RICE UNIVERSITY

Virtual Ring Buffer for Camera Application Concurrency

by

Jose Eduardo Reyes, Jr.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE Master of Science

APPROVED, THESIS COMMITTEE:

Lin Zhong, Chair
Associate Professor of Electrical and Computer Engineering and Computer Science

Joseph R. Cavallaro
Professor of Electrical and Computer Engineering and Computer Science

Ashok Veeraraghavan
Assistant Professor of Electrical and Computer Engineering and Computer Science

Houston, Texas
January, 2015
ABSTRACT

Virtual Ring Buffer for Camera Application Concurrency

by

Jose Eduardo Reyes, Jr.

Smartphones with integrated cameras have inspired a growing number of real-time, computer vision applications. Existing camera software architectures, however, do not support concurrency: only one application accesses the image stream at any time. A naive solution that makes a copy of every image for every application is inherently inefficient. Towards a computation- and power-efficient solution, this work presents a driver-level architecture, wherein a single, copy-on-write, shared-memory ring buffer delivers images to all applications via virtual interfaces. The architecture guarantees application isolation, minimizes data redundancy, and provides an illusion to applications that they are the sole consumers of the image stream. This work implements the architecture in Android 4.3.1 and characterizes its performance on a modern, multi-core smartphone. Measurements show the architecture increases CPU utilization at half the rate of the naive solution and reduces power consumption by several hundred milliwatts.
# Contents

List of Illustrations v
List of Tables viii

1 Introduction 1

2 Background 4
   2.1 Camera Driver/Service Architectures .......................... 4
   2.2 Ring Buffer Concurrency ........................................ 7

3 Design 9
   3.1 Design Goals ..................................................... 9
   3.2 Hardware Assumptions .......................................... 9
   3.3 Virtual Ring Buffer Design .................................... 10

4 Features 13
   4.1 Lock-free Memory Sharing .................................... 13
   4.2 Interface Virtualization ....................................... 14
   4.3 Frame Rate Isolation .......................................... 15
   4.4 Data Integrity and Redundancy ............................... 16

5 Implementation 19
   5.1 Zero-Copy JNI Boundary .............................. 19
   5.2 Copy on Write ashmem ................................. 21
   5.3 Thread-Safe Android Camera Service ..................... 22
Illustrations

2.1 Model of the driver architecture used by the UVC driver in Linux and the Camera Service in Android. A single application controls the flow of images through a ring buffer in memory. The driver tracks the state of individual bins via two queues.

3.1 (Left): naive stream replication. Every application controls a private ring in memory. The driver copies images to meet demand. (Right): virtual ring buffer. Every application interacts with a virtual ring. The driver maps virtual bins to a shared ring in memory on demand. Shared memory is copy on write. Superscripts indicate virtual indices used by applications. Normal numbers indicate memory mappings. Matching colors represent the same image.

4.1 Example sequence of bin utilization for two applications with different frame rates. By coupling memory map lifetimes to enqueue operations, App A can reuse its first virtual bin at time step 2 without causing data loss for App B.

4.2 Illustration of pixel layout in memory. 4 KB pages store a row of pixels sequentially and continue at the next row. The alternating colors highlight the page boundaries of a typical image.
4.3 A common scenario in image processing. A drawn bounding box
incurs copies of just the pages that contain overwritten pixels.
Without reordering pixels, the virtual ring buffer driver generates the
minimum number of page copies. ......................................... 18

5.1 Image delivery across the JNI boundary for (Left) stock Android and
(Right) the modified implementation. Using a Java Byte Buffer,
instead of a Java Byte Array, images cross the boundary multiple
times without a single copy. ................................................. 20

6.1 CPU utilization to deliver 176x144 pixel thumbnail images (.03 MP)
and 800x480 pixel preview images (.40 MP) at 30 fps with an
exposure time of 9 ms. The CPU supports twice as many
applications using the virtual ring buffer over naive stream replication. 31

6.2 CPU utilization to deliver 176x144 pixel thumbnail images (.03 MP)
and 800x480 pixel preview images (.40 MP) at 15 fps with an
exposure time of 72 ms. The CPU supports twice as many
applications using the virtual ring buffer architecture over stream
replication. ................................................................. 32

6.3 Power consumption to deliver 800x480 pixel preview images (.40 MP)
at 30 fps with an exposure time of 9 ms. The virtual ring buffer
architecture reduces power consumption by more than 500 mW. ... 33

6.4 CPU utilization for applications that read (R) or overwrite (W)
pixels in the first column of 800x480 pixel images delivered at 30 fps
with an exposure time of 9 ms. The virtual ring buffer uses
significantly less CPU time than naive stream replication. ............ 34
6.5 CPU utilization for applications that either read (R) or write (W) pixels in the first row of 800x480 pixel images delivered at 30 fps with an exposure time of 9 ms. The virtual ring buffer uses significantly less CPU time than naive stream replication.

6.6 CPU utilization for applications that calculate the first level of a Gaussian pyramid for 800x480 pixel images delivered at 30 fps with an exposure time of 9 ms. Results are stored in separate memory (R) or overwrite the image in place (W).

6.7 Power consumption for applications that calculate one level of a Gaussian pyramid for 800x480 pixel images (.40 MP) delivered at 30 fps with an exposure time of 9 ms. The virtual ring buffer shows a 500 mW improvement over stream replication with two concurrent applications. The gap widens as concurrency increases.
Tables

5.1 Thermal management unit states and actions ................................ 24
5.2 CPU frequency governor policies ........................................... 25
5.3 Memory bus frequency governor levels .................................... 26

6.1 Power measurements for worst-case group of applications. Results show no advantage to either architecture because the applications have few sharing opportunities. .................................................. 39
6.2 Breakdown of image sharing for the group of worst-case applications. Share1 indicates the percentage of images that were distributed to one application only. Share4 indicates the percentage of images that were distributed to all four applications. Applications from Chapters 6.3 and 6.4 shared maximally at all times. ......................... 40
Chapter 1

Introduction

Despite the growing variety of mobile camera applications, only a single application can access the camera at any given time. Mobile devices with head-mounted cameras and multi-core smartphones with embedded camera hardware have produced a new class of mobile, real-time camera applications [15]. Chapter 2 describes the camera software architectures that currently support these applications. These architectures use a ring buffer to capture and deliver images from the camera. Concurrency is challenging for them because they provide applications with total control over this ring buffer. Support for mobile camera application concurrency, however, should allow application developers to create specialized applications and users to take full advantage of different real-time computer vision algorithms.

Naive stream replication, the only known concurrency solution, is power-hungry and memory-inefficient. On a general purpose chip, the energy consumed to read and write pixels from off-chip DRAM during image processing dominates that of computing new data [8]. The reason data fetches cost more energy than computations is that new values are often computed from weighted sums of neighboring pixels in a regular, multi-dimensional grid or stencil [18]. The localized multiply-and-add operations are cheap; the resulting volume of data fetches is not. By creating image copies, stream replication allows energy consumption to scale with the number of camera applications. Consequently, the Virtual Webcam applications [5, 11] that implement stream replication only exist for non-mobile operating systems, whose
systems have more than enough power to replicate hundreds to thousands of kilobytes of data per image in the stream. On mobile devices with small power budgets, stream replication is not a scalable concurrency solution.

This work designs and characterizes a concurrency architecture that achieves energy and memory efficiency by minimizing data redundancy across applications. The idea stems from the observation that concurrent camera applications consume the same data at the same time and that ASICs for computer vision, computational photography, and image processing achieve many of their efficiency gains by eliminating redundant data transfers [16]; accordingly the virtual ring buffer architecture presented in this work identifies all applications that request the same image and creates shared memory mappings to the original image in memory.

The design of the virtual ring buffer architecture caters to concurrency-unaware camera applications. The architecture eliminates all contention for shared memory and exposes an interface identical to that of a non-concurrent camera driver. Chapter 2 outlines the core interface and behavior of the USB Video Class driver for Linux [14] and the Camera Service for Android [7]. They do not extend easily to support concurrent applications. The virtual ring buffer architecture adds a layer of abstraction to virtualize the state of the underlying ring buffer for each application. To eliminate contention for shared memory, the architecture creates and destroys memory mappings every time an application consumes an image. Consequently, each application interacts with the camera as if it is the sole consumer of the image stream.

The creation and destruction of memory maps for every shared image adds zero overhead to image delivery, at least in Android, because the Android Camera Service already requires it. The Camera Service is split between a camera-side Server process and an application-side Client process. Both rely on the Android Binder IPC
mechanism to communicate [17]. The Binder, which provides all interprocess communication in Android, passes file descriptors from one process to the other. The process that receives this file descriptor will create a memory mapping, access the data produced by the other process, and delete the mapping when done. As a result, in Android, the virtual ring buffer architecture saves hundreds of milliWatts and increases CPU utilization per additional application at half the rate of naive stream replication.

The rest of this thesis is organized as follows. Chapter 2 describes a model of the camera driver architecture Linux and Android use. Chapter 3 outlines the goals of this work and overviews the design of the virtual ring buffer architecture. Chapter 4 elaborates on the components of the architecture. Chapter 5 presents the implementation of the virtual ring buffer and stream replication architectures in Android 4.3.1 for a multi-core Galaxy S2 smartphone. Chapter 6 presents and discusses results for CPU utilization and power consumption on primitive and realistic computer vision applications that use the OpenCV library. Chapter 7 concludes the thesis and discusses future extensions to the work.
Chapter 2

Background

This chapter describes a generic camera driver architecture that Android and Linux use, reviews the way other domains use ring buffers to support concurrent users, and ends with a brief discussion on the relation of this work to both.

2.1 Camera Driver/Service Architectures

A study of the USB Video Class (UVC) driver for Linux and the Camera Service for Android 4.3.1 provides the necessary information to derive a generic model for camera architectures in use today. The UVC driver conforms to the Video for Linux 2 (V4L2) specifications and supports a broad range of compliant USB video and web cameras. The Camera Service is effectively a user space driver; it lacks kernel-mode privileges and operates the camera on behalf of the user application. The UVC driver and Camera Service represent the gamut of camera drivers: desktop and mobile, USB and embedded, kernel-mode and user-mode. Their architectures, however, are nearly identical.

As illustrated in Figure 2.1, the UVC driver and Camera Service share the following interface and behaviors, which the virtual ring buffer architecture complies with:

- **Full control of private ring buffer:** applications consume the image stream via a single-consumer, single-producer ring buffer and two operations: enqueue
Figure 2.1: Model of the driver architecture used by the UVC driver in Linux and the Camera Service in Android. A single application controls the flow of images through a ring buffer in memory. The driver tracks the state of individual bins via two queues.

and dequeue. Using enqueue and dequeue, the application controls the frame rate at which images travel through the ring.

- **Bin enqueue**: the application calls enqueue to notify the camera driver to write new data to a specific bin. Enqueue simultaneously signals that an application finished using the image data in a particular bin and requests data to replace it. The driver tracks the enqueued bin by adding it to a "Released" queue. The driver drops images when the Released queue is empty. As soon as a new image occupies a bin, the camera driver pops the bin reference and adds it to the end of the "Ready" queue.

- **Bin dequeue**: the application calls dequeue to acquire a newly occupied bin from the Ready queue. The driver blocks if the Ready queue is empty. Once
dequeued, a bin is neither ready nor released; it is acquired, or in use, by the application. In Android, where interrupts deliver images to applications, dequeue is implicit. Bins change state directly from released to acquired.

- **Application-configured ring size**: the application configures the number of bins the driver should allocate for the ring buffer. As long as sufficient memory is available to it, the driver allocates exactly the requested number of bins in its own address space.

- **Memory-mapped bins**: after allocating memory for the ring, the driver creates a memory map from the application’s address space to the ring’s bins.

- **Bin ordering**: the application dequeues bins from the Ready queue in the same order it enqueues them to the Released queue. If the application enqueues bins faster than the camera can produce images, the driver writes images to bins in the order the bins appear on the Released queue.

- **Variable frame rate**: the camera hardware defines the maximum frame rate. The actual frame rate varies with changes in ambient lighting that cause changes in exposure time and with the rate at which the application enqueues bins.

- **Camera lock**: the driver maintains an exclusive lock that an application acquires to access the camera. As long as one application holds the lock, the driver rejects the others. A lock precludes concurrency; thus it is not included as part of the interface the virtual ring buffer adheres to.
2.2 Ring Buffer Concurrency

This review of the Linux Trace subsystem, the Linux Circular Buffers API, and Xen I/O Rings highlights the strategies that ring buffer-based architectures use to support concurrency for concurrency-aware users.

- **Linux Trace [6]:** The Trace subsystem logs program execution details to a multi-consumer, multi-producer ring buffer with tight performance specifications. Logging takes priority, so producers are concurrent and consumers are made sequential by having them acquire an exclusive lock. The implementation is lock free for producers, relying on an atomic compare-and-swap operation to update the ring’s head and tail pointers. The buffer itself is a doubly linked list of memory pages whose pointers are 4-byte aligned. The subsystem ensures data integrity for consumers by designating a reader page and removing it from the linked list while a consumer reads from it.

- **Linux Circular Buffers [9]:** The API provides a generic multi-producer, multi-consumer ring buffer implementation for the kernel. It uses memory barriers [12] to achieve concurrent reading and writing, without locks or atomic operations. The extent of concurrency is limited: consumers acquire an exclusive lock to read; producers acquire an exclusive lock to write. The user specifies the buffer size, but the API expects a power of two so that it can use bit-masking for fast indexing. The API supports variable-size objects, as needed, by occupying multiple bins.

- **Xen I/O Rings [4]:** Xen Hypervisor uses an I/O Ring per device, per Guest OS to paravirtualize device I/O. An I/O Ring is a single-producer, single-consumer shared memory ring buffer. The Hypervisor and Guest OS use it
to receive and deliver asynchronous I/O request and response messages. By using atomic writes and monotonic index counters, incremented on write by Guest requests and decremented on write by Hypervisor responses, the shared memory access is free of locks and memory barriers. I/O Rings are always powers of two so that the implementation can use bit-masking for fast indexing.

A concurrency architecture to support concurrency-unaware applications needs both a lock-free implementation and sufficient abstraction to hide any shared resources. Like Linux Trace, Linux Circular Buffers, and Xen I/O Rings, most concurrency architectures focus on the reduction of lock contention. These architectures are sufficient when their users are aware of shared resources. For example, their users can safely receive indices out of order or expect to wait. The concurrency-unaware camera applications that this work targets cannot safely receive indices out of order and do not expect to wait. Consequently, much of the virtual ring buffer architecture consists of an interface virtualization to guarantee that the architecture appears to behave in a way that is consistent with the behavior of prevailing camera drivers that do not support concurrency.
Chapter 3

Design

3.1 Design Goals

Towards efficient application concurrency for concurrency-unaware camera applications, this work designs a camera driver architecture that achieves the following design goals:

- Supports concurrent access to the image stream
- Minimizes data redundancy and data fetches
- Isolates image data and application frame rates
- Conforms to the interface defined in Chapter 2.1

3.2 Hardware Assumptions

To minimize data fetches, the virtual ring buffer architecture assumes that the device’s processor uses a physically indexed, physically tagged (PIPT) cache. Virtually indexed cache lines require a cache flush from the CPU on a context switch. PIPT cache allows shared data, fetched for one application, to remain on chip for the others. Beginning with the ARMv7 ISA, ARM-based processors use a PIPT L1 D-Cache. The Cortex A9 and A15 use PIPT L2 cache controllers as well [1, 2]. Given the popularity of ARM-based processors for mobile devices, this assumption is commonly satisfied.
Figure 3.1: (Left): naive stream replication. Every application controls a private ring in memory. The driver copies images to meet demand. (Right): virtual ring buffer. Every application interacts with a virtual ring. The driver maps virtual bins to a shared ring in memory on demand. Shared memory is copy on write. Superscripts indicate virtual indices used by applications. Normal numbers indicate memory mappings. Matching colors represent the same image.

3.3 Virtual Ring Buffer Design

As illustrated in Figure 3.1 (right), the virtual ring buffer architecture is logically split into two parts: a virtual interface for each application and a shared ring buffer in memory. Concurrency-unaware applications will generally configure their rings to different sizes and consume images at different rates; yet the shared ring is the only data structure used to deliver images here. The virtual interfaces serve as an abstraction layer between the actual state of the ring buffer and the state each application believes it to be in. In addition to delivering images to every application, the virtual ring buffer driver must update the virtual interfaces.

The virtual interfaces reproduce the behavior described in Chapter 2.1 for each
application. Each interface consists of the operation to configure the size of a ring buffer, the operation to enqueue a bin, and the operation to dequeue a bin. Unlike the driver described in Chapter 2.1, this driver does not allocate memory for a ring when one is configured by any application. The virtual ring buffer driver simulates the existence of a private ring for each application instead. The number of bins the application configures serves as a limit to the number of active memory maps the driver creates for the application. The driver maintains a Released and Ready queue for each application to ensure correct frame delivery and index consistency. After each enqueue operation (i.e., a bin release and data request), the driver can create a memory map from the virtual bin at the index the application enqueued to the newest image in the shared ring. Chapter 4.2 describes the technique used to update each interface in constant time.

The virtual ring buffer driver allocates memory for the minimum number of ring buffer bins that it needs to satisfy the frame rate of all applications. Like the driver in Chapter 2.1, the virtual ring buffer driver allocates memory for the ring buffer in the driver’s address space. It does not, however, allocate the number of bins configured by any application. The driver allocates as few as one bin in memory and adds more on demand. The driver inspects the Released queues and the availability of bins as images arrive to determine whether the ring buffer needs to grow or not. Chapter 4.1 details the lock-free mechanisms the driver uses to do so.

The virtual ring buffer architecture guarantees data integrity and minimum data redundancy across all applications. To identify applications that should share an image, the driver inspects every Released queue when a new image is ready. The driver then creates a memory map to the new image for every application that has a non-empty Released queue. This partially minimizes data redundancy. The driver
also applies a copy-on-write policy to every memory map; thus every write, if one occurs, generates a copy of affected memory pages only, not necessarily of the whole image. The virtual bin, being composed either partially or entirely of private memory, is temporarily non-virtual after a write. Chapter 4.4 describes the worst-case copy-on-write scenario and Chapter 6 shows that copy-on-write image sharing is always more power- and CPU-efficient than copying images up front.

The virtual ring buffer driver also isolates every application’s frame rate from the others’ frame rates. The driver couples the lifetimes of shared memory maps to enqueue operations. Chapter 4.3 details how creating a memory map at the time of image delivery, and destroying it when the application enqueues the corresponding virtual bin, allows faster applications to share memory with slower applications. As reviewed in Chapter 2.1, ambient lighting, algorithm workload, and camera hardware affect frame rate. This frame rate guarantee does not assert that the system will perform the same under concurrent workloads; it asserts that each application can enqueue and dequeue bins without interference from other applications.
Chapter 4

Features

To elaborate on the design of the virtual ring buffer architecture, this chapter describes how its components work in concert to meet the goals defined in Chapter 3.

4.1 Lock-free Memory Sharing

The virtual ring buffer driver shares every image in the stream maximally. The driver inspects the length of every application’s Released queue at the moment each new image arrives. If all Released queues are empty (i.e., no application has enqueued a virtual bin), the driver drops the image. If at least one is non-empty, the driver finds the first unoccupied bin in memory to write to. If no such bin exists, the driver allocates a new bin in memory and writes the new image to it. The driver then creates a memory map from the image to the virtual bin at the top of every non-empty Released queue; thus every application that requests an image will receive a shared memory map to the image as soon as the image is available.

The virtual ring buffer driver critically must have the capacity to identify unoccupied bins in memory in real time. It uses lock-free reference counts to do so. When a new image arrives, the driver reads the reference counts to identify unoccupied bins and, just after creating the memory maps, sets the count for the newly occupied bin. Every enqueue event decrements one reference count. Lock-contention can delay image delivery or cause superfluous memory allocations. This architecture achieves
lock-free reference counting in the same way Xen I/O Rings do [4]. The applications only reduce the reference counts when they enqueue virtual bins. The driver sets a reference count just before it delivers an image. The strict ordering of enqueue after image delivery is sufficient to keep the reference counts lock free.

4.2 Interface Virtualization

Interface virtualization is the task of dynamically mapping the virtual bin indices each application uses to the actual bin indices in the shared ring buffer that the driver created a memory map to. A correct mapping is necessary to both simulate the expected state of the ring for each application and to decrement the correct reference count when each application enqueues a bin. The virtual ring buffer driver virtualizes each interface in constant-time.

The driver uses an array, in addition to the Released and Ready queues, to manage the mappings for each application. The array stores actual indices to bins in memory. The length of the array is equal to the number of bins the application configures for its virtual ring. After the driver creates a memory map to a bin in memory, the driver writes the bin’s index to an entry in the array. The driver determines which entry to write to by inspecting the virtual index at the top of the application’s Released queue. The driver then sets the entry at the virtual index of the array equal to the index of the bin in memory. When the application enqueues a virtual bin again, the driver uses the virtual index supplied by the application to recover the index of the bin in memory. The driver can then decrement the reference count for the correct bin in memory and destroy the appropriate memory map.
Figure 4.1: Example sequence of bin utilization for two applications with different frame rates. By coupling memory map lifetimes to enqueue operations, App A can reuse its first virtual bin at time step 2 without causing data loss for App B.

### 4.3 Frame Rate Isolation

The sequence in Figure 4.1 illustrates the type of bin contention that the virtual ring buffer successfully eliminates. In the example, App B configures one virtual bin and enqueues at a rate $X$. App A configures two virtual bins and enqueues at a rate $3X$. Suppose the driver has allocated two bins in memory. At time step 0, both A and B dequeue virtual bins mapped to Bin 0 in memory. By time step 1, A enqueues Bin 0 in memory and calls dequeue. A receives its second virtual bin, mapped to Bin 1 in memory; B continues to use a virtual bin mapped to Bin 0 in memory. By time step 2, A enqueues Bin 1 in memory and calls dequeue. At time step 2, A expects to have its first virtual bin, which was previously mapped to Bin 0 in memory, filled with new data, but B is still using a virtual bin mapped to Bin 0 in memory.

The virtual ring buffer architecture isolates one application’s frame rate from the others’ frame rates via the virtualization of interfaces and the coupling of memory maps to enqueue events. In the example above, the driver does not allocate more
memory or prevent A from receiving a new image. By time step 2, the driver has already deleted the memory map from App A’s first virtual bin to Bin 0 in memory and from App A’s second virtual bin to Bin 1 in memory. The driver is free to memory map App A’s first virtual bin to Bin 1 in memory. Without a virtual interface, App A receives a reference to the bin it most recently enqueued, rather than to the bin it expects to receive. Enqueue-destroyed memory maps decouple applications from the shared images; virtualization keeps interfaces consistent with expectations.

True frame rate isolation also requires that the time to enqueue and dequeue bins remains constant for each consumer. The driver uses lock-free reference counting and virtualizes each interface in constant time. Compared to the creation and destruction of memory maps in Android, these operations are negligible. The memory map overhead is, thus, the primary scalability bottleneck of the virtual ring buffer driver.

### 4.4 Data Integrity and Redundancy

The virtual ring buffer driver applies a copy on write policy to all memory maps. Concurrency-unaware applications believe they consume private images and expect overwrites to succeed. The OS kernel handles any write to a copy-on-write memory map transparently, by replacing the targeted page in the memory map with a private copy; thus every application is free to write to the shared image without corrupting it for the others.

Page-specific copies are advantageous in computer vision. Figure 4.2 shows the memory layout of the first few rows of a YUV420sp format image that is 800 pixels wide. The alternating colors highlight the coverage an individual page provides. As shown in Figure 4.3, computer vision applications commonly draw thin bounding boxes around detected objects or points of interest. Because the bounding boxes
Figure 4.2: Illustration of pixel layout in memory. 4 KB pages store a row of pixels sequentially and continue at the next row. The alternating colors highlight the page boundaries of a typical image.

and interest points are usually localized, many pages are left untouched. A copy on write memory map can often expect to replicate just a fraction of the total image in memory.

In the worst case, the virtual ring buffer driver can temporarily and repeatedly allocate more memory than naive stream replication allocates. If all applications overwrite every page in every image, the driver creates full, private copies of every image in memory and keeps the pages from the original image. Chapter 6 shows that copy on write, even in the worst case of full image copies, outperforms the copy strategy used by stream replication.
Figure 4.3: A common scenario in image processing. A drawn bounding box incurs copies of just the pages that contain overwritten pixels. Without reordering pixels, the virtual ring buffer driver generates the minimum number of page copies.
Chapter 5

Implementation

This chapter summarizes the implementation challenges and optimizations that are specific to Android, OpenCV, and the Exynos 4210 platform. This work implements the virtual ring buffer and naive stream replication architectures, as described in the previous chapters, directly into the framework for Android 4.3.1 on the Galaxy S2 smartphone. OpenCV camera applications provide different workloads for the evaluation of the two architectures. This chapter documents all major modifications to Android, OpenCV, and the Exynos 4210 platform configuration.

5.1 Zero-Copy JNI Boundary

Images streaming from the Camera Service to the application must cross the JNI boundary. This boundary separates the native C++ code used by the Android framework from the Java code used by Android applications. Broadly, to cross the boundary, the framework must instantiate a Java object that the application can interact with. As shown in Figure 5.1, to deliver an image across the boundary, stock Android copies the image from the bin in native memory to a Java Byte Array in Java managed memory. Android’s copy solution to cross the JNI boundary invalidates the shared memory optimizations in the virtual ring buffer architecture.

The implementation of this work modifies the Android framework to cross the JNI boundary with Java Byte Buffers instead of Java Byte Arrays. Java Byte Buffers are
Java objects that can wrap Java memory from a Byte Array or native memory from a C/C++ array. Using a Byte Buffer, a Java application can read and write from Java or native memory; thus Android can deliver an image by instantiating a Byte Buffer with a pointer to the shared native memory. Applications that choose to perform image processing on the native side, like those that use the OpenCV library, can recover the data pointer directly from the Byte Buffer too. With a Java Byte Array, the native code has to protect the managed memory from the garbage collector, copy the image data into a region of native memory, and unprotect the managed memory from the garbage collector. The Byte Buffer solution allows an image to cross the JNI boundary an arbitrary number of times without incurring a single copy.

To support image delivery via Byte Buffers, the implementation of this work overloads the Android PreviewCallback interface and one method in the OpenCV library. The overloaded PreviewCallback allows the Android application to receive images in Byte Array or Byte Buffer objects. The overloaded OpenCV function can initialize an OpenCV Mat object by reusing the data pointer in a Byte Buffer or by protecting the Java memory in a Byte Array, copying its data to a native array,
and supplying the Mat object with a pointer to the native array. Consequently, the
implementation of the virtual ring buffer driver in Android only works for Android
applications that use the Byte Buffer interface.

5.2 Copy on Write ashmem

The anonymous shared memory (ashmem) driver in Android breaks conventional
mmap syntax. The flags argument for mmap designates whether a memory map is
shared or private. If the MAP_SHARED bit is set, writes to the mapping are visible
to the other processes that map the same region. If the MAP_PRIVATE bit is set,
writes to the mapping are invisible to the other processes because it implements a
copy on write policy. The ashmem driver incorrectly handles the flags argument.
Instead of creating a copy on write memory map, the ashmem driver allocates a
new region of anonymous shared memory for any process that calls mmap with the
MAP_PRIVATE bit set.

The implementation of this work modifies the ashmem driver so that it correctly
interprets the flags argument to mmap when the Camera Service calls mmap. Android
allows a process to apply, at the time of allocation, a name to a new region of ashmem.
For the implementation of this work, the Camera Service applies a name to the
ashmem regions that make up the shared ring buffer. A small modification to the
ashmem driver forces it to test the name of the ashmem region before the driver
creates a memory map. If the name provided to the ashmem driver during a call to
mmap matches the name used by the Camera Service, the ashmem driver honors the
flags argument and creates a copy on write memory map.
5.3 Thread-Safe Android Camera Service

The Camera Service source code is not thread-safe. The Client-Server split in Android causes race conditions. On one side, in its own process, is the Camera Service, and on the other side, in the application’s process, is the Camera Client. The Camera Client maintains the Released queue, but the Camera Service needs to know when the queue is non-empty. Stock Android implements a message-passing scheme from Client to Service. With each enqueue to the Released queue, the Client sends a message to the Service to resume the image stream. As soon as the Released queue is empty, the Client sends a message to the Service to pause the image stream. The message passing latency, when multiple Camera Clients operate, is too high. Consequently, Camera Clients receive more images than they can handle or not enough images when they can.

To eliminate races, the implementation of this work moves the Released queues to the Camera Service. The Camera Service itself inspects the Released queues to make decisions about pausing and resuming the image stream. If all Released queues are empty, the stream pauses; otherwise the stream continues.

5.4 Concurrent Android Camera Applications

The Android Camera Service requires an Android Surface to display images to. Android defines four application components: Activity, Service, Broadcast Receiver, and Content Provider. An application can consist of zero or more of every component, but only the Activity and Service are relevant to this discussion. An Activity represents a single user interface and thus can supply a Surface. A Service performs long-running background operations and cannot supply a Surface. Only an application with an
Activity on display can start the camera.

Activity-based applications cannot run concurrently and thus cannot access the Camera concurrently. Because stock Android allows just one Activity to occupy the display at any time, only one application can provide a Surface to the Camera Service at any time. Furthermore, Android deallocates a Surface object as soon as its Activity leaves the display; thus an application that uses both an Activity and a Service component cannot maintain camera access, by running in the background, after its Activity leaves the display.

To allow concurrent access to the Camera Service, the implementation of this work modifies the Camera Service to accept two types of users. The first type is the usual Client user; it supplies a Surface to the Camera Service and can start the image stream. The second type is a Shared user; it registers with the Camera Service without a Surface, but cannot start the image stream. A Shared user, consequently, must wait for a Client user to supply a valid camera ID and Surface and to start and stop the stream. These two users are otherwise identical. The Camera ID, provided as an argument to the camera API to specify whether the Camera Service should start the front-facing or back-facing camera, also distinguishes Client users from Shared users. Valid IDs range from zero to the number of cameras minus one. For implementation purposes, the Camera Service identifies a user that supplies an ID equal to negative one as a Shared user that is trying to register. This solution allows one Activity-based application and zero or more Service-based applications to concurrently access the camera.
### Table 5.1: Thermal management unit states and actions

<table>
<thead>
<tr>
<th>Event</th>
<th>Temp (F)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop_1st_throttle</td>
<td>61</td>
<td>N/A</td>
</tr>
<tr>
<td>start_1st_throttle</td>
<td>64</td>
<td>CPU: 800 MHz</td>
</tr>
<tr>
<td>stop_mem_throttle</td>
<td>80</td>
<td>DRAM Refresh: 1.95 µs</td>
</tr>
<tr>
<td>start_mem_throttle</td>
<td>85</td>
<td>DRAM Refresh: 3.90 µs</td>
</tr>
<tr>
<td>stop_2nd_throttle</td>
<td>87</td>
<td>N/A</td>
</tr>
<tr>
<td>start_2nd_throttle</td>
<td>103</td>
<td>CPU: 200 MHz</td>
</tr>
<tr>
<td>stop_2nd_throttle</td>
<td>110</td>
<td>Shutdown</td>
</tr>
<tr>
<td>start_2nd_throttle</td>
<td>120</td>
<td>Kernel Panic</td>
</tr>
</tbody>
</table>

### 5.5 Fixed Exposure Time

The kernel-level driver for the Galaxy S2 camera employs a light metering algorithm to adjust imaging parameters. To produce a nice image, the exposure time changes constantly to adapt to ambient light intensity. Because exposure times vary constantly and unpredictably, frame rate varies unpredictably. For consistent evaluation, the implementation of this work disable the metering algorithm and fixes the exposure time manually.

### 5.6 Exynos Microarchitecture Configuration

The Exynos 4210 SoC implements four mechanisms to automatically manage the availability of system resources: the thermal management unit, the CPU frequency governor, the memory bus frequency governor, and the hot plug governor. For consistent measurement of system utilization, the implementation of this work configures
Table 5.2: CPU frequency governor policies

<table>
<thead>
<tr>
<th>Level</th>
<th>MHz</th>
<th>Governor Policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>1200</td>
<td>OnDemand, Conservative, Performance</td>
</tr>
<tr>
<td>L1</td>
<td>1000</td>
<td>OnDemand, Conservative</td>
</tr>
<tr>
<td>L2</td>
<td>800</td>
<td>OnDemand, Conservative</td>
</tr>
<tr>
<td>L3</td>
<td>500</td>
<td>OnDemand, Conservative</td>
</tr>
<tr>
<td>L4</td>
<td>200</td>
<td>OnDemand, Conservative, PowerSave</td>
</tr>
</tbody>
</table>

each one to maximize the availability of system resources.

The thermal management unit overrides the CPU frequency and reduces the DRAM refresh rate to protect against overheating. Table 5.1 summarizes the events that trigger action from the thermal management unit. Running the camera for several minutes at a time can easily trigger the first throttle. Because CPU utilization increases when CPU frequency decreases, thermal management skews results unpredictably. The implementation of this work configures thermal management to do nothing until the processor temperature is high enough to trigger the second throttle.

The CPU frequency governor sets the target frequency of the CPU according to one of several policies. Table 5.2 shows the frequency levels available to each policy on the Exynos 4210. The default on most Android devices is the OnDemand policy. It increases the target frequency quickly when work is pending at the CPU. Its counterpart, the Conservative policy, increases the target frequency slowly. The Performance and PowerSave policies are independent of the pending workload and respectively pin the target frequency to the maximum and minimum level. Because the smartphone can idle without work between measurements, or vary in the amount of work to be done at any given time, the default OnDemand policy produces significant variation
Table 5.3: Memory bus frequency governor levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Memory (MHz)</th>
<th>Bus (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>400</td>
<td>266</td>
</tr>
<tr>
<td>L1</td>
<td>400</td>
<td>200</td>
</tr>
<tr>
<td>L2</td>
<td>267</td>
<td>160</td>
</tr>
<tr>
<td>L3</td>
<td>267</td>
<td>160</td>
</tr>
<tr>
<td>L4</td>
<td>160</td>
<td>160</td>
</tr>
<tr>
<td>L4</td>
<td>133</td>
<td>133</td>
</tr>
<tr>
<td>L4</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

in results. The implementation of this work pins the frequency of all cores to 1.2 GHz by selecting the Performance policy for the CPU frequency governor.

The memory bus frequency governor sets the target frequency of both the DRAM and the bus. Table 5.3 shows the available frequency levels for the Exynos 4210 chip. A per-pin measurement unit in hardware provides a bus traffic measurement at regular intervals. Based on the traffic load, the memory bus governor scales the frequencies up or down. The resulting performance variation is unpredictable and difficult to measure. The implementation of this work configures the governor to ignore traffic levels and pin the frequencies to the L0 level.

The hot plug governor inspects the amount of pending CPU workload to determine how many cores are on or off at any time. The Exynos 4210 chip has two cores; the hot plug governor periodically inspects the workload queued for the first core. It starts the second core when the pending workload exceeds some threshold. Below some threshold, the hot plug governor turns the second core off. Some of the evaluation applications produce workloads near the threshold; results vary wildly as the second
core oscillates on and off. The implementation of this work configures the hot plug governor to keep all cores on at all times.
Chapter 6

Evaluation

6.1 Test Applications

The evaluation of this work consists of measuring system utilization under workloads from three groups of applications. The first group generates the workload for frame delivery only. The second group generates workloads for three primitive image processing algorithms. The third group generates a maximum workload from four applications that implement a worst-case scenario of realistic computer vision applications with severely mismatched frame rates. The three groups provide sufficient context to characterize the system efficiency for both concurrency solutions.

The frame delivery group consists of four identical applications that enqueue an image buffer as soon as it is received. To ensure that demand paging does not prevent the memory map from being created, the applications read the first pixel of each image. Results from this group measure the system resources needed to deliver the image stream to concurrent applications for both concurrency solutions.

The primitive algorithms group consists of four identical applications that execute one of three different algorithms, each in a read and a write implementation. The first algorithm iterates along the first column of each image; the second iterates along the first row of each image. Results from these two primitives simulate workloads from point-wise algorithms, such as those that rely on histograms [10] or dynamic programming [3]. For the read implementations, the algorithms read the pixel values
of the first row and first column. For the write implementation, the algorithms set the pixels in the first row and first column to a value of one. The third algorithm generates the first level of a Gaussian pyramid by downsampling the image, by two, in both dimensions and convolving the downsampled image with a 5x5 Gaussian kernel [13]. Results from this primitive simulate workloads from stencil-based algorithms. For the read implementation, the algorithm calculates the first level and stores it in a separate variable. For the write implementation, the algorithm calculates the first level and overwrites the image with the result. The primitives group simulates the workload for the earliest step of many computer vision algorithms.

The algorithms for the primitive group do not execute complete algorithms by design. One row, one column, and one pyramid level provide all the insight to determine the impact of the driver architectures on complete algorithms. Usually, after the earliest computations on the original image, computations continue on newly generated data or on copies incurred by copy on write. The driver architecture has less impact on system utilization after the early stages of a complete algorithm.

The worst-case group consists of four applications that implement complete computer vision algorithms. The algorithms span a spectrum of light to heavy workloads. The first performs a conversion of the received image from YUV420sp format to RGBA format and draws the result to the display. The second application applies a Sobel filter to a gray scale version of the image and overwrites the image with the resulting black and white edge intensity image. The third application detects faces using a cascade classifier of local binary patterns on a gray scale version of the image; a bounding box drawn over the image identifies any detected faces. A panel of faces in the camera’s field of view guarantees work. The fourth application applies a pyramid-SURF feature detector, FREAK feature descriptors, and a FLANN-based
point-matcher to two gray-scale instances of the same image. Lines drawn over the image pair identify matches. Using the same image maximizes the matching workload. The worst-case group motivates the discussion for future work in Chapter 7.

6.2 Measurements

This work measures CPU utilization, power consumption, and frame rate for several workload scenarios. The camera streams images at 176x144 pixel resolution and at 800x480 pixel resolution. One to four copies of each application in the first two groups run concurrently, at 09 ms and 72 ms exposure times, for each image resolution. The four applications in the worst-case group run concurrently at 09 ms exposure time for the 800x480 pixel resolution only. The resulting measurements provide sufficient data to compare the efficiency of the two concurrency solutions.

This work measures each statistic as follows. The mpstat utility for Linux measures CPU utilization; it collects per-core and aggregate statistics during twelve ten-second windows. A Monsoon Power Monitor measures power consumption; it collects average power statistics during intervals of thirty seconds, repeated five times for each data point. Idling the phone at minimum CPU frequency for several minutes between measurements mitigates thermal influences on power measurements. Each application calculates frame rate for itself and writes the value to a file using the default Completely Fair I/O scheduler for Android. For all but the worst-case scenario, frame rates match the frame rate set by the exposure time. A presentation and discussion of the statistics follows.
Figure 6.1: CPU utilization to deliver 176x144 pixel thumbnail images (.03 MP) and
800x480 pixel preview images (.40 MP) at 30 fps with an exposure time of 9 ms. The
CPU supports twice as many applications using the virtual ring buffer over naive
stream replication.

### 6.3 Results for Frame Delivery

Figure 6.1 and Figure 6.2 show the CPU utilization to deliver the image stream at
30 and 15 frames per second (fps), respectively, for 176x144 pixel thumbnail images
and 800x480 pixel preview images. The curves for 30 fps are representative of the
curves for 15 fps across all measurements; only the 30 fps scenarios, achieved by 09 ms
exposure time, are presented hereafter. For the same CPU workload, the virtual ring
buffer driver can deliver the image stream to twice as many concurrent applications
than naive stream replication can.
Figure 6.2: CPU utilization to deliver 176x144 pixel thumbnail images (.03 MP) and 800x480 pixel preview images (.40 MP) at 15 fps with an exposure time of 72 ms. The CPU supports twice as many applications using the virtual ring buffer architecture over stream replication.

A small offset appears to separate the single-application data points for the two solutions in both Figure 6.1 and Figure 6.2. In both solutions, the system delivers the exact same amount of data. The naive solution, however, creates two additional copies of every image: one to place it in a Java Byte Array by Android and one to place it back in a C++ array by OpenCV. Thus the zero-copy JNI optimization causes a small improvement in CPU utilization.

Figure 6.3 shows the power consumption at 30 fps. A 400 mW offset separates the data points for one application. Given that no concurrency is present for one
Figure 6.3: Power consumption to deliver 800x480 pixel preview images (.40 MP) at 30 fps with an exposure time of 9 ms. The virtual ring buffer architecture reduces power consumption by more than 500 mW.

application, the zero-copy JNI optimization appears to achieve a substantial power saving. As concurrency increases, the virtual ring buffer curve takes the shape of a log curve, whereas the stream replication curve appears linear. These curves show that the virtual ring buffer with a zero-copy JNI optimization is a power-efficient solution for concurrent image delivery in Android.

6.4 Results for Image Processing Primitives

Figure 6.4 shows CPU utilization for the primitive algorithm that reads or writes the pixels in the first column of the image. As illustrated by Figure 4.2, column
Figure 6.4: CPU utilization for applications that read (R) or overwrite (W) pixels in the first column of 800x480 pixel images delivered at 30 fps with an exposure time of 9 ms. The virtual ring buffer uses significantly less CPU time than naive stream replication.

Traversal produces a non-sequential access pattern in memory. Naive stream replication produces read and write curves that are nearly identical; the virtual ring buffer produces diverging read and write curves because of the copy-on-write policy. Given the dimensions of the image, this primitive accesses 480 1.5-byte pixels. Because the operating system uses 4 KB pages to manage memory, approximately every third write to a pixel triggers a page copy. Writing to every pixel in the column generates a full image copy. Although both architectures produce linear curves, the virtual ring buffer is always more efficient.
Figure 6.5: CPU utilization for applications that either read (R) or write (W) pixels in the first row of 800x480 pixel images delivered at 30 fps with an exposure time of 9 ms. The virtual ring buffer uses significantly less CPU time than naive stream replication.

Figure 6.4 suggests copy-on-write is cache friendly. The write implementation of this primitive generates an equal number of copies of the image stream in both architectures. The copy-on-write policy, however, makes a copy of the 4 KB page of the image as the processor needs it. The data moves into cache during copy and the next three to four pixel accesses are free; thus duplicating the entire image by copy on write generates a smaller CPU workload than duplicating the entire image from the start.

Figure 6.4 shows CPU utilization for the primitive algorithm that reads or writes
the pixels in the first row of the image. Whereas the column primitive accesses 480 pixels in a non-sequential pattern, the row primitive accesses 800 pixels in a sequential pattern. Comparing the curves for the two primitives, the row primitive shows no improvement in CPU efficiency for naive stream replication and a near-10% improvement for the virtual ring buffer. The virtual ring buffer is again always more CPU-efficient than naive stream replication.

The copy-on-write policy might explain the virtual ring buffer’s near-10% CPU-efficiency improvement from the column primitive to the row primitive. The write implementation of the row primitive incurs a single page copy; the write implementation of the column primitive incurs 144. However the read curve, which incurs zero copies, improves as much as the write curve in the row primitive; thus copy-on-write overhead is minimal and cannot account for the improvement.

The processor’s hardware prefetcher and PIPT cache produce the near-10% improvement in CPU-efficiency for the virtual ring buffer. The technical specifications for the ARM Cortex A9 cores on the processor indicate that the data prefetcher will load up to eight 32-Byte cache lines into the 32 KB L1 cache when the CPU requests memory in a regular pattern. The prefetcher continues to prefetch as long as cache hits occur on the prefetched data. The prefetcher stops when it crosses a 4 KB page boundary or the CPU changes context. The row primitive accesses pixels sequentially and never crosses a page boundary; the prefetcher has optimal benefit. Consequently the read curve is nearly identical to the .40MP VRB image delivery curve in Figure 6.1. The overwrite curve, however, is significantly higher than the image delivery curve. The ARM Cortex A9 uses a PIPT data cache. It does not need to flush cache on a context switch and can reuse shared data across processes. The difference in the read and write curves in Figure 6.4, when compared to the .40MP VRB image deliv-
Figure 6.6: CPU utilization for applications that calculate the first level of a Gaussian pyramid for 800x480 pixel images delivered at 30 fps with an exposure time of 9 ms. Results are stored in separate memory (R) or overwrite the image in place (W).

eyery curve, shows that applications are able to reuse shared data that was loaded into cache by the other applications. The write implementation works from private copies of the same page; consequently every application loads its own data every time. Thus the prefetcher accounts for the near-10% improvement from the column primitive to the row primitive and the PIPT cache accounts for most of the difference in the read and write curves for both primitives.

Analysis of CPU utilization for the third primitive builds on the insights drawn from the previous two. Figure 6.6 shows CPU utilization for the primitive algorithm that calculates the first level of a Gaussian pyramid. The read implementation stores
Figure 6.7: Power consumption for applications that calculate one level of a Gaussian pyramid for 800x480 pixel images (.40 MP) delivered at 30 fps with an exposure time of 9 ms. The virtual ring buffer shows a 500 mW improvement over stream replication with two concurrent applications. The gap widens as concurrency increases.

the result to a separate variable; the write implementation writes the result over the image. Because the Gaussian kernel is separable, the OpenCV library implements the convolution for a 5x5 Gaussian kernel by convolving in one dimension at a time. The algorithm first convolves the kernel with each row horizontally and stores the result to an intermediate array. This step approximately performs the row primitive for several rows. The intermediate array, however, precludes data reuse during the rest of the algorithm for either architecture using either read or write implementation. The algorithm then vertically convolves five rows at a time in vectorized form. This step approximately performs the write implementation of the column primitive. The
Table 6.1: Power measurements for worst-case group of applications. Results show no advantage to either architecture because the applications have few sharing opportunities.

<table>
<thead>
<tr>
<th>Arch.</th>
<th>CPU(%)</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSR</td>
<td>100</td>
<td>6.70 ± 0.15</td>
</tr>
<tr>
<td>VRB</td>
<td>100</td>
<td>6.85 ± 0.07</td>
</tr>
</tbody>
</table>

write implementation of the third primitive for the virtual ring buffer incurs thirty-six page copies at the very end because the first pyramid level is a quarter of the size of the original image. The reason for the offset between the read and write curves for naive stream replication remains unclear from the data.

Figure 6.7 shows power consumption for the read implementation of the third primitive algorithm only. The write implementation would never occur in practice. For one application, the stream replication and virtual ring buffer architectures consume approximately the same power. The virtual ring buffer shows a 500 mW improvement over stream replication with two concurrent applications. As more applications are added, the gap widens to nearly one Watt of power savings with the virtual ring buffer architecture.

6.5 Results for Worst-Case Group

The worst-case group implements complete algorithms for face detection, image matching, edge filtering, and image format conversion across four applications. Table 6.1 shows that this combination of applications saturates the CPU and consumes an extreme amount of power. The results for both architectures are equivalent. The worst-case group eliminates the remarkable power and CPU savings reported in Chapter
Table 6.2: Breakdown of image sharing for the group of worst-case applications. Share1 indicates the percentage of images that were distributed to one application only. Share4 indicates the percentage of images that were distributed to all four applications. Applications from Chapters 6.3 and 6.4 shared maximally at all times.

<table>
<thead>
<tr>
<th>Arch.</th>
<th>Share1(%)</th>
<th>Share2</th>
<th>Share3</th>
<th>Share4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSR</td>
<td>56</td>
<td>37</td>
<td>07</td>
<td>00</td>
</tr>
<tr>
<td>VRB</td>
<td>50</td>
<td>42</td>
<td>08</td>
<td>00</td>
</tr>
</tbody>
</table>

6.3 and Chapter 6.4. To understand why, the Camera Service is modified to capture the number of applications that receive each image. Table 6.2 shows the results. The Camera Service delivers the majority of images to one application only and over 90% of images to just one or two applications. Naive stream replication is able to process 98% of the images that the virtual ring buffer is able to capture. These results indicate that, in the absence of sharing opportunities, the virtual ring buffer architecture degenerates to that of naive stream replication.
Chapter 7

Conclusion

This work presents the design and characterization of a software architecture that supports camera application concurrency for concurrency-unaware camera applications. It incorporates a virtual interface to create an illusion that each application is the only application consuming images from the camera. Under the hood, a single ring buffer provides shared memory access to all images produced by the camera. By creating and destroying memory mappings from each application’s virtual interface to the shared buffer, the virtual ring buffer driver can guarantee frame rate isolation among applications. By sharing images in a single buffer and protecting the shared memory with a copy-on-write policy, the architecture achieves minimum data redundancy. Compared to the current desktop solution for camera application concurrency, stream replication, the virtual ring buffer solution can support nearly twice as many applications for a given level of CPU utilization and saves several hundred milliWatts of power on a Galaxy S2 smartphone.

Evaluation on a realistic, worst-case group of computer vision applications reveals that the benefits of the virtual ring buffer over naive stream replication are maximized when applications request images at the same time. This happens when applications process images at the same or similar rates; such an assumption is only realistic if all applications are real time. If the frame rate isolation requirement, that each application must receive the next image to arrive after the application enqueues a bin, is relaxed, one could implement a novel technique to improve sharing. For example,
the camera driver could decide to delay image delivery by one interval or provide an image that is from the previous interval. Such a technique should coerce sharing with imperceptible impact for most applications; thus a more intelligent image delivery deadline for the virtual ring buffer should provide more consistent energy and CPU efficiency gains with negligible performance impact.
Bibliography


