly stress a particular IP. We ran in airplane mode, turned off GPS, and only bare minimum background activities were enabled on the phone for over a minute (shown in Figure 3.1). We iteratively tuned the model such that the maximum error on the simulation infrastructure was well within ∼10% of the actual measurements for the chosen applications.

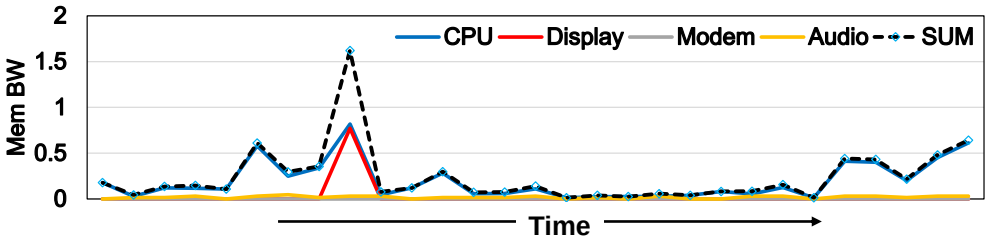


Figure 3.2: Execution where display IP reads a frame from memory at the same time core is writing a new frame.

3.3 Capabilities of GemDroid

The infrastructure can be used to conduct multiple types of studies starting from the core, memory and individual IPs to system-level performance/power analyses. The first insight we can get from using GemDroid is the usage pattern of IPs for different applications. Currently, we have incorporated multiple IP models in GemDroid and have analyzed the

behavior of a wide spectrum of applications such as games, video recording and video playback. While GemDroid may not simulate (cycle) accurately, it models the performance of various components based on a combination of various available resources (online reference manuals, architecture designs, etc.,) faithfully with reasonable accuracy. Hence, variation with real world performance and simulator performance is possible. With GemDroid, in addition to understanding IP usage behavior, the platform can help in studying contention for shared resources. Consider the example shown in Figure 3.2. The figure illustrates a scenario when YouTube video playback traces were simulated on the system and different IP's memory accesses were analyzed. We note that there are instants when two IPs perform memory access at the same time. Further, multi-core studies are also possible once we collect application-level traces for multiple applications. We leave this as a future work. Instead, in this thesis, we analyze the memory system of current SoCs to quantify its impact on application performance, explain how a heterogeneous memory controller design can help mitigate some of the problems the memory system brings and explore methods to circumvent memory through IP-to-IP direct communication. The rest of the chapters will explore in detail each of the above.

Chapter 4

Design of a Heterogeneous Memory Controller (HMC)

We provide a brief overview of the existing baseline memory design, and explain the proposed heterogeneous memory controller (HMC) design, and finally show some initial results with such a design.

4.1 Locality-Parallelism Tradeoff in Memory Design

Baseline Memory System: Figure 4.1 (A) shows the memory design of our baseline system. It consists of 2 memory controllers (MCs) controlling two distinct regions of memory. As shown, the cores and IPs *share* this memory subsystem. Traditionally, these MCs are the gateway to access data in memory, which is logically organized as DRAM banks. Each bank has cells (memory elements) laid out in arrays of rows and columns. The data can be striped across banks at various granularity, for example, at *page-level* or at *cache-line-level*. In *page-level*, the distribution of data across banks is at a granularity of a OS page, which is a chunk of multiple consecutive cache lines. For example, if page

size is 4KB (as used in this work), the first 4KB of consecutive data is mapped to the first bank, and the next 4KB to the next bank, and so on. In our baseline system, we use this page-level striping for both the memory controllers, as shown in Figure 4.1 (A). In *cache-line-level* striping, the distribution of data across banks is at a much finer granularity – at cache-line granularity. In this case, every other cache line is mapped to a different bank.

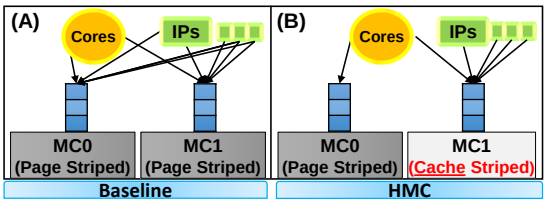


Figure 4.1: Schematic of (A) Baseline memory design and (B) Proposed HMC memory design.

Locality vs. Parallelism: When accessing a cache line from memory, the row that contains the cache line is brought to a buffer called *row buffer*, which is associated with every DRAM bank. Once the contents are placed in the *row buffer*, subsequent memory requests to the same row are served from the row buffer (row-buffer hits), instead of fetching them again from the memory array. This reduces access latency, improves performance, and saves the energy-expensive job of reading the row from memory array. Instead, if a different row from the one in the row buffer is requested, the current row is closed, and the new row gets placed in the buffer. This incurs high memory latency and is a high energy consuming task. Therefore, it is optimal to receive and serve requests from the row buffer. For this very reason, the most popular form of data distribution in CMPs is page-level striping, where up to 4KB of consecutive data can be mapped to a bank, and if requests are scheduled timely, all the data can be fetched from the row-buffer, thereby improving DRAM locality and energy efficiency. On the other hand, page-level striping restricts parallelism, as not many DRAM banks can be utilized in parallel. This is because if the requests possessing good locality are scheduled roughly at the same time, only a limited set of the banks will be accessed and the other DRAM banks will be idle. This limitation can be addressed by

cache-line striping, where the same 4KB of data is striped across banks, and hence the same requests will access multiple banks. Such striping, although increases parallelism, it reduces locality. It is apparent that both techniques of data-distribution have pros and cons.

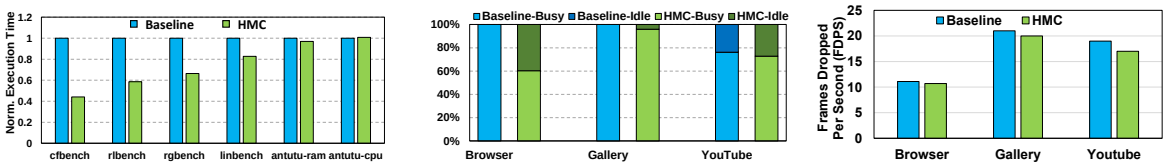
Auxiliary Metrics: In this context, we define two auxiliary metrics, which will be used to understand this trade-off. First, Bank Level Parallelism (BLP), which is defined as the average number of memory banks that are busy when there is at least one request being served at this memory controller [79]. Improving BLP enables better utilization of DRAM bandwidth. Second, Row-Buffer Locality (RBL), which is defined as the average row-buffer hit rate across all memory banks [79]. Improving RBL decreases average latency for memory requests, and increases the memory service rate.

4.2 Overview of the Proposed Design

The IPs have significantly higher memory bandwidth requirements compared to cores. This manifests into two primary problems: (1) the IP requests arrive in bursts thereby causing large queuing delays for CPU requests reducing the core performance, and (2) the IP memory requests interfere with core requests, thereby impacting the row-buffer locality of all the requests. Due to these two issues, the DRAM bandwidth utilization is severely affected leading to degradation of system performance. To address this, we propose having separate memory regions for mobile systems.

4.2.1 Memory Region Separation

In this design, we divide the address space into two regions: first region – associated with a dedicated memory controller (MC1) for CPU data which is accessed only by the CPUs, and the second region – associated with MC2 for IP data, which can be accessed by both cores and IPs. Note that, we cannot have completely dedicated memory controllers for IP and CPU requests, because the data produced by IPs need to be used by cores (or vice versa).



(a) Reduction in execution time for CPU-bound applications.

(b) Reduction in CPU busy cycles with HMC.

(c) Improvement in average FDPS for Onscreen applications

Figure 4.2: Performance improvements of HMC with respect to baseline system.

The goal of this design is to offer dedicated memory controller for core requests, as these requests are more latency critical. On the other hand, requests for the IP region are bandwidth intensive, as they arrive in bursts and access large chunks of data. Now, with separate memory regions, the bursts of requests coming to IP regions access consecutive cache lines. Due to this, these requests have very good row buffer locality. But, the downside of such an access pattern is that the bank-level-parallelism is very low.

4.2.2 Heterogeneity in Data Striping

To address the above problem, we enhance the design of memory with appropriate data striping. We adopt two different data striping techniques for MCs: MC0 uses page-level striping, and MC1 uses cache-line striping as shown in Figure 4.1(B). The motive of having two different striping techniques is to increase the BLP for IP memory region, while retaining the row-locality at CPU memory region. Note that, in general, cache-line striping reduces row-buffer locality. However, in this scenario (especially for IPs), typically the row-buffer locality is not affected, because these regions receive requests to large chunks of contiguous data. Consider the example where the system has cache-line striped n memory banks, and the display IP is accessing a large frame region. In such a system, consecutive cache lines are mapped to different banks in a cyclic manner, such that every n^{th} cache line is mapped to the same bank. Similarly, as there is a large number of IP requests for each consecutive cache line, every n^{th} request is mapped to the same bank. All the requests that are mapped to a bank hit in the row buffer, leading to high row-buffer-locality. Because the IPs typically

access consecutive cache lines, the requests that are mapped to the same bank are likely to hit in the row buffer, leading to high row-buffer locality. Also, as the IP requests are sent to different banks, they take advantage of BLP. Note that in the proposed design, there are no extra overheads in terms of data copies or data duplication. All IP-associated data are written to DRAM through a separate memory controller(MC1) by cores or other IPs. While MC0 can be accessed by the cores, MC1 can be accessed by the cores and the IPs.

4.3 Results

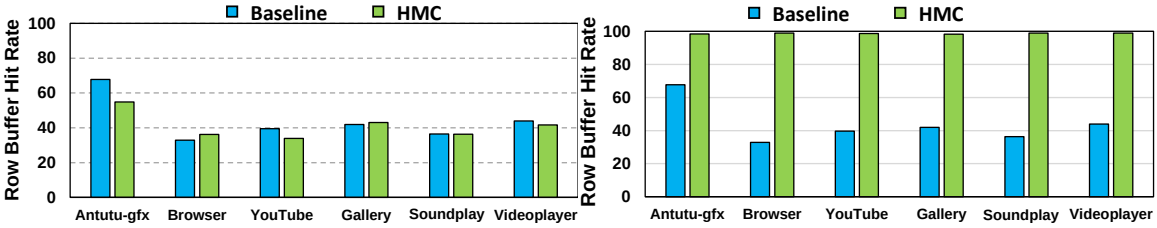
We compare our HMC design to an iso-resource baseline system with 2 memory controllers which are page-striped. In the baseline system, the memory controllers are not aware of the characteristics of IPs' and cores' requests. We do not consider cache-line striped memory controllers as they increase the memory latency for all core's memory operations, thus reducing system performance and energy efficiency. In the proposed design, *Heterogeneous Memory Controller* (HMC), we isolate the requests targeted to IP and CPU memory regions. Figure 4.2a shows the performance comparison of HMC with the baseline system for representative CPU bound and Onscreen applications. We report their respective evaluation metrics (execution times for CPU bound applications, and CPU-busy cycles and FDPS for Onscreen applications). From Figure 4.2a, we observe that, on average, the execution time of CPU-bound applications is reduced by 25% (up to 56% in cfbench). This improvement is primarily attributed to two reasons. First is the reduced interference from IP accesses on the CPU requests at MC 0, because of memory region separation. Second is the reduction in latency at MC1 because of increased RBL and BLP.

The variance in reduction in execution times are attributed to the impact of IP accesses on the CPU accesses. If an application has relatively more number of IP accesses, it is likely to perform better with our HMC design. Note that for core bound applications, which do

not have *any IP calls* (antutu-ram and antutu-cpu), will mostly not take advantage of HMC's optimizations. In fact, in some cases, they might lose performance due to reduced memory channel parallelism for CPU requests. In our studies, we find that the execution time of Antutu-CPU application increases by less than 1%.

The graph in Figure 4.2b shows the CPU activity under different memory system designs. In on-screen applications, CPU has to process data before an IP can consume it or vice versa. By employing our HMC design, the CPU processes the data quicker leaving it idle for more cycles. This can be seen in the second set of bars in the graph. This reduction in busy cycles directly translates to power savings.[1] Figure 4.2c shows the metric Frames Dropped Per Second (FDPS) under different memory system designs. HMC design makes the memory subsystem faster for both CPU and IP memory requests leading to fewer frame drops per second.

To understand the impact of our HMC design, we analyze some auxiliary metrics below. First, we look at how the locality at the memory controllers is affected due to HMC in Figure 4.3. Sub-graph (a) shows that the locality (row-buffer hit rates) at MC0 which is receiving only CPU requests in the HMC case did not change much, while (b) shows the row-buffer hit rates increase to almost 100%. This is mainly because, when the address regions are partitioned, only requests to IP memory space arrive at MC1. These requests typically access consecutive cache lines, contributing to high number of row-buffer hits.

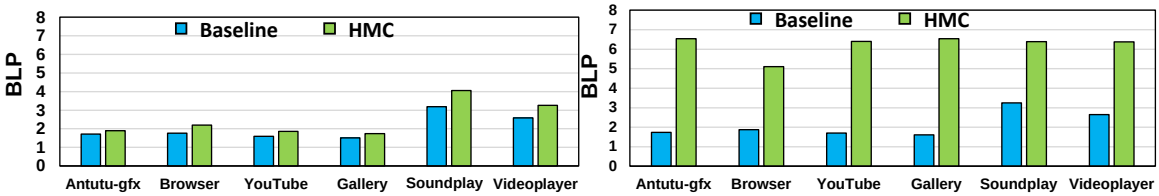


(a) Effect on locality in CPU-region addresses. (b) Effect on locality in IP-region addresses.

Figure 4.3: Impact of HMC on locality. Baseline has both MCs serving both CPU and IP requests without distinguishing between them.

[1] In this work, we focus on performance and do not have a comprehensive power model for the system components.

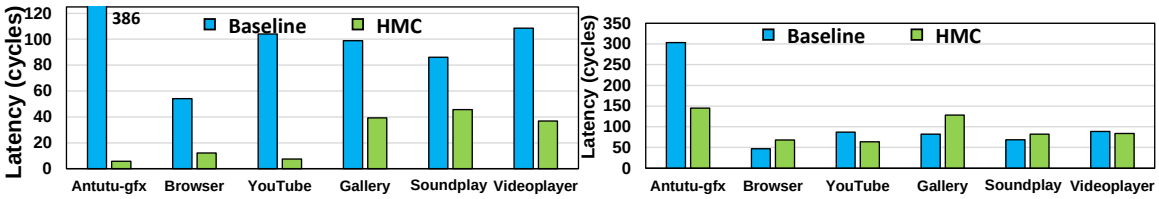
In HMC, though there is significant locality, because consecutive accesses go to the different banks due to cache-line striping, the bank level parallelism is also observed to be substantially higher than the base case. Particularly, this can be seen in Figure 4.4 (b), where the BLP for base-case averages around 1.25 banks only, whereas for HMC averages around 5.8 banks across all applications. In this IP memory region, as the requests that arrive typically go to consecutive banks in cyclic fashion, BLP tends to remain so high. Thus, Figures 4.3 and 4.4 together clearly show that our design did not lose locality when striping cache lines across banks. It is also clear from these graphs that with intelligent data mapping in memory, like in HMC, we can get the benefits of both, locality and parallelism.



(a) Effect on bank-level parallelism in CPU-region addresses.

(b) Effect on bank-level parallelism in IP-region addresses.

Figure 4.4: Impact of HMC on Bandwidth. Baseline has both MCs with default page-striped addresses.

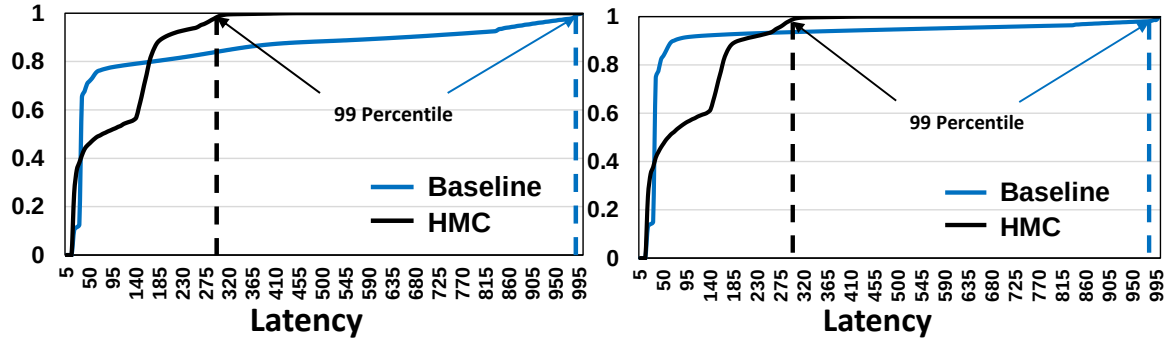


(a) Effect on latency in CPU-region addresses.

(b) Effect on latency in IP-region addresses.

Figure 4.5: Impact of HMC on Latency. Baseline has both MCs serving both CPU and IP requests without distinguishing between them.

We observe that locality and parallelism in this system have significant impact on the latency of memory requests. Figure 4.5 shows the average latency of requests arriving at the memory controllers. We observe that, with HMC, in MC0, average latency improves because the core's requests are isolated from IP's requests. Thus, the latency critical core requests are served much faster, leading to performance improvements. In IP-region memory



(a) CDF of latencies of memory accesses sent to IP address region in YouTube application.

(b) CDF of latencies of memory accesses sent to IP address region when using Browser application.

Figure 4.6: Impact of HMC showing how increase in bank-level parallelism reduces latency of requests in IP-address region.

controller, the latencies were not affected significantly even though the requests are coming in bursts.

Finally, in Figure 4.6, we plot the cumulative distribution function of latencies of memory requests that arrived at the IP-region memory controller for YouTube and Browser application. The x-axis in this plot is the memory latency in cycles. We can observe that, with HMC, 99 % of the requests have latencies less than 300 cycles, while in baseline system, only 82% (youtube) and 95% (browser) of the requests have this latency. This clearly shows the benefits of HMC in reducing the memory latencies.

4.4 Conclusions

Analyzing a spectrum of applications with GemDroid, we observed that the memory subsystem is a vital cog in the mobile platform because, it needs to handle both core and IP traffic, which have very different characteristics. Consequently, we present a heterogeneous memory controller (HMC) design, where we divide the memory physically into two address regions, where the first region with one memory controller (MC) handles core-specific application data and the second region with another MC handles all IP

related data. The proposed modifications to the memory controller design results in an average 25% reduction in execution time for CPU bound applications, up to 11% reduction in frame drops, and on average 17% reduction in CPU busy time for on-screen (IP bound) applications.

Chapter 5

Short-Circuiting Memory Traffic

Real-time interactive applications, including interactive games, video streaming, camera capture, and audio playback, are amongst the most popular on today's tablets and mobiles apart from email and social networking. Such applications account for nearly 65% of the usage on today's handhelds [82], stressing the importance of meeting the challenges imposed by such applications efficiently. This important class of applications has several key characteristics that are relevant to this study. First, these applications work with input (sensors, network, camera, etc.) and/or output (display, speaker, etc.) devices, mandating real-time responsiveness. Second, these applications deal with "frames" of data, with the requirement to process a frame within a stipulated time constraint. Third, the computation required for processing a frame can be quite demanding, with hardware accelerators often deployed to leverage the specificity in computation for each frame and delivering high energy efficiency for the required computation. The frames are then pipelined through these accelerators one after another sequentially. Fourth, in many of these applications, the frames have to *flow* not just through one such computational stage (accelerator) but possibly though several such stages. For instance, consider a video capture application, where the camera IP may capture raw data, which is then encoded into an appropriate form by another IP, before being sent either to a flash storage or a

display. Consequently, the frames have to flow through all these computational stages, and typically the memory system (the DRAM main memory) is employed to facilitate this flow. Finally, we may need to support several such flows at the same time. Even a single application may have several concurrent flows (the video part and audio part of the video capture application which have their own pipelines). Even otherwise, with multiprogramming increasingly prevalent in handhelds, there is a need to concurrently support individual application flows in such environments.

5.1 Overview

Apart from the computational needs for real-time execution, all the above observations stress the memory intensity of these applications. Frames of data coming from any external sensor/device is streamed in to memory, from which it is streamed out by a different IP, processed and put back in memory. Such an undertaking places heavy demands on the memory subsystem. When we have several concurrent flows, either within the same application or across applications in a multiprogrammed environment, all of these flows contend for the memory and stresses it even further. This contention can have several consequences: (i) without a steady stream of data to/from memory, the efficiencies from having specialized IPs with continuous dataflow can get lost with the IPs stalling for memory; (ii) such stalls with idle IPs can lead to energy wastage in the IPs themselves; and (iii) the high memory traffic can also contend with, and slow down, the memory accesses of the main cores in the system. While there has been a lot of work covering processing – whether it be CPU cores or specialized IPs and accelerators (e.g. [61] [48] [16]) – for these handheld environments, the topic of optimizing the data flows, while keeping the memory system in mind, has drawn little attention. Optimizing for memory system performance, and minimizing consequent queueing delays has itself received substantial interest in the past decade, but only in the area of high-end systems (e.g., [34] [79] [83] [78]). This work addresses this critical issue in the design of handhelds,

where memory will play an increasingly important role in sustaining the data flow not just across CPU cores, but also between IPs, and with the peripheral input-output (display, sound, network and sensors) devices.

In today's handheld architectures, a *System Agent* (SA) [84–87] serves as the glue integrating all the compute (whether it be IPs or CPU cores) and storage components. It also serves as the conduit to the memory system. However, it does not clearly understand data flows, and simply acts as a slave initiating and serving memory requests regardless of which component requests it. As a result, the high frame rate requirements translate to several transactions in the memory queues, and the flow of these frames from one IP to another explicitly goes through these queues, i.e., the potential for data flow (or data *reuse*) across IPs is not really being exploited. Instead, this work explores the idea of *virtually integrating accelerator pipelines* by "short-circuiting" many of the read/write requests, so that the traffic in the memory queues can be substantially reduced. Specifically, this work explore the possibility of *shared buffers/caches* and *short-circuiting communication* between the IP cores based on requests already pending in the memory transaction queues. In this context, the following specific **contributions** are made:

- It is shown that the memory is highly utilized in these systems, with IPs facing around 47% of their total execution time stalling for data, in turn, causing 24% of the frames to be dropped in these applications. We cannot afford to let technology take care of this problem since with each DRAM technology advancement, the demands from the memory system also become more stringent.

- Blindly provisioning a shared cache to leverage data flow/reuse across the IP cores is also likely to be less beneficial from a practical standpoint. An analysis of the IP-to-IP reuse distances suggests that such caches have to run into several megabytes for reasonable hit rates (which would also be undesirable for power).

- This work shows that this problem is mainly due to the current frame sizes being

relatively large. Akin to tiling for locality enhancement in nested-loops of large arrays [88–90], it introduces the notion of "sub-frame" for restructuring the data flow, which can substantially reduce reuse distances.

- With this sub-framing in place, it is shown that reasonably sized shared caches – referred to as *flow buffers* in this work – between the producer and consumer IPs of a frame can circumvent the need to go to main memory for many of the loads from the consumer IP. Such reduction in memory traffic results in around 20% performance improvement in these applications.

- While these flow buffers can benefit these platforms substantially, it also explores an alternate idea of not requiring any separate hardware structures – leveraging existing memory queues for data forwarding from the producer to the consumer. Since memory traffic is usually high, recently produced items are more likely to be waiting in these queues (serving as a small cache), which could be forwarded to the requesting consumer IP. It is also shown that these can be accommodated in recently-proposed memory queue structures [91], and demonstrate performance and power benefits that are nearly as good as that of the flow buffer solution.

Typically, DRAM is shared between the cores and IPs and is used to transfer data between them. There is a high degree of data movement and this often results in a high contention for memory controller bandwidth between the different IPs. Depending on the type of IPs involved, frames get written to memory or read from memory at a certain rate. For example, cameras today can capture video frames of resolution 1920x1080 at 60 FPS and the display refreshes the screen with these frames at the same rate (60 FPS). Therefore, 60 bursts of memory requests from both IPs happen in a second, with each burst requesting one complete frame. While the request rate is small, the data size per request is high – 6MB for a 1920x1080 resolution frame (this will increase with 4K resolutions [10]). If this amount of bandwidth cannot be catered to by the DRAM, the memory controller and DRAM queues fill up rapidly and in turn the devices and accelerators start experiencing performance drops.

The performance drop also affects battery life as execution time increases.

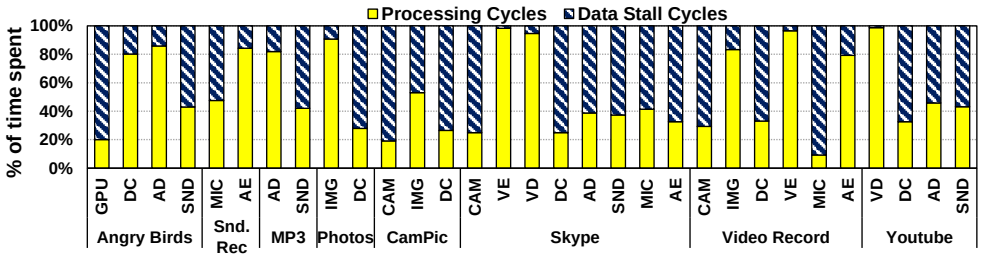


Figure 5.1: Total data stalls and processing time in IPs during execution.

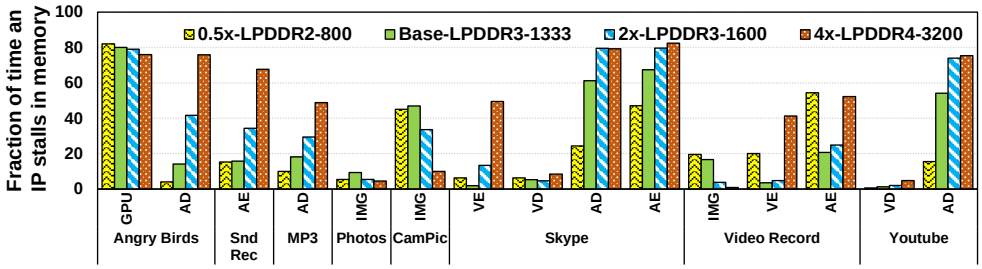


Figure 5.2: Trends showing increase of percentage of data stalls with each newer generation of IPs and DRAMs.

To explain how much impact the memory subsystem and the system-agent can have on IPs' execution time (*active cycles* during which the IPs remains in active state), in Figure 5.1, we plot the total number of cycles spent by an IP in processing data and in data stalls. Here, we use "data stall" to mean the number of cycles an IP stalls for data without doing any useful computation, after issuing a request to the memory. We observe from Figure 5.1 that the video decoder and video encoder IPs spend most of their time processing the data, and do not stress the memory subsystem. IPs that have very small compute time, like the audio decoder and sound engine, demand very high bandwidth than what memory can provide, and thus tend to stall more than compute. Camera IP and graphics IP, on the other hand, send bursts of requests for large frames of data at regular intervals. Here as well, if memory is not able to meet the high bandwidth or has high latency, the IP remains in the high-power mode stalling for the requests. The high data stalls seen in Figure 5.1 translate to frame drops which is shown in Figure 5.3 (for 5.3 GBPS memory bandwidth). We see that on average 24% of the frames are dropped with the default baseline system, which can hurt user experience with the device. With higher memory bandwidths (2x and

4x of the baseline bandwidth), though the frame drops decrease, they still do not improve as much as the increase in bandwidth. Even with 4x baseline bandwidth, we observe more than 10% frame drops (because of higher memory latencies).

As user demands increase and more use-cases need to be supported, the number of IPs in the SoC is likely to increase [9] along with data sizes [10]. Even as the DRAM speeds increase, the need to go off-chip for data accesses places a significant bottleneck. This affects performance, power and eventually the overall user experience.

Further, to establish the maximum gains that can be obtained if we had an ideal and perfect memory, we did a hypothetical study of *perfect memory with 1 cycle latency*. The cycles-per-frame results with this perfect memory system are shown in Figure 5.4. As expected, we observed drastic reduction in cycles per frames across applications and IPs (as high as 75%). In some IPs, memory is not a bottleneck and those did not show improved benefits. From this data, we conclude that reducing the memory access times does bring the cycles per frame down, which in turn boosts the overall application performance. Note that, this perfect memory does not allow any frames to be dropped.

5.2 Data Reuse and Reuse Distance

In a flow, data get read, processed (by IPs) and written back. The producer and consumer of the data could be two different IPs or sometimes even the same IP. We capture this IP-to-IP reuse in Figure 5.5, where we plotted the physical addresses accessed by the core and other IPs for `YouTube` application. Note that this figure only captures a very small slice of the

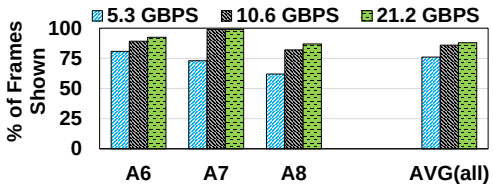


Figure 5.3: Percentage of frames completed in a subset of applications with varying memory bandwidths.

entire application run. Here, we can see that the display-controller (DC) (red points) reads a captured frame from a memory region that was previously written to by video decoder (black points). Similarly, we can also see that the sound-engine reads from an address region where audio-decoder writes. This clearly shows that the data gets reused repeatedly across IPs, but the reuse distances can be very high. As mentioned in Section 2.1.2, when a particular application is run, the same physical memory regions get used (over time) by an IP for writing different frames. In our current context, the reuse we mention is only between the producer and consumer IPs for a particular frame and nothing to to do with frames being rewritten to the same addresses. Due to frame rate requirements, reuse distances between display frame based IPs were more than tens of milli-seconds, while audio frame based IPs were less than a milli-second. Thus, there is a large variation across producer-consumer reuse distances across IPs that process large (display) frames (e.g., VD, CAM) and IPs that process smaller (audio) frames (e.g., AD, AE).

5.2.1 Converting Data Reuse into Locality

Given the data reuse, the simplest solution is to place a on-chip cache and allow the multiple IPs to share it. The expectancy is that caches are best for locality and hence they should work. In this subsection, we evaluate the impact of adding such a shared cache to hold the data frames. Typical to conventional caches, on a cache-miss, the request is sent to the transaction queue. The shared cache is implemented as a direct-mapped structure, with multiple read and write ports, and multiple banks (with a bank size of 4MB), and

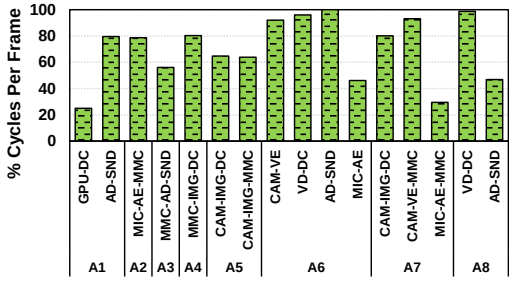


Figure 5.4: Percentage reduction in Cycles-Per-Frame in different flows with a perfect memory configuration.

the read/write/lookup latencies are modeled using CACTI [92]. We evaluated multiple cache sizes, ranging from 4MB to 32MB, and analyzed their hit rates and the reduction in cycles taken per frame to be displayed. We present the results for 4MB, 8MB, 16MB and 32MB shared caches in Figure 5.6 and Figure 5.7 for clarity. They capture the overall trend observed in our experiments. In our first experiment, we notice that as the cache sizes increase, the cache hit rates either increase or remain the same. For applications like `Audio Record` and `Audio Play` (with small frames), we notice 100% cache hit rates from 4MB cache. For other applications like `Angry Birds` or `Video-play` (with larger frames), a smaller cache does not suffice. Thus, as we increase the cache capacity, we achieve higher hit rates. Interestingly, some applications have very low cache hit rates even with large caches. This can be attributed to two main reasons. First, frame sizes are very large to fit even two frames in a large 32MB cache (as in the case of `YouTube` and `Gallery`). Second, and most importantly, if the reuse distances are large, data gets kicked out of caches by the other flows in the system or by other frames in the same flow. Applications with large reuse distances like `Video-record` exhibit such behavior.

In our second experiment, we quantify the performance benefits of having such large shared caches between IPs, and give the average cycles consumed by an IP to process a full-frame (audio/video/camera frame). As can be seen from Figure 5.7, increasing the cache sizes does not always help and there is no optimal size. For IPs like SND and AD, the frame sizes are small and hence a smaller cache suffices. From there on, increasing cache size increases lookup latencies, and affects the access times. In other cases, like DC, as the frame sizes are large, we observe fewer cycles per frame as we increase the cache size. For

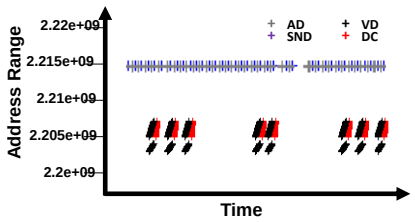


Figure 5.5: Data access pattern of IPs in YouTube application.

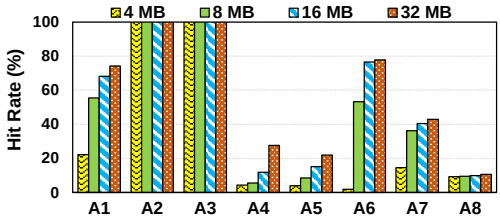


Figure 5.6: Hit rates under various cache capacities.

other accelerators with latency tolerance, once their data fits in the cache, they encounter no performance impact.

Further, scaling cache sizes above 4MB is not reasonable due to their area and power overheads. Figure 5.8 plots the overheads for different cache sizes. Typically, handhelds operate in the range of 2W – 10W, which includes everything on the device (SoC+display+network). Even the 2W consumed by the 4MB cache will impact battery life severely.

Summary: To summarize, the high number of memory stalls is the main reason for frame drops, and large IP-to-IP reuse distances is the main cause for large memory stalls. Even large caches are not sufficient to capture the data reuse and hence, accelerators and devices still have considerable memory stalls. All of these observations led us to re-architect how data gets exchanged between different IPs, paving way for better performance.

The primary goal is to reduce the IP-to-IP data reuse distances, and thereby reduce data stalls, which are a major impediment to performance.

To achieve this, this work proposes a novel approach of *sub-framing* the data. One of the commonly used compiler techniques to reduce the data reuse distance in loop nests that

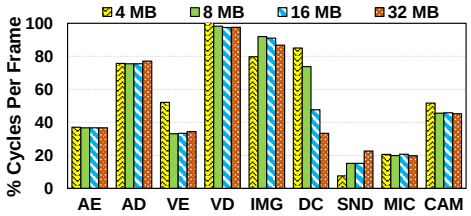


Figure 5.7: Cycles Per Frame under various cache capacities.

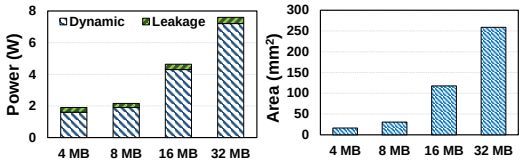


Figure 5.8: Area and power-overhead with large shared caches.

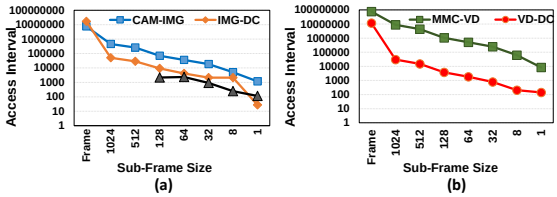


Figure 5.9: IP-to-IP reuse distance variation with different sub-frame sizes. Note that the y-axis is in the log scale.

manipulate array data is to employ *loop tiling* [88, 89]. It is the process of partitioning a loop's iteration space into smaller blocks (tiles) in a manner that the data used by the loop remains in the cache enabling quicker reuse. Inspired by tiling, we propose to break the data frames into smaller *sub-frames*, that reduces IP-to-IP data reuse distances.

In current systems, IPs receive a request to process a data frame (it could be a video frame, audio frame, display frame or image frame). Once it completes its processing, the next IP in the pipeline is triggered, which in-turn triggers the following IP once it completes its processing and so on. In our solution, we propose to sub-divide these data frames into smaller sub-frames, so that once IP1 finishes it's first subframe, IP2 is invoked to process it. In the following sections, we show that this design reduces the hardware requirements to store and move the data considerably thereby bringing both performance and power gains. The granularity of the subframe can have a profound impact on various metrics.

To quantify the effects of subdividing a frame, we varied the sub-frame sizes from 1 cache line to the current data frame size, and analyzed the reuse distances. Figure 5.9 plots the reduction in the IP-to-IP reuse distances (on y-axis, plotted on *log-scale*), as we reduced the size of a sub-frame. We can see from this plot an inverse exponential decrease in reuse distances. In fact, for very small sub-frame sizes, we see reuse distances in less

than 100 cycles. To capitalize on such small reuse distances, we explore two techniques – *flow-buffering* and opportunistic IP-to-IP *request short-circuiting*.

5.2.2 Flow-Buffering

In Section 5.2.1, we showed that even large caches were not very effective in avoiding misses. This is primarily due to very large reuse distances that are present between the data-frame write by a producer and the data-frame read by a consumer. With sub-frames, the reuse distances reduce dramatically. Motivated by this, we now re-explore the option of caching data. Interestingly, in this scenario, caches of much smaller size can be far more effective (low misses). The reuse distances resulting from sub-framing are so small that even having a structure with few cache-lines is sufficient to capture the temporal locality offered by IP pipelining in SoCs. We call these structures as *flow-buffers*. Unlike a shared cache, the flow-buffers are private between any two IPs. This design avoids the conflict misses seen in a shared cache (fully associative has high power implications). These flow-buffers are write-through. As the sub-frame gets written, the sub-frame is written to memory. The reason for this design choice is discussed next.

In a typical use-case involving data flow from IP-A→IP-B→IP-C, IP-A gets its data from the main-memory and starts computing it. During this process, as it completes a sub-frame, it writes back this chunk of data into the flow-buffer between IP-A and IP-B. IP-B starts processing this sub-frame from the flow-buffer (in parallel with IP-A working on another sub-frame) and writes it back to the flow-buffer between itself and IP-C. Once IP-C is done, the data is written into the memory or the display. Originally, every read and write in the above scenario would have been scheduled to reach the main memory. Now, with the flow-buffers in place, all the requests can be serviced from these small low-latency cost and area efficient buffers.

Note that, in these use-cases, cores typically run device driver code and handle

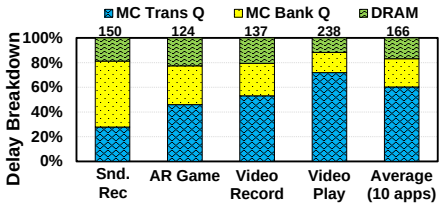


Figure 5.10: Delay breakdown of a memory request issued by IPs or cores. The numbers above the bar give the absolute cycles.

interrupts. They have minimal data frames processing. Consequently, we do not incorporate flow-buffers between core and any other accelerator. Also, when a use-case is in its steady-state (for example, a minute into running a video), the IPs are in the active state and quickly consume data. However, if an IP is finishing up on an activity or busy with another activity or waking up from sleep state, the sub-frames can be overwritten in the flow-buffer. In that case, based on sub-frame addresses, the consumer IP can find its data in the main memory since the flow-buffer is a write-through buffer. From our experiments, we found that a flow-buffer size of 32 KB provides a good trade-off between avoiding a large flow-buffer and sub-frames getting overwritten.

5.2.3 IP-IP Short-circuiting

The flow-buffer solution requires an extra piece of hardware to work. To avoid the cost of adding the flow-buffers, an alternate technique would be to enable consumers directly use the data that their producers provide. Towards that, we analyzed the average round-trip delays of all accesses issued by the cores or IPs (shown in Figure 5.10) and found requests spend maximum time queuing in the memory subsystem. `MC Trans Queue` shows the time taken from the request leaving the IP till it gets to the head of the transaction queue. The next part `MC Bank Queue`, is the time spent in bank queues. This is primarily determined by whether the data access was a row buffer hit, or miss. And, finally `DRAM` shows the time for DRAM accessing along with the response back to the IPs. As can be seen, most of the time is spent in the memory transaction queues (˜100 cycles). This means that data that

could otherwise be reused lies idle in the memory queues and we use this observation towards building an opportunistic IP-to-IP short-circuiting technique, similar in concept to "store-load forwarding" in CPU cores [93, 94][1] though our technique is in between different IPs. There are correctness and implementation differences, which we highlight in the following paragraphs.

IPs usually load the data frames produced by other IPs. Similar to store-load forwarding, if the consumer IP's load requests can be satisfied from the memory transaction queue or bank queues, the memory stall time can be considerably reduced. As the sub-frame size gets smaller, the probability of a load hitting a store gets higher. Unlike the flow-buffers discussed in Section 5.2.2, store data does not remain in the queues till they are overwritten. This technique is opportunistic and as the memory bank clears up its entries, the request moves from the transaction queue into the bank queues and eventually into main memory. Thus, the loads need to follow the stores quickly, else it has to go to memory. This distance between the consumer IP load request and producer IP store request depends on how full the transaction and bank queues are. In the extreme case, if both the queues (transaction-queue and bank-queue) are full, the number of requests that a load can come after a store will be the sum of the number of entries in the queues.

The overhead of implementing the IP-IP short-circuiting is not significant since we are using pre-existing queues present in the system agent. The transaction and bank queues already implement an associative search to re-order requests based on their QoS requirements and row-buffer hits, respectively [95]. Address-searches for satisfying core loads already exist and these can be reused for other IPs. As we will show later, this technique works only when the sub-frame reuse distance is small.

[1] Core requests spend relatively insignificant amount of time in transaction queues as they are not bursty in nature. Due to their strict QoS deadlines, they are prioritized over other IP requests. They spend more time in bank queues and in DRAM.

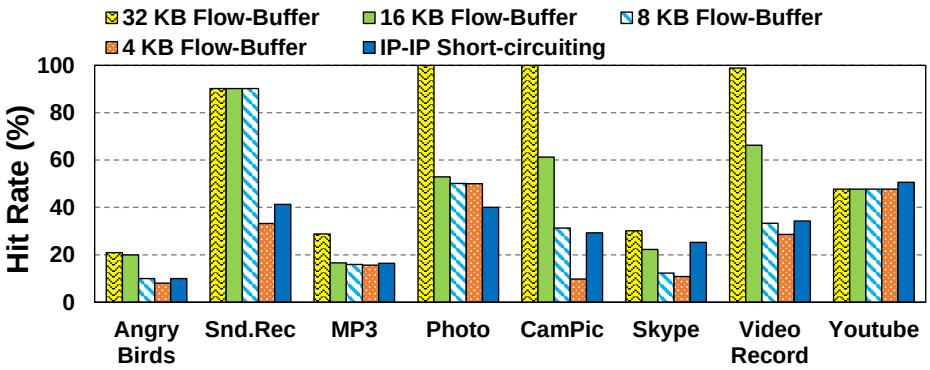


Figure 5.11: Hit rates with flow-buffering and IP-IP short-circuiting.

Effects of Sub-framing Data

The benefits of sub-framing are quantified in Figure 5.11 in terms of hit rates when using flow-buffering and IP-IP short-circuiting. We can see that the buffer hit rates increase as we increase the size of flow-buffers, and saturate when the size of buffers are in the ranges of 16KB to 32KB. The other advantage of having sub-frames is the reduced bandwidth consumption due to the reduced number of memory accesses. As discussed before, accelerators primarily face bandwidth issues with the current memory subsystem. Sub-framing alleviates such bottleneck by avoiding fetching every piece of data from memory. Redundant writes and reads to same addresses are avoided. Latency benefits of our techniques, as well as their impact on user experience will be shown in Section 5.4.

5.3 Implementation Details

In implementing our sub-frame idea, we account for the probable presence of dependencies and correctness issues resulting from splitting frames. Below, we discuss the correctness issue and the associated intricacies that need to be addressed to implement sub-frames. We then discuss the software, hardware and system-level support needed for such implementations.

5.3.1 Correctness

We broadly categorize data frames into the following types – (i) video, (ii) audio, (iii) graphics display, and (iv) the network packets. Of these, the first three types of frames are the ones that usually demand sustained high bandwidth with the frame sizes varying from a megabyte to tens of MBs. In this work, we address only the first three types of frames, and leave out network packets as the latency of network packet transmission is considerably higher compared to the time spent in the SoC.

Video and Audio Frames

Encoding and decoding, abbreviated as *codec* is compression and decompression of data that can be performed at either hardware or software layer. Current generation of smartphones such as Samsung S5 [96] and Apple iPhone [97] have multiple types of codes embedded in their phone.

Video Codecs: First, let us consider the flows containing video frames, and analyze the correctness of sub-dividing such large frames into smaller ones. Among the video codecs, the most commonly used are H.264 (MPEG-4) or H.265 (MPEG-H, HEVC) codecs. Let us take a small set of video frames and analyze the decoding process. The encoding process is almost equivalent to the inversion of each stage of decoding. As a result, similar principles apply there as well. Figure 5.12 shows a video clip in its entirety, with each frame component named. A high-quality HD video is made up of multiple frames of data. Assuming a

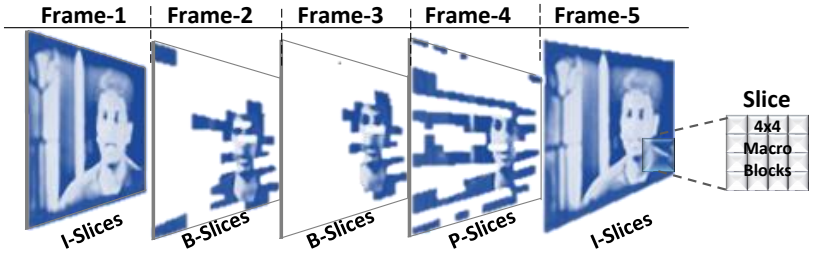


Figure 5.12: Pictorial representation showing the structure of five consecutive video frames.

default of 60 FPS, the amount of data needed to show the clip for a minute would be 1920x1080(screen resolution) x 3(bytes/pixel) x 60 (frame rate) x 60 (seconds) = 21.35 GB. Even if each frame is compressed individually and stored on today's hand-held devices, the amount of storage available would not permit it. To overcome this limitation, codecs take advantage of the temporal redundancy present in video frames, as the next frame is usually not very different from the previous frame.

Each frame can be dissected into multiple slices. Slices are further split into macroblocks, which are usually a block of 16x16 pixels. These macroblocks can be further divided into finer granularities such as sub-macroblocks or pixel blocks. But, we do not need such fine granularities for our purpose. Slices can be classified into 3 major types: I-Slice (independent or intra slices), P-Slice (predictive), and B-Slice (bi-directional) [98] as depicted in Figure 5.12. I-slices[2] have all data contained in them, and do not need motion prediction. P-slices use motion prediction from one slice which belongs to the past or future. B-slices use two slices from past or the future. Each slice is an independent entity and can be decoded without the need for any other slice in the *current frame*. P- and B-slices need slices from a previous or next frame only.

In our sub-frame implementation, we choose *slice-level granularity* as the finest level of sub-division to ensure correctness *without having any extra overhead of synchronization*. As slices are independently decoded in a frame, the need for another slice in the frame does not arise, and we can be sure that correctness is maintained. Sub-dividing any further would bring in dependencies, stale data and overwrites.

Audio Codecs: Audio data is coded in a much simpler fashion than video data. An audio file has a large number of frames, with each audio frame having the same number of bits. Each frame is independent of another and it consist of a header block and data block. Header block (in MP3 format) stores 32-bits of metadata about the coming data block

[2]Earlier codecs had frame level classification instead of slice level. In such situations, I-frame is constructed as a frame with only I-slices.

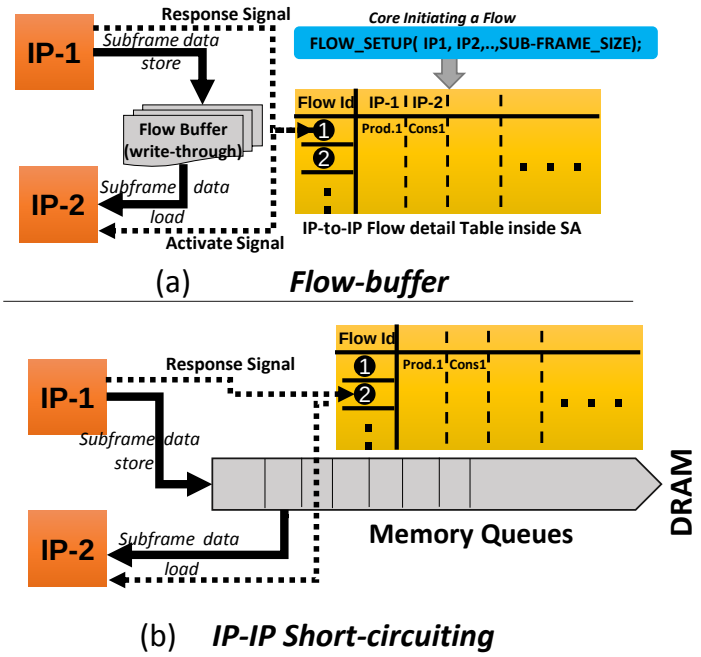


Figure 5.13: High level view of the SA that handles sub-frames.

frame. Thus, each audio frame can be decoded independently of another as all required data for decoding is present in the current frames header. Therefore, using a full audio frame as a sub-frame would not cause any correctness issue.

Graphics Rendering: Graphics IPs already employ tiled rendering when operating on frames and for the display rendering. These tiles are similar to the sub-frames proposed in this work. A typical tiled rendering algorithm first transforms the geometries to be drawn (multiple objects on the screen) into screen space and assigns them into tiles. Each screen-space tile holds a list of geometries that needs to be drawn in that tile. This list is sorted front to back, and the geometry behind another is clipped away and only the ones in the front are drawn to the screen. GPUs renders each tile separately to a local on-chip buffer, which is finally written back to main-memory inside a framebuffer region. From this region, the display controller reads the frame to be displayed to be shown on screen. All tiles are independent of each other, and thus form a sub-frame in our system.

5.3.2 OS and Hardware Support

In current systems, IPs are invoked sequentially one-after-another per frame. Let us revisit the example considered previously – a flow with 3 IPs. The OS, through device drivers, calls the first IP in the flow. It waits for the processing to complete and the data to be written back to memory and then calls the second IP. After the second IP finishes its processing, the third IP is called. With sub-frames, when the data is ready in the first IP, the second IP is notified of the incoming request so that it can be ready (by entering to the active state from a low power state) when the data arrives. We envision that the OS can capture this information through a library (or multiple libraries) since the IP-flows for each application are pretty standard. In Android [99] for instance, there is a layer of libraries (Hardware Abstraction Layer – HAL) that interface with the underlying device drivers and these HALs are specific to IPs. As devices evolve, HAL and the corresponding drivers are expected to enable access to devices to run different applications. By adding an SA HAL and its driver counterpart to communicate the flow information, we can accomplish our requirements. From the application's perspective, this is transparent since the access to the SA HAL happens from within other HALs as they are requested by the applications. Figure 5.13 shows a high level view of the sub-frame implementation in SA along with our short-circuiting techniques. From a hardware perspective, to enable sub-framing of data, the SA needs to have a small matrix of all IPs – rows corresponding to producers and columns to consumers. Each entry in the row is 1 bit per IP. Currently, we are looking at about 8 IPs, and this is about 8 bytes in total. In future, even as we grow to 100 IPs, the size of the matrix is small. As each IP completes its sub-frame, the SA looks at its matrix and informs the consumer IP. In situations where we have multiple flows (currently Android allows two applications to run simultaneously [100]) with an IP in common, the entries in the SA for the common IP can be swapped in or out along with the context of the application running. This will maintain the correct consumer IP status in the matrix for any IP.

5.4 Evaluation

The performance and power benefits obtained by using sub-frames compared to the conventional baseline system which uses full frames in IP flows is presented below. We used a modified version of the GemDroid infrastructure [44] for the evaluation. For each application evaluated, we captured and ran the traces either to completion or for a fixed time. The trace lengths varied from about 2 secs to 30 secs. Usually this length is limited by frame sizes we needed to capture. The transaction queue and bank queues in SA can hold 64 entries (totaling 8KB). For the flow buffer solution, we used a 32 KB buffer (based on the hit rates observed in Figure 5.11).

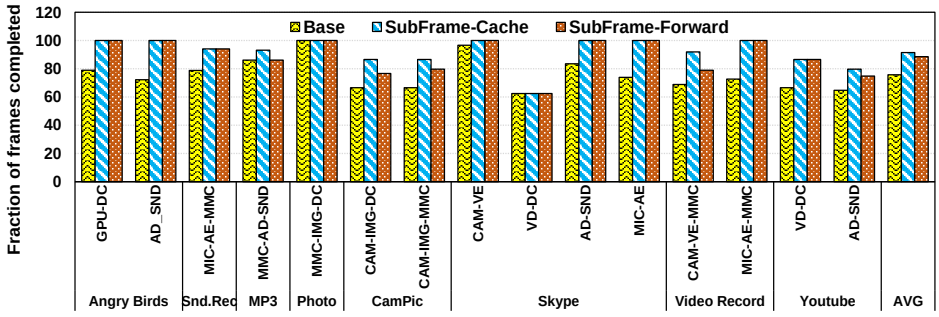


Figure 5.14: Percentage of Frames Completed (Higher the better).

User Experience: As a measure of user experience, we track the number of frames that could complete in each flow. The more frames that get completed, lesser the frame drops and better is the user experience. Figure 5.14 shows the number of frames completed in different schemes. The y-axis shows the percentage of frames completed out of the total frames in an application trace. The first bar shows the frames completed in the baseline system with full frame flows. The second and third bars show the percentage of frames completed with our two techniques. In baseline system, only around 76% of frames were displayed. By using our two techniques, the percentage of frames completed improved to 92% and 88%, respectively. Improvements in our schemes are mainly attributed to the reduced memory bandwidth demand and improved memory latency as the consumer IP's

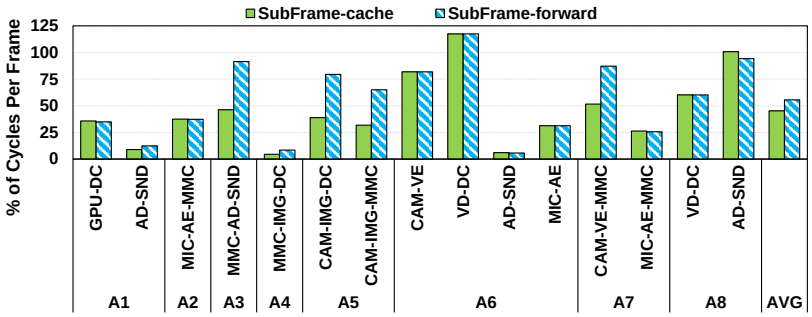


Figure 5.15: Reduction in Cycles Per Frame in a flow normalized to Baseline (Lower the better).

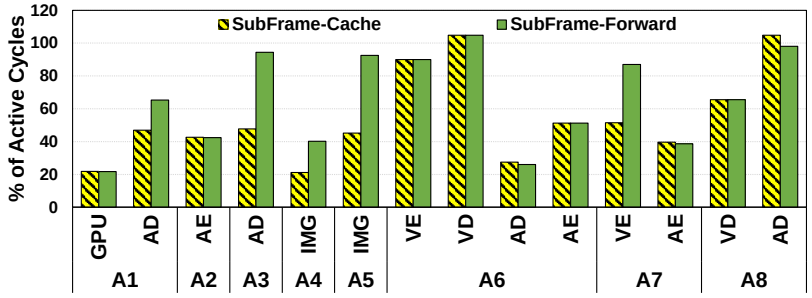


Figure 5.16: Reduction in Number of Active Cycles of Accelerators (Lower the better).

requests are served through the flow-buffers or by short-circuiting the memory requests. The hit-rates of consumer's requests were previously shown in Figure 5.11. In some cases, flow-buffers perform better than short-circuiting due to the space advantage in the flow buffering technique.

Performance Gains: To understand the effectiveness of our schemes, we plot the average number of cycles taken to process a frame in each flow in Figure 5.15. This is the time between the invocation of first IP and completion of last IP in each flow. Note that, reducing the cycles per frame can lead to fewer frame drops. When we use our techniques with sub-framing, due to pipelining of intra-frame data across multiple IPs instead of sequentially processing one frame after another, we are able to substantially reduce the cycles per frame by 45% on average. We also observed that in A6-Skype application (which has multiple flows), through the use of sub-framing, the memory subsystem gets overwhelmed because, we allow more IPs to work at the same time. This is not the case in

the base system. If IPs do not benefit from flow-buffers or IP request short-circuiting, the memory pressure is more than the baseline leading to some performance loss (17%).

Energy Gains: Energy efficiency is a very important metric in handhelds since they operate out of a battery (most of the time). Exact IP design and power states incorporated are not available publicly. As a proxy, we use the number of cycles an IP was active to correspond to the energy consumed when running the specific applications. In Figure 5.16, we plot the total number of active cycles consumed by an accelerator compared to the base case. We plot this graph only for accelerators as they are compute-intensive and hence, consume most of the power in a flow. On average, we observe 46% and 35% reduction in active cycles (going up to 80% in GPU) with our techniques, which translates to substantial system-level energy gains. With sub-framing, we also reduce the memory energy consumption by 33% due to (1) reduced DRAM accesses, and (2) memory spending more time in low-power mode. From the above results it can be concluded that sub-framing yields significant performance and power gains.

5.5 Conclusion

Memory traffic is a performance bottleneck in mobile systems, and it is critical that we optimize the system as a whole to avoid such bottlenecks. Media and graphics applications are very popular on mobile devices, and operate on data frames. These applications have a lot of temporal locality of data between producer and consumer IPs. This work show that by operating a frame as an atomic block between different IPs, the reuse distances are very high, and thus, the available locality across IPs goes unexploited. By breaking the data frames into sub-frames, the reuse distance decreases substantially, and we can use flow buffers or the existing memory controller queues to forward data from the producer to consumer. Our results show that such techniques help to reduce frame latencies, which in turn enhance the end-user experience while minimizing energy consumption.

Chapter 6

Virtualizing Flows in SoCs

6.1 Inefficiencies in Current Systems

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.

Performance Inefficiencies due to shared resources We also consider 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 average memory bandwidth increases with the number of applications and the percentage of time when memory is close to its peak bandwidth (¿80%) is typically high.

In nutshell, 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

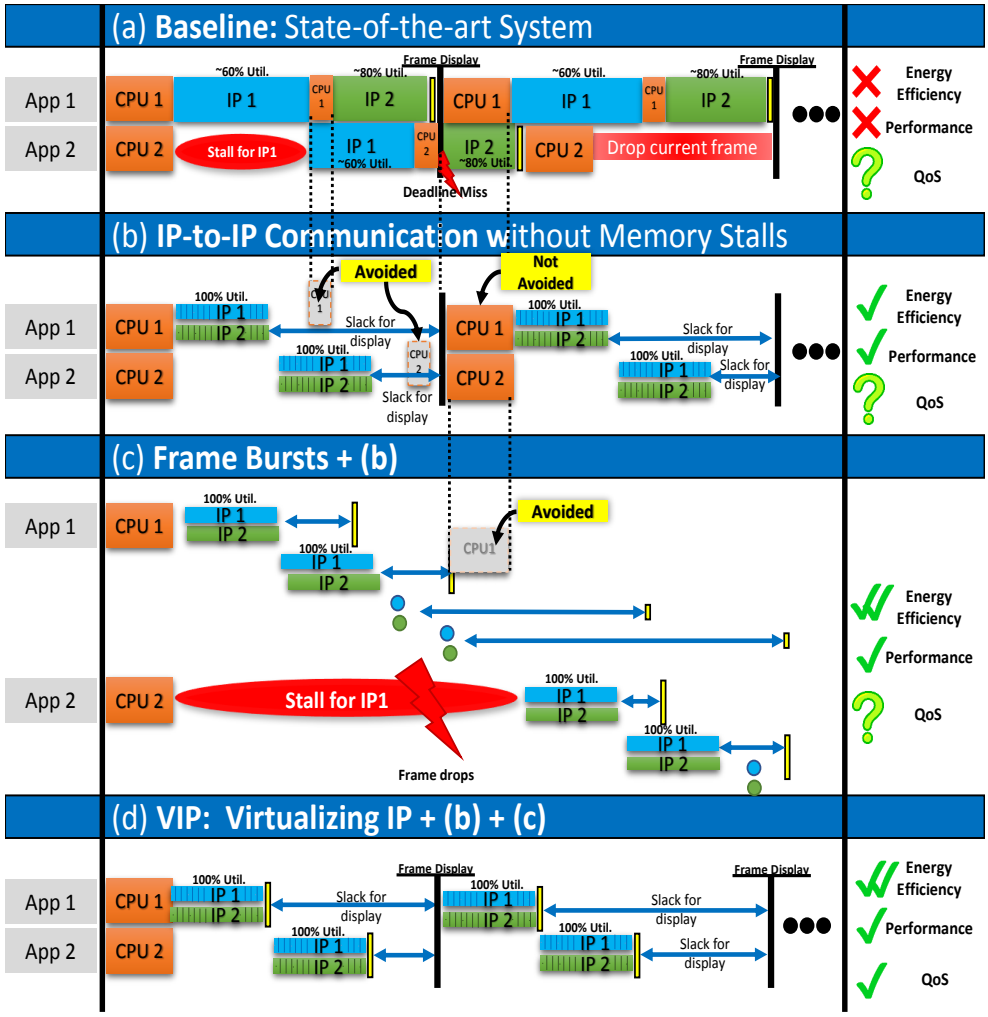


Figure 6.1: Detailed Overview Of Virtualization at Multiple Levels

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.*

6.2 VIP: Virtualizing IP Chains

We propose to tackle the issues mentioned above 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 [70, 71] in the shared resources.

We propose to chain individual IPs and expose a smarter interface to this integrated chain, there by relieving 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, this work additionally incorporates enhancements to conduct rate-based flow control/scheduling mechanisms to meet each frame's QoS requirements.

Overview of Our Proposed Solutions

Figure 6.1 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. (b) on the other hand 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. 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 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 parts below, we discuss each of the above-mentioned parts in greater detail.

IP-to-IP Communication

As discussed earlier in this dissertation, 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* [20, 58, 101], 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 proposal, 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 the CPU just instantiates the chain of IPs in different flows. Instead data flows through a sequence of IPs avoiding intermediate memory operations.

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 6.1.)

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 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 finally, (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 [102]. 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.

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 an open-source versions of a popular game, Flappy Bird [103] (touch-based). We instrumented the code of the game 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.

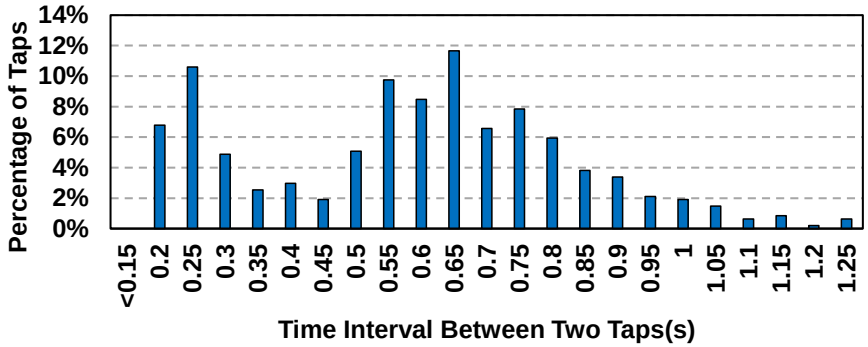


Figure 6.2: Distribution of percentage of frames in between two taps in `FlappyBird`*

In Figure 6.2, 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.

Note that, in applications similar to above, frame bursts will *always* improve frame time and FPS. We observed that for gaming apps, with 10 frames per burst, system responsiveness remains unaffected.

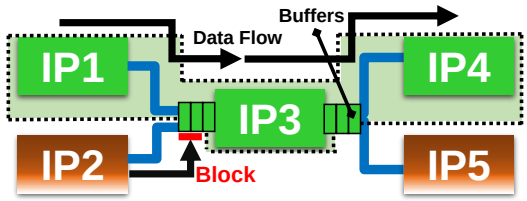


Figure 6.3: Combining Frame-Bursts and IP-to-IP communication leads to Head-Of-Line blocking by shared IPs.

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 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 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 6.3), 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.

6.3 Advantages of VIP

The aim of VIP is to enable IP-to-IP communication, circumventing source (CPU) involvement as well as avoiding head-of-line blocking phenomena caused in the multiple application execution scenario, as explained above. One of the commonly used networking techniques to avoid Head Of Line (HOL) blocking in routers is to *virtualize the channels*

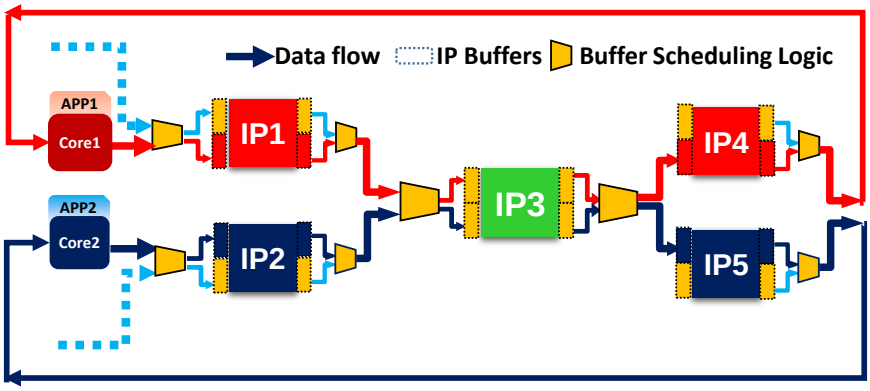


Figure 6.4: VIP Solution with Multiple Applications

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, 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 utilization are maximized.

An illustration of multiple apps sharing IP3 is shown in Figure 6.4. 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. We propose to use a simple EDF (earliest deadline first) scheduling policy when there are multiple requests.

6.4 Evaluation and Results

Applications and Workloads Used

Table 6.1 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 6.2 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 [44] for IP-abbreviations).

Evaluation Platform: For the initial part of the work, we used real systems including Nexus 7, Asus Memo Pad8, Samsung S4 and S5 to understand application and system behavior. In all the systems, as noted before, we use instrumented version of the the

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 6.1: 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 6.2: Multiple Applications Workloads.

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 6.3: Evaluation platform details.

Grafika application suite released by Google for video playback, encoding, and recording. We use `ftrace` [104] 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 [44], 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 components parameters is in Table 6.3.

Results

While motivating the need for virtualizing IP chains in Section 6.2, we showed in Figure 6.1 (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 work. 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).

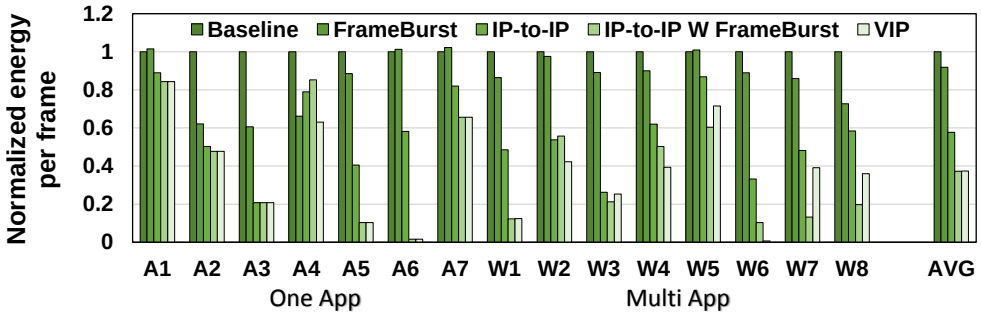


Figure 6.5: Energy Efficiency of VIP.

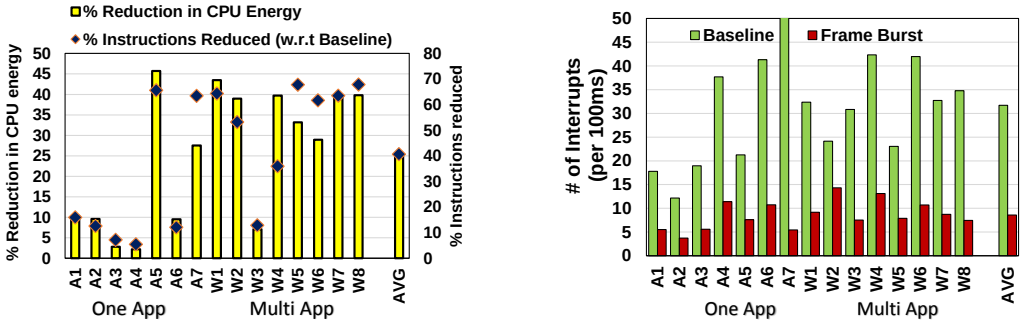


Figure 6.6: Improved energy efficiency of CPUs handling interrupts and scheduling frames with Frame Burst. (a) Shows reduction in CPU energy and executed instructions. (b) Shows reduction in the number of interrupts processed by the CPU.

Energy Efficiency: First, we plot the energy benefits of the evaluated schemes in Figure 6.5 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 6.5 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 6.6. On an average, the CPU consumes ∼25% lower energy when frame 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 shown in Figure 6.5.

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.

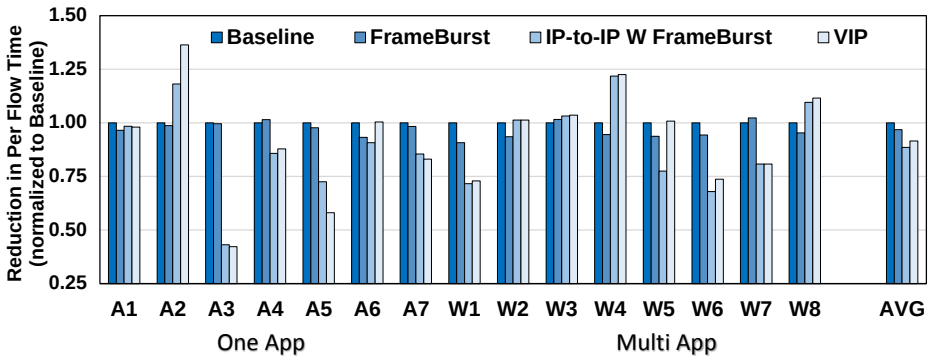


Figure 6.7: Normalized flow time per frame with VIP.

Performance: Frame bursts with IP-to-IP communication provides two benefits. First, as we can see in Figure 6.6, 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 6.7. 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 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 6.8 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.

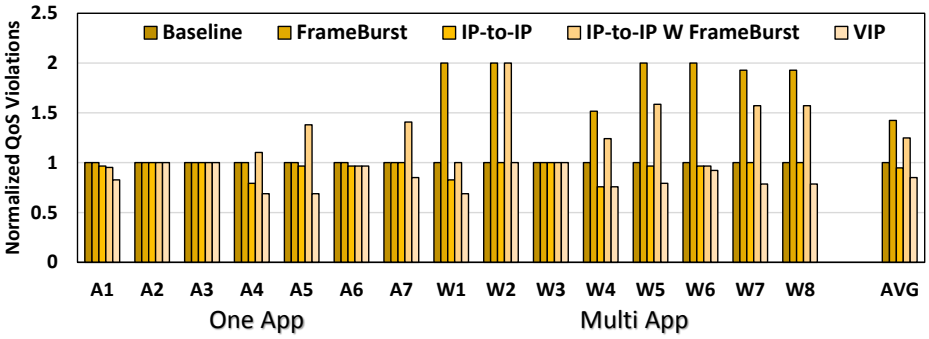


Figure 6.8: VIP enabling meeting QoS deadlines with IP-to-IP communication and Frame bursts.

Overall, while IP-to-IP communication reduces frame drops by 5%, with frame bursts and EDF in place, VIP helps reduce ∼15% frame drops compared to the baseline.

6.5 Conclusions

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 work 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.

Chapter 7

Energy Management in Handhelds

SoC platforms employ DVFS policies to manage the energy consumption and performance of a system. These governors adjust the frequencies of different hardware components in the system to minimize the energy consumption or achieve the required performance.

7.1 Existing Power Management Policies

From a broader perspective, energy management policies can be viewed along two different dimensions: (a) infusing domain awareness into the policy, and (b) co-ordination across different components. Existing works in this area mostly fall under one of the combinations of the above two dimensions – ❶ policies that are *domain-aware, but only one component* is taken into account; ❷ policies that *independently optimize multiple components* but do not consider domain knowledge; and ❸ policies that *coordinatedly manage multiple components* but do not consider domain knowledge. To our knowledge, there are NO prior ❹ ***domain-aware coordinated*** policies, where multiple components like CPU cores, accelerators, and memories are considered together on handhelds, and domain-knowledge application-slack is leveraged.

Optimal: As a yardstick for comparison, we describe an optimal DVFS scheme which assumes that the exact slack for each frame is known at the beginning of the frame. Using this slack as input to a dynamic programming (brute-force) algorithm, the most energy efficient frequency settings for core, memory and IPs are found while making sure performance constraints are met. This is the upper-bound scheme, which gives the best energy efficiency but not implementable. Note that even if the slack is known ahead for each frame, this scheme incurs a complexity of $O(M*N^n*A^n)$, where M is number of frequency settings for the memory, N is number of cores and n is number of frequency settings for core/accelerator, A is number of accelerators.

In the following discussion, we describe some of the previously proposed schemes in categories ❶, ❷, and ❸, and motivate our proposal for ❹ using their deficiencies.

7.1.1 Domain-Aware, Single-Component DVFS

Application specific systems like MPEG-decoders, image-recognition devices, and smart-cameras have specific application properties that allow for specialized DVFS policies [105–109]. These policies use the domain knowledge to their advantage to achieve their goal. They target the frequency of the highest power consuming component on board, typically the main processor.

OracleCPU: As there is a large body of such work [105–113]) that have proposed DVFS solutions for various domains, we choose to implement an ideal *oracle* single-component DVFS policy that targets only the CPU cores (called `Oracle-CPU`). This policy will utilize oracle knowledge of each frame's slack to push the main CPUs into their optimal most energy-efficient frequency point. Frequencies of all other components are set at their highest values.

7.1.2 Independent, Multi-Component DVFS

These policies manage energy consumption of different components on-board independently (without coordination). Each component has its own energy management policy to manage its frequency, either to reduce the energy or increase performance [38, 39, 56, 114]. In current SoC platforms, many components including CPU cores, GPUs, and other high power consuming accelerators have their own DVFS policies which are independent of each other. Specifically, in Android, CPU DVFS policy comes in multiple flavors (called *governors* [115] as they govern the performance of cores). We consider the following 4 representative policies in this space of *independent multi-component DVFS policies*.

HighPerformance governor puts all components at their highest frequency settings with the goal to achieve the best performance.

PowerSave governor locks all the components in lowest frequency setting to achieve least power (not necessarily least energy).

OnDemand governor (default in Android) sets the frequencies of components based on the utilization. Frequency of a component is set to maximum when utilization goes beyond a threshold (typically 90%) and gradually reduces once the utilization falls below the threshold.

ComponentOptimal runs all components (CPU and IPs) at their respective energy-efficient optimal points (the knees in the respective energy curves for various frequencies) and memory at default 433 Mhz *without* taking performance into consideration. Although this governor has the least energy consumption, as all components are in their most energy-efficient point, it may not deliver the required performance.

7.1.3 Coordinated, Multi-Component DVFS

In a system with multiple components, frequency settings for one component can effect other components. For example, setting the frequency of memory can impact the performance of core and vice versa. A coordinated DVFS policy, manages frequencies of multiple components [37, 116], taking such mutual impact into consideration.

CoScale [37] is an example of a coordinated DVFS policy, which scales CPU and memory frequencies together to achieve minimal energy consumption in servers. We note that there has been no such prior proposal in the context of handheld platforms and we simply use CoScale, a coordinated strategy across CPUs and memory, for our own platform even though it has been originally proposed for servers. The idea behind CoScale is to scale the CPU and memory frequencies together while keeping the performance within user defined tolerance. At the beginning of each epoch (5 ms), CoScale profiles the CPU and memory behavior for a short period (300 us) and builds models for energy and performance estimation. Using these models, it runs a greedy heuristic algorithm to determine the frequencies for CPU and memory that achieves minimal energy consumption.

The main problem with DVFS policies like `CoScale` is that they do not have application domain knowledge – how much slack is there, what components are needed to process a frame, and how much processing is needed from each such component, etc. Sampling may not really reveal all the details of the flow within a frame. While these techniques can work well for general server systems, in our scenario, lack of domain awareness can be a serious setback.

7.2 Energy Savings with Existing DVFS Policies

In Figure 7.1, we plot the energy consumed by the above schemes for different applications. All the energy numbers on y-axis are normalized to the `Optimal` scheme which yields the

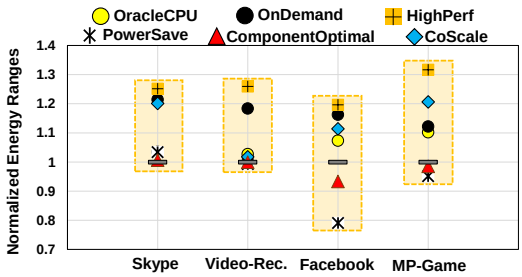


Figure 7.1: Energy consumption of different DVFS schemes, normalized to `Optimal`.

least energy consumption when there are no frame drops. Their corresponding frame drop-rates are shown in Table 7.1.

First, the `Optimal` policy is not only ideal in terms of meeting the performance needs of all the frames, but it also has energy consumption close to `ComponentOptimal` in most cases. We can see that the domain-aware single-component policy (`Oracle-CPU`) fares well in two applications (Skype and Videorecord) but lags behind in others. The four independent multi-component policies have different energy and performance characteristics. The `HighPerformance` governor consumes the maximum energy, while meeting the deadlines to display all frames. `PowerSave` runs all components at the lowest frequency, thus missing most frames (sometimes even complete frames), thus doing less work. `OnDemand` governor is relatively better in energy efficiency than `HighPerformance` as it has the same performance as `HighPerf` with slightly lower energy consumption. The `ComponentOptimal` governor consumes the least energy, as all components run at their most energy-efficient point, but misses around 75% of the frames on average. `CoScale` presents different behaviors depending on the performance degradation parameter selected (10% degradation results are shown here) – being very energy efficient in some cases (as in Videorecord) and missing most deadlines in other cases. Overall, one can conclude that the existing policies are optimized for either energy saving *or* high performance but *not* both. In this work, we propose DVFS policies that try to achieve the best of both worlds.

Policy	Performance			
	Skype	Video-Record	Facebook	MP-Game
OracleCPU	Displays all frames			
OnDemand	Displays all frames			
HighPerformance	Displays all frames			
PowerSave	Misses all frames			
ComponentOptimal	37	95	94	0
CoScale	0	58	17	0
Optimal	Displays all frames			

Table 7.1: Performance of different DVFS schemes, given in terms of number of frame deadlines that were met.

Governor	Performance
On Demand	Displays all frames
High Performance	Displays all frames
Power Save	Misses all frames
Optimal	Displays 75% frames
Optimal (Perf)	Displays all frames

7.2.1 Inefficiency of existing approaches

To understand their shortcomings, we delve deeper into the gaps of two representative DVFS policies, the default `OnDemand` governor (independent multi-component) and `CoScale` (coordinated multi-component), and motivate the need for a slack-aware coordinated multi-component DVFS.

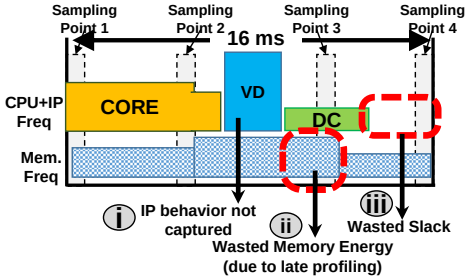


Figure 7.2: Problems with profiling based DVFS policies.

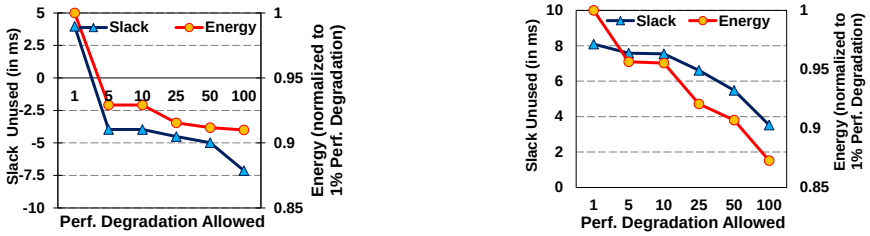
• **Utilization based DVFS may be too late to react:** `OnDemand` is a utilization (history) based DVFS policy. Based on the profile information obtained in the last 10 ms, it decides the core frequencies for the next 10 ms (5 ms if set to aggressive). Assuming similar policies for the accelerators, we notice that a reactive change in frequency based on load at

a coarse granularity (of 5 or 10 ms) does not yield the best results. This is because most IPs typically take a very short time (around 5 ms) when processing a frame and may not even be sampled. As shown in Figure 7.2, the third sampling point occurs only at the end of the 16 ms and hence misses the opportunity to change memory frequency to its optimal value. Thus, appropriate frequencies for the IPs need to be set at the *start* of processing a frame, as being reactive misses energy saving opportunities.

• **Profile and Prediction based DVFS may not capture application semantics:** Profile-based DVFS policies, similar to CoScale, profile for a short span (300 us) and predict the performance of the system for the next 'n' ms. These policies assume that the same set of IPs running during the profiling phase will operate till at least the next sampling period. Unfortunately, such an assumption does not hold true in these platforms, where all IPs are not active all the time. Without such information when processing a flow, these policies are handicapped and miss out on the opportunity to operate in the most energy-efficient frequency. This is also shown in Figure 7.2 case (i) where frequency of `VD` cannot be set appropriately as the component behavior was not at all sampled. Note that sampling at a higher rate adds substantial overhead in terms of energy and performance. In fact, Android does not support sampling faster than 5 ms.

• **Difficulty of finding an appropriate performance degradation tolerance per application and the need for a dynamic tolerance:** In server and other throughput based systems targeted in CoScale, there is a well-defined performance degradation metric (like Cycles Per Instruction) that is more continuous. In mobile platforms, particularly for display-based applications, performance is a rather discrete variable - frame deadline is either met or missed. For example, a tolerance of 20% performance degradation might translate to zero frame drops and a tolerance of 21% can lead to significant number of frame drops. There is no well-defined translation from required frame rate (used in handheld systems) to CPI (used in server systems). To demonstrate this problem, we plot the unused slack (indication of frame rate) and energy saving (normalized to 1% tolerance) for different

tolerance% values (between 1% to 100%) in Figure 7.3. As we see, tolerance value has a dramatic effect on unused slack and energy savings in different applications. In Video-Recorder, moving the tolerance from 1% to even 5% causes significant increase in frame drops. We also observe that, the same tolerance percentage in one application might leave a lot of slack (like Video-Player), or, it misses a lot of frames (as in Video-Recorder) making a case for per application tolerance values. Even within an application, one tolerance setting for the whole execution also may not be suitable because of the dynamic behavior.



(a) Video-Recorder App. (b) Video-Player App.

Figure 7.3: CoScale's performance and energy behavior for different performance degradation thresholds (shown on x-axis from 1% to 100%).

Need For Slack Awareness and Coordination across IPs: The above analysis suggests that we need to (1) leverage the frame-level slack information, and (2) take into account the specific IPs involved in an application flow and their processing times for good energy performance trade-offs. However, trying to get all such information in an application oblivious manner through just sampling is very hard.

7.3 Proposed Domain Aware Energy Management

This section describes our proposed coordinated DVFS schemes which takes domain-awareness (application slack) into account. Towards this goal, we need a prediction of slack per frame (described next), which is then used by our policies. In this section, we describe our slack prediction mechanism, and explain how our policies use it to determine energy-efficient frequency settings.

Slack Prediction Mechanism: Our target applications have more-or-less uniform slack distribution throughout the execution. From our experiments, we observed that a *moving average* of the slack from the 3 previous frames can be a good predictor of the slack in the next frame, because if there is contention for resources, it is experienced across multiple frames.

Allocating the available slack to multiple IPs with different energy characteristics is a multi-parameter optimization problem. Although the `Optimal` (brute-force) scheme described in previous section is slack-aware and coordinated, the search space for this optimization includes *all* possible combinations of frequency settings for all components. Recall that the complexity of such a policy is $O(m*N^n*A^n)$, where m is number of frequency settings for the memory, N is number of CPU cores, and n is number of frequency settings possible for a core/accelerator. Therefore, we resort to more practical heuristic-based algorithms that have a complexity of $O(m*(N+A)*n)$.

1. **Collect time took and energy spent by each component in previous frame**
2. **Predict *slack* based on recent history**
3. **For each IP in the application flow and memory**
   i. Compute the energy difference from energy at optimal frequency
   ii. Compute the time difference from time at optimal frequency
   iii. Compute the ratio of energy difference and time difference
4. **Sort the components based on the energy-time ratio**

(a) Common Algorithm to both Policies.

Greedy Policies

```
While (slack < 0) {
    i. Select component with lowest energy-
       time ratio
    ii. Increase the frequency of the
       selected component as much as
       possible to make slack positive
    iii. If slack > 0, then return
}
While (slack > 0){
    i. Select the component with next
       highest energy-time ratio
    ii. Reduce the frequency of the selected
       component till optimal freq. to use
       the slack
}
```

Kaldor-Hicks (K&H)

```
If (slack < 0), [do same as in Greedy case]
While (slack > 0) {
    i. Select component with highest energy-time
       ratio
    ii. Reduce the frequency of the selected component
       till optimal frequency is reached
}
While (optimal frequency is not reached) {
    i. Estimate energy saving & slack needed to
       reduce the frequency by one step
    ii. Inc. frequency of component with lowest
       energy-time ratio to create slack
    iii. Use created slack to reduce the frequency of
       the component selected in step 1
}
```

(b) Policy Description.

Figure 7.4: Algorithm employed to compute frequencies for components using greedy and K&H policy

7.3.1 Greedy Policies

Towards this, we explore using greedy algorithms that push different components to lower frequencies. For these policies, when slack is predicted to be available (positive slack), a particular component is chosen (as per policy), and its frequency is reduced until either the optimal frequency for the component is reached or the remaining slack becomes zero. If further slack remains, the next component is chosen and the process continues. If the

predicted slack is negative, we move the chosen components to higher frequencies. The order in which the components are chosen is based on the following heuristics.

Max Energy First (MEF): In this policy, we first order the components used by the application based on the total amount of energy consumed in the previous frame for each flow. We then target the components in increasing order of energy consumed.

Most to Gain First (MGF): While the above scheme targets the component consuming maximum energy, it ignores the possibility that the highest energy consumer might already be near its optimal point. Because the component energy curves are concave in nature, as we move closer to the optimal energy point, benefits in moving to even lower frequencies diminish. To take this into account, in this policy, we rank the components based on the difference between energy consumed by it at its current frequency and its optimal frequency, since this is the most we can hope to gain from this particular component.

Slope ($\triangle$E/$\triangle$T) First (SDT): The above two policies consider only the energy side of the picture. However, reducing the frequency of a component has also a cost associated with it in terms of increasing processing time, diminishing the slack. The consequent lower slack left provides less room for energy savings in other components. On the other hand, if we target a component that will reduce energy but does not use up the slack much, it can potentially give more overall benefits since multiple components involved in the flow can be moved to lower frequencies. Particularly, components which are already near optimal give minimum benefits in terms of energy, but consume a large portion of slack. We implement this scheme by ranking the components based on the ratio of energy saved to the slack consumed for the next frequency jump for each component, and choose the one with the maximum value.

7.3.2 Kaldor-Hicks Compensation Policy (K&H)

Note that, in the above three policies, if enough slack is present, we *never* move a chosen component to a higher frequency. Therefore, one can view these policies as **Pareto-efficient**, because at *no* point during the execution the algorithm would make a component worse off to make another well off.

On the other hand, it may sometimes pay off to slightly increase the energy of some component to get significant savings in another. *Kaldor-Hicks Compensation* [8] captures some of the intuitive appeal of Pareto efficiency, but has less stringent criteria allowing more such options to be considered. Under this theory, an outcome is more efficient if those components that are made better off could in theory compensate those that are made worse off and in total can lead to a globally optimal outcome.

In this policy, we target the component that will create the most slack with least increase in energy when its frequency is increased. We then increase its frequency to a point of enough slack that allows another (more energy consuming component, i.e., $\triangle$E/$\triangle$slack) to move to a lower frequency. This process is repeated until such a trade-off seems inefficient.

7.4 Results

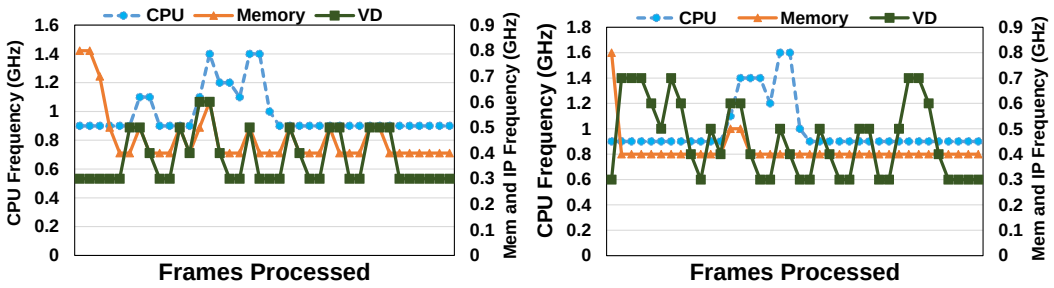


Figure 7.5: Dynamic behavior of SDT Greedy Policy and Kaldor&Hicks policy in YouTube app.

Evaluation Metrics: We use the amount of energy consumed (in milli-joules) per

frame as the metric to compare the energy efficiency of different DVFS policies. For performance, we use the percentage of frames dropped by the application. Note that, if a frame is dropped during execution, it is not counted towards FPS, but it contributes towards the energy consumed.

Results We present the energy and performance results seen across the set of DVFS policies evaluated in this work. We added support for all these policies in our evaluation platform. Also, an application's run time is kept constant (few tens of seconds), while we ran the same part of the trace for all policies, with the same 60 FPS requirement.

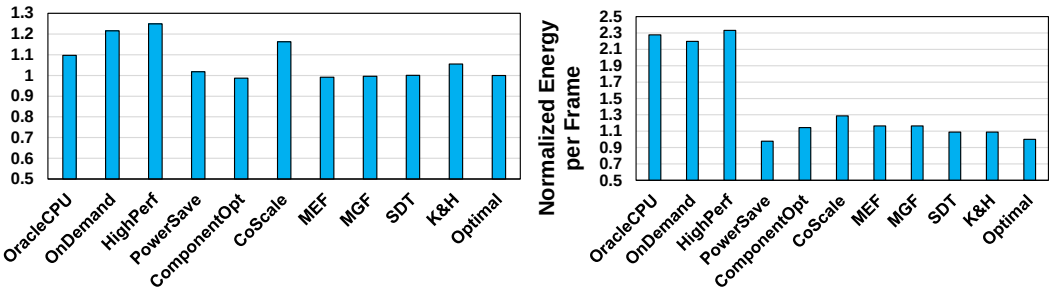


Figure 7.6: Energy Per Frame normalized to the `Optimal` policy for Skype and Angry Birds applications.

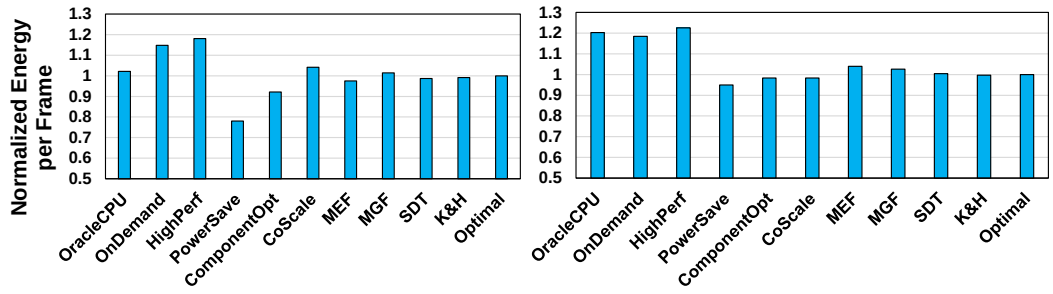


Figure 7.7: Energy Per Frame normalized to the `Optimal` policy for Facebook and VideoRecord applications.

The plots in Figure 7.6, Figure 7.7, Figure 7.8, show the energy per frame for six different applications We normalized the energy consumed per frame to the `Optimal` policy. Recall that `Optimal` is an ideal energy-efficient policy with performance guarantees.

In the results shown, the `OracleCPU` policy is the ideal version of any single-component DVFS policy. We see that even such an ideal implementation consumes 37.8% more energy

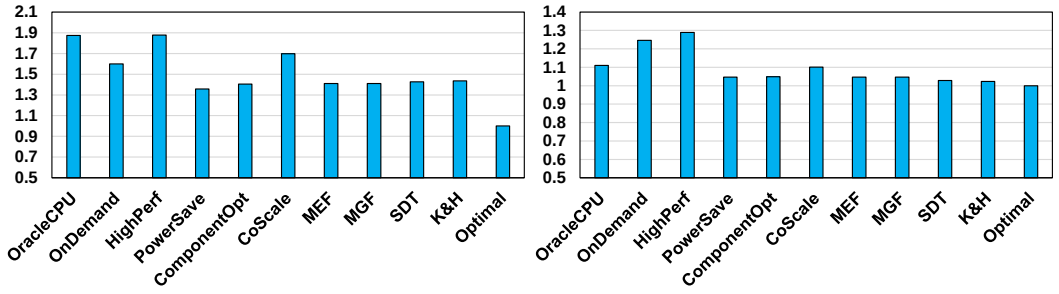


Figure 7.8: Energy Per Frame normalized to the `Optimal` policy for multi-player game and VideoPlayback applications.

DVFS Policy	VidPlay	VidRec	Skype	AngryBirds	Facebook	MPGame
OracleCPU	**0**	**0**	**0**	**0**	**0**	**0**
OnDemand	**0**	**0**	**0**	**0**	**0**	**0**
HighPerf	**0**	**0**	**0**	**0**	**0**	**0**
PowerSave	**All frames dropped (100%)**					
ComponentOpt	**0**	**95**	**37**	**0**	**94**	**0**
CoScale	**0**	**58**	**0**	**0**	**17**	**0**
MEF	**0**	**3**	**3**	**0**	**12**	**0**
MGF	**0**	**3**	**4**	**0**	**10**	**0**
SDT	**0**	**3**	**9**	**0**	**12**	**0**
K&H	**0**	**3**	**4**	**0**	**10**	**0**
Optimal	**0**	**0**	**0**	**0**	**0**	**0**

Table 7.2: Frames dropped (in %) for different policies.

than the optimal case on average. This is attributed to the fact that this policy does not adjust the frequencies of components other than CPU cores. However, for the Facebook app, `OracleCPU` is only 2% worse than optimal. This is because the application is core bound, and thus CPU cores consume the major part of energy. Note that, this policy does not miss any frame deadlines since all the components other than CPU are running at highest frequency.

Among the independent multi-component DVFS policies, as expected, `HighPerformance` and `OnDemand` policies are among the highest energy consumers of all the governors shown, while they guarantee the best performance. On the other hand, while `PowerSave` and `ComponentOptimal` policies conserve more energy than the `Optimal` case, they do less work because they drop a significant fraction of frames, as can be seen in Table 7.2.

For the domain-unaware coordinated policies, we focus on `CoScale`[1]. As shown previously in Figure 7.3, the degradation tolerance value selected can have a significant impact on energy efficieny and performance characteristics. We chose the tolerance value for CoScale in 3 ways, tuned for – *performance* (in Skype and Youtube), *energy efficiency* (in VideoRecord and Facebook) and *balancing* both (in VideoPlay, Angrybirds and MPGame). Correspondingly, we see no frame drops in the case of *performance*, close to optimal energy consumption with *energy efficienct* tolerence value, and the *balanced case* strikes a middle-ground. Please note that, although CoScale algorithm can select the best available frequency combination for different IPs, it performs sub-optimally due to not being able to effectively use all the slack for each frame because of the reasons explained in Section 7.2.1.

Next, we analyze our greedy policies: `MEF, MGF and SDT`. We find that `SDT` performs better than or equal to `MEF` and `MGF` in all cases. While all three focus on the maximum energy consuming component in some form, `SDT` is the only one which considers the time contribution of the component towards the slack consumed.

Finally, let us analyze the energy efficiency and performance of the K&H compensation policy. Overall, we observe that our greedy and K&H policies are within 9% of the optimal; which is 22% and 23% better respectively, than the Android default `OnDemand` governor. Further, in almost all cases, K&H is more energy-efficient than `CoScale` for an *iso-performance* scenario. We note that `CoScale` can be tuned for energy efficiency, by selecting a high *performance degradation tolerance*, in which case it will be very energy-efficient (in some cases within $\sim$1% of Optimal) but at the cost of higher frame drops. For example, in case of `VideoRecord` and `Facebook`, we see that the energy consumption of `CoScale` is close to `Optimal`. However, from the performance perspective, `CoScale` performs much worse than our policies in those applications, as shown in Table 7.2.

[1]For `CoScale`, we set the performance degradation value (for each application) to strike a trade-off between energy-efficiency and performance.

To understand the reason why K&H is better than Greedy, let us consider Figure 7.5, which shows the dynamic nature of these two policies over a duration of 40 frames in `YouTube` app. Particularly, note the VD frequency variations across these policies. We observe that K&H pushes VD (an energy efficient component) to higher frequencies (than SDT) and saves memory (a high-energy consuming component) from increasing its frequency. This is because K&H pushes some low-energy consuming/high-slack providing components to higher frequency, and keeping other components at lower frequency. We wish to stress that, although the energy improvements from K&H seem marginal overall, it outperforms the greedy policies in frame-drop rate (shown in Table 7.2), particularly when there are multiple concurrent flows happening (e.g., `Skype`).

Based on the above analysis, one can clearly conclude that *domain-aware coordinated energy management policies have an edge over the rest of the policies.*

7.5 Conclusions

Minimizing energy consumption in mobile devices is critical to improve battery life. Current DVFS techniques are sub-optimal as they do not take into account – the voltage-frequency states of the multiple IPs and the application characteristics such as available slack time per frame in a coordinated fashion. Towards this, the dissertation presents a coordinated, multi-IP and slack-aware DVFS policies to minimize the overall system energy with minimal performance impact. Experimental evaluations show that our DVFS policies saves on average ∼23% energy across the system, and is within ∼9% of a theoretically-optimal policy. This is one of the first works that presents a slack-based, coordinated DVFS framework for energy management in handheld platforms, an area that is relatively little explored compared to server platforms.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The propensity of tablets and mobile phones in this handheld era raises several interesting challenges. At the application end, these devices are being used for a number of very demanding scenarios (unlike the mobiles of a decade ago), requiring substantial computational resources for real-time interactivity, on both input and output sides, with the external world. On the hardware end, power and limited battery capacities mandate high degrees of energy efficiencies to perform these computational tasks. Meeting these computational needs with the continuing improvements in hardware capabilities is no longer just a matter of throwing high performance and plentiful cores or even accelerators at the problem. Instead, a careful examination and marriage of the hardware with the application and execution characteristics is warranted for extracting the maximum efficiencies. In other words, a co-design of software and hardware is necessary to design energy- and performance-optimal systems, which may not be possible just by optimizing the system in parts. With this philosophy, this dissertation proposes a set of four software and hardware optimizations at different levels of system stack for high performance and energy-efficient handheld platform design.

• First, the dissertation describes the infrastructure that was built to simulate and evaluate handheld platforms. This comprehensive simulation infrastructure is called the **GemDroid**, and is comprised of two primary layers. The first layer provides emulation of Android OS by the Google Android Emulator [7] and allows us to capture system-level interaction between multiple IPs and I/O devices, including OS activities. What the emulator cannot provide is the timing information of different IP activities and therefore, as the second layer, we integrate/build the timing piece using existing simulation platforms or model them analytically as needed for different IPs. The framework is flexible for integrating models of varying complexities for the cores and IPs. Such an infrastructure would open-up possibilities in evaluating and proposing new micro-architectural solutions to improve performance and energy efficiency of handheld systems.

• Second, the dissertation proposes a novel heterogeneous memory controller (HMC) design for SoCs, where one MC is dedicated for latency critical core requests and the second MC is optimized to enhance the bank-level parallelism of the memory requests it serves. These controllers can be further customized by employing different (memory request) scheduling strategies and targeting different optimization metrics. These controllers handle the core and IP requests differently. For IP requests, it maximizes the bank-level parallelism as they are bandwidth constrained requests. For core requests, it minimizes the latency by improving row buffer hits, as they are latency constrained requests. The two memory controllers are designed such that they are still responsible for two distinct (non-intersecting) address ranges, there by avoiding synchronization issues. We conclude that incorporating such an heterogeneous design for memory controllers results in better performance and user experience.

• Third, the dissertation proposes a new paradigm for creating hardware IP chains and virtualizing them to address the throughput and energy inefficiencies of current designs. This paradigm 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. Experimental evaluations with two-application workloads shows substantial energy saving, and improvement in frame processing time compared to just enabling direct IP-to-IP communication. We conclude that IP-chaining and virtualization in handhelds can have a transformational impact on designing such platforms, without having to simply throw more hardware resources at meeting future demands. We could "do more with less".

• Fourth, the dissertation makes a case for the need for recognizing performance slack in the applications when performing such DVFS for the different components. Towards that, we propose a *frame-aware* coordinated multi-component DVFS framework that leverage frame-level slack and utilization information of components predicted at frame boundaries. In this context, we propose two mechanisms, called *Greedy policy* and *Kaldor-Hicks* [8] *compensation policy* (K&H). The *Greedy policy* in turn examines three different options: Maximum Energy First (MEF), Most to Gain First (MGF) and Slope ($\triangle$E/$\triangle$T) First (SDF). The K&H policy can even choose energy inefficient options for one component, while offsetting this with higher energy savings in another. Experimental evaluations show that such schemes improves energy efficiency by approaching ideal energy usage scenario, as all components use minimal amount of energy with the constraint that they meet the deadline together. We conclude that such co-ordinated domain aware DVFS schemes are a promising way to improve energy efficiency in handheld platforms.

8.2 Future Research Directions

The next generation platforms are heading towards wearables and Internet-Of-Things (IoT) where all devices are always "connected" to each other. Each device is specialized for it's own functionality, with energy efficiency built-in each one of them. However, they have been built agnostic to a Li-ion battery's physical and chemical characeristics, as well as agnostic to other co-running applications/components in the system. Due to these factors, possible exciting future research includes 1) utilizing battery characteristics for energy savings, 2) scheduling among accelerator requests and memory requests for high performance and more energy savings.

8.2.1 Utilizing Battery Characteristics for Energy Savings

Towards the end of battery run-time, around last 15% of battery, peaks in power trace start to play a major role. A bigger peak at that time can drain the voltage much quicker, and hit the cut-off voltage quicker. Otherwise, having a peak or not does not seem to make any difference. The run-time improvements that we obtain are only because of voltage not reaching cut-off voltage quickly, thus we are able to use more capacity from the battery.

In ideal case, that is, when we compare a real application's power trace which have big peaks (and where no DVFS is applied) to a constant power draw of their average power, we see average improvement of less than 3-4%. For some applications, particularly if the applications are not drawing more than 1.5 C current, only limited benefits can be obtained. For a 2600mAH battery, 1.5 C equals to around 15W.

Our smart-phones (with 2600MAh) have to be stressed a lot to touch even 8 Watts once is a while. This is less than 1C actually. For example, when we run a graphics intensive game, along with artificially forcing all cores to always stay at maximum frequency whenever they turn on. Plus, our phones forcibly throttle back if they reach 8W, mainly because of

thermal reasons. The maximum in all use-cases possible with multiple applications, we touched 10 to 11 W once in a while, after which we could not measure power because our power monitor shuts down.

Potential Benefits: A simple experiment was conducted, where we plotted the difference between amplitude of voltage drops in a power profile with peaks vs a constant draw. As explained above, peaks reduce voltage considerably towards the end 70% of state of charge, when the battery reaches the cut off 3.2V and stops. When we run workloads such that we execute 5 minutes of an application, and a standby period, we notice that the time increases by some 5 to 10%. We noticed that this is because, even 1% availability of extra capacity provided by normalizing the peaks gives a lot more stand-by time, because standby reduces voltage very slowly. Thus even 15 to 20% of increase in standby time in possible if we start from 20% state of charge. When we run workloads from 100% SOC, we see those big numbers averaging out for the whole duration, and end up with 4% even for ideal experiments.

Thus, we expect that if handhelds operate at high C-rates or their battery model is such that it provides them with higher voltage drops when peaks are present in their power, then there is substantial energy benefit by applying peak reduction technique. Peaks with 10.5W and valleys with 2.5W drains the battery a lot compared to a constant draw for a 2000 mAh battery.

Using "unaffected battery with peaks" to our advantage:
Utilizing the different behavior of the battery at different SoCs and proposing techniques that suit the regions is also an interesting problem to explore. Because we see that peaks do not affect the voltage much in the initial 95% to 25% of the battery, maybe we can ride the wave and try to gain benefits from it. Currently, at times, CPU sleeps for less than 1 millisecond, and wakes up. It performs some job for less than 1 ms, and goes back to sleep. Instead of intermittently waking up and getting back to sleep in C3 power state, cores can do all jobs at once, and sleep at deeper power stats till the next frame.

To implement this proposal, the handheld platform needs to move into deeper power states like C6 - where the whole core and platform is power gated, and they come back up only when the next set of frames are to be processed. For more benefits, the core should decode multiple frames (say 5 frames) at once, and go to deep sleep state C6, and wake up only after 5 frames are shown.

8.2.2 Effective Scheduling

As the next generation platforms are fitted with more components likes more cores with heterogeneous core, heterogeneous IPs with more functionality, there is a pressing need to manage them more efficiently. Hence, an efficient way of scheduling tasks to cores and requests to IPs are necessary for better performance and energy.

Accelerator Request Scheduling

Scheduling at the System agents (a.k.a. the north bridge in SoCs) is a critical task, and can determine the performance in any mobile system. Such scheduling at the IP-level can be achieved through two techniques:

• a non-work-conserving scheduler: either predicting the presence of concurrent flows which access the same IPs and wait till the request from the shortest flow is served before serving the longer flows, or,

• preemption within a frame: by preempting a long-running request from an IP and serving the shorter flow-first (or earliest deadline first) before getting back to the preempted request. Preemption of requests in an IP hardware leads to a work-conserving scheduler and such a scheduler is needed for optimality.

Memory Scheduling

While this memory bottleneck problem can be tackled through provisioning more hardware such as adding more memory/buffer as has been studied in the VIP (virtualizing IP chains) part of this dissertation, this may not be a viable approach since not only the amount of extra

hardware grows with number of flows, but also the power consumption grows accordingly. Instead, smarter solutions for addressing memory contention can be investigated.

As one option, one could explore intelligent memory scheduling of different flows. Just meeting the timing requirements of individual frames or IPs is *not* sufficient to improve the performance of multiple flows. Instead, an effective memory scheduler should consider three intersecting parameters: (i) per-frame QoS requirements; (ii) awareness of existing live flows in the system to schedule the required IPs in a coordinated fashion; and (iii) any flow dependency in terms of shared IPs between flows. None of the existing scheduling techniques for handhelds consider these three issues in *unison*.

Rate-proportional scheduling has been used extensively for scheduling periodic jobs with timing constraints. Hence, borrowing these concepts and employing a *rate-proportional scheduling* of memory requests at the memory controller can potentially improve the performance of handheld devices.

Chapter 9

Publications

9.1 Five Significant Publications

[**ISCA 2015**]

Nachiappan Chidambaram N., Haibo Zhang, Jihyun Ryoo, Niranjan Soundararajan, Mahmut Kandemir, Anand Sivasubramaniam, Chita R. Das

VIP: Virtualizing IP Chains on Handheld Platforms,

In International Symposium on Computer Architecture (ISCA), 2015.

[**HPCA 2014**]

Nachiappan Chidambaram N., Praveen Yedlapalli, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut Kandemir, Ravi Iyer, Chita R. Das

Domain Knowledge Based Energy Management in Handhelds,

In High Performance Computer Architecture (HPCA), 2014.

[**SIGMETRICS 2013**]

Nachiappan Chidambaram N., Praveen Yedlapalli, Niranjan Soundararajan, Mahmut Kandemir, Anand Sivasubramaniam, Chita R. Das

GemDroid: A Framework To Evaluate Mobile Platforms,

In International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), 2014.

[**MICRO 2014**]

Praveen Yedlapalli, **Nachiappan Chidambaram N.**, Niranjan Soundararajan, Mahmut Kandemir, Anand Sivasubramaniam, Chita R. Das

Short-Circuiting Memory Traffic in Handheld Platforms,

In Microarchitecture (MICRO), 2014.

[**ASPLOS 2013**]

Adwait Jog, Onur Kayiran, **Nachiappan Chidambaram N**, Asit Mishra, Mahmut Kandemir, Onur Mutlu, Ravi Iyer, Chita Das

OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU performance,

In Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2013.

9.2 Other Significant Publications

[**MICRO 2014**] Onur Kayiran, **Nachiappan Chidambaram N.**, Adwait Jog, Rachata Ausavarungnirun, Mahmut Kandemir, Gabriel Loh, Onur Mutlu, Chita Das

Managing Concurrency in Heterogeneous CPU-GPU Architectures,

In Microarchitecture (MICRO), 2014.

[**PACT 2012**] **Nachiappan Chidambaram N.**, Asit K. Mishra, Mahmut Kandemir, Anand Sivasubramaniam, Onur Mutlu, Chita Das

Application-aware Prefetch Prioritization in On-chip Networks,

In Parallel Architectures and Compilation Techniques (PACT), 2012.

Bibliography

[1] Gartner (2013) "Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013," .

[2] Cisco (2013) "Cisco Visual Networking Index: Forecast and Methodology, 2012:2017," .

[3] for Semiconductors, T. I. T. R. (2008), "2008 Update," .
URL `http://www.itrs.net/`

[4] Henning, J. L. (2006) "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, **34**(4).

[5] Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood (2011) "The Gem5 Simulator," *SIGARCH Computer Architecture News*.

[6] del Barrio, V., C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa (2006) "ATTILA: a cycle-level execution-driven simulator for modern GPU architectures," in *ISPASS*.

[7] Google (2014) "Android SDK - Emulator," .

[8] Newman, P. (2004) in *The New Palgrave Dictionary Of Economic and The Law*.

[9] Steve Scheirey, D. S. (2013) *Sensor Fusion, Sensor Hubs and the Future of Smartphone Intelligence*, *Tech. rep.*, ARM Techreport.

[10] Engwell, J. (2013) *The High Resolution Future Retina Displays and Design*, *Tech. rep.*, Blurgroup.

[11] Engwell, J. Y. C. *GPU Technology Trends and Future Requirements*, *Tech. rep.*, Nvidia Corp.

[12] Google (2014), "Android Developers," `http://developer.android.com/`,.

[13] ——— (2014), "OpenGL ES - Android Developers," `http://developer.android.com/guide/topics/graphics/opengl.html`,.

[14] ——— (2014), "RenderScript - Android Developers," `http://developer.android.com/guide/topics/renderscript/compute.html`.

[15] ——— (2014), "Google/Grafika," `https://github.com/google/grafika`.

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

[17] Akesson, B., K. Goossens, and M. Ringhofer (2007) "Predator: A Predictable SDRAM Memory Controller," in *CODES+ISSS*.

[18] Lin, Y.-J., C.-L. Yang, T.-J. Lin, J.-W. Huang, and N. Chang (2010) "Hierarchical Memory Scheduling for Multimedia MPSoCs," in *ICCAD*.

[19] Jeong, M. K., 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*.

[20] Yedlapalli, P., N. C. Nachiappan, N. Soundararajan, M. Kandemir, A. Sivasubramaniam, and C. R. Das (2014) "Short-Circuiting Memory Traffic in Handheld Platforms," *MICRO*.

[21] Lim, K., D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch (2013) "Thin servers with smart pipes: designing SoC accelerators for memcached," in *International Symposium on Computer Architecture (ISCA)*.

[22] Fenney, S. (2003) "Texture Compression Using Low-frequency Signal Modulation," in *HWWS*.

[23] Han, K., A. Min, N. Jeganathan, and P. Diefenbaugh (2013) "A hybrid display frame buffer architecture for energy efficient display subsystems," in *ISLPED*.

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

[25] Patel, K., E. Macii, and M. Poncino (2005) "Frame Buffer Energy Optimization by Pixel Prediction," in *ICCD*.

[26] Shim, H., N. Chang, and M. Pedram (2004) "A Compressed Frame Buffer to Reduce Display Power Consumption in Mobile Systems," in *ASP-DAC*.

[27] Siqueira, H. M., I. S. Silva, M. E. Kreutz, and E. F. Correa (2011) "DDR SDRAM Memory Controller for Digital TV Decoders," in *SBESC*.

[28] Hauswald, J., T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge (2014) "A hybrid approach to offloading mobile image classification,"

in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*.

[29] Gao, C., A. Gutierrez, R. Dreslinski, T. Mudge, K. Flautner, and G. Blake (2014) "A study of Thread Level Parallelism on mobile devices," in *ISPASS*.

[30] Wang, Y., B. Krishnamachari, Q. Zhao, and M. Annavaram (2010) "Markov-optimal sensing policy for user state estimation in mobile devices," in *Information Processing in Sensor Networks (IPSN)*, ACM.

[31] Goulding-Hotta, N., J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor (2011) "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future," *IEEE Micro*, **31**(2).

[32] Zhu, Y. and V. J. Reddi (2013) "High-performance and Energy-efficient Mobile Web Browsing on Big/Little Systems," in *High Performance Computer Architecture (HPCA)*.

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

[34] Balasubramanian, N., A. Balasubramanian, and A. Venkataramani (2009) "Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications," in *IMC*.

[35] Carroll, A. and G. Heiser (2010) "An Analysis of Power Consumption in a Smartphone," in *USENIX ATC*.

[36] Chung, Y.-F., C.-Y. Lin, and C.-T. King (2011) "ANEPROF: Energy Profiling for Android Java Virtual Machine and Applications," in *ICPADS*.

[37] Deng, Q., D. Meisner, A. Bhattacharjee, T. Wenisch, and R. Bianchini (2012) "CoScale: Coordinating CPU and Memory System DVFS in Server Systems," in *MICRO*.

[38] Deng, Q., D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini (2011) "MemScale: Active Low-power Modes for Main Memory," in *ASPLOS*.

[39] Deng, Q., D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini (2012) "MultiScale: Memory System DVFS with Multiple Memory Controllers," in *ISLPED*.

[40] Lee, B., E. Nurvitadhi, R. Dixit, C. Yu, and M. Kim (2005) "Dynamic voltage scaling techniques for power efficient video decoding," *Journal of Systems Architecture*, **51**(1011).

[41] Xiao, Y., R. S. Kalyanaraman, and A. Yla-Jaaski (2008) "Energy Consumption of Mobile YouTube: Quantitative Measurement and Analysis," in *NGMAST*.

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

[43] Pandiyan, D., S.-Y. Lee, and C.-J. Wu (2013) "Performance, Energy Characterizations and Architectural Implications of An Emerging Mobile Platform Benchmark Suite : MobileBench," in *IISWC*.

[44] Chidambaram Nachiappan, N., P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. Kandemir, and C. R. Das (2014) "GemDroid: A Framework to Evaluate Mobile Platforms," in *SIGMETRICS*.

[45] Huang, Y., Z. Zha, M. Chen, and L. Zhang (2014) "Moby: A mobile benchmark suite for architectural simulators," in *ISPASS*.

[46] Lee, C., E. Kim, and H. Kim *The AM-Bench: An Android Multimedia Benchmark Suite*, *Tech. rep.*, Georgia Institute of Technology.

[47] Pathak, A., Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang (2011) "Fine-grained Power Modeling for Smartphones Using System Call Tracing," in *EuroSys*.

[48] Zhu, Y. and V. J. Reddi (2013) "High-performance and Energy-efficient Mobile Web Browsing on Big/Little Systems," in *HPCA*.

[49] Janapa Reddi, V., B. C. Lee, T. Chilimbi, and K. Vaid (2010) "Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency," in *ISCA*.

[50] Falaki, H., D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin (2010) "A First Look at Traffic on Smartphones," in *IMC*.

[51] Sunwoo, D., W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. Paver (2013) "A Structured Approach to the Simulation, Analysis and Characterization of Smartphone Applications," in *IISWC*.

[52] Ubal, R., B. Jang, P. Mistry, D. Schaa, and D. Kaeli (2012) "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in *PACT*.

[53] Pathak, A., Y. C. Hu, and M. Zhang (2012) "Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof," in *EuroSys*.

[54] Chen, L., W. Chen, B. Wang, X. Zhang, H. Chen, and D. Yang (2011) "System-level simulation methodology and platform for mobile cellular systems," *IEEE Communications Magazine*.

[55] Guo, P., "Simulation and Testing of Mobile Computing Platforms using Fujaba," .

[56] Diniz, B., D. O. G. Neto, W. M. Jr., and R. Bianchini (2007) "Limiting the power consumption of main memory." in *ISCA*.

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

[58] Sungho, P., A. Ahmed, M. Kevin, C. Aarti, C. Matthew, C. Nandhini, D. Michael, and N. Vijaykrishnan (2012) "System-On-Chip for Biologically Inspired Vision Applications," *IPSJ transactions on system LSI design methodology*.

[59] Irick, K. and N. Chandramoorthy (2014) "Achieving High-Performance Video Analytics with Lightweight Cores and a Sea of Hardware Accelerators," in *ISVLSI*.

[60] Ozer, E., N. Chong, and K. Flautner (2010) *Processor and System-on-Chip Simulation*, chap. IP Modeling and Verification, Springer.

[61] Saleh, R., S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov (2006) "System-on-Chip: Reuse and Integration," *Proceedings of the IEEE Volume 96 Issue 6*.

[62] Zhu, Y. and V. J. Reddi (2014) "WebCore: Architectural Support for Mobileweb Browsing," in *International Symposium on Computer Architecuture*, ISCA.

[63] Zhu, Y., A. Srikanth, J. Leng, and V. Reddi (2014) "Exploiting Webpage Characteristics for Energy-Efficient Mobile Web Browsing," *Computer Architecture Letters*, **13**(1).

[64] "Yuhao Zhu, V. J. R., Matthew Halpern (2015) *"Event-based Scheduling for Energy-Efficient QoS (eQoS) in Interactive Mobile Web Applications*.

[65] Bhatt, V., 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*.

[66] Choi, K., K. Dantu, W.-C. Cheng, and M. Pedram (2002) "Frame-based Dynamic Voltage and Frequency Scaling for a MPEG Decoder," in *International Conference on Computer-aided Design*, ICCAD.

[67] Mohapatra, S., R. Cornea, N. Dutt, A. Nicolau, and N. Venkatasubramanian (2003) "Integrated Power Management for Video Streaming to Mobile Handheld Devices," in *International Conference on Multimedia*.

[68] Choi, K., R. Soma, and M. Pedram (2004) "Off-chip Latency-driven Dynamic Voltage and Frequency Scaling for an MPEG Decoding," in *Design Automation Conference (DAC)*.

[69] Popek, G. J. and R. P. Goldberg (1974) "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, **17**(7).

[70] Dally, W. and C. Seitz (1987) "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*.

[71] Dally, W. (1992) "Virtual-channel flow control," *IEEE Transactions on Parallel and Distributed Systems*, **3**(2).

[72] Tanasic, I., I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero (2014) "Enabling preemptive multiprogramming on GPUs," in *International Symposium on Computer Architecture (ISCA)*.

[73] Nvidia, "CUDA Multi Process Service Overview," `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[74] ———, "NVidia Shared Virtual GPU (vGPU) Technology," .
URL `http://www.nvidia.com/object/virtual-gpus.html`

[75] Menychtas, K., K. Shen, and M. L. Scott (2014) "Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators," in *ASPLOS*.

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

[77] Bienia, C. (2011) *Benchmarking Modern Multiprocessors*, Ph.D. thesis, Princeton University.

[78] Das, R., O. Mutlu, T. Moscibroda, and C. R. Das (2010) "Aergia: Exploiting Packet Latency Slack in On-chip Networks," in *ISCA*.

[79] Kim, Y., M. Papamichael, O. Mutlu, and M. Harchol-Balter (2010) "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*.

[80] Xu, K. (2008), "Nova : H.264/AVC Baseline Decoder," OpenCores, rTL verified.

[81] Inc., M. S., "Monsoon Power Monitor," `http://msoon.com/LabEquipment/PowerMonitor/downloads/PowerMonitor_ManualVer1.4.pdf`.

[82] Bosomworth, D., "Mobile Marketing Statistics 2014," `http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics`.

[83] Kim, Y., D. Han, O. Mutlu, and M. Harchol-Balter (2010) "ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*.

[84] Keltcher, C. N., K. J. McGrath, A. Ahmed, and P. Conway (2003) "The AMD Opteron processor for multiprocessor servers," *IEEE Micro*, **23**(2), pp. 66–76.

[85] Conway, P., N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes (2010) "Cache hierarchy and memory subsystem of the AMD Opteron processor," *IEEE Micro*, **30**(2), pp. 16–29.

[86] Naveh, A., D. Rajwan, A. Ananthakrishnan, and E. Weissmann (2011) "Power management architecture of the 2nd generation Intel ® Core microarchitecture, formerly codenamed Sandy Bridge," .

[87] Yuffe, M., E. Knoll, M. Mehalel, J. Shor, and T. Kurts (2011) "A fully integrated multi-CPU, GPU and memory controller 32nm processor," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, IEEE.

[88] Lim, A. W. and M. S. Lam (1997) "Maximizing parallelism and minimizing synchronization with affine transforms," in *Principles of programming languages (POPL)*, ACM.

[89] Song, Y. and Z. Li (1999) "New tiling techniques to improve cache temporal locality," in *ACM SIGPLAN Notices*, vol. 34, ACM.

[90] Zhang, Y., W. Ding, M. Kandemir, J. Liu, and O. Jang (2011) "A data layout optimization framework for NUCA-based multicores," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM.

[91] Bojnordi, M. N. and E. Ipek (2012) "PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards," in *International Symposium on Computer Architecture (ISCA)*.

[92] Shivakumar, P. and N. P. Jouppi (2001) *Cacti 3.0: An integrated cache timing, power, and area model*, *Tech. rep.*, Technical Report 2001/2, Compaq Computer Corporation.

[93] Sha, T., M. M. K. Martin, and A. Roth (2005) "Scalable Store-Load Forwarding via Store Queue Index Prediction," in *MICRO*.

[94] Loh, G. H., R. Sami, and D. H. Friendly (2002) "Memory Bypassing: Not Worth the Effort," in *WDDD*.

[95] Rixner, S., W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens (2000) "Memory Access Scheduling," in *International Symposium on Computer Architecture (ISCA)*, ISCA.

[96] Samsung (2014), "Samsung Galaxy S5," `http://www.samsung.com/global/microsite/galaxys5/`.

[97] Apple, "Apple Iphone 5S," `https://www.apple.com/iphone/`.

[98] Schwarz, H., D. Marpe, and T. Wiegand (2007) "Overview of the scalable video coding extension of the H. 264/AVC standard," *IEEE Transactions on Circuits and Systems for Video Technology*, **17**(9).

[99] Google, "Android HAL," https://source.android.com/devices/index.html.

[100] Report, S. T. (2012), "WQXGA Solution with Exynos Dual," http://www.samsung.com/global/business/semiconductor/minisite/Exynos/data/Enjoy_the_Ultimate_WQXGA_Solution_with_Exynos_5_Dual_WP.pdf.

[101] Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel (2002) "Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems," in *CODES*.

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

[103] cubei (2014), "cubei/FlappyCow," https://github.com/cubei/FlappyCow.

[104] Linux, "ftrace - Function Tracer," https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[105] Chen, J.-J. and C.-F. Kuo (2007) "Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms," in *Real-Time Computing Systems and Applications*, RTCSA '07.

[106] Sanz, C., J. I. Gómez, C. Tenllado, M. Prieto, and F. Catthoor (2013) "System-level Memory Management Based on Statistical Variability Compensation for Frame-based Applications," *ACM Transactions on Embedded Computer Systems*, **13**(1s).

[107] Lee, B., E. Nurvitadhi, R. Dixit, C. Yu, and M. Kim (2005) "Dynamic voltage scaling techniques for power efficient video decoding," *Journal of Systems Architecture*, **51**(1011), pp. 633 – 652.
URL http://www.sciencedirect.com/science/article/pii/S1383762105000330

[108] Cao, Z., B. Foo, L. He, and M. van der Schaar (2008) "Optimality and improvement of dynamic voltage scaling algorithms for multimedia applications," in *Design Automation Conference (DAC)*.

[109] Choi, K., K. Dantu, W.-C. Cheng, and M. Pedram (2002) "Frame-based Dynamic Voltage and Frequency Scaling for a MPEG Decoder," in *International Conference on Computer-Aided Design (ICCAD)*.

[110] Isci, C., A. Buyuktosunoglu, and M. Martonosi (2005) "Long-term workload phases: Duration Predictions and Applications to DVFS," *IEEE Micro*.

[111] Aydin, H., R. Melhem, D. Mosse, and P. Mejia-Alvarez (2004) "Power-aware scheduling for periodic real-time tasks," *IEEE Transactions on Computers*, **53**.

[112] Chen, G., K. Huang, J. Huang, C. Buckl, and A. Knoll (2013) "Effective Online Power Management with Adaptive Interplay of DVS and DPM for Embedded Real-Time System," in *Euromicro Conference on Digital System Design (DSD)*.

[113] Devadas, V. and H. Aydin (2012) "On the Interplay of Voltage/Frequency Scaling and Device Power Management for Frame-Based Real-Time Embedded Applications," *IEEE Transactions on Computers*, **61**.

[114] Mishra, A. K., R. Das, S. Eachempati, R. Iyer, N. Vijaykrishnan, and C. R. Das (2009) "A Case for Dynamic Frequency Tuning in On-chip Networks," in *MICRO*.

[115] Tyler, J. and W. Verdusko (2012) in *XDA Developers' Android Hacker's Toolkit: The Complete Guide to Rooting and theming*.

[116] Chen, X., Z. Xu, H. Kim, P. Gratz, J. Hu, M. Kishinevsky, and U. Ogras (2012) "In-network Monitoring and Control Policy for DVFS of CMP Networks-on-Chip and Last Level Caches," in *International Symposium on Networks on Chip (NoCS)*.

Vita

Nachiappan Chidambaram Nachiappan

Nachiappan Chidambaram N. was born in Tamil Nadu, India, in 1989. Nachi holds a bachelor's degree in Computer Science and Engineering from the Anna University, Tamil Nadu. He was a part of WARFT (Waran Research Foundation) as research trainee during 2008-2010. Nachi joined Penn State as a PhD student in 2010, and worked on SoC architectures under the supervisions of Prof. Mahmut Kandemir and Prof. Chita R. Das. He served as a teaching assistant for C++ and Systems Programming courses. His research has been published in top-tier computer architecture conferences such as ISCA, HPCA, MICRO, ASPLOS, SIGMETRICS, and PACT. Nachi worked as an intern at Intel Server Performance group in 2013, and Intel Software Services group in 2015, and was rated "Outsanding" for his contributions.