F4GRAPH: AN API FOR CAMERA SCHEDULING AND HETEROGENEOUS IMAGE PROCESSING ON MOBILE DEVICES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Steven Bell
August 2022

Abstract

Mobile device photography is in the midst of a transformation: fixed processing pipelines are being replaced by flexible and programmable image processors, and single cameras are being replaced by clusters of sensors with diverse capabilities and new modalities. However, the camera APIs for mobile operating systems rarely expose the full capabilities of the hardware, because these capabilities depend on low-level and real-time code or platform-specific hardware details.

This work introduces F4graph, an application model and software interface which opens up many more capabilities of the imaging hardware to userspace applications. Within F4graph, captures on multiple image sensors or other hardware can be linked together to produce complex sequences of actions using a straightforward constraint-based scheduling model. This model enables the user to specify only the constraints that matter to the application, leaving freedom for the system to work around hardware and occupancy constraints, and to share the hardware between applications. The system provides strong guarantees by checking as many constraints as possible before committing actions to hardware and by only breaking constraints at well-defined points. Additionally, F4graph allows the application to define the processing pipeline along with the capture sequence, allowing image data to be streamed directly to a reconfigurable image processor or other hardware. This includes the ability to stream continuously, to process multiple images from a stream, and to create feedback cycles for metering and similar applications.

This thesis also describes how F4graph can be implemented in a real system, using a Xilinx Zynq Ultrascale board as a prototyping platform. Our demonstration system uses the Zynq's FPGA fabric as a reconfigurable image processing accelerator and its real-time core to orchestrate timing-critical events while the application processor cores run a normal multitasking Linux OS. Using the F4graph API on our prototype, userspace applications are able to schedule and execute actions with a temporal resolution under 20 µs — within a single image sensor scanline. The complexity of this heterogeneous hardware system is managed by a build system which generates the entire hardware/software stack from the application code and a short human-readable system description file. This makes it possible to develop complete applications that use precise capture sequencing and hardware-accelerated image processing without the need to write platform-specific or real-time code.

Acknowledgment

Numerous people have supported, participated in, critiqued, encouraged, and patiently endured this work, and without them it would not have been possible. It is my joy to be able to thank them briefly here.

First of all, thanks to my advisor, Mark Horowitz. It has been an immense privilege not only to work with with you on technical research, but also to share a love of teaching and a love of taking stuff apart. Getting to develop E40M with you was a load of fun and continues to influence how I think and teach today. It seems that we spent a good third of our meetings together happily disassembling and speculating on the inner workings of some gadget instead of talking about research. Looking back, it is only appropriate that I started working with you by volunteering to disassemble and reverse-engineer Micro Four Thirds lenses. As an advisor, you have been generous and supportive, and over the past few years you have been patient with me far beyond any reasonable obligation. Thanks.

Thanks to my readers, Pat Hanrahan and Marc Levoy. Your advice and feedback along the way provided both inspiration and focus for this work. You've also demonstrated that's it's possible to be a busy professor and still get your hands dirty building real stuff like Magma and Synthcam once in a while. I hope I can do the same!

Thanks also to Kayvon Fatahalian and Roger Howe for being part of my oral defense committee. It was a pleasure to work with each of you in various meetings and conversations about image processing and E40M respectively.

Many people contributed ideas and code (and occasionally sweat and blood over a lab bench) to the various projects that eventually became F4graph. Thanks to John

Brunhaver, Artem Vasilyev, Jing Pu, James Hegarty, Jeff Setter, Xuan Yang, Keyi Zhang, Holly Chiang, Gedeon Nyengele, and Zach Myers.

Steve Richardson was always ready to talk through a problem and worked with me though an untold number of server upgrades and system tweaks and emergency fixes. Mary Jane Swenson was (in the words of our group wiki) “scary good at her job” — friendly and efficient and always ready to help with all manner of random requests. Thanks to the office-mates I had over the years and the rest of the Horowitz group. I miss our weekly “what’s new and interesting in the world” conversations!

Outside of Packard and Gates, I’m thankful to my friends at the Stanford Center for Teaching and Learning for helping me grow as a teacher, and to the students in Intervarsity Graduate Christian Fellowship who became close friends, examples, and spiritual encouragers.

Thanks to my grandfather, who told me many years ago that I ought to pursue a PhD and become a professor. To my parents, who helped make it possible for me to start at Stanford, and who seemed to know exactly how to ask about my progress without ever being pushy. Countless thanks to my endlessly patient wife, who has put up with my regular stress and occasional despair for the entirety of our marriage. I’m really done now, dear.

And finally, praise be to God. Whatever good I have accomplished in this work or in my time at Stanford is by His generous grace.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Photography today is dominated by mobile phones. As shown in Figure 1.1, mobile phones have almost completely supplanted the point-and-shoot camera market over the last decade. In 2017 over 1.5 billion mobile phones with cameras were sold, which is roughly sixty times more than all other types of consumer cameras put together [1, 2]. Even on Flickr — which caters to enthusiasts and professionals — the recent iPhone models are the most popular cameras by a huge margin [3]. As the adage goes, "the best camera is the one you have with you," and mobile phones are seemingly always at hand. The megapixel wars are long over, and given that the picture quality on most mobile phone cameras is good enough for the casual user, the convenience of a pocketable form factor and a large vivid touchscreen is hard to beat.

As a result, photography itself is changing. Mobile phones did not merely replace point-and-shoot cameras; they opened up an enormous new market that now includes more than half of the world population. With the ability to take an image anytime and anywhere, cameras are used for all sorts of things, from capturing meeting notes to depositing checks to sharing snaps of every meal on social media.

There are three consequences of this shift to mobile devices. First, due to the thin form factors of mobile devices and the harsh realities of physics, mobile device cameras have significantly worse imaging performance compared to dedicated cameras. Although phone cameras have megapixel counts only slightly lower than their larger SLR cousins (12MP for the iPhone 8 vs 20.9MP for Nikon's D500 "flagship"

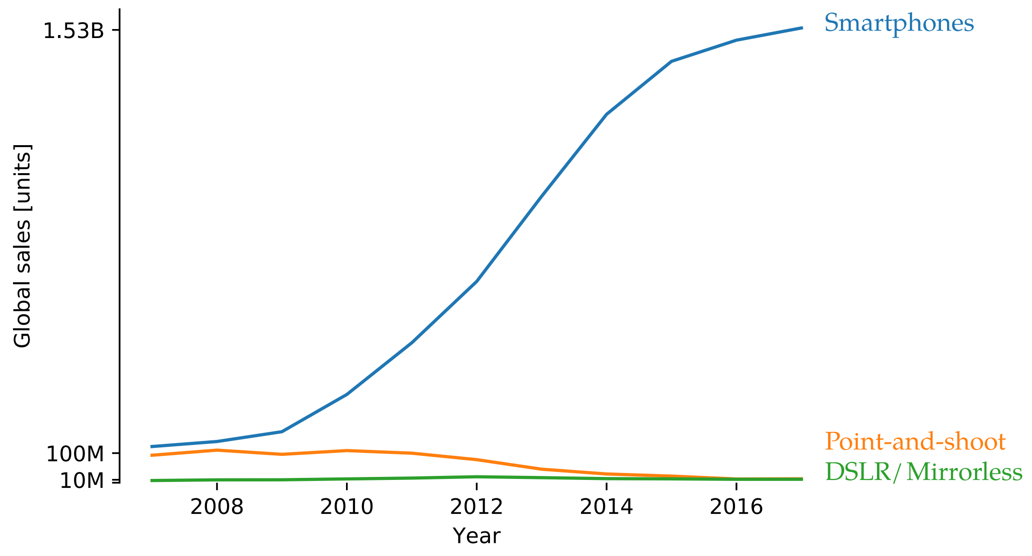


Figure 1.1: Global camera sales over the last decade. Data from [1, 2].

DSLR [4, 5]), the physical image sensors are smaller by a factor of four or five in each linear dimension, meaning that the area of each pixel is smaller by roughly a factor of 20. A pixel pitch of about one micron is typical for mobile phone sensors, while SLR cameras using the APS-C sensor format have a pitch around 4 µm [5].

The smaller pixel size has two drawbacks. At low light levels, fewer photons strike each pixel as compared to a larger sensor, meaning that the gain must be increased (leading to grainy images) or that the exposure must be longer (often leading to blur). At higher light levels there are plenty of photons, but a small sensor's electron wells fill up more quickly, limiting the overall dynamic range that the sensor can capture.

The compact form factor and lower manufacturing costs also limit the optical quality of mobile cameras. It is difficult to fit a lens with a wide aperture into the tight space constraints of a mobile device, meaning that these cameras have a larger depth of field and a less pleasing bokeh for close-ups and portraits. Moreover, the optical components are simple and inexpensive relative to the glass assemblies used in larger cameras, which increases optical aberrations such as radial distortion, color fringing, and dark shading near the corners of the image.

A second consequence of the shift to mobile phone cameras is that every camera now comes with a computer attached, in the form of the phone's processor. Computation has been necessary to correct for dead pixels and to produce full-color images from the Bayer mosaic since the advent of digital cameras, but its role on mobile devices has come front and center.

More and more, computation is used to offset the shortcomings described above: denoising algorithms clean up grainy images in low light [6], and calibrated filters warp and brighten images to correct lens aberrations. Capturing multiple images and fusing them together for improved rendering of both light and dark areas of the image (known as high-dynamic range imaging) is now standard [7]. And most recently, synthetic blurs are being used to simulate the pleasing bokeh of lenses with much larger apertures [8, 9].

But computation goes beyond making up for the limitations of the camera; it is also used to enhance images and to provide entirely new capabilities. Automatic panorama stitching makes it easy for novices users to capture 360-degree scenes [10]. Image "filters" add artistic effects, and various video effects (whether called "filters", "AR stickers" or "animoji") analyze a live stream of images to create and overlay virtual elements.

These examples are just the beginning of what computational cameras can and will do. Today's flagship phones are all flat slabs with ultra-high-resolution screens covering the whole front face of the device, leaving cameras as one of the few exciting areas for improved capabilities and market differentiation.

Computation will be used to further improve low-light performance, to compensate for handshake, and to produce synthetic long exposures [11]. There are opportunities to use two or more cameras together, whether for sensing depth, producing sharper images, or capturing video for virtual reality. Augmented reality is in its infancy, and will be a rich field for computational imaging. Depth sensors and other imaging modalities are beginning to appear in mobile devices, and these will open even more doors to compute with image data. In short, mobile device cameras are computational cameras.

The third consequence of the shift to mobile devices is that cameras are now programmable. There is of course lots of software running inside a camera, but they have traditionally been closed platforms, and the imaging algorithms that run on them are tightly guarded secrets. Although several camera manufacturers have recently released APIs for their cameras, these essentially provide a software interface which replicates what is ordinarily done with physical controls on the camera [12, 13].

In contrast, the camera app on a mobile device presents physical controls as an interface to the underlying software API. The camera app is simply another application on the device, and it is easy for a developer independent of the device manufacturer to write software to control the camera. Thanks to the robust app development ecosystem, a developer can create a new application from scratch in a matter of hours and experiment with ideas never conceived of by the original designers of the device. As a result, both Android and iOS have dozens of third-party camera applications which boast features not available in the stock camera applications.

Mobile operating systems also provide an extensive application distribution platform. Traditional cameras typically have their algorithms “baked in”, and consumers only replace them every few years. By contrast, apps can be installed and uninstalled in a matter of seconds, and improvements can be delivered wirelessly by overnight updates. Taken together, the availability of computation and the ease of developing and distributing camera-based applications means that imaging on mobile devices is a field ripe for continued innovation.

However, there are two impediments to further innovation and adoption of computational photography on mobile devices: access to powerful computational resources and precise control of the camera pipeline. Said another way, mobile devices provide computational resources and software control of the camera, but emerging and potential applications demand more of both.

We will first consider the challenge of computation. Processing a 1080p video stream with 60 frames per second requires handling over 120 million pixels per second. Performing even five or ten mathematical operations per pixel is sufficient to fully occupy a CPU, to say nothing of the complex algorithms we would like to run. In fact, applications such as Google’s HDR+ take several seconds to process a single frame [7].

The solution is to build specialized compute engines tailored to particular tasks. For example, the process of converting raw pixels from the image sensor into a presentable image requires a few hundred operations per pixel, which is infeasible on a CPU at video rates. Instead, both standalone cameras and mobile SoCs contain specialized *image signal processors* (ISPs) which perform this task very efficiently. Because the data buffering and the mathematical precision are optimized for this one task, ISPs (and similar fixed-function modules) are able to achieve orders of magnitude higher performance for the same power budget. A modern ISP can process around a gigapixel per second while using less than 2 W, which is roughly 25× more efficient than a mobile CPU [14].

The problem is that ISPs are fixed-function pipelines; they are efficient in part because they only do one thing. Graphics processing units (GPUs) are another category of specialized processors, targeted at a more general problem: highly parallel applications with lots of floating-point math. Because this application set is more general, GPUs are more flexible and can be programmed with languages like OpenGL Shading Language (GLSL), OpenCL, and CUDA. However, this increased flexibility relative to ISPs comes at the cost of efficiency.

Over the past few years a handful of programmable ISPs have been created to fill the space between fixed-function ISPs and GPUs. Examples include Google's Pixel Visual Core [15] and the the Movidius Myriad 2 [16]. There are several approaches to creating programmable ISPs: some add more flexibility to a traditional ISP pipeline, others look more like GPUs specialized for fixed-point integer processing. Chapter 4 will describe an approach based on an FPGA, which uses fixed designs programmed into a reconfigurable compute fabric. But regardless of the architecture, creating specialized hardware engines is only part of the challenge. The other challenge is to integrate them into the complete system, such that application-level code can make use of the hardware.

The system interface for traditional ISPs is quite simple: Either the data coming off the camera is streamed through the ISP, or it isn't. A handful of parameters are sufficient to control the behavior of the ISP, and any further processing can be done on the CPU or GPU using existing languages and libraries. This becomes more complex with programmable ISPs, because both the computation and the dataflow pattern must be configured. Some operations consume two images or more; others might take one image and run it through several different pipelines in parallel. Still others may consume input from device memory and a live camera simultaneously. Unfortunately, programming this specialized hardware is difficult, because both coding the algorithm for the hardware and configuring the interfaces requires low-level programming that is slow and error-prone. To enable productivity and portability, we need an interface to this type of hardware that is capable of expressing these more complex dataflow patterns.

The second impediment is control over the capture process. A cornerstone of computational photography is the ability to precisely control the capture pipeline, in order to capture more information about the scene in some way. Despite some recent improvements, camera system control remains rather limited on current platforms. Operating systems such as Android and iOS are running dozens of tasks in parallel and have many layers of abstraction separating the user code from the underlying hardware, which means that the timing precision of application code is sometimes horrendous. For example, the timestamps obtained from different sources (such as an image sensor and an IMU) can have varying offsets as large as twenty or thirty milliseconds, making it difficult to reliably correlate them. It is similarly difficult to synchronize actions which should take place in the system, such as frame captures from multiple sensors, lens motions, or flashes. There are built-in mechanisms for the traditional use cases — firing the flash during an exposure, or capturing an image with stereo cameras simultaneously — but anything outside of these is difficult or impossible.

To address these two impediments, this thesis introduces a set of abstractions that enable applications to capture multiple data streams with precise timing constraints and to process them in real time on heterogeneous hardware. Rather than specifying the capture and computation via timing-critical interaction with a stateful system, an application drives the system by submitting a set of stateless requests. Said differently, instead of specifying *how* to capture and compute each output frame, the application simply specifies *what* to capture and compute, and a runtime system orchestrates the process on the underlying hardware.

The abstract model unifies capture and processing, treating them as two parts of a single pipeline. The entire process from capture to output is modeled as a dataflow graph, with one or more capture requests as the data sources. Captures themselves are timed relative to other captures or external events in the system, meaning that an application can create precise schedules, and then delegate the work to a thread suited for real-time control.

This work introduces the following contributions:

- A set of abstractions that enable mobile-device applications to capture multiple data streams with precise timing constraints and to process them in real time on heterogeneous hardware.
- A practical implementation of this API on a Xilinx Zynq SoC, making extensive using of the Zynq's heterogeneous hardware resources.
- Characterization of the performance of this system, with application toward similar systems where a multitasking OS running user applications must make use of embedded hardware with hard real-time constraints.

To lay the foundation for the rest of the thesis, the following chapter takes a closer look at how existing cameras work, and discusses the existing interfaces for controlling them. Chapter 3 describes a new software interface for controlling capture and processing together, and Chapter 4 explains how such an API can be implemented on real hardware, specifically a Xilinx Zynq SoC. Finally, Chapter 5 demonstrates a number of working applications and several benchmark results which test the abilities and limits of the system, and Chapter 6 concludes the thesis.

Chapter 2

Background

The API introduced in this thesis builds on several streams of work. First are camera control APIs which enable applications to capture images and drive camera peripherals without needing direct access to the hardware. Second are multimedia streaming frameworks, which are flexible toolkits for creating applications which must efficiently capture, transcode, or view media at live framerates. Third are tools for writing high-performance image processing code. This chapter introduces each of these in turn. First, though, it is worth a brief look at modern cell-phone cameras to understand what makes them tricky to control.

2.1 Characteristics of mobile image sensors

In a conventional camera, the exposure is controlled by opening and closing a mechanical shutter, allowing photons to strike the sensor (or film) for a brief period of time. By contrast, the cameras used in mobile devices generally do not have a mechanical shutter for space and cost reasons, and instead rely on an “electronic shutter”. That is, the pixels are electronically reset at the beginning of the exposure, and gather light until the pixel value is read out. There is nothing to stop pixels from collecting photons, so the exposure duration is simply the time between reset and readout.

However, reading out all of the pixels takes a significant amount of time — somewhere around 20 milliseconds, depending on the specific sensor and the size of the

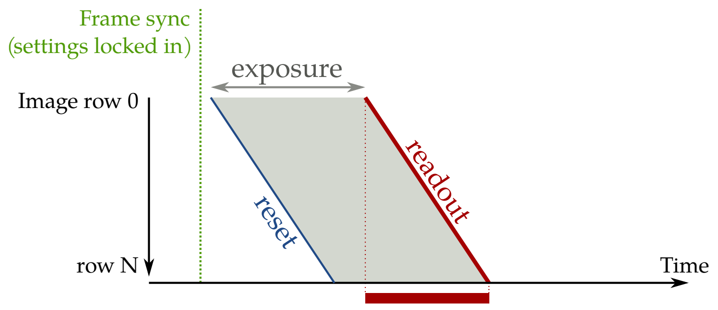


Figure 2.1: Conceptually, a rolling-shutter camera uses a reset and readout pointer, which are steadily incremented in lockstep down the image. This results in each row having the same exposure duration, but each row captures a slightly different slice of time.

image being read out. During the readout time, rows of the image are read out sequentially from top to bottom as illustrated in Figure 2.1. Since image rows are read out at different times, they must also be reset at different times to keep the exposure duration uniform across the frame. This scheme is known as "rolling shutter", because the shutter action is not a single event but rather operates in a rolling fashion across the sensor [17].

In addition to simplifying the mechanical design of the camera and eliminating the need to store pixel values until they can be read out, rolling shutter allows higher continuous framerates than a comparable global-shutter sensor. As shown in Figure 2.3, it is possible to partially overlap the rolling shutter readout with the exposure, with the result that the sensor hardware spends less time idle than in a global shutter setup (Figure 2.2).

This also presents a challenge: when capturing shots repeatedly (such as for a viewfinder or video), most CMOS sensors are free running. That is, the sensor is not synchronized to anything else in the system, and produces a stream of image data without it being explicitly requested. And because of the overlap, it is possible that the next frame is already exposing by the time the current frame is finished reading out, depending on the exposure duration and inter-frame blanking time. Any configuration settings such as the exposure time and gain must be applied at precise

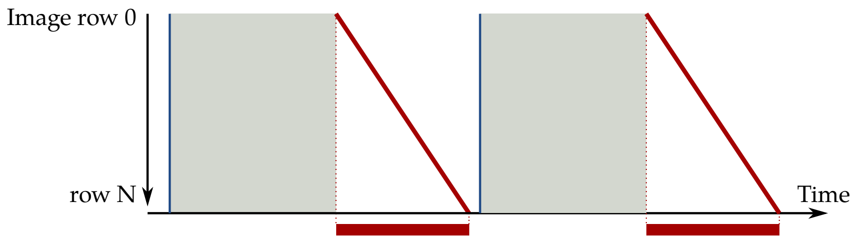


Figure 2.2: Sequential frames captured with a global-shutter sensor. Every line of the image captures the same point in time, but the pixel values must be stored until all of the data can be read out.

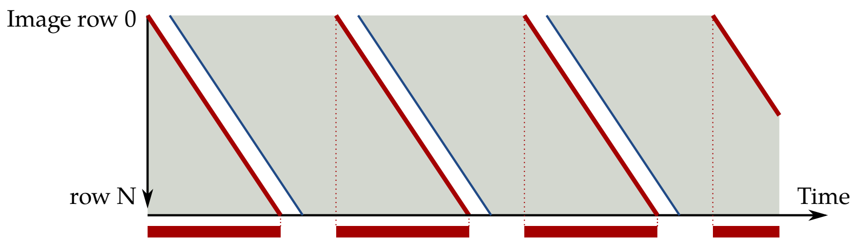


Figure 2.3: Overlap of exposure and readout for a rolling-shutter sensor. Once a row is read out, it can be reset and begin exposing the next frame while the remaining rows of the current frame are read. This results in less idle time and higher framerates than an equivalent global-shutter sensor capturing with the same exposure.

intervals when they will not affect the image capture currently in process. In some sensors, changing the settings in the middle of an exposure causes part of the image to be captured with the new settings, producing an abrupt variation in the middle of the image.

Other sensor options introduce even more timing intricacies, particularly those which control the readout such as pixel binning, image size, or readout area. For example, switching to a 2× binning scheme where four pixels are combined to produce a single value requires that the pixel clock be reset and reconfigured for this double readout, causing a stutter in the regular timing of the image stream. The result of all this is that merely keeping up with the sensor is a substantial task, to say nothing of the frame-to-frame adjustments we need for computational photography.

To alleviate this burden, many image sensors include one or more hardware conveniences to simplify particular aspects of the timing and coordination problem. Some sensors only apply settings at the end of each frame, so changes to the registers only apply to the subsequent frame and cannot corrupt a frame currently being captured. Additionally, some (such as the Sony IMX219, which will be discussed in more detail later) have two sets of frame settings registers, so that one set can be configured while the other is currently used [18]. A single register write swaps the active set of registers. This makes register settings atomic, ensuring that either all of the new settings are applied or none of them are.

Other sensors drive a special flash trigger pin which pulses to synchronize an external flash with the image capture, or can trigger their capture based on an input. Some sensors also have a “fast switch” option where the sensor interrupts the exposure it was currently taking, sets the reset pointer back to the top of the image, and begins an exposure with new settings [19].

Even with all of these aids, however, the camera pipeline remains difficult to control. The remainder of this chapter describes systems and abstractions that have been developed to make the capture and processing pipeline more manageable.

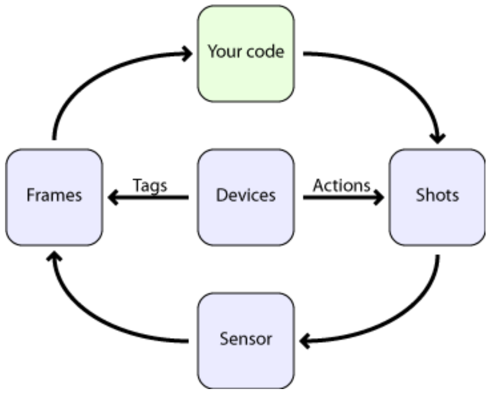


Figure 2.4: FCam model: User code specifies `Shots` and passes them to a `Sensor`, which asynchronously returns `Frames` containing the image data and metadata `Tags` describing what actually happened during the capture. Image from [22].

2.2 FCam

The FCam API [20, 21] was created to enable precise control of the camera capture process on the unwieldy pipelines we have just described. FCam is designed around the observation that the camera capture pipeline is just that — a pipeline. Because it is a pipeline, it operates most efficiently when there are multiple images “in flight” at any given time, but traditional stateful APIs (set some settings, take a shot, repeat) make this rather difficult. At the same time, users at the application level do not care about the state of the pipeline at any given time; they simply need to capture images with various settings.

Therefore, instead of specifying the camera’s behavior in terms of its settings and state, FCam allows a user to specify a set of shots which should be captured. A runtime engine takes these shot requests and handles all of the implementation details to capture the shots correctly. Since the user can specify a whole sequence of images at once, there is no need to wait for one image to finish before the next one begins, and the pipeline can run as rapidly as the hardware will allow.

The core objects in FCam and their interactions are shown in Figure 2.4. As described already, the user specifies one or more requests encapsulated in the `Shot` class. Each `Shot` contains information such as the exposure time for the shot, the analog and digital gain to apply, and whether the resulting data should be passed through the ISP or returned immediately as raw pixel data.

The runtime asynchronously returns frames, which are the result of executing shot requests. A `Frame` contains the resulting image data, but also the original shot request and some metadata describing what actually took place. In some cases it is impossible to satisfy the shot request exactly, either because the user specified something that the hardware does not support, or because the timing at that moment does not allow it. In these cases, FCam makes its best effort to capture the shot and returns whatever information it can about what happened. The user program can examine the resulting metadata and decide what to do.

To handle camera components other than image sensors, FCam introduces the `Device` and `Action` objects. A `Device` represents a physical hardware device such as a lens, flash, or gyroscope. An `Action` is a request that a device perform some action, such as a flash firing or a lens moving to a particular position. These actions are bound to shot requests, and as each shot is captured the system executes the corresponding actions on the appropriate devices. Since actions are bound to shots, the user can specify when they should take place relative to the beginning of the exposure. In this way, each `Shot` specifies a micro-timeline of events: Move the lens 5 ms before the start of the exposure, begin a 20-ms exposure, fire the flash 10 ms after the start of the exposure, and so forth. Like shots, actions are performed on a best-effort basis: the camera makes an attempt to satisfy the request, and then returns a data structure describing what actually took place. Actions may also append other useful metadata to the frame. For example, an inertial measurement unit (IMU) device can be attached to a `Shot`, and this adds a set of metadata tags to the `Frame` describing the orientation of the sensor during the exposure.

The runtime engine is encapsulated inside the `Sensor` object. Shot requests are queued up with the `Sensor.capture()` method and are captured asynchronously by worker threads running in the background. The resulting frames are retrieved later with `Sensor.getFrame()`.

FCam acknowledges the streaming nature of image sensors and provides the `Sensor.stream()` method to stream a particular shot or sequence of shots, capturing it repeatedly as long as there are no other shot requests with a higher priority. This allows the user to keep the pipeline full and the sensor busy all the time, even if the application-level code is busy with another task.

FCam provides a clean and expressive abstraction for controlling the image capture process, and dramatically reduces the real-time burden of controlling camera pipelines. By recognizing and exploiting the pipeline nature of camera systems, FCam facilitates a whole host of computational photography applications that would otherwise require building a specialized camera system from the ground up.

However, there are several limitations to the FCam model. First, FCam was designed with a single camera in mind, and the original implementation did not have any explicit support for multi-camera systems. It is possible to instantiate multiple sensors within FCam, but they operate as completely independent devices rather than as two components of a unified camera system. To remedy this, Troccoli et al. introduced a set of extensions for a multi-camera system [23]. In their proposed API, the platform can enumerate all of the sensors connected to it, along with their physical locations and orientations on the device. Multiple identical cameras can be grouped together in a logical "sensor array" which share a shot queue, making it possible to queue up corresponding arrays of requests to be executed simultaneously. Finally, they introduce a basic synchronization object which allows semaphore-like synchronization between devices which are not part of the same sensor array.

As with multiple cameras, FCam has limited support for timing multiple shots on the same sensor. The user can specify a `frameTime` for a shot, which (at least in theory) controls the duration from the start of one shot to the start of the next. This is sufficient to set the framerate for a stream of requests but becomes unwieldy for more complex timing scenarios, particularly when the requests have to be modified due to the timing constraints of the sensor hardware.

Finally, all customized image processing happens outside the FCam pipeline. This is not so much a drawback as an intentional limitation of scope, but due to the recent innovations in programmable image processing hardware, capture and processing can no longer be treated independently. FCam provides a mechanism to specify what processing should be done in terms of the basic options of a conventional ISP: either process the image, or don't. As the previous chapter argued, that landscape has become far more complex and we need a more expressive way to specify the processing alongside the capture.

Because processing happens in user code outside the FCam pipeline, it is difficult (or impossible) to build tight feedback pipelines. For example, an autofocus loop ought to evaluate the sharpness of the images coming out and use this information to control the lens position in a feedback loop. Because the FCam pipeline is completely opaque, this feedback has a latency of two frames or more. The autofocus examples in FCam instead perform an entire focal sweep and pick the sharpest frame.[1]

Perhaps unsurprisingly, FCam is also limited by the platforms it runs on. The initial implementations ran on top of the Video4Linux2 (V4L2) API, which facilitates low-level access to various video devices on Linux platforms. Certain operations which are perfectly reasonable and easily expressed in FCam, such as capturing shots at different resolutions, are unsupported by V4L2 and required drastic workarounds like closing the V4L2 device and reopening it at the new resolution. Another more recent and important example of platform limitations comes from the Android mobile operating system, which we discuss next.

2.3 Android Camera2

Android's "Camera2" API [24] is essentially an implementation of FCam in Android, which takes the key ideas and abstractions from FCam and adapts them to the features and constraints of the current Android operating system. This implies both some restrictive limitations, but also a number of additional features.

Because Android must run on a large and diverse collection of hardware, it also imposes some limitations which make parts of Camera2 less expressive than the corresponding elements of FCam. Unlike FCam, it is not possible to time lens or flash actions relative to image captures. In fact, although most of the objects in FCam have

[1] At the time, even this level of control was an important innovation — one student wrote an autofocus routine that was better than the stock implementation on the Nokia N900.

direct correspondences in Camera2, the `Device` and `Action` objects were removed entirely. Instead, the various controls for the lens, flash, and so forth are specified as options on the shot request. The result is that it is possible to specify whether the flash should fire or not, but the timing and duration of the flash cannot be controlled [25].

This simplification makes sense for Android, where the peripheral set is generally limited to a lens and a single flash, and where there are hundreds of different hardware devices attempting to support the API. Unfortunately this also pushes many computational photography algorithms out of reach of application code and into the domain of device-specific subroutines.

On the flip side, one added feature is support for multiple cameras: following the ideas introduced by multi-camera FCam [23], application code can query all of the available cameras on the phone, and each one reports the position and orientation of the image sensor relative to the phone's global coordinate system. With the introduction of "logical cameras" in Android Pie [26], it is also possible to request and process streams from multiple cameras, with at least basic support for synchronization [27]. Camera2 also adds more output formats and more complete representations for various camera configurations common in mobile devices (e.g., fixed-lens EDOF cameras), as well as extensive mechanisms for querying the capabilities of the hardware.

Finally, and perhaps most importantly, Camera2 integrates capture with the Android media pipeline. Unlike FCam, where the resulting frame data is handed back to user code, Camera2 allows the application to specify one or more output "surfaces" as targets. The image data is forwarded directly to these targets without further intervention from the application, allowing full-framerate preview and capture without the overhead of returning to user code. This kind of delegation by linking outputs to targets points to a more general theme in multimedia processing, which we turn to in the next section.

2.4 Media streaming frameworks

Multimedia capture and playback is an important part of many applications, from video chat to playing movies to editing video. These use cases have many overlapping components (such as sharing the same set of video codecs) and need to be designed with performance in mind in order to support the high data rates of streaming video. As a result, every major operating system today includes a media streaming framework that provides an API for capturing, encoding, reading, decoding, and displaying video and audio streams.

Microsoft develops DirectShow for Windows [28], Apple has AV Foundation for iOS and macOS [29], Android's user-facing APIs wrap around the Stagefright library [30], and Linux systems predominantly use GStreamer[31]. OpenMAX ("Open Media Acceleration") is an API developed and hosted by the Khronos group which aims to be a standard multimedia framework for mobile devices [32]. These systems each have their own peculiarities, but the functional core is very similar. In all of these systems a multimedia application is built from a set of "nodes," each having a set of ports which produce or consume data. The nodes are linked together to form a graph, and the entire graph is executed by the framework. The terminology differs between frameworks, but the basic components are universal:

	Node	**Graph**	**Port**
GStreamer	Filter	Filter graph	Pad
DirectShow	Filter	Filter graph	Pin
AVFoundation	AVCaptureDevice, AVCaptureOutput	AVCaptureSession	AVCaptureInputPort
OpenMAX	Component		Port

An important aspect of these frameworks is that nodes are interchangeable. It is straightforward to exchange a camera source for a file on disk, or to swap one media codec for another. For example, most of GStreamer's useful functionality is implemented as a set of plugins which can be swapped in or out, or replaced entirely by user-defined components. Likewise, because Android uses OpenMAX IL under the hood, phone manufacturers can integrate their hardware with Android by implementing media-handling hardware modules as a set of OpenMAX IL components.

This thesis borrows and extends several ideas from these frameworks. The stream processing API which is introduced in Chapter 3 is also based around a set of data-processing nodes linked together to form a graph. Likewise, some of the implementation details of the system are inspired by the optimized implementations in these other frameworks.

These media streaming frameworks also share some common limitations. Because they are designed for feedforward processing of multimedia streams, they generally do not have facilities for dynamically controlling an input device (e.g., driving a camera the way FCam is able to) or for feedback within the pipeline.

In the academic space, Alexandre François published a series of papers in the early 2000's describing a multimedia architecture called SAI ("Software Architecture for Immersipresence") and a runtime implementation called MFSM ("Modular Flow Scheduling Middleware") [33, 34]. François argued that traditional multimedia frameworks structured around "pipes and filters" (such as those just described) are inadequate for emerging interactive multimedia applications, particularly because they do not permit sharing data between components except in a streaming fashion. This makes it difficult to implement systems which maintain state as part of their processing, or which require communication between multiple pieces of the algorithm.

Based on the observation that data in such a system can be categorized as either *volatile* or *persistent*, SAI uses a hybrid data model which includes both dataflow message-passing and persistent shared data. Data is organized into temporally-sequenced *pulses*, which maintains temporal coherence of data while breaking up the FIFO pipes model for improved parallelism.

Although multimedia processing has evolved significantly since the creation of SAI, its basic critique is still valid: the pipes and filters model by itself is a poor choice for handling interactivity and feedback. In the following chapter, we apply the notion of volatile and persistent data to our camera system.

2.5 OpenVX

Although most multimedia graph frameworks are focused on capturing, transcoding, and playing video streams, one graph framework that has focused on processing is OpenVX, also from the Khronos group [35]. At a high level, the aim of OpenVX is to solve the problem of how to write portable image processing code and run it on hardware with varying acceleration capabilities — in short, to be for computer vision what OpenGL is for graphics.

Users write their computer vision applications against the OpenVX API, and hardware vendors (Intel, NVIDIA, Synopsys, and half a dozen others [36]) create implementations of the API for their hardware. Like OpenGL, the OpenVX API enables the application to be (mostly) hardware-agnostic, and to transparently take advantage of whatever hardware resources are available.

The design of OpenVX is based on an enumeration of important image processing operations from OpenCV, which are defined as "kernels" in OpenVX. Applications are constructed by linking together these kernels into graphs, which are then executed by the vendor implementation. Because the application passes the complete graph to the runtime system, it is possible for the implementation to statically schedule the graph, manage memory, and possibly perform cross-kernel optimizations to improve the overall efficiency of the application.

A number of projects have examined various compilation and scheduling techniques for OpenVX graphs [37, 38], and a few have specifically examined compilation of OpenVX kernels into hardware. JANUS [39] uses a library of OpenVX kernels implemented in Vivado HLS to produce FPGA designs, and can maximize the throughput for a given FPGA resource budget by transforming loops and fusing kernels within an heuristic-guided optimization. AFFIX [40] is a similar project for Intel FPGAs which implements OpenVX as a library of OpenCL kernels. The AFFIX tool performs a series of optimizations on the compute graph, partitions the graph into CPU and FPGA sections to minimize storage and data transfer, and finally uses an OpenCL compiler to generate the FPGA bitstream and CPU binaries.

Unfortunately, OpenVX's "fixed-kernel" approach imposes a number of limitations, both algorithmically and performance-wise. Computer vision is an extremely fast-moving field, and it is unrealistic to expect that a fixed set of kernels is sufficient for algorithms emerging over the next few years.[2] Most applications do use some common kernels, but also combine this with some "secret sauce" — custom code or algorithmic tweaks that make the application function. Recognizing this, OpenVX allows user-defined kernels written in OpenCL or other languages, but this muddies the collection of clean and easily optimized kernels.

The performance limitation comes from the fact that fast kernels don't necessarily make a fast algorithm. As we'll discuss in more detail in the following section, many speed improvements come from inter-kernel optimization: tiling the image and fusing kernels to improve locality. Again, OpenVX attempts to address this by allowing the implementation to optimize the graph, but there is no straightforward way to integrate custom kernels into this optimization. Recognizing this problem, Özkan et al. use the image-processing domain-specific language (DSL) Hipacc as a backend for OpenVX [41] and add support for custom OpenVX nodes written in Hipacc, which enables proper optimization of the entire processing graph. This is the approach we describe in the next section with the Halide DSL.

Finally, like some multimedia frameworks, OpenVX has no notion of cameras or live inputs; this is left to other system APIs. There was a proposal for "OpenKCam" which aimed to be like FCam for the Khronos OpenGL/OpenVX/OpenMAX ecosystem, but it never materialized [42].

2.6 Halide

Compared to OpenVX, the Halide image processing language takes a more flexible approach to creating high-performance image processing applications [43, 44]. Halide begins with the observation that image processing code optimized by experts generally uses a small toolkit of optimizations applied in various patterns, including

[2]Only a few years after its introduction, OpenVX is already overshadowed by neural-network frameworks such as Caffe and Torch.

vectorization to make use of SIMD hardware, tiling for cache locality, loop unrolling to uncover additional parallelism, and multithreading to use multiple cores.

Instead of providing a set of pre-optimized kernels as OpenVX does, Halide provides a way to describe an entire algorithm and then rapidly experiment with combinations of these common optimizations. The central idea in Halide is to separate the result to be computed (the *algorithm*) from the order in which the computation is performed (the *schedule*). The *algorithm* is coded in a functional (side-effect free) language which specifies the value for every pixel coordinate in terms of functions of the inputs. The *schedule* then specifies the storage and compute order. Because the schedule is specified concisely and separately from the algorithm, it is possible to rapidly experiment with optimizations to find the best implementation. Moreover, there have been several recent and ongoing attempts to automatically create optimal (or near-optimal) schedules using some basic information about the hardware [45].

While Halide is a useful tool for optimizing CPU code, it particularly shines on the task of porting algorithms to specialized accelerators. Because all of the storage and ordering directives are contained within the schedule, the algorithm can remain unchanged, and the only thing that must be rewritten for the target architecture is the schedule. The main Halide codebase contains backends for CUDA and OpenCL-capable GPUs and Qualcomm Hexagon DSPs [46]. Pu [47, 48] recently developed a new backend for Halide which compiles a subset of Halide programs into efficient designs for FPGAs using a line-buffered pipeline template.

Halide is focused specifically on the task of creating fast routines for processing images, and as such it does not address a number of system-level concerns. For example, Halide operates on image streams as simply a sequence of frames, and does not contain any notion of state. Likewise, Halide is not concerned with where its images come from or where they go; these things are left to a higher-level system.

To summarize, Halide solves the problem of creating a high-performance implementation of an image processing algorithm for a particular piece of hardware. It intentionally does not solve the problem at the next conceptual layer: creating a complete high-performance system for capturing and processing images. The remainder of this thesis approaches this latter problem, using Halide as one building block.

As we'll describe in the following chapters, we leverage Halide as a kind of "shader language" for processing images within our system. We use the Halide-to-FPGA flow from Pu [47] to produce executables for our platform.

Chapter 3

Camera system abstraction

Each of the languages and APIs described in the previous chapter addresses a different variation of the same challenge: we need interfaces which enable both high developer productivity and high runtime performance. This is particularly true for mobile devices, which have numerous tasks with real-time constraints (inertial tracking and processing, multimedia streaming and playback, wireless communication, and more) and which must present all their features to application developers via a convenient software interface.

The solution repeatedly applied in these domains is to delegate the work to a separate low-level thread, which is designed and executed with timing and hardware constraints in mind. In many cases, this thread runs on a completely separate processor where it can operate in hard real time isolated from the actions of the rest of the system. Wireless modems have their own processors, audio and video codecs have their own hardware, and motion coprocessors are now built into MEMS inertial measurement units [49].

This is possible because the development and manufacturing cost of adding another processor to a system-on-chip (SoC) is almost zero — at least relative to the cost of producing the SoC in the first place. IP designs for simple processors are inexpensive or even free [50, 51], and they take only a tiny fraction of a square millimeter of silicon on a modern IC process [52]. As Moore's law has marched forward, even the central quad-core (or even octa-core) processor complex on a typical SoC has been reduced to a modest fraction of the total die size, with the rest filled in by GPU cores and a host of specialized coprocessors [53].

This work builds on the assumption that the hardware cost of adding a small co-processor is negligible, and thus that it is feasible to integrate dedicated a camera co-processor which orchestrates image capture and manipulation in order to achieve fine-grained control.

However, if the low-level work is delegated to a separate subsystem (and especially a whole coprocessor), then it is essential to have a clean and expressive software interface to delegate work. FCam, Camera2, OpenMAX, and OpenVX all attempt to do this for their respective domains, albeit with some limitations. The rest of this chapter describes a new API called F4graph which is tailored for the task of image capture and processing on multi-camera, heterogeneous processing systems.

Specifically, F4graph is designed to solve two problems: First, it allows an application to request that the hardware execute a set of actions with time constraints, while also satisfying the constraints of the individual hardware devices and the overall system. Second, it allows an application to make use of specialized hardware accelerators connected directly to the camera for streaming applications. By enabling the user to define the processing along with the capture requests, the capture-processing pipeline can run without additional intervention. Further, it allows the application to describe iterative and feedback algorithms so these can run continuously without intervention.

To achieve these goals, the F4graph API is designed to balance three objectives in tension. First, it aims to provide the developer with as much "specification power" as possible. That is, the application author should be able to control the cameras and peripherals as precisely and flexibly as we can allow. At the same time, we want to minimize the chance that the developer will request a set of actions which cannot be executed on the hardware due to resource limits or conflicting constraints.

These first two objectives are in competition: A very restrictive API would allow all requests to be correct by construction and eliminate the potential for errors altogether, at the risk of being useless for all but trivial applications. A wide-open API gives programmers ultimate flexibility and control, but exposes all sorts of pitfalls, which the user would have to discover the "hard way". Section 3.1 explores this tension and our approach to solve it, and Section 3.2 describes the basic constructs in our API.

The third and final objective is to provide hard guarantees about the system behavior given the uncertainties of a complex multitasking hardware/software system. At what point are requests guaranteed to happen? Are timing requests ever ignored, and if so, under what circumstances? What happens when the system runs out of memory? Again, these come into tension with the first objective. The less control is ceded the user, the fewer promises the system is responsible for keeping — but a system which makes no promises is hardly better than one that makes promises but doesn't keep them. Section 3.3 discusses this second tension after the core API objects have been introduced.

3.1 Scheduling requests

We will formalize the term "request" to mean any request that the software makes of the hardware platform. This is a generalization of the "shot requests" and "actions" in FCam, both of which are programmatic requests that some action be executed on the camera hardware. These are "requests" rather than "demands" since it may not be possible to fulfill the request exactly, or even to execute it at all. This tension between what can be specified and what can actually be achieved is precisely the challenge we must address.

There are many possible ways to programatically specify the ordering and timing of requests, but they each work by choosing some set of constraints which user code can specify. From most restrictive to least restrictive, these constraints are:

- **Exact time**: Specify the exact time that a request executes in terms of some global time system (e.g., the system clock).
- **Exact delay**: Specify exact nonzero delays between the execution of two requests.

- **Synchronization**: Two or more requests execute at exactly the same time (presumably on different hardware modules). This can be thought of as limited special case of exact delay, where the timing delay between two requests is zero.
- **Ordering**: Specify the order in which requests should be executed.

At the same time, the system should provide as strong a guarantee as possible that the specified schedule can actually be achieved. We will consider three different guarantees.

- **Accommodate present occupancy**: The system can find a schedule that fits around the current occupancy of the hardware, or even reschedule if preempted by higher-priority tasks.
- **Preserve timing after failure**: When something goes wrong (e.g., a request took longer than expected), timing relationships between remaining requests are preserved.
- **Always valid**: All schedules which can be specified are valid. That is, the nature of the scheduling specification is such that there is always a valid schedule which meets the specification while conforming to the constraints of the hardware.

The benefit of the first and third guarantees should be apparent, but the second deserves more discussion. In particular, is it even necessary to preserve temporal relationships after a timing failure? An alternative would be to simply require that no event fail, i.e., that all requests complete in a deterministic amount of time. With this requirement, it would be possible to fully calculate the schedule and resolve all constraints before committing any of it to execution. Every schedule, once running, would complete exactly as it was specified, barring a catastrophic hardware or system-level failure.

Unfortunately, the requirement that all requests complete in deterministic time is quite restrictive. There are cases where the request execution time depends on an external event, such as capturing a shot when a button is pushed or triggering a flash based on a sound sample or external trigger signal. Triggering on external events is a powerful and important feature, but it inherently introduces non-determinism.

Additionally, many instances of continuous capture or metering require that a software function process one set of results to generate the parameters for another image in the capture sequence. If this software is not designed for real-time performance or runs on a multitasking OS, it is impossible to make any guarantees, even if it works in practice.

A third scenario which induces non-determinism is running multiple schedules simultaneously, such as capturing an image while a viewfinder is running. Interrupting the stream of viewfinder requests to take a shot breaks the deterministic timing of those requests. We could maintain determinism by pausing the viewfinder schedule before launching the capture schedule, but this pushes the burden onto the user and introduces overhead. Instead, we will allow such interruptions and develop a scheduling system which can handle them gracefully.

We next consider seven possible designs for a scheduling API, discussing both the capabilities each design allows the user and the guarantees that the system can provide under each one. This discussion is summarized in Table 3.1.

	Fixed timeline	Moveable timeline	Constraint-based	Order + delay (FCam)	Semaphore (multi-thread ordering)	Ordering	No constraints
hit exact timepoints	✓						
exact (nonzero) delays between requests	✓	✓	✓	✓			
synchronize requests	✓	✓	✓				
order on multiple devices	✓	✓	✓	(actions only)	✓		
order requests	✓	✓	✓	✓	✓	✓	
preserve timing after failure			✓	✓	✓	✓	N/A
accommodate current busy-ness		✓	✓	✓	✓	✓	✓
all schedules are valid						✓	✓

Table 3.1: Possible scheduling choices with their corresponding capabilities and guarantees.

The scheduling options lie between two extremes: at one end, the user specifies exact timepoints for every request; at the other, the user cannot put any timing constraints on requests at all. Specifying exact timepoints allows a very simple interface: a single command enables every possible schedule. Order, relative delays, and exact timepoints can all be specified simply by pinning every request onto an absolute timeline. Of course, giving the user this power means that the system cannot make even basic guarantees. The user could easily specify requests that conflict with each other, or which conflict with requests already queued up for the hardware. And since the interface does not embed any notion of timing relationships between events, it is impossible to preserve them, even if they exist in the user's mind or code.

At the other extreme, the system could prevent the user from specifying any timing constraints. With this restriction, the system can easily guarantee that all schedules are valid and that every request will eventually be executed. The runtime has total freedom to rearrange, reschedule, and optimize requests to execute in the most efficient manner possible. However, the inability to specify the timing of requests means that the system is useless for many practical applications. To continue our survey of possible APIs, we'll incrementally add more user control.

One option is to use only a simple ordering constraint: the system executes requests in the order in which they were enqueued. This still leaves flexibility for the system to work around the current and future occupancy constraints of the platform, and delays due to non-deterministic requests can easily be accommodated without breaking the ordering.

However, this simple ordering model has two shortcomings: first, there is no option to time requests relative to each other, such as specifying that a flash should fire simultaneously with a frame capture. Second, it is unclear what an "ordering" even means for more than one device. A strict linear ordering (specifying that a request does not execute until the previous one is finished, regardless of which device it is on) would be a valid definition, but does not allow synchronization or multiple devices operating in parallel.

We might solve this multiple-device issue by specifying an ordering of requests for each device, and then allowing inter-device synchronization. This bears remarkable

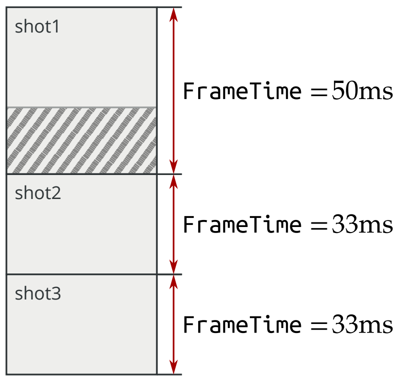


Figure 3.1: Order+Delay scheduling, used by FCam and Camera2. Requests are executed in sequence, and the duration of each request is controlled by the `FrameTime`.

similarity to the problem of multicore software scheduling, which suggests that the synchronization primitive we want is a semaphore. A semaphore provides a particular kind of synchronization: when process X reaches the synchronization point, it does not continue until process Y sets the semaphore. This solution is proposed by Troccoli et al. in their extension of FCam for multiple cameras [23]. This one-way synchronization covers many cases in a camera system, such as waiting until until a lens finishes moving before capturing the next frame. However, it does not address other cases, such as scheduling two actions to occur at precisely the same time or with some fixed time offset. [1]

The FCam API (and Camera2) add a different extension to the simple ordering model to provide timing control. Shot requests are queued up either individually or as ordered groups (vectors) of shots, but each shot request additionally contains a `FrameTime` (renamed `SENSOR_FRAME_DURATION` in Android Camera2) which controls the duration of the request. That is, in addition to specifying the order, each request specifies the time from the beginning of the request until the beginning of the next request. The overall timing of the sequence can be controlled simply by setting the `FrameTime` of each shot in a sequence, as shown in Figure 3.1.

This scheme is conceptually straightforward and easy for the developer to specify.

[1]As with software semaphores, there are also potential problems with deadlock, but these can be prevented as long as the user is prevented from creating cycles in the constraint graph.

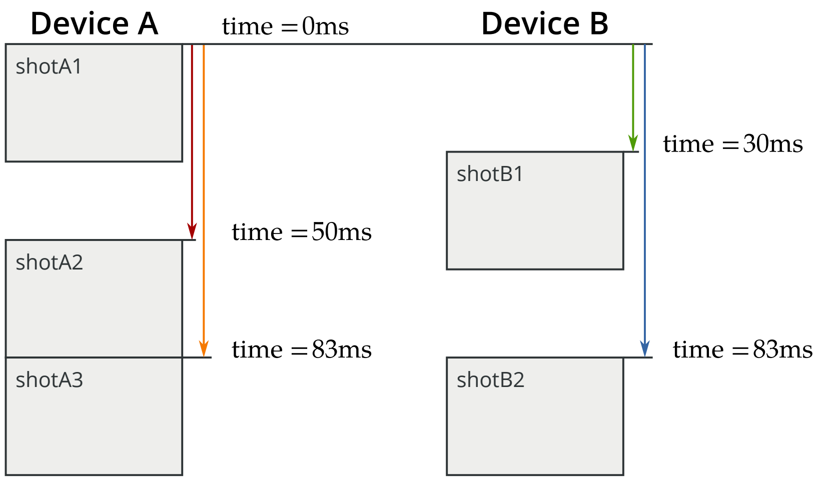


Figure 3.2: Relative timeline scheduling, where requests are scheduled using a timeline relative to the start of the entire capture.

Additionally, because it does not contain any reference to absolute time, it neatly enforces the limitation that the hardware may be busy for some time into the future. Unfortunately, the `FrameTime` only works for sequences of requests on the same device, and does not offer any way to control the timing between multiple devices.

It seems natural to combine the semaphore approach with FCam's ordering and delay mechanism, providing more control than an ordering alone. However, this still does not allow the user to specify timing relationships between devices except for the one-way "wait until" synchronization described earlier.

One way to achieve both a consistent global ordering and inter-request timing would be to specify the time of each request relative to the start of the sequence, as shown in Figure 3.2. The relative timing of requests is fixed, but the entire timeline can be shifted to accommodate hardware constraints. For a single camera, this is functionally equivalent to the FCam model: both methods create a timeline and attach timepoints to each request. But specifying times rather than durations makes it easy to link together requests across multiple devices, and removes an additional quirk of the `FrameTime` model. When sequencing shots in FCam, each one is temporally contiguous with those before and after it, and all intervals must be created by stretching out some requests to waste time. For example, to capture a time-lapse sequence with one image every 10 seconds, the user code must set the `FrameTime` to 10 seconds, which (at least conceptually) blocks the sensor for that entire time. By specifying relative times, the user can express the same intent, without the side effect of limiting what happens in the interim.

Unfortunately, the “relative timeline” approach is quite brittle in the face of timing errors. Although it does offer some flexibility for finding an open time window, the failure of any time constraint is likely to cause subsequent ones to also be delayed in domino progression. FCam’s “frame time” scheduling has the small advantage that, in the event of failure, only one timing relationship will be broken at a time. Because each request is timed relative to the one before, one delay does not result in an immediate cascade of failures.

This observation suggests an improvement to the “relative timeline” scheme: rather than specifying all of the timepoints relative to the beginning of the first request, we can allow the user to specify times relative to any other request. If some request cannot be executed at the desired time, then at least the requests scheduled relative to it can be moved to preserve their timing relationships, as illustrated in Figure 3.3. This scheduling method gives the user near complete control of timing while allowing the system to accommodate the hardware constraints and preserve as many timing relationships as possible after a failure.

One apparent drawback to this “relative constraint” scheme which is avoided by the FCam model is that it is possible to schedule multiple requests such that they interfere with each other, for example by scheduling two requests on the same device at overlapping intervals. However, some variation of this conflict is almost inevitable if we allow synchronization across multiple devices.

To understand why, it is helpful to consider schedules as graphs, where each node is a Request and edges are timing relationships between requests. Scheduling conflicts occur when a request has multiple competing constraints. As long as there are no undirected cycles in the graph, then the system cannot be overconstrained.

We could restrict the user to specifying only a single timing constraint per request, forcing the graph to be a tree, and many practical schedule graphs do in fact look like trees. However, it is not only user-defined constraints that could be the problem:

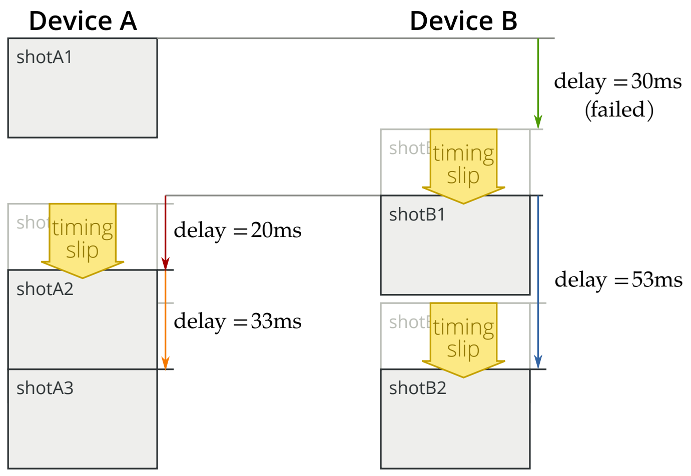


Figure 3.3: Constraint-based scheduling, where requests are scheduled relative to other requests, not necessarily to the beginning of the capture.

device utilization also imposes constraints, and these must coexist with the user constraints. An example in shown in Figure 3.4, where two synchronization constraints force a timing conflict.

For these reasons, it is unrealistic to construct an interface that is useful while maintaining restrictions such that all schedules are correct by construction. The question instead is how to manage these failure cases. In F4graph, this happens in two phases. As the schedule is being built, the system can construct the constraint graph and check for any conflicts based on the known timing parameters. This catches

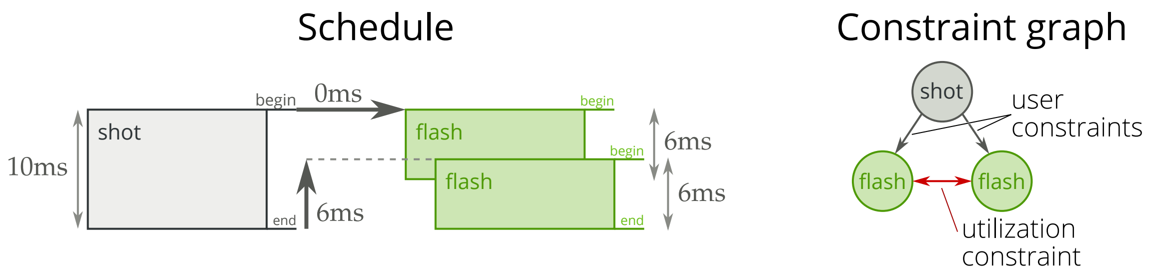


Figure 3.4: A shot on an image sensor and two flash requests on a flash, synchronized to the start and end of the exposure. The constraint graph would be a tree if we considered only the user constraints, but there is an additional requirement that the flashes not overlap.

errors such as scheduling two requests on the same device with overlapping times. In the less common cases where the timing cannot be statically determined (because it depends on external events or non-deterministic processing), the system checks for errors while the schedule runs, completes the schedule on a best-effort basis, and provides error handlers and metadata to inform the user.

Having laid out the basic scheduling model, we now formalize it and introduce the API objects and operations that are used to define requests and specify schedules.

3.2 Scheduling constructs

To implement this constraint-based schedule, F4graph defines a set of software objects (`Device`s, `Request`s, `Event`s) and provides methods for linking together their timing relationships.

Basic objects: Requests, Events, and Devices

Our notion of a "request" is formalized in the API as a `Request` object. Like the "shot requests" and "actions" in FCam, it represents a request that a device perform some action, such as capturing a shot or moving the lens. Requests produce data as a result of executing: Requests to image sensors produce frames of image data, and requests to IMUs produce samples of the current orientation. Requests also return metadata describing when the action was executed and what parameters were actually applied, which gets passed along with the data. The format and use of this metadata is discussed further in Section 3.8.

`Request`s must be scheduled relative to events of some type; these are represented with `Event` objects. Events are fired by `Request`s or by downstream processing nodes. For example, an event is fired when a image sensor begins exposing a frame, when a timer reaches a particular value, when a lens reaches its target position, or when an image finishes processing.

Similar to FCam and Camera2, a piece of physical hardware in the system is represented by a `Device`. A `Device` could be an image sensor, a lens, a gyroscope, a modulated light source, or some other piece of hardware.

Putting these together, we can write a simple program that captures a single image and saves it to a file:

```
Graph g;
Camera cam; // Camera is a subclass of Device
g.addDevice(cam);

ShotRequest shot = cam.makeRequest();
shot.exposure = 50.0; // 50ms exposure
g.addRequest(shot);

FileWriter file("out.jpg");
g.add(file);
g.connect(shot.output, file.input); // Pipe result to a file

g.execute(); // Run the graph and execute the ShotRequest
```

Creating schedules

With these objects defined, we can begin to compose schedules of Requests. To capture an image with a flash, for example, we create requests for both the image capture (a `ShotRequest`) and the flash (a `FlashRequest`), and then schedule the flash request to occur when the exposure begins:[2]

```
// Set up objects
Graph g;
Camera cam;
Flash flashDev;
g.addDevice(cam);
g.addDevice(flash);

// Create camera and flash requests and add them to the graph
ShotRequest shot = cam.makeRequest();
g.addRequest(shot);
FlashRequest flash = flashDev.makeRequest();
g.addRequest(flash);
```

[2]In this example and in those that follow in the remainder of this section, nothing is done with the data that is captured. The subsequent section describes in detail how to pipe the output of requests to various processing and output nodes.

```

// Fire flash 5ms after shot begins exposing
g.schedule(flash, AT, shot.begin, +5);

g.execute();
```

The key API call in this example is `Graph.schedule()`, which takes four parameters:

- **Request** : The request being scheduled.
- **Relation** : When the request should occur relative to the event; one of `AT`, `AFTER`, or `WITHIN`.
- **Event** : The event which the request is scheduled relative to.
- **Delay** : An optional delay to insert between the event firing and the Request initiation, specified in milliseconds.

In the example above, `flash` is scheduled to occur `AT` the time `shot` begins plus five milliseconds. That is, the flash should fire 5 ms after the sensor begins exposing.

We can synchronize two shots on separate cameras so that both are captured simultaneously using `schedule(shot2, AT, shot1.begin, +0)`:

```
Graph g;
Camera cam1; // Create and add both cameras
Camera cam2;
g.addDevice(cam1);
g.addDevice(cam2);

ShotRequest shot1 = cam1.makeRequest(); // Create and add both requests
g.addRequest(shot1);
ShotRequest shot2 = cam2.makeRequest();
g.addRequest(shot2);

g.schedule(shot2, AT, shot1.begin, +0); // Capture at the same time

g.execute();
```

Because we can create and manipulate timing constraints programmatically, it is straightforward to set up captures with many cameras. For example, to create a Matrix-style bullet-time sequence where a large number of cameras capture the same scene with a slight time offset, we can set up a large array of cameras and schedule requests with small delays:

```
Camera cams[N_CAMS];
ShotRequest shots[N_SHOTS];
for(int i = 0; i < N_CAMS; i++){
  shots[i] = cam[i].makeRequest();
  g.addRequest(shots[i]);
  if(i > 0){
    g.schedule(shots[i], AT, shots[i-1].begin, +0.5); // 500 us delay
  }
}
```

The event need not be the start of the exposure; it could just as easily be the end of the exposure or the completion of a lens motion. In the example below, we move the lens in between capturing two images, waiting for the first exposure to complete before moving the lens, and then waiting for the lens to stabilize before capturing the second.

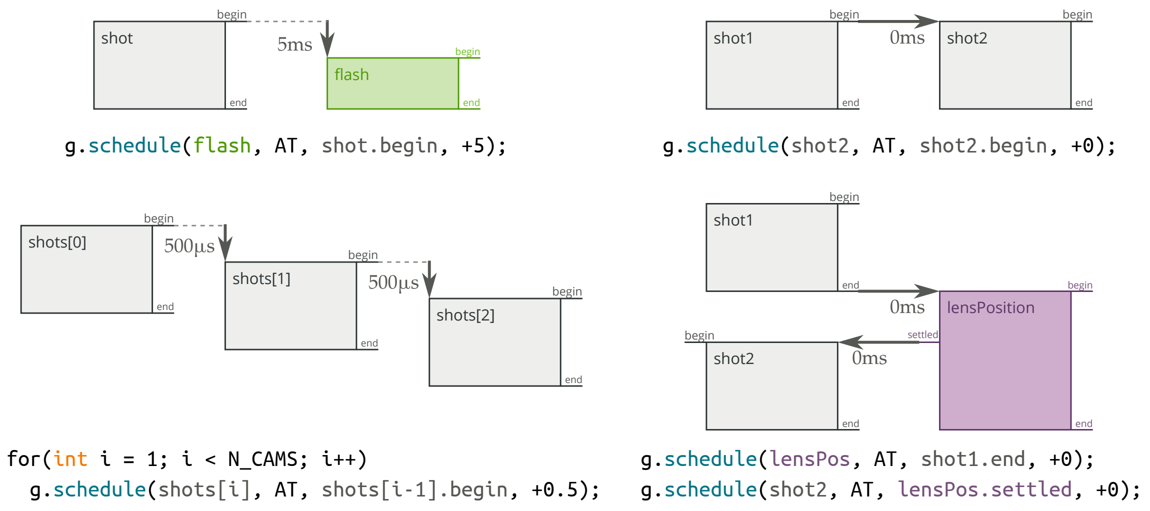
```
Graph g;
Camera cam;
Lens lens;

g.addDevice(cam);
g.addDevice(lens);

// Create two shot requests on the same camera
ShotRequest shot1 = cam.makeRequest();
g.addRequest(shot1);
ShotRequest shot2 = cam.makeRequest();
g.addRequest(shot2);

// We must specify the duration for the lens to hold its position
LensRequest lensPosition = lens.makeRequest(0.8);
g.addRequest(lensPosition);

```



```
g.schedule(flash, AT, shot.begin, +5);
```

```
g.schedule(shot2, AT, shot2.begin, +0);
```

```
for(int i = 1; i < N_CAMS; i++)
  g.schedule(shots[i], AT, shots[i-1].begin, +0.5);
```

```
g.schedule(lensPos, AT, shot1.end, +0);
g.schedule(shot2, AT, lensPos.settled, +0);
```

Figure 3.5: Visual depiction of the constraint graphs for the schedules introduced up to this point. Arrows point from each timing "anchor" to the corresponding request.

```
// Start moving the lens after the first shot;
// take the second after the lens is settled.
g.schedule(lensPosition, AT, shot1.end, +0);
g.schedule(shot2, AT, lensPosition.settled, +0);
```

Each scheduling directive implies a timing constraint between some set of requests, building up a *constraint graph*. This is a useful visual and mental model for describing complex sequences of requests, and will be particularly useful for reasoning about what happens when constraints must be broken. Figure 3.5 visually illustrates the constraint graphs for the code examples introduced so far.

Scheduling relations

As alluded to earlier, devices also impose constraints on the schedule: a sensor can only capture one image at a time, and a flash may need some recharge time between firing. As described in Section 2.1, most CMOS image sensors do not capture images on demand, so the precise time when a particular request can be satisfied is often tightly constrained. For example, if a sensor is 25 ms into capturing a 33 millisecond shot, the next frame may be available in exactly 8ms. If that is too soon, the next available time slot might not be until $8 + 33 = 41$ ms.

To provide flexibility for these constraints, we can specify that a Request execute on an open time interval after an event, rather than after an exact interval. For example, we could rewrite the previous lens motion example more flexibly using the open-ended constraint `AFTER` rather than the exact constraint `AT`:

```
g.schedule(lensPosition, AFTER, shot1.end, +0);
g.schedule(shot2, AFTER, lensPosition.settled, +0);
```

In this application, the intent is not that the lens move at precisely the end of the exposure nor that the second shot be captured the instant the lens settles, but instead that the lens not start moving until the first shot is over, and that the second shot not occur until the lens is settled. An `AFTER` constraint ensures that the latter is true while leaving flexibility for the scheduler to meet other constraints. If it happens that there are no other constraints, the system will execute the request as soon as possible. That is, the `AFTER` relation will behave as if it were an `AT`. By eagerly executing requests, the system preserves as much future flexibility as possible.

Given `AT` and `AFTER`, we might ask whether there is a corresponding "`BEFORE`" relation. The answer is a qualified yes. "`BEFORE`" specifies that a Request must execute before an event plus some time interval, which is not necessarily unreasonable if the delay is greater than zero.

Since in the general case it is not possible to predict when an event will occur, the normal use case for "`BEFORE`" is to set an upper bound on the delay between an event and a subsequent request. Because of this, we instead refer to this "`BEFORE`" relation as `WITHIN`. For example, to schedule a shot to occur within 10 ms of a button being pressed, we could write:

```
g.schedule(shot1, WITHIN, button.press, +10)
```

Said another way, `shot1` should occur before 10 milliseconds have elapsed since the button press event.

Events: static and dynamic

The `WITHIN` relation brings up an important distinction between two types of events. Some (including most of those introduced so far) are *static events*, meaning that they occur at a known time relative to their parent requests. The clearest example of a static event is a Request's `start` event, which fires at exactly the same time as the request begins executing. An `end` event is also static if the Request's duration can be statically determined by its parameters. For example, the duration of an image capture can be fully determined by the exposure and readout times, and the duration of a flash event is specified explicitly. For timing constraints relative to static events, the scheduling problem reduces to finding a time offset between two Request executions.

There are also *dynamic events*, whose execution time is non-deterministic. One example of a dynamic event is an external synchronization signal, perhaps from the system, from a high-precision time source such as a GPS, or from another mobile device. Another example is a node further down the processing pipeline which controls the capture settings. A processing node might evaluate IMU samples at 200 Hz and fire an event when it detects that the camera is still in order to minimize motion blur, or an autofocus block might fire an event when it has processed enough of the image to know which way to move the lens. Similarly, a program to capture lightning strikes might run a detection loop on low resolution images at a high framerate, and fire an event when it detects a pre-strike, launching a Request to capture a full-resolution image. These types of event-based feedback loops are discussed in more detail in Section 3.7.

Delays

Since the firing time of dynamic events is unknown, it is obviously impossible to schedule actions to happen before they occur. To enforce this, F4graph requires all Event-Request delays to be non-negative. In the cases where a negative delay might be meaningful and useful, it is always possible to rewrite the constraint with a positive delay. For example, the constraint

```
// Make A start at least 5ms before B starts
g.schedule(A, BEFORE, B.start, -5);
```

can be written equivalently as

```
// Make B start at least 5ms *after* A starts
g.schedule(B, AFTER, A.start, 5);
```

Constraints to events other than the start are a little trickier, but the same result holds. A constraint to an `end` event, such as

```
// Make A start at least 5ms before B finishes
g.schedule(A, BEFORE, B.end, -5);
```

cannot be rewritten directly, since we cannot set `B.end`. However, we can use the duration of the event to achieve the same result:

```
// Make A start at least 5ms before B finishes
g.schedule(A, BEFORE, B.start, B.duration - 5);

// Or if B is shorter than 5ms:
g.schedule(B, AFTER, A.start, 5 - B.duration);
```

If the event is dynamic and the duration is therefore unknown, doing math with the duration is not possible, which effectively prevents the user from illegally scheduling a Request to occur before a dynamic event.

Streaming and looping

Up to this point we have only considered burst captures, where the Request (or group of requests) is executed once. However, a common use case is to capture a stream of frames, with one or more Requests executed repeatedly. This is the case for a live viewfinder and for capturing video, but also potentially for capture modes which grab data until some event occurs (e.g., continue capturing frames until all the subjects have their eyes open). To enable this, F4graph offers the `executeContinuous()` method. The behavior is the same as the `execute()` method except that instead of running a fixed number of iterations, `executeContinuous()` runs the graph until the `stop()` method is called.

However, simply executing the graph repeatedly is not sufficient; there are instances where it is valuable to constrain the timing from the end of one burst to the beginning of the next. To do this, F4graph includes an additional method, `Graph.repeat()`. This allows the user to specify a timing constraint much the same way as `Graph.schedule()`, except that the constraint is applied between an event in the current execution iteration and an event in the *next* iteration.

This can be thought of as creating a cycle in the timing graph. In principle, it would be possible to achieve the same outcome by simply detecting any cycles in the timing graph, and applying those as repeat constraints. However, unintentional cycles in the graph are highly problematic, so we instead refuse to create any cycles in the graph with `Graph.schedule()`, and offer `Graph.repeat()` as an explicit mechanism for defining this feature.

An example of streaming a single shot at a consistent framerate of 30 FPS is shown below.

```
Graph g;
Camera cam;

ShotRequest shot = cam.makeRequest();
g.addRequest(shot);

g.scheduleRepeat(shot, AT, shot.begin, 1000.0f / 30);
```

Likewise, it is straightforward to stream a group of shots, such as to alternate short and long exposures in order to produce HDR video.

```
Graph g;
Camera cam;

ShotRequest s_short = cam.makeRequest();
s_short.exposure = 0.5; // Short exposure: 1/2000 sec
g.addRequest(s_short);

ShotRequest s_long = cam.makeRequest();
```

```
s_long.exposure = 20; // Long exposure: 1/50 sec
g.addRequest(s_long);

g.schedule(s_long, AFTER, s_short.end, +0);
g.scheduleRepeat(s_short, AFTER, s_long.end, +0);
```

3.3 Scheduling errors and guarantees

As discussed at the end of Section 3.1, giving the user an expressive scheduling interface means that we must be prepared to handle invalid schedules. Fortunately most event timing is static, so our scheduling method can easily check that the requested schedules are legal as they are constructed. Errors can still occur in cases where event timing is dynamic and depends on the runtime behavior of the system. These are handled by relaxing a minimal number of constraints and executing the remaining requests as scheduled.

To explore these issues in detail, let us define a "scheduling subgraph" to be a collection of one or more Requests which are linked together only by static events. Because the firing times can be calculated for static events, all of the relative times between events can be computed, and the entire subgraph can be scheduled via an optimization problem without any additional information. Constraints based on dynamic events are the broken links between scheduling subgraphs. Because the event time is unknown, a subgraph cannot be scheduled until all of the dynamic events driving it have been resolved. With this terminology out of the way, we consider the types of errors which could occur.

3.3.1 Static errors

The first type of error is a *statically-detectable conflict*, where a user-applied timing constraint directly conflicts with another constraint in the graph. The most obvious example of this would be two constraints on the same request which specify conflicting times, such as in the example below:

```
g.schedule(A, AT, B.start, +0);
g.schedule(A, AT, B.start, +20);
```

An equally problematic case is when two requests are scheduled on a device such that their execution would overlap, such as asking one sensor to capture two images simultaneously:

```
Graph g;
Camera cam;

g.addDevice(cam);

ShotRequest shot1 = cam.makeRequest();
g.addRequest(shot1);
ShotRequest shot2 = cam.makeRequest();
g.addRequest(shot2);

// This works if shot1 and shot2 are on different cameras, but not
// if they are on the same camera:
g.schedule(shot2, AT, shot1.start, +0);
```

These conflicts are "statically-detectable" because it is not necessary to analyze the state of the system or to fully schedule the graph in order to detect the problem; all of the necessary information is embedded in the static parts of the schedule. On each call to `Graph.schedule()`, the scheduler checks the new constraint against the existing set, and returns a failure code to the user if it detects any conflicts.

3.3.2 Static/dynamic overconstraint

A related problem is when a Request has both a static and dynamic constraint, or possibly more than one of each. Consider the example shown in Figure 3.6 where one shot is analyzed and used to meter a second shot, scheduled to be taken 100 ms after the first. If the metering calculations take place on a non-real-time system, then it is impossible to make hard guarantees about when the computation will finish. The data dependency from the metering calculation to the second shot implies a dynamic `AFTER` constraint: the completion of the processing is a dynamic event, and the shot may be taken any time after the event fires.

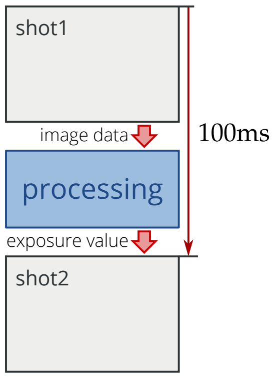


Figure 3.6: A static/dynamic overconstraint created by specifying a fixed interval between shots plus a non-deterministic processing operation which uses the result of the first shot to determine the exposure of the second shot. If the processing takes too long, the 100 ms interval constraint will be broken.

Because of this dynamic constraint, the shots are not in the same scheduling subgraph and cannot be committed for execution together. If the processing takes takes too long, the static 100 ms `AT` constraint must be broken.

3.3.3 Run-time conflict

A third possible error is a *dynamic* or run-time conflict. Unlike statically-detectable conflicts, which are the result of intrinsically impossible constraints, dynamic conflicts are caused by a conflict between the schedule and the current runtime state of the system. For example, consider the following schedule, where a shot is to be taken at the instant a button press is detected:

```
Graph g;
Camera cam;
ShutterButton button;

ShotRequest shot = cam.makeRequest();
g.addRequest(shot);

ButtonRequest capture = button.makeRequest();
g.addRequest(capture);
```

```

// Take the shot the instant the button is pressed
g.schedule(shot, AT, capture.pressed, +0);
```

There is nothing intrinsically broken about this schedule. However, depending on the state of the image sensor at the instant the request is fired, it may not be possible to capture a shot immediately.

Consider also the case of streaming a sequence of images 33 ms apart while running an autoexposure loop, which is a normal case for a viewfinder or video capture. If the exposure time is driven dynamically and not clamped to be less than about 20 ms, then it is possible that the exposure will be driven such that a frame takes longer than 33 ms and the frame-to-frame constraint cannot be met.

In the case of dynamic conflicts, it is not helpful to immediately return a failure code to the user since the requests have been submitted and are already in the pipeline. Instead, the system breaks the dynamic constraint(s), and records exactly what took place in the capture metadata. In the shutter example, if the image sensor is unable to capture when the button is pressed, the `(shot, AT, capture.pressed, +0)` constraint will be converted to an `AFTER` constraint and `shot` will be captured at the earliest opportunity. Static constraints are still preserved as long as they do not have other conflicts, since the run-time system schedules complete schedulable subgraphs.

3.3.4 Streaming errors

A final category of errors appears when we consider streaming applications that execute continuously. It is not possible to reserve an infinite amount of space for all future captures; nor is it reasonable to mandate that all downstream processing operate under hard real-time constraints. Thus, it is possible that the downstream hardware is unable to keep up with the schedule, and the system runs out of resources, whether pre-allocated memory buffers or something else.

Some constraint must be broken, whether by dropping frames or stalling the pipeline. We choose to handle this by breaking the `repeat` constraint and stalling the pipeline. For example, if insufficient buffers are available after a run through the schedule, the next iteration will not be committed for execution until enough buffers are freed, regardless of the inter-iteration timing constraint.

3.3.5 Guarantees

To summarize, the runtime system makes the following guarantees. If a graph is accepted (i.e., a call to `execute()` succeeds), then the following are true:

- The graph contains no statically-detectable conflicts. This is enforced during each API call as the graph is created, providing a clear indication which constraint is at fault.
- Scheduling subgraphs will execute atomically, with all of their constraints satisfied. Said another way, the scheduler will find a way to satisfy at least the internal static constraints, and then the subgraph will be committed for execution as a whole.
- Constraints based on dynamic events are executed on a best-effort basis. All of the requests will be executed and processed, but the timing constraints may not be satisfied.
- The graph execution will not fail on the first iteration due to resource limitations. This can be ensured by reserving resources at the beginning of the `execute()` call.

  If the graph fails on subsequent iterations due to resource constraints (i.e., the camera is capturing continuously and all of the image buffers are in use), then the pipeline will stall and the `repeat` constraint will be broken.

3.4 Basic dataflow graphs

So far we have seen how to schedule a group of requests for execution, but have not done anything with the resulting data. In this and the following sections, we describe

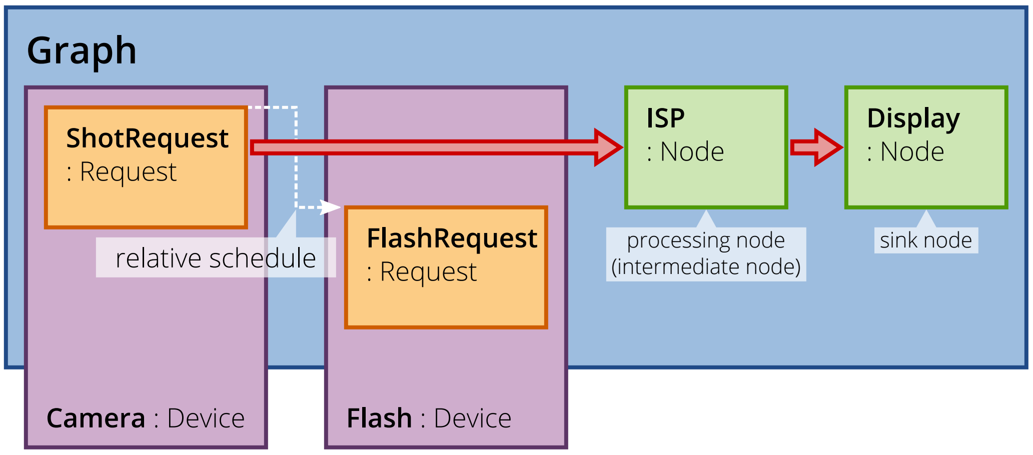


Figure 3.7: A simple graph which takes a shot with a flash, processes it, and displays the result.

the mechanisms in F4graph for defining dataflow and computation after the requests are executed.

Data processing is specified by linking together nodes which produce and consume data to form a graph. When executed, the requests produce data which is passed through intermediate processing nodes and eventually to sink nodes which do not produce further outputs. The code example in Listing 1 on page 49 creates a simple graph where the output of a shot request is fed through a processing node to a display window, and Figure 3.7 shows a visual representation of this graph.

The first half of Listing 1 are familiar from Section 3.2; lines 1-15 create a graph, instantiate devices and requests, and schedule a flash to occur during an exposure. Lines 18 and 23 create graph nodes for the processing and display windows, and the subsequent lines add them to the graph, as is done for `Requests`.

Lines 20 and 25 link the nodes together in the graph. Every node has one or more ports ("pads" in GStreamer; "pins" in DirectShow) through which the node produces and consumes data. The output of one node is connected to the input of another with `Graph.connect()`:

```
// Connect the output of sourceNode to the input of destinationNode
// Graph object and hardware devices
Graph g;
Camera* cam = new Camera;
Flash* flash = new Flash;

// Set up the requests and add them to the graph
ShotRequest* shot = cam->makeRequest();
shot->exposure = 25.0; // 25ms
g.addRequest(shot, "shot");
FlashRequest* f = flash->makeRequest();
flash->duration = 5.0; // 5ms
g.addRequest(f, "flash");

// Schedule the flash to occur 5ms into the exposure
g.schedule(f, AT, shot.begin, +5);

// Create the processing node
HalideNode* isp = new HalideNode("isp.kernel", "pipeline_zynq_argv");
g.add(isp, "isp");
g.connect(shot->result, isp->getInput("input0"));

// Create the display
WindowEGL* display = new WindowEGL("Example viewer");
g.add(display, "display");
g.connect(isp->result, display->input)

g.executeContinuous();
```

Listing 1: Graph example which captures a shot with a flash, processes the result, and renders it in an EGL window. Figure 3.7 provides a visual depiction of this graph.

```
g.connect(sourceNode->result, destinationNode->input);
```

A given output may connect to multiple inputs, but each input can only be driven by a single output. Data ports are typed with a C++ datatype and dimensions, and ports can only be connected if their types match (or if there is an available type conversion), and an output is being connected to an input.

In addition to passing streams of data, ports can also pass parameter values. All of the parameters to a `Request` (such as sensor gain or flash duration) are in fact ports, which means they can be driven by other nodes in the graph.

However, it's often the case that some inputs should be constants, which brings up a distinction between two ways an input port can behave. *Statically-assigned ports* accept a persistent value which is used for multiple invocations of the node, while *dynamically-assigned ports* are fed exactly one new data pulse per invocation.[3] Statically-assigned ports may have their values updated while the pipeline is running, but the value is persistent. For example, the exposure value for a camera shot might be statically-assigned: it can be updated while the pipeline is running (perhaps due to user input or an autoexposure algorithm), but in the absence of any updates the current value will continue to be used repeatedly. Ports are statically assigned simply by assigning a value to them, as shown in the example statements below and in line 8 of Listing 1.

```
shot.exposure = 20.0; // 20ms
halideDemosaic.getInput["ccm"] = colorCorrectionMat;
```

Conversely, the image data input for a processing node is dynamically-assigned; it is connected to another node which produces new data for each invocation.

All static ports must be given an initial value, either by the Node constructor or by user code. Remaining ports are assumed to be dynamic and must be connected to matching output ports. Once the pipeline is configured and running, each node executes when all of its dynamically-assigned ports have been provided with a new data pulse.

[3]This terminology is borrowed from the notion of static and dynamic pulses used in SAI [33], although our use is somewhat different.

F4graph uses a simple model for graph execution: the graph executes a sequence of one or more "iterations", where each node executes exactly once. On each iteration, every `Request` is scheduled to execute once according to its scheduling constraints. Downstream nodes execute once — when all of their dynamically-assigned inputs have a valid data pulse — and produce one result on each output port.

This "one execution per iteration" restriction enforces a set of clear runtime semantics:

- The entire graph executes at the same rate, so it is not necessary to synchronize or correlate data pulses from different streams. In cases where it is necessary for data to be produced at different rates (e.g., an IMU sampling at 10× the framerate), it is straightforward to make more than one request to the same hardware device within the graph to produce an integer multiple of the base rate.
- A call to `Graph.execute()` executes the entire graph exactly once, and `execute(N)` executes the graph N times.
- If the user calls `Graph.executeContinuous()`, the runtime can begin the second iteration as quickly as it can be scheduled (following any resource utilization and `repeat()` constraints), and multiple iterations can be in flight simultaneously. However, each iteration stays in order; mismatches in rates and runtimes cannot scramble the ordering.

3.5 Processing images

Having introduced the core framework, we now turn to the task of actually processing images. In theory, custom nodes could be implemented to perform all types of image processing operations, or to encapsulate user image-processing code. However, rather than implementing a fixed set of nodes (as OpenVX does) or asking the user to interface all of their code with F4graph, a more flexible and powerful approach is to create an interface to a language designed for image processing. As described in Section 2.6, Halide is a powerful language for rapidly optimizing image processing applications, and works nicely as a kind of "shading language" for the system. Moreover, because Halide code can be compiled into FPGA modules which integrate seamlessly with F4graph, it is possible for users to write Halide code and use it to accelerate image-processing code on an FPGA without needing to know Verilog or even write a line of driver code. The details of this implementation are described in Chapter 4.

To use Halide within F4graph, the user writes a Halide function and a corresponding metadata file describing the input and output ports for the node, including any size restrictions. The metadata file also includes default values for any ports which should be statically assigned on construction. Within the F4graph code, the user instantiates a `HalideNode` with the metadata file, and links it into the graph like any other node. An example is shown in Listing 1, which uses an ISP implemented in Halide to transform raw camera pixels into a full-color image.

3.6 More complex graphs

The applications described so far have been straight pipelines, where a single source produces data that is processed by a linear sequence of nodes. Many real applications are more complex than this, and this section introduces constructs to handle these cases.

The first case is a node which consumes more than one input stream. One example of this is computing depth from a stereo pair, where two cameras physically spaced apart capture images simultaneously, and the resulting image parallax is used to estimate the depth of objects in the scene. Halide nodes can take multiple inputs, so connecting multiple shots to one processing block is straightforward, as shown below. The resulting graph is depicted in Figure 3.8.

```
HalideNode* stereo = new HalideNode("stereo_cpu.kernel",
                                    "pipeline_zynq_argv");
g.add(stereo, "stereo");
g.connect(left->result,  stereo->getInput("input0"));
```

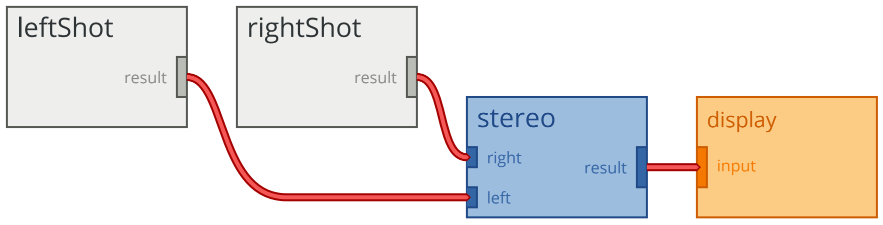


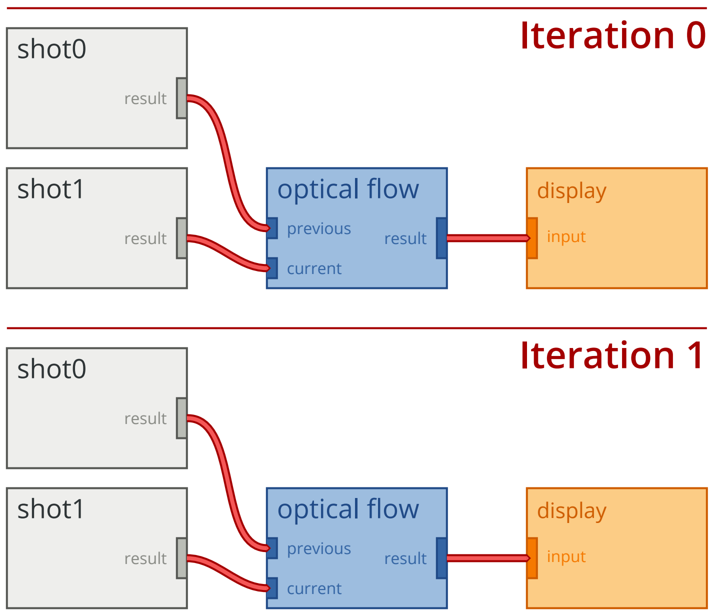
Figure 3.8: Visual depiction of the processing graph for the stereo application. Two images are captured simultaneously and passed to a Halide node for processing. The result is displayed in a window.

```
g.connect(right->result, stereo->getInput("input1"));
```

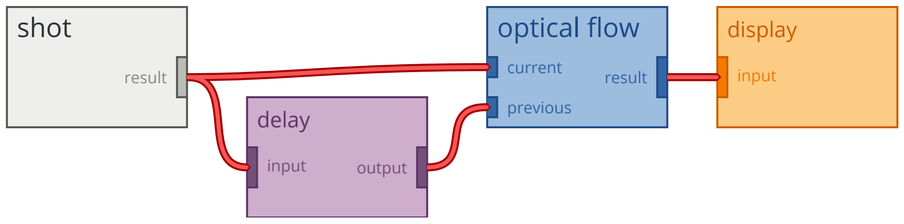
Since each dynamically-assigned input port holds its inputs in a queue until ready to be consumed, multi-input nodes do not require any special handling. The two images are pushed to the `stereo` input ports as soon as they are captured, and the stereo processing node runs once both are available.

A related case is where nodes need to access frames from past iterations of the graph. Consider running optical flow on a stream of image frames, where each pair of frames is compared to find the relative motion of the scene. A naive implementation would schedule two requests, and pipe those into a single processing node which calculates the optical flow, as shown in Figure 3.9a. However, by setting up the graph this way, we have ignored half of the potential pairs of frames and the output framerate will be half of the camera framerate. In Figure 3.9a, we also want to compare `shot1` from iteration 0 with `shot0` from iteration 1, `shot1` from iteration 1 with `shot0` from iteration 2, and so on.

This is achieved using a *delay node*, a FIFO buffer that delays a data pulse for one or more iterations of the graph. The new graph sends the shot result down two paths, as shown in Figure 3.9b. The first connects the shot directly to the processing node; this is the current frame. The second connects the incoming shot to a delay node, the output of which is the frame from the previous frame. On the first iteration of the graph, the image is passed into the delay node, which returns a null image to be passed downstream. On the second iteration, the delay node stores the new image and returns the image from the previous iteration.



(a) Naive implementation of optical flow which combines pairs of frames. One output is produced for every two input frames.



(b) Improved implementation of optical flow, using a delay node. After the first frame, the optical flow node will produce a new output frame for every new input frame.

Figure 3.9: Two ways to implement optical flow, with different output framerates.

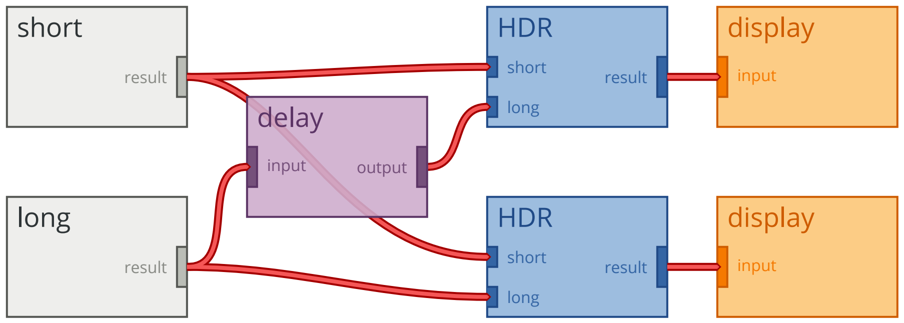


Figure 3.10: HDR video using alternating short and long exposures. A delay node allows the long frame of one iteration to be combined with the short frame of the next.

This illustrates an important consequence of using a delay node: in order to maintain the invariant that every node executes once per iteration of the graph, the delay node must produce something as soon as the first item is pushed in. This is a *null image* (or null data pulse), which is an empty data structure with a "null" flag set. Null images are propagated through the processing graph until they reach sink nodes, which simply discard them.

Running live high-dynamic-range imaging with alternating long and short exposures presents a similar problem to optical flow. To begin, we schedule a short and long exposure on the image sensor, and connect them to the HDR node which merges the two to produce a single output. This brings up the same problem as optical flow — this graph operates on disjoint pairs of images, but we can achieve a framerate that matches the input by also computing a result using the short frame from the current iteration and the long frame from the previous run. Because there are two distinct shots (rather than one as in the optical flow case) each run of the graph includes two shot requests and produces two outputs. However, only one delay node is necessary to link subsequent iterations of the graph, as shown in Figure 3.10.

Delay nodes provide one simple mechanism to hold state between iterations of the otherwise stateless graph, making it easy to implement a large number of continuous multi-frame algorithms. The following section looks at state in more detail, and describes the functions F4graph provides to implement and control state.

3.7 State and feedback

Many applications require some kind of state beyond linking together a continuous sequence of shots. As one example, it is often desirable to change parameters gradually while running applications on a viewfinder or capturing video, so the resulting sequence appears smooth rather than containing abrupt adjustments in brightness or color. This implies some state to track at least the settings for the previous frame, so the parameter can be updated incrementally. Similarly, algorithms which track objects or faces are often initialized with information from a previous run, so that subsequent iterations of the algorithm need only to search a local region.

At the same time, we desire a modular abstraction with a clean separation between algorithms and the data they operate on. Embedding state into the nodes themselves makes their operation more opaque and reduces the flexibility to reuse them in novel ways. Additionally, many nodes are created as stateless implementations of algorithms, and so it is appropriate that we encapsulate the state variables outside the nodes themselves. Halide functions in particular have no notion of state that persists between executions.

Other applications need not only state, but *feedback*. That is, the output of one node (typically some kind of processing or analysis) drives a parameter of node earlier in the pipeline, such as for autoexposure or auto-white-balance (AWB). With AWB, one node computes some color statistics and these are used to adjust the white balance parameters. It is possible to implement this in a feed-forward fashion, where the statistics are first computed and later applied to the same image. However, on streaming ISPs which read pixels directly off the image sensor and only buffer a few lines of the image, it is too late to apply any correction by the time the statistics block has read and analyzed the last pixels of the image. Instead, feedback is used: the AWB statistics block drives the settings for the next image, which is very likely to be similar to the current one.

How should F4graph implement state and feedback? One option is to keep the entire API stateless, and to implement the necessary state and feedback constructs outside of the API. This is the approach taken by FCam, and it works for algorithms where the state evolves slowly. However, other algorithms update state rapidly, and the output of one run is needed immediately on the next. In such cases, maintaining the state outside the runtime system introduces a latency penalty, because the graph must be modified and re-run for each state update. Conversely, it may make portions of the user code time-critical, because settings must be immediately applied and re-submitted to keep the pipeline busy.

Instead, state and feedback in F4graph are managed inside the graph but outside individual nodes using specially-created feedback paths. Any values which a node wishes to preserve for a subsequent run must be pushed to an output port, which is connected back to an input. Because the state is external to the node, it is easy to log the state or to drive it from other sources. Because it is within the graph, it can run without the overhead of returning to the user.

At a high level, feedback is easy: simply connect an output port to an input port somewhere upstream. However, this raises several concerns which must be carefully addressed:

1. When is feedback applied — as soon as the result is ready, or a deterministic number of iterations later?

2. Since feedback paths are inputs which do not have real values until the graph has been executed once, what should nodes do for the first iteration?

3. Can feedback paths stall the pipeline? In other words, if the next request depends on the result of a long-running computation, should the request be postponed?

Both options for concern #1 have valid use cases. Applying each result as soon as it is generated permits the tightest possible feedback loops, since the value can be picked up immediately on the next run of the node. However, this means that the latency (in terms of frames) might not be consistent, which is problematic for algorithms that attempt to smoothly vary parameters or otherwise need deterministic latency. It is actually straightforward to implement both, since these two choices directly correspond to the behavior of statically and dynamically-assigned ports.

In all cases, a node's input port values are locked in when the node begins to execute, not when the whole graph begins or partway through the execution. If a port is statically-assigned, then the feedback values are applied immediately and replace the old values, and the node can use them on the next iteration. Conversely, if a port is dynamically-assigned, it contains a FIFO buffer which consumes one feedback value on each iteration.

Thus, F4graph has two methods for creating feedback paths, `connectFeedback()` and `connectImmFeedback()` which correspond to fixed-latency (dynamic) and immediate (static) feedback respectively. As with `Graph.repeat()`, it would be possible to infer feedback cycles by analyzing the processing graph, but instantiating them explicitly reduces the potential for confusion. To address challenge #2, both methods have a required parameter which specifies one or more default values to use during the initial runs of the graph. For a fixed-latency feedback path, the number of default values specified controls the FIFO size and therefore the latency of the path. An immediate feedback path needs only a single value, which is applied directly to the port.

It might seem reasonable here to simply fill in null pulses rather than force the user to supply default values, but this quickly becomes problematic. If a node receives a null pulse on one of its inputs, then it also produces a null pulse, and so on down the graph, until a null pulse gets pushed back along the feedback path, leaving the system stuck with no results.

The answer to the third difficulty follows quite naturally from the solutions to the first two. For immediate feedback paths, the node always has data values and will never stall as a result. Conversely, a lack of data values on a fixed-latency feedback path will stall the pipeline until the results become available. If such stalls (and the resulting pipeline "bubbles") are undesirable, the solution is to increase the feedback FIFO depth. Otherwise, stalls are an unfortunate consequence of the requirement to observe a fixed frame latency.

Moving to practical implementation, the code snippet below illustrates an example of using feedback to control an auto-white-balance block.

```
Shot* s = cam->makeShot();
g.addRequest(s, "shot0");

// ISP / color-correction block which applies white balance
// Takes two inputs: the image, and a color-correction matrix
HalideNode* colorCorrect = new HalideNode("ccm.kernel", "ccm_cpu_argv");
g.add(colorCorrect, "colorCorrect");
g.connect(s->result, colorCorrect->getInput("input0"));

// Estimation block which computes new white balance parameters
HalideNode* colorStats = new HalideNode("stats.kernel", "stats_cpu_argv"
    );
g.add(colorStats, "colorStats");
g.connect(colorCorrect->result, colorStats->getInput("input0"));

// Connect the feedback loop to update parameters for the next frame
// Because this is a feedback loop, we must specify a default
uint8_t ccm[12] = {190, 0, 0, 128,
                   0, 90, 0, 128,
                   0, 0, 120, 128};
Image ccmDefault(ImageFormat::GRAY_U8, 4, 3, 1, ccm);
g.connectFeedback(colorStats->result, colorCorrect->getInput("ccm"),
    ccmDefault);
```

An additional concern applies when the feedback controls a `Request`, rather than a processing node. Because changing the parameters of a Request can alter the time it takes to execute the request, any such requests must be dynamically scheduled. The following code illustrates a request driven by a processing block to perform auto-exposure. The processing block calculates a histogram of pixel brightness values for the image, and uses this information together with the exposure time of the captured shot to calculate an updated exposure time.

```
Graph g;
Camera* cam = new Camera();

Shot* s = new Shot(cam);
g.addRequest(s, "shot0");

// autoexpose() is a user-defined function that takes and image and
// returns a float (exposure time in ms).
```

```
UserNode<Image, float>* autoexp =
    new UserNode<Image, float>(&autoexpose);
g.add(autoexp, "autoexposure");
g.connect(s->result, autoexp->input);

// Drive the exposure using the result of the AE function,
// using immediate feedback.  Use 0.25 ms as the default value.
g.connectImmFeedback(autoexp->result, s->exposure, 0.25);

WindowEGL* display = new WindowEGL("Example viewer");
g.add(display, "display");
g.connect(s->result, display->input);

g.executeContinuous();
```

3.8 Metadata

One of the key features of FCam is that metadata is returned with every shot, which provides a way to identify each returned frame and to control the processing based on the parameters the frame was taken with. The metadata describes what parameters the user requested, and more importantly, the parameters that were actually applied as the image was captured. Like FCam, metadata in F4graph is represented with a key-value store, referred to as a set of *tags*. However, because F4graph describes capture and processing on multiple imagers, we need to extend the FCam model in two ways.

First, every node can create metadata, including Requests and processing nodes. It isn't sufficient to simply know the capture parameters, we also need a "processing pedigree". Each node class defines a standard set of tags to facilitate portability and reuse. Every node's input ports are used to create two sets of tags: the values which were requested (i.e., the value of the input ports at the moment the node began to execute), and the values which were actually applied.

Second, because the capture and processing nodes are arranged into a graph, the metadata is arranged in a corresponding tree. Simply adding new tags to a flat metadata store is insufficient, because in many cases there are multiple identical shots which form part of a single capture, and there must be some way to differentiate them.

Obviously this tree can grow indefinitely in the case of feedback paths, which can create an IIR filter. After a few seconds of streaming, several hundred frames have contributed to the output, creating an enormous (and generally unhelpful) glob of metadata. To mitigate this, feedback paths strip off the tree and pass only the root values back, although this can be disabled for debugging purposes.

3.9 Summary

Each feature of the API introduced in this chapter is designed to give application developers as much expressive power as possible while maintaining guarantees about the behavior of the resulting application. Constraint-based scheduling (together with the associated rules about when and how constraints may be broken) allows developers to specify the constraints that matter for their application, leaving flexibility to work around hardware constraints or coexist with other applications. State storage is modeled explicitly with the graph, making it straightforward to develop iterative and streaming applications which rely on results from previous runs of the pipeline. Data computed for feedback paths can be applied immediately or with a constant frame delay. Failures are reported, and metadata records what actual capture and processing parameters were used.

The next chapter describes the prototype hardware platform we created to implement and validate this API, discussing the design choices and purpose-built hardware features that make a practical implementation possible.

Chapter 4

System implementation

The previous chapter introduced an abstract interface which promised stateless control of the camera capture and processing pipeline, with flexible processing and precise timing control. This chapter describes the camera system we built to rapidly prototype and validate the API on real hardware, showing how particular features of the design enable the API to perform as intended.

To achieve these goals, the camera system must support the following:

- **A rich hardware environment**, meaning multiple cameras, lenses, and flashes, a high-resolution display, standard interfaces including USB and Ethernet, and the flexibility to add more peripherals.
- **Custom accelerators**, which can perform image processing faster and/or more efficiently than the CPU.
- **Microsecond-level control** of the attached camera hardware.

At the same time, the platform must support a standard development environment that reasonably emulates a modern mobile device, which in practice means the system should run Linux or Android on a multi-core CPU with a standard (i.e., complex) memory hierarchy. Finally, the development system should be accessible for non-expert users, meaning that the hardware designs as well as any real-time or hardware-specific code must be automatically generated.

The following sections describe how each component of our system was designed to meet these requirements. First we show the physical hardware, and then detail the hardware architecture used on the FPGA fabric. The following section describes the software architecture, including code on the real-time core, in kernel space, and in userspace. Finally, we explore the build system that produces this combined hardware/software stack.

4.1 Hardware platform

In terms of hardware, the above requirements imply a sophisticated SoC which contains multiple CPUs, a flexible programmable engine (such as an FPGA, course-grained reconfigurable array, or highly-flexible DSP), cameras, hardware to receive camera input and display graphics, and the ability to drive other common peripheral devices. At first glance, this is very demanding feature list. However, both Xilinx and Intel have been integrating hardened CPU cores and other peripherals with their FPGA fabrics to produce SoCs with all of these features and more. Moreover, the FPGA fabric can implement capabilities not included in the hardened logic, such as integrating additional cameras or controlling other peripherals. The main contenders in this space are the Xilinx Zynq-7000 series, the newer Zynq UltraScale+ series, and Intel's line of Stratix and Arria SoC FPGAs.

4.1.1 Processing platform

The original Halide-to-FPGA work described in Section 2.6 [47] used the first-generation Zynq parts, specifically the XC7020 and XC7045. The Halide-to-FPGA compiler is not part-specific but does depend on the Xilinx Vivado HLS toolchain, so for simple compatibility we chose to use a Xilinx SoC.

Since the time of the original Halide-to-FPGA work, Xilinx introduced the UltraScale+ series [54] of chips, which are a major step above the original Zynq-7000 parts. These include a quad-core ARM A53 processor with improved capabilities for cache-coherency with the FPGA fabric, two ARM R5 "real-time" cores, an integrated

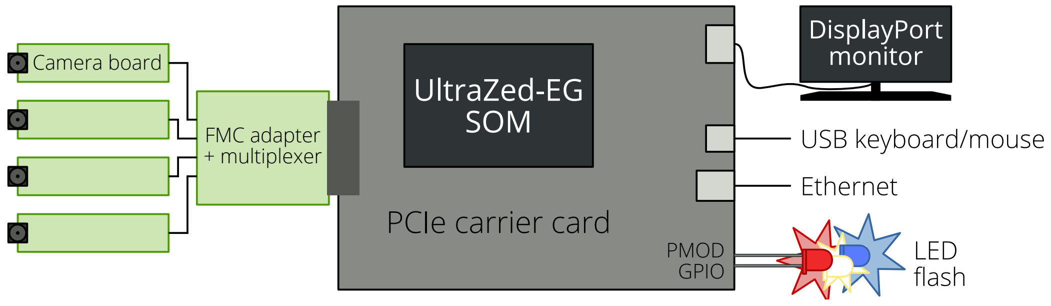


Figure 4.1: Main hardware components and their interconnections.

Mali GPU, and a dedicated DisplayPort controller. In our experience, HDMI support on the Zynq-7000 was unreliable, and the limited support for cache coherency via the accelerator coherency port (ACP) proved to be a performance bottleneck. The hardened DisplayPort controller and high-performance coherency (HCP) ports on the UltraScale+ parts promised to correct these shortcomings. The additional CPU and GPU cores also enable more on-board processing, without needing a separate board and processor for the user interface.

Given these benefits, we selected the Avnet UltraZed-EG platform [55], which is a modestly-priced ($535 USD) system-on-module development kit for the Xilinx Zynq UltraScale+ XZU3EG MPSoC. The system on module (SoM) design offers the opportunity to develop a custom baseboard (without the headache of routing the BGA package or connecting DRAM), but for speed of development we used an off-the-shelf baseboard, specifically Avnet's UltraZed PCIe carrier card. The board provides an FMC breakout connector for the FPGA I/O, as well as USB, Ethernet, and DisplayPort connections in a modest-sized package (4"×10"). Figure 4.1 summarizes the hardware setup built around the UltraZed-EG.

Overall, the UltraZed platform worked well for our implementation. The A53 complex runs Linux, which provides the conveniences of a full-fledged operating system, while real-time-critical tasks are delegated to the R5 cores. The FPGA implements the camera interfaces and processing accelerators, as will be described later in this chapter. Output images are scaled with the GPU and shown via DisplayPort, while a USB keyboard and Ethernet connection provide local and remote shell access for development.

While this setup worked well for our initial prototype, a couple of issues could be addressed in future. First, Xilinx and Avnet's support for DisplayPort is neither robust nor well-documented, so we were constrained to a particular 1080p monitor that happens to work with the system. Second, the PCIe carrier card was an unfortunate compromise. We selected it because it was the only off-the-shelf carrier card with an FMC slot, which we needed for the high-density, high-speed I/O from the cameras. However, it dedicated one of the chip's four gigabit transceivers to the PCIe interface (which went unused) rather than using two for DisplayPort (which would allow resolutions up to 4K). This could be improved in the future by building a custom carrier board, which could also integrate the camera interface described in the following section.

4.1.2 Camera hardware

The camera hardware needs to reasonably approximate a modern cell phone camera, allow for multiple image sensors to be connected, and permit fine-grained, low latency control of the module. An obvious solution is to use actual cell phone sensors, since they are abundant, tiny, and inexpensive. However, this has two hurdles, one technical and one logistical.

The technical challenge is merely connecting to the sensor and receiving data at the physical level. Essentially all cell phone sensors use the MIPI (Mobile Industry Processor Interface) CSI-2 (Camera Serial Interface 2) protocol. Since the specification is tailored for mobile devices where low pin counts and tiny connectors are essential, CSI-2 uses a small number of high-speed (500 MHz – 2 GHz) differential data lanes overlaid with low-speed ($< 50\,\text{kHz}$) common-mode data, in a scheme known as D-PHY.

Compared with a parallel LVDS link operating at a few hundred megahertz, D-PHY is difficult to interface with. Fortunately, the Xilinx UltraScale+ chips have D-PHY-capable I/O pins, which eliminates the need for external interface circuitry — each camera clock and data lane can be connected directly to a pair of differential pins. Moreover, Xilinx provides a free IP core which configures the I/O pins correctly, decodes the PHY signal, and deserializes the data to produce an output stream clocked at typical FPGA speeds.

The logistical hurdle is that very few cell-phone image sensors have publicly available documentation and reliable sourcing options in small quantities. Although it is possible to purchase practically any replacement sensor for a few dollars on Ebay, these often have no public documentation at all, let alone a datasheet or register map. One exception is the Sony IMX 219 [18], which is used in the popular Raspberry Pi V2 camera board [56]. Although the board itself is not open-source, it has been publicly reverse-engineered, and a datasheet for the IMX 219 has been shared online [57]. There are open-source code samples available [58] as well as designs for connecting the camera to Zynq-based systems [59].

The sensor itself has a resolution of 3280×2464 (8 MP) and physical diagonal of 4.6 mm (1.2 µm pixels). Its performance and feature set is modest compared to top-of-the-line parts used in flagship phones, but it is a decent imager that has been adopted in a number of mid-range phones [60].

Our system uses up to four Raspberry Pi cameras and interfaces them directly to the Xilinx D-PHY IP and a custom CSI-2 receiver. By building this interface from the ground up, we obtain "bare-metal" control over the data processing and timestamping. Additionally, CSI-2 specifies that low-speed configuration and control is handled by I2C, which gives easy and direct access to the full register space of the sensor.

The Raspberry Pi camera boards use a fixed-focus lens, so to experiment with focus control we built our own board using commercially-available autofocus modules integrated with the IMX 219 image sensor. The popularity of the IMX 219 helps again here — only a handful of sensors can be purchased with AF modules in single-digit quantities, and the IMX 219 happens to be one of them. For simplicity of integration, we designed our boards to be pin-compatible with the Raspberry Pi V2 camera boards. This allowed us to begin development with the latter and switch to our customized boards when they were completed.[1]

[1]Unfortunately, our boards are not "backwards-compatible" with the Raspberry Pi, due to a

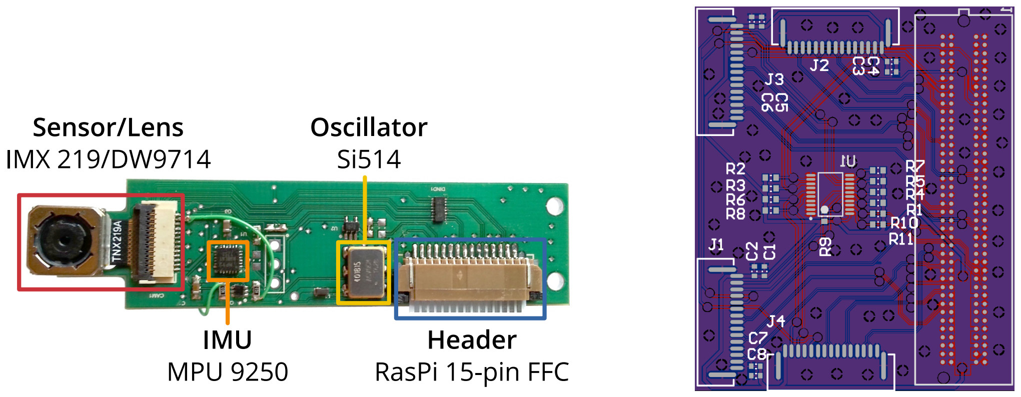


Figure 4.2: Custom IMX219 camera board and FMC adapter board. Up to four cameras can be connected to the adapter, which plugs into the UltraZed carrier card.

Figure 4.2 shows an image of the board and the 4-camera adapter PCB. Each off-the-shelf camera module contains both the sensor itself and and an independently-controlled focus module. The focus is actuated by a voice-coil motor (VCM) which is controlled by a dedicated VCM driver chip (Dongwoon DW9714) built into the module.

The IMX 219 requires a 24 MHz input clock, which is provided by a Silicon Labs Si514 programmable oscillator. Combined with the IMX 219's internal clock tree, the programmable clock offers the potential for "overclocking" or otherwise tweaking the operating frequency of the image sensor. Each board also integrates an Invensense MPU-9250 9-axis inertial measurement unit (IMU) to enable applications such as video stabilization, rolling shutter correction, and deblurring.

The imaging modules all connect to the UltraZed via a breakout board that plugs into an FPGA Mezzanine Card (FMC) slot.[2] Each of the peripherals — the VCM controller, IMU, and the image sensor itself — are configured over an I2C bus. Because all of the cameras have the same fixed I2C address, the breakout board contains an I2C multiplexer that addresses a single camera module at a time. The I2C tree

cryptography chip built into the RasPi camera boards to prevent third-party camera modules from operating with the Pi [57].

[2]Interestingly, Digilent released an almost identical board, the "FMC PCam adapter", as a product in May 2019 [61].

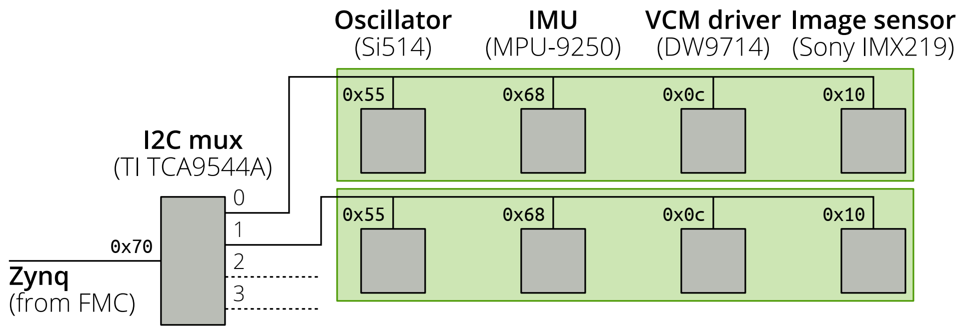


Figure 4.3: I2C tree. Addresses are shown as 7 bits (without the R/W LSB). Up to four camera boards can be connected to our FMC adapter board, each with their own oscillator, IMU, and camera assembly.

for a two-camera configuration is shown in Figure 4.3.

Camera flashes are simulated using high-brightness LEDs of varying colors, driven by additional GPIO pins from the FPGA.

4.2 FPGA architecture

Since the FPGA fabric is an unstructured “blank canvas”, we need a hardware architecture which defines the macro-components and their possible interconnections in order to automatically generate the FPGA images, drivers, and runtime system. This architecture consists of a set of static components that are always included, plus a set of user-defined pipelines constructed from standard modules.

The static components include the following:

- An I2C controller which controls the cameras and related peripherals.
- A GPIO register which drives the camera “enable” pins and LED flashes.
- Required logic to drive the DisplayPort clock generator, copied from the UltraZed reference design.
- A memory-mapped register that controls coherency by driving the `ARPROT` and `AWPROT` (read and write protection) signals on all AXI buses going to main

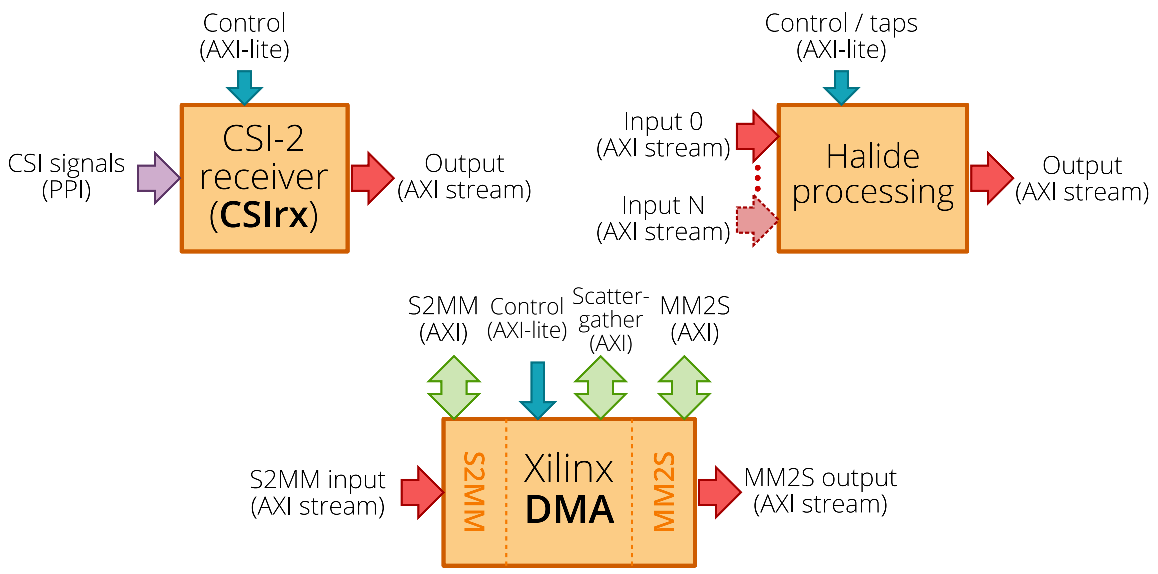


Figure 4.4: Configurable block types in the FPGA design, with their inputs and outputs. FPGA designs are composed by linking these blocks together.

memory.

Figure 4.4 shows the three types of configurable components: CSI-2 receivers, Halide processing blocks, and DMA engines which push and pull data from main memory. All of these transfer data using AXI-stream, a minimal protocol which consists of a variable-width data signal mediated by one-bit ready and valid signals (designated `TREADY` and `TVALID`). An additional `TLAST` signal is used to mark the last data beat of a transaction (in our case, the end of an image).[3] The following sections describe each of these in turn.

4.2.1 MIPI CSI-2 receiver

The Zynq SoCs do not have hardened CSI-2 receivers, so camera data is read and unpacked using the FPGA. The physical interface is created using the Xilinx MIPI D-PHY IP core (v3.1), which uses FPGA logic for high-speed deserialization and

[3]The specification defines a number of other optional signals, but these are not used in our implementation.

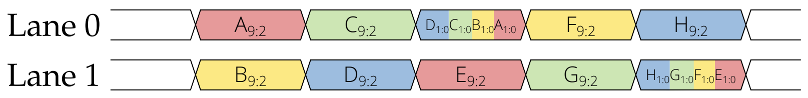


Figure 4.5: MIPI CSI-2 packing of four 10-bit pixels into five bytes split across two data lanes.

generates the appropriate pin I/O and timing constraints for D-PHY signals around 1 GHz. The output of the Xilinx D-PHY IP is a stream of bytes along with some control signals indicating when a transmission is beginning, and whether the data is valid [62].

We developed our own IP core to decode this raw byte stream while extracting timing information. The CSI-2 data format includes a frame header with a checksum and row headers on each line of the image, and groups of four 10-bit pixels are packed together into five bytes, as shown in Figure 4.5. Our receiver parses the headers and swizzles the data bits to produce complete 10-bit pixels. The result is presented as a standard AXI-4 stream, with 2 bytes per pixel and 4 pixels per clock cycle, valid two out of every five clock cycles. This stream can be passed directly into a processing core or DMA engine, or reshaped into a stream with more or fewer pixels per cycle.

Because we designed our own receiver, we were able to build in three simple features that enable and simplify the API implementation. First, the receiver can generate interrupts when it receives the first byte of a new frame (by watching for the frame header), and when it completes a frame (by counting the number of rows received). These interrupt lines are routed to the real-time core, which records the frame timestamp with a latency well under 1 µs. Combined with knowledge about the timing and buffering of the sensor, this allows the real-time system to calculate the beginning and end of the exposure in terms of the global clock. Second, the data output can be turned off while leaving the receiver on and the interrupts enabled. This allows the real-time system to track the camera timing, even when the frame data is being thrown away. This is particularly useful for tracking "dummy frames" which are inserted to control timing — the image data is useless, but the timing information is critical. Last, the hardware buffers the settings and applies them only on frame boundaries, so that the system can configure the next frame without affecting the one currently being read. This prevents partial frames from coming out and hanging up the downstream hardware, and relaxes the timing restrictions for configuring the hardware. The software can configure the frame any time during the previous frame, not only in the tiny inter-frame window.

All of these features are extremely simple: they are implemented with just a few lines of Verilog code, and took only a couple of days to develop, test, and debug. However, they illustrate the principle we are arguing for: a little hardware support goes a long way. It would be trivial for SoC manufacturers to include these enabling features in their designs, and they ought to be standard.

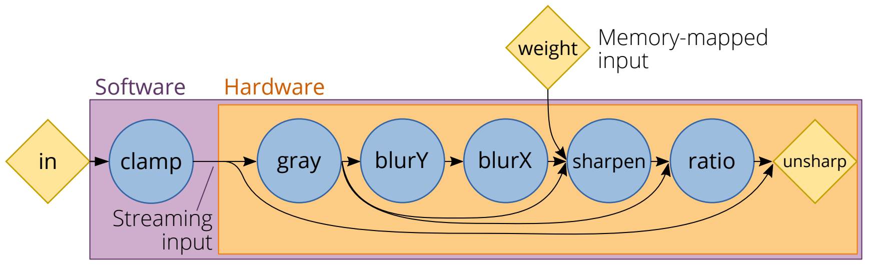
4.2.2 Hardware generation from Halide

We use the Halide-to-FPGA compiler developed by Pu [47] to compile Halide functions into IP cores that can be integrated into the FPGA design. Each Halide function defines an output image in terms of one or more input images or previously-defined functions. This creates a chain of functions, from input images to final result.

When a Halide function is compiled into a software library, the inputs and output become parameters of a C function. For a hardware accelerator, however, inputs can be passed either as streaming ports or via memory-mapped registers. Streaming inputs are best for image data, while registers are more suited to “process parameters” or “tap values” where a handful of values are used to process thousands or millions of pixels.

The FPGA extensions to the Halide language allow the developer to define both the scope of accelerator and the type of input used for each input. Specifically, a call to `accelerate()` defines the output and all of the streaming inputs. The scope of the accelerator is everything in between, and any additional necessary inputs become memory-mapped registers. The output is always presented as a stream.

An example from Pu [47] of the “unsharp” image filter is shown in Figure 4.6. In a series of operations, the image is converted to grayscale, blurred and then



```
unsharp.accelerate({clamp}, x, y);
```

Figure 4.6: Generating hardware and software from a Halide function. Each blue circle represents a function within the code; inputs and outputs are yellow diamonds. The parameters `x` and `y` represent the tile dimensions for the accelerator.

subtracted from the original to pick out the edges. This edge image is recombined with the grayscale by a weighted addition to produce a sharper result, and this is then applied back to the full-color image. Using the Halide accelerate call `unsharp.accelerate(clamp, x, y)`, everything from the output of `clamp` to `unsharp` will be part of the hardware accelerator, with `clamp` being computed in software. The result of `clamp` is fed as a streaming input, while `weight` (which was not included in the input list of the `accelerate` call) is configured as a memory-mapped input.

When part of an algorithm is accelerated in hardware, the normal software function is replaced by one with the same arguments, but which configures and runs the hardware rather than computing the result in software. The software section later in this chapter will describe the kernel drivers and intermediate software calls which connect compiled software library to the hardware.

There are two limitations that are a result of using the current Halide-to-FPGA compiler. First, the system can only use FPGA acceleration for kernels which fit into a line-buffered pipeline model. This is because the Halide-to-FPGA compiler works by mapping Halide algorithms into line-buffered pipelines and then using a flexible template to generate hardware following this design pattern. Unfortunately, this excludes multiscale algorithms which work on image pyramids or require random access to the pixels, but these kernels can still run on the CPU. Ongoing and future work will allow for hardware generation using other templates, such as a memory-backed systolic array [63].

The other limitation is that Halide code must be precompiled rather than being compiled on the fly as graphics shaders typically are, because the FPGA build process takes tens of minutes on a desktop computer and is infeasible in any sort of runtime context. This is an unfortunate restriction, since it limits the portability of the system. OpenGL and Direct3D are useful APIs partly because shaders in source form can be compiled and optimized for many different GPU architectures at runtime. It is not necessary for the application distributor to provide binaries targeting all possible GPUs, nor for the end user to manually recompile the entire application.

On systems with other image processing accelerators where rapid compilation is possible, it might be feasible to relax this restriction and provide processing blocks to the system as uncompiled Halide sources, enabling target-specific optimization as well as optimizations across Halide nodes in F4graph. While these sorts of optimizations are not exploited in the current system, they are active areas of research. Since we currently only use a single demonstration platform, the need to precompile Halide code results in only a minor complication of the build process.

4.2.3 DMA engines

Data movement between the FPGA and main memory is handled by one or more DMA engines implemented in the FPGA fabric. For speed of development, we used the Xilinx-provided AXI DMA (v.7.1). Each DMA engine includes a read and write channel (less ambiguously referred to as a "memory-mapped-to-stream" or "stream-to-memory-mapped" channel) which share a few registers and configuration bus. Each channel consists of an AXI master port which makes read/write requests to main memory, and a corresponding AXI-stream port which interacts with the CSI receiver or processing module. A "complete" interrupt line from each channel is routed to the APU cores, where it is picked up by the Linux kernel driver and triggers the next memory operation.

The DMA engines are configured in "2D mode," which allows them to fetch or write data in 2-dimensional tiles, using a small descriptor containing the starting address, width, height, and stride. This is useful when images are processed from memory, since breaking the image into tiles (or vertical strips) reduces the size of the intermediate line buffers. For hardware modules connected directly to the camera input, it isn't possible to process the image in tiles without buffering it first, so the processing hardware operates on the sensor line width.

The Xilinx DMA engines turned out to be one of the most troublesome components of the system, and the most frustrating to debug. Chapter 6 discusses several ways the DMA engine could be improved, both to fix bugs and to enable new capabilities.

4.2.4 Design patterns

CSI-2 receivers, Halide processing modules, and DMA engines are linked together according to one of two basic design patterns. In one pattern, the data is received from a camera (or cameras) and unpacked by a CSI decoder, optionally passed through a processing module, and finally streamed into memory with a DMA engine. In the other pattern, data is read from memory by one or more DMA engines, processed by a single Halide module, and streamed back into main memory. Limiting the pipelines to these two patterns simplifies the design of the device drivers, allowing us to reuse most of the driver code from the Halide-to-FPGA work. Figure 4.7 shows a block diagram of an example FPGA design that uses both of these patterns.

These two design patterns impose a number of limitations. Because Halide functions (and the corresponding hardware designs) only have a single output, all processing modules also produce exactly one output. Further, every output is connected to exactly one source and controlled with a ready/valid handshake, so a camera stream cannot be sent to multiple processing blocks in parallel. Finally, all of a module's inputs must come from the same source. The patterns allow multiple cameras or multiple images from memory as inputs, but because of our software architecture (discussed later in this chapter) it is not currently possible to feed a module with a mix of memory and camera inputs.

However, most useful patterns can still be achieved within these limitations, at the

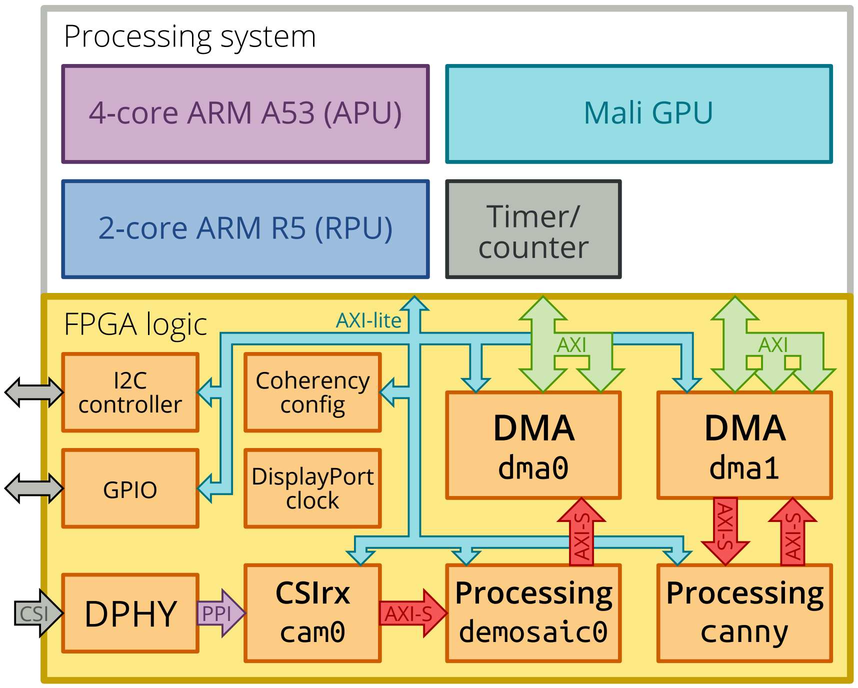


Figure 4.7: Example configuration for the Zynq Ultrascale system. Here, the FPGA logic is configured with two paths: one which captures pixels from a camera (via `cam0`) processes them (`demosaic0`) and dumps them to memory (`dma0`). The second path can pull images from memory (via `dma1`), process them in hardware (`canny`) and write the results back to memory. The hardware blocks on the left (GPIO, I2C, coherency controller) are included in every design.

cost of some performance and/or FPGA resources. While multiple processing blocks cannot be operated in series, it is straightforward to combine the Halide functions to produce a single hardware block, possibly with switches to turn those functions on and off. Multiple outputs can be produced by running multiple functions on the same memory inputs. Restrictions on camera streams can be worked around by streaming the images to memory first, and then processing them from memory.

These workarounds have a cost — additional transfers to and from memory increase the latency and power consumption, and duplicated pipelines waste FPGA resources. To allow higher performance and better sharing of resources, a more flexible architecture would allow each input of a processing module to be selected from more than one source, such as using one ISP pipeline for two cameras, or switching between camera and memory inputs. Data could be processed by a chain of kernels, with an option to pull it out partway or pipe data in at points other than the beginning. This would require switchable connections (rather than fixed point-to-point AXI-stream connections) and software that is able to configure the switches and feed data using these various patterns (rather than assuming a single dataflow).

4.2.5 Design specification

Ideally, FPGA designs would be automatically generated by F4graph and dynamically loaded as the application needs them, much as GPU shaders are. However, the limitations of partial reconfiguration and hour-plus FPGA compile times make this difficult. As a result, our prototype uses a pre-compiled FPGA image which can contain hardware for multiple applications. The contents of this image are defined by the user with a YAML configuration file, where each component is defined and parameterized by a collection of key-value pairs. Listing 2 is an example configuration file corresponding to the design in Figure 4.7.

Each component includes key-value pairs with the identifier to use in the hardware design and the file path to the IP core design files. Since each component has at most one output, connections between components are made by setting the `outputto` key on each component that produces output. For DMA engines — where the stream width can be configured — the hardware generator adjusts the width of the stream to match the other component. For the remaining mismatches, the hardware generator inserts a Xilinx AXI-Stream Data Width Converter between the components. Although the stream width is converted automatically, it is still up to the user to ensure that the data semantics match. For example, the hardware generator can handle the conversion between 1 pixel/cycle and 4 pixels/cycle, but cannot convert RGB to grayscale data or unpack a 10 bits/pixel stream into 2 bytes/pixel. There is no need to specify the AXI-lite control bus or AXI connections to DRAM; these can all be inferred automatically.

4.3 Software architecture

The software architecture consists of three components running on different parts of the system, as illustrated in Figure 4.8. At the lowest level is a small processing thread running on one of the Zynq's real-time processing units (RPUs), which handles timestamping and peripheral control tasks which must happen with minimal latency. The real-time loop runs "bare-metal" on the R5 processor, so it is able to achieve timing precision that would be impossible within the Linux kernel. The main application processors (APUs) run a set of Linux kernel drivers, which provide simplified access to the image processing designs implemented in the FPGA. These require frequent attention, but not microsecond-level response times. We use kernel drivers rather than userspace drivers in order to make use of contiguous memory buffers and to access hardware interrupts. Between the drivers and the user application code is a userspace library which performs request scheduling and delegates work to the kernel drivers and real-time loop. The following sections describe each of these components, which we will refer to the "real-time loop", "kernel drivers", and "APU runtime" respectively.

```
# Example project configuration file
name: example
productionsilicon: y
board: pciecc

# This example specifies two paths: One which processes data directly
# from a CSI receiver, and one which processes images in DRAM via DMA.
# Both return the images to DRAM using DMA.
hw:
- type: csi
  name: cam0
  clk_loc: AB6
  data0_loc: AD8
  data1_loc: AE7
  i2c_loc: 2
  path: /ip_repo/vlsiweb.stanford.edu_csi_axi_csi_1.1/
  outputto: demosaic0
- type: hls
  name: demosaic0
  path: /ip_repo/demosaic/xilinx_com_hls_hls_target_1_0/
  outputto: dma0
- type: dma
  name: dma0

- type: dma
  name: dma1
  outputto: canny
- type: hls
  name: canny
  path: /ip_repo/canny/xilinx_com_hls_hls_target_1_0/
  outputto: dma1
```

Listing 2: Sample project configuration file which instantiates a camera connected to a hardware demosaicker, and a second processing module. See Figure 4.7 for a diagram of the generated hardware design.

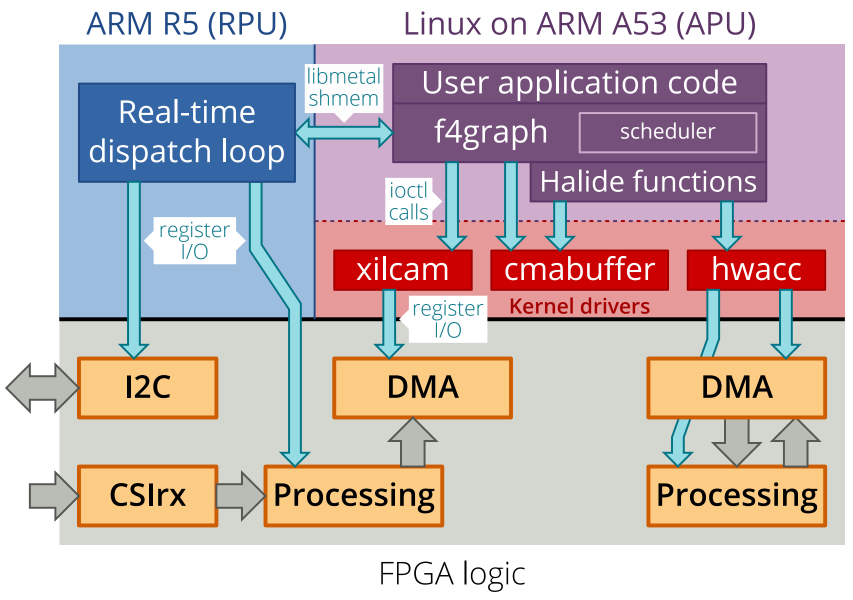


Figure 4.8: The three software components (real-time loop, kernel drivers, and APU runtime) and their interactions. The real-time dispatch loop (blue) controls the camera hardware and communicates with the F4graphruntime via a shared memory mailbox. The runtime (purple) runs in userspace and calls the kernel drivers (red) to interact with the accelerator and DMA hardware.

4.3.1 APU runtime

The APU runtime is a C++ library which implements the F4graph API and serves as the glue between the user code and the hardware-specific software layers. When the user calls `Graph.execute()`, it runs the scheduler (described in Section 4.3.4) to create a timeline for the requests in the graph. It then queues up the requests for execution. Requests which use hardware controlled by the real-time loop (CSI cameras, flashes, and buttons) are passed to the real-time core through the shared memory interface. For camera requests, the runtime simultaneously makes a call to one of the kernel drivers (described in Section 4.3.3) to capture the image that will be generated and stream it into memory. Requests which do not use real-time hardware are simply executed by another thread on the APU.

The runtime executes downstream nodes as their data becomes available. As each node completes, it notifies downstream nodes, which execute in turn. In the case of Halide nodes, the runtime calls the Halide function which was dynamically loaded from the shared object library (`.so`) when the node was created, which may invoke the hardware. Because the developer specified the hardware/software boundary using the Halide schedule, and because this interface is well defined, the Halide compiler can automatically insert calls to the kernel drivers at that boundary. The generated code makes a kernel driver call to allocate memory, performs whatever portion of the algorithm was mapped into software, and then calls another driver to run the hardware.

Since the driver calls are automatically generated and wrapped inside the software library, the Halide function call appears essentially the same, as illustrated in Figure 4.9. The only difference is the need to pass Linux file handles to the `cmabuffer` and `hwacc` device driver nodes.

If the graph is set to run continuously, the thread sets a timer and wakes up a little before the end of the current schedule to launch the next scheduling iteration. It is not necessary for the graph to finish executing before the next iteration begins, so multiple iterations of the graph may be in flight at once.

If a graph is run continuously with `Graph.executeContinuous()`, the function returns quickly while the execution proceeds asynchronously. If an error occurs during

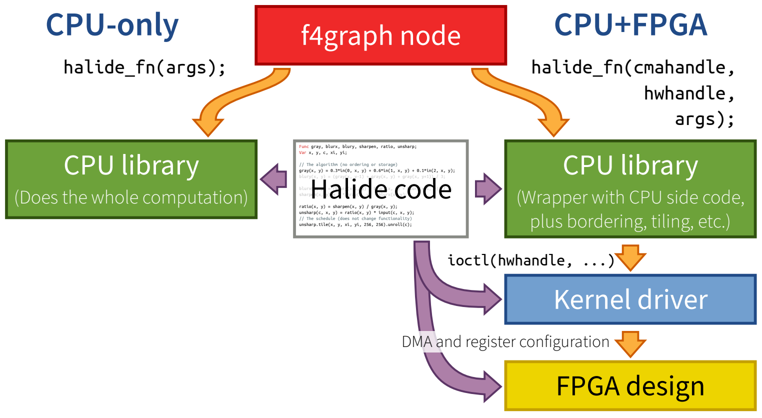

CPU-only
halide_fn(args);
f4graph node
CPU+FPGA
halide_fn(cmahandle, hwhandle, args);
CPU library
(Does the whole computation)
Halide code
CPU library
(Wrapper with CPU side code, plus bordering, tiling, etc.)
ioctl(hwhandle, ...)
Kernel driver
DMA and register configuration
FPGA design

Figure 4.9: Call stack from userspace down to the hardware. Purple arrows show generation/parameterization from Halide; orange arrows show the call chain.

a future iteration of the graph, then an error callback is used to notify the user. As described in Section 3.3, Graph iterations execute atomically: either the whole iteration is schedulable and has sufficient resources to run, or no part of the graph will execute.

4.3.2 Real-time dispatch loop

The real-time dispatch loop runs by itself on one of the ARM R5 cores. When launched, the software configures the CSI receiver interrupts, programs the Si514 oscillators with the correct frequency for the cameras, and starts the image sensors running. Then it enters a processing loop where it executes requests delegated by the F4graph runtime.

To do this, the RPU loop interacts with the APU runtime via a shared memory interface built with the Xilinx libmetal library [64]. A device tree entry reserves a small block of memory within Linux at a specific address, and the same address and size is configured within the RPU code. The memory block is used as a circular buffer for communication between the APU and RPU. For each new request being delegated to the RPU, the APU inserts a small data structure into the circular buffer with the request type, parameters, and target execution time. On the RPU side, the realtime loop polls the circular buffer for new requests, and copies each new request into a

```
$dt \leftarrow targetTime_{N+1} - (startTime_N + duration_N)$
if $dt <$ MIN_FRAME_TIME then
    $exposure \leftarrow$ (desired exposure for N+1)    ▷ $dt$ is $\approx 0$ when working correctly.
else if $dt < 2 \cdot$ MIN_FRAME_TIME then
    $exposure \leftarrow dt -$ BLANKING_TIME    ▷ Not enough time for 2 frames, so use a single extended dummy frame.
else
    $exposure \leftarrow 100\mu$s    ▷ We have a long time to wait; just do a short frame.
end if
```

Algorithm 1: Calculate the exposure time for the next sensor frame given the current time and the desired starting time for a frame.

local queue for the corresponding device. Each queue is implemented as a linked list sorted by scheduled execution time, with the soonest first. After this check for new requests, the main loop examines the front of each queue to determine whether it is time to execute any requests. When it is, it takes the appropriate action, such as toggling a GPIO pin for a flash request.

Image sensors are a bit more complex, since (unlike the flash) the sensor does not simply operate on demand. Instead of checking every loop iteration whether the request time has been reached, the camera control executes once per frame, just before the sensor locks in the settings. The control routine calculates the time when the next frame will begin and checks it against the desired time for the upcoming target frame. If the time delta is less than one frame, then the next frame must be the target. If the time delta is less than two frames, then we can insert a variable-length dummy frame so that the subsequent frame will begin at exactly the target time. If the time delta is greater than or equal to two frames, then there is time for at least two dummy frames, so we insert the shortest possible frame now, and add any necessary padding later. This is summarized formally in Algorithm 1. The use of "dummy frames" is similar to the technique introduced by Ansari et al. to synchronize mutliple video streams [65], but applied to a single camera.

Only tasks which must be executed or timestamped with microsecond-level accuracy are handled by the real-time loop. It would be possible to delegate more —

for example, the work done by the kernel drivers could be just as easily done by the real-time loop. However, adding more tasks for the real-time loop increases the potential latency of that loop. Keeping the accelerator interface on the Linux/APU side also means that it is possible to work on hardware accelerators without booting up the RPU, and that the software remains compatible with the Zynq-7000 series parts which do not have an RPU.

4.3.3 Kernel drivers

A set of three kernel drivers perform work delegated directly by the F4graph APU runtime and indirectly via Halide functions. One driver handles allocation of physically contiguous memory buffers, which are easier for the hardware to use but cannot be allocated by userspace code. A second driver abstracts the DMA $\rightarrow$ accelerator $\rightarrow$ DMA design pattern and provides a simple queue-based interface to the software layers above. The third driver provides a similar abstraction for transferring camera images into memory. While the camera hardware is controlled by the RPU, the image data must be read into an Linux-allocated memory buffer. To do this, the driver maintains a queue of requests (with corresponding memory buffers) and configures the DMA engine connected to the camera for each new image.

Reasons for kernel drivers

Some of these operations could be performed in userspace without kernel drivers, but we chose to build kernel drivers to take advantage of hardware interrupts, which help the drivers achieve low-latency responses without continuous polling. The other challenge which must be handled in kernel space rather than userspace is memory paging. Userspace buffers are scattered page-by-page across physical memory, and userspace code is prevented from reading or controlling the physical addresses of these pages. While it is possible for a kernel driver to walk the kernel page tables and construct a list of physical page addresses (known in Xilinx terminology as a "descriptor chain") for the DMA engine to read or write using "scatter-gather" mode, this is costly both in terms of execution time and memory. With a standard 4 kB page size, an 8 MP RGB image is spread across 6000 pages, and the descriptor chain itself requires a substantial 384 kB (94 pages!).

Even more importantly, paged memory cannot be accessed arbitrarily. With a contiguous block of memory, it is possible to access sub-blocks of an image by specifying an offset and stride (i.e., the address difference between subsequent rows of the image). An image can easily be cropped or processed in tiles simply by manipulating the offset, stride, and size. Doing this with paged memory would require re-walking the descriptor chain even when only a small piece of the image is accessed, or using a TLB-like structure within the hardware accelerator.

Instead, we use special buffers which are contiguous in physical memory, provided by the aptly-named Linux Contiguous Memory Allocator (CMA). At boot time, we allocate between 100 and 500 MB for contiguous memory using the boot parameter `cma=100MB` (or `cma=500MB`, as appropriate). Once this space is reserved, the standard Linux kernel function `dma_alloc_coherent()` returns contiguous buffers allocated from this memory.[4] Using the CMA also solves the page-locking problem. With paged memory, the pages must be “locked” so they are not paged out to disk, since the hardware accelerator is unable to handle page faults. Because the CMA sidesteps paged memory entirely, these buffers are guaranteed to be resident in memory.

The downside is that the CMA allocation size must be chosen carefully and may never be optimal as the system load shifts. If the allocation is too large, then memory is wasted that could have been used by the OS; if it is too small, the camera and image-processing system may fail to allocate the memory it needs. While this was not an issue or our system (which only runs our image-processing applications), it could become a problem on real devices where the end user is running many programs simultaneously.

[4]Despite the name, this allocation does not ensure that the buffers are kept coherent between the CPU and FPGA hardware; that must be handled via hardware configuration or explicit flush/invalidation operations.

Kernel driver interfaces

Linux provides several mechanisms for userspace code to interface with kernel drivers, often summarized by the phrase, "everything is a file". That is, every hardware device attached to the system is represented by a file node in `/dev`, and these can be opened, closed, and read like other files. Of course, not every possible device operation fits into the `read()`/`write()` paradigm, so there are some additional operations to cover these cases. More concretely, Linux device drivers implement one or more of the following interfaces:

- `read` and `write`, which are the common operations on files. The Video4Linux framework, for example, implements `read` for the user to read a frame from a webcam or other device [66].
- `mmap`, which maps the contents of a "file" into the caller's memory space. This is typically used to create a userspace mapping into the device address space, which allows subsequent user code to read and write registers or device memory directly.
- `ioctl`, which is a sort of catch-all for other operations that drivers need to perform, but which do not fit neatly into the other interfaces. For example, the Video4Linux2 driver uses `ioctl` to set camera parameters such as resolution and white balance settings [66].

Since the accelerator reads and writes blocks of memory (much like Video4Linux devices), the `read`/`write` interface seems like a natural choice. However, `read` and `write` operate on userspace buffers, which would require copying the data back and forth from contiguous buffers or using scatter-gather DMA.

The second option is `mmap`. By mapping the accelerator registers into user memory space, the user has direct control of the hardware. In many cases, `mmap` can be used as a minimal kernel-mode driver, while a user-space driver does the heavy lifting. However, if we used `mmap` to provide access to the hardware registers, we would still need other mechanisms for contiguous buffers and interrupts.

Instead, we chose to implement the interface primarily with `ioctl`, using custom commands to dispatch and fetch contiguous buffers. An `ioctl` command takes the form:

```
ioctl(int fd, unsigned long request, void* argp)
```

The three parameters are

- `fd`, the file descriptor which has been `open`ed.
- `request`, which specifies which command to run. These commands are defined in a header file accompanying the driver.
- `argp`, which is a pointer to one or more arguments for the command. Arbitrary data can be passed with the `void*` pointer, provided the driver and user code agree on the layout and semantics.

This is a raw interface with no safety net, but it provides the most flexibility of the available driver interfaces in Unix. In a sense, each `ioctl` sub-command is like its own syscall, with its own semantics and behavior.

CMA allocation driver (`cmabuffer`)

Because memory operations are independent of the hardware, they are wrapped in a separate driver with two `ioctl` operations, `GET_BUFFER` and `FREE_BUFFER`. The former allocates a buffer and populates a structure with pointers for it; the latter releases the buffer to be used again elsewhere. To access the contents of the buffer, it must be mapped into the program memory space, which is performed with `mmap`. A typical usage follows the pattern below:[5]

```
int cma = open("/dev/cmabuffer0", O_RDWR);

Buffer buf;
buf.width = 256; // Width of the image
```

[5]Error-handling code is omitted for brevity, but proper usage would check for failures after `open`, `ioctl`, and `mmap` calls.

```
buf.height = 256; // Height of the image
buf.depth = 4; // Bytes per pixel
buf.stride = 256; // Pixels between successive lines

// Request a buffer of the size specified in buf
int ok = ioctl(cma, GET_BUFFER, (long unsigned int)&buf);

// Get a userspace pointer
long* data = (long*) mmap(NULL, buf.stride * buf.height * buf.depth,
                          PROT_WRITE, MAP_SHARED, cma, buf.mmap_offset);

// Fill the buffer with data, and do things with it
// ...

// Release the userspace pointer (and the corresponding kernel
   structures)
munmap((void*)data, buf.stride * buf.height * buf.depth);

// Release the buffer
ok = ioctl(cma, FREE_BUFFER, (long unsigned int)&buf);
```

Hardware accelerator driver (`hwacc`)

A second driver handles the interface with the accelerator itself. The essential `ioctl` command exposed by this driver is `PROCESS_IMAGE`:

```
int id = ioctl(hwacc, PROCESS_IMAGE, (long unsigned int)bufs);
```

Here, `bufs` is a pointer to an array of `Buffer` structs from the CMA allocation driver. The array contains at least an input and output image, but depending on the algorithm, may contain multiple input images, and/or small images holding "tap values" such as convolution weights or processing parameters.

As shown in Figure 4.10, the driver maintains a queue of buffer sets be processed, and each call to `PROCESS_IMAGE` drops another buffer set into that queue. Each time the driver receives a completion interrupt, it pops the next set (if any) off the queue, configures the DMA engines with the data addresses and sets the hardware to run. Having an input queue greatly relieves the timing burden from the userspace code: As

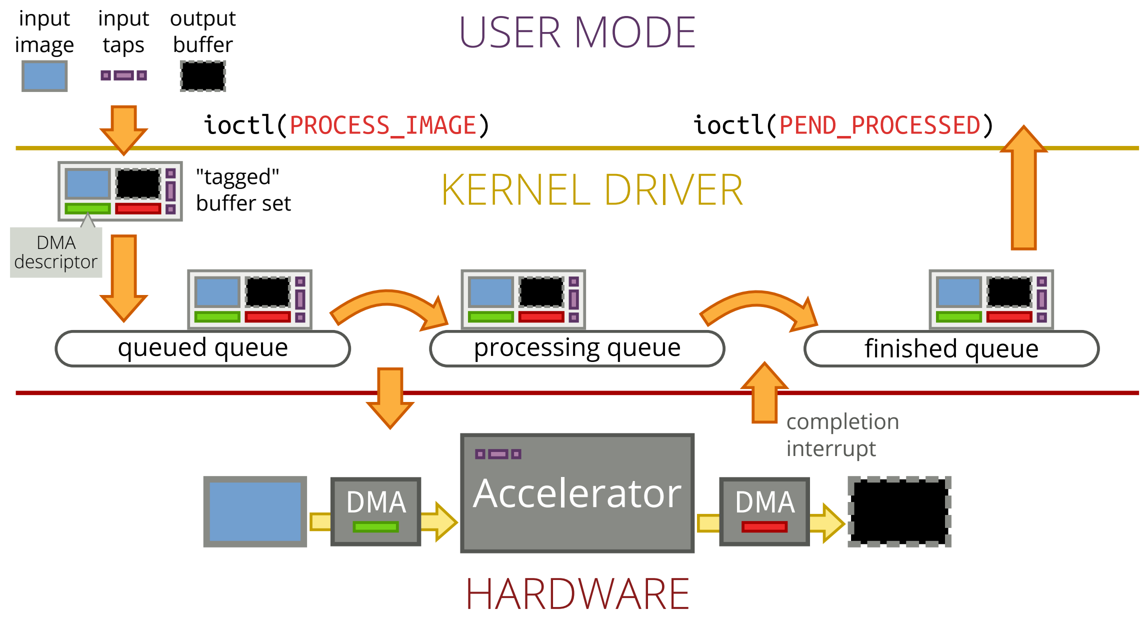


Figure 4.10: Internal operation of the driver. `PROCESS_IMAGE` places a buffer set onto the queue. Each time a completion interrupt fires, the driver starts the next buffer processing, and puts the completed one on the finished queue. `PEND_PROCESSED` blocks until the specified buffer set is in the "finished" queue, and then returns it to the user. Note that the "processing queue" isn't truly a queue, as it never holds more than one item.

long as there is at least one item in the queue, the driver will keep feeding the hardware continuously with no intervention needed. This also removes the burden of setting configuration registers in between processing runs. The configuration is bundled with the input and output buffers, and the driver applies the values immediately before kicking off the hardware.

Because the accelerator can be busy for hundreds or thousands of microseconds at a time, it is not appropriate for the `PROCESS_IMAGE` call to block the calling thread. Instead, it pushes the set of buffers onto the queue, and immediately returns a unique "tag" value. Later, the user code can make another `ioctl` call, `PEND_PROCESSED`, passing in the same tag. This call will block until the operation is complete, indicating that the buffer set is free to be processed by the CPU again. If the operation has already finished, the call simply pops the buffer off the output queue and returns immediately.

The `hwacc` driver must be configured to match the hardware it is driving, including the number of input and output streams, configuration addresses of the attached DMA engines, and the size and datatype of the image tiles. Our initial approach was to automatically generate a customized driver for each hardware accelerator, using the same templating scripts that generate the FPGA project. While this works, an approach more in line with modern embedded driver design is to use the device tree.[6] Our scripts extract the necessary information from the system configuration file and generated IP cores, and rather than parameterizing the driver, they construct a device tree entry for each module defining its inputs and output and linking to the device tree entries for the corresponding DMA engines. The kernel drivers read the device tree as they are loaded and use this to configure all of the necessary parameters. This streamlines the build process, since the drivers no longer have to be rebuilt with every change to the the hardware design. The device tree is always rebuilt for other reasons, so this was not an additional complication.

[6]A device tree is a standardized text file that defines the hardware peripherals connected to a Linux system. The kernel reads it on boot, and uses the information to initialize and configure kernel drivers as needed. The device tree is used primarily for embedded systems, where hardware is connected directly to the system bus and generally does not feature autodiscovery mechanisms present in traditional interface buses.

Camera DMA driver (`xilcam`)

Ideally, the `hwacc` driver would be the only necessary interface to the accelerator and DMA engines. However, `hwacc` is an adaption of the driver used in the original Halide-to-FPGA work, and was designed from the start only for the DMA $\rightarrow$ accelerator $\rightarrow$ DMA dataflow pattern. The driver does not have any notion of camera inputs, where data simply appears at an input without having come from main memory — and more crucially, neither does Halide. Rather than redesign the driver and extend the Halide code generator, it was easier for our prototype to create a simplified sibling of `hwacc` to handle the Camera $\rightarrow$ (optional) accelerator $\rightarrow$ DMA pattern.

This driver, called `xilcam`, provides the same queue abstraction as `hwacc` but only handles the launch of the AXI-stream-to-memory side of the DMA. The camera streaming and hardware accelerator configuration (if any) is initiated by the APU runtime and executed by the R5 real-time loop. The operation is similar to `hwacc`: the `ioctl` call `ENROLL_BUFFER` requests an operation by passing a buffer to the driver (akin to `PROCESS_IMAGE`), and the `WAIT_COMPLETE` call blocks until the buffer is filled with data and ready for the user (like `PEND_PROCESSED`). Because the input data comes from one or more cameras, the `ENROLL_BUFFER` call only needs to pass a single empty buffer for the output image.

4.3.4 Scheduling

The simple goal of scheduling is to take a complete `Graph` and compute an execution time for each request. The core of this task is delegated to an off-the-shelf optimization tool, so scheduling essentially consists of the following three steps:

1. **Compute the schedulable subgraphs.** First, the scheduler calculates the portion of the graph that can presently be scheduled for execution, which consists of all schedulable subgraphs which no longer have any dependencies on dynamic events (i.e., events with non-deterministic execution time). The scheduler is re-run after each dynamic event in order to schedule any pieces of the graph which were constrained to that event and now have known execution times.

2. **Map the problem into solver constraints.** A free variable is created for the start time of each request, and calls to the solver API create constraints between these variables. User-defined constraints imply $>$, $<$, or $=$ constraints between variables, current occupancy implies a start time after some known time point, and device utilization implies $a_{start} > b_{start} + b_{duration}$ OR $b_{start} > a_{start} + a_{duration}$, for every pair of requests on the same device.

3. **Run the solver.** After the solver computes the result, the scheduler sends each request to the hardware queue with its corresponding execution time. Finally, if the graph is executing repeatedly, the solver sets a timer to wake up and run the next scheduling iteration.

To implement the constraint solver, we experimented with both mixed-integer programming and constraint programming formulations of the problem, and tried four different solvers: MIP solvers Gurobi [67] and SCIP [68], and CP solvers Facile [69] and Google or-tools [70].[7]

Mixed-integer programming seems like a natural choice for the problem — integer variables handle the various ordering permutations, and the linear programming solution produces a real-valued time point for each request. Conversely, constraint programming works only on finite integer domains and is tailored for combinatorial optimization problems. However, by discretizing time into fixed intervals, the scheduling problem maps easily into a constraint programming framework. Using 1 µs increments provides sufficient granularity and does not appreciably slow down the solver.

All four solvers have Python interfaces, so we were able to write a front-end that maps scheduling problems into each of the four solvers for comparison. Figure 4.11 compares the speed of the four solvers on 350 random schedules generated by a simulation. This test was executed on a laptop computer (Intel Core i5-5200 at 2.2 GHz and 4 GB RAM) using the Python bindings for each tool.

The general trend is the same for all four solvers — the solve time increases

[7]Gurobi 8.0.0, SCIP 5.0.1, Facile 1.3, and or-tools 6.7.1

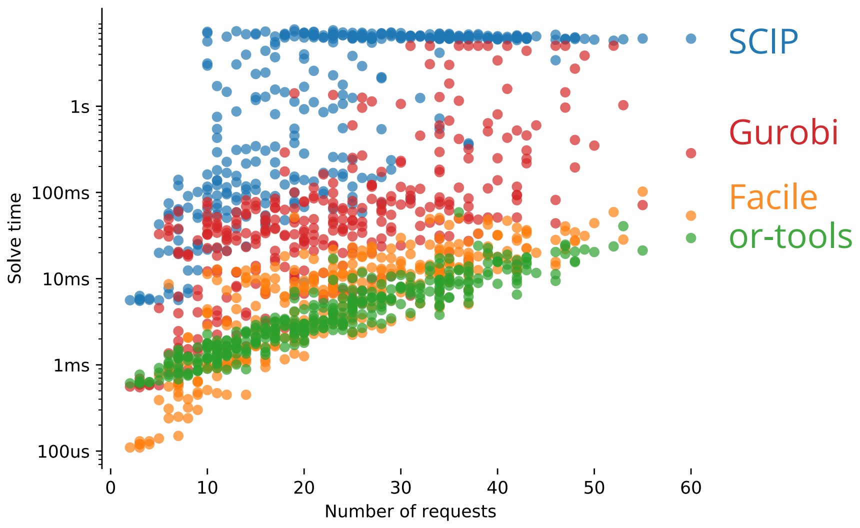


Figure 4.11: Comparison of solver speed on 350 randomly-generated schedules. Solvers were set to time out after 5 seconds.

exponentially with the size of the problem — with Google or-tools and Facile outperforming Gurobi by roughly a factor of 10 and SCIP by a factor of 100 or more for problems with more than about 5 requests. Facile is the fastest for very small problems, probably because it is a lightweight tool with a comparatively fast start-up time. However, Facile is only guaranteed to find a feasible solution, not necessarily an optimal one. Even so, Facile's feasible solution was identical to the optimal solution in 85% of the simulated cases.

We use the total overall schedule time as the objective function to minimize (i.e., `max(endTimes)`). Minimizing the end time for all requests (i.e., `sum(endTimes)`) has the potential to free up some devices earlier, but is significantly more constrained and more than doubles the average time to find a solution.

For the hardware implementation, we used or-tools (version 6.7.1), because it is open-source and could be ported to the aarch64 architecture, and because it has a supported C/C++ interface.[8] Only a couple of changes to the build files and one

[8]Facile's primary interface is in OCaml, and Gurobi is closed-source and does not run on aarch64.

architecture-specific file were necessary to run on the 64-bit Zynq ARM processor.[9] On the Zynq, the scheduler reliably finds an optimal solution in 1-2 ms for typical schedules with 10 or fewer requests. This is discussed in more detail later in Section 5.2.3. These are tiny optimization problems, and this scheduling overhead is small given the duration of the schedules being computed (tens to hundreds of milliseconds). However, if it were necessary to improve the performance of the solver, it would be possible to "pre-solve" parts of the problem using simple rules or heuristics, and then to use a lightweight algorithm to complete the solution. As we found with Facile, a search for a feasible solution will frequently arrive at an optimal one; additional heuristics might allow very rapid scheduling with near-optimal results.

4.4 Build system

The build system is an integrated collection of tools which produces software for the application, the real-time loop, kernel drivers, the FPGA bitstream, and everything else needed to boot the Zynq system. Figure 4.12 gives an overview of the tools and their interactions.

4.4.1 Build inputs

The user-written input to the system consists of the following:

- Halide source code for the image processing kernels. This includes both the algorithm and schedule, which specifies what parts (if any) will run on the FPGA.
- Halide kernel definitions, which are short metadata files describing the parameters that the Halide function expects. The Halide function is compiled into a shared library with only generic image parameters, so information about size and datatype must be passed through the metadata file. These files could be

[9]The version of `config.guess` within the COIN sub-build had to be updated, and the `ARCH_K8`-specific lines in `ortools/base/integral_types.h` were removed.

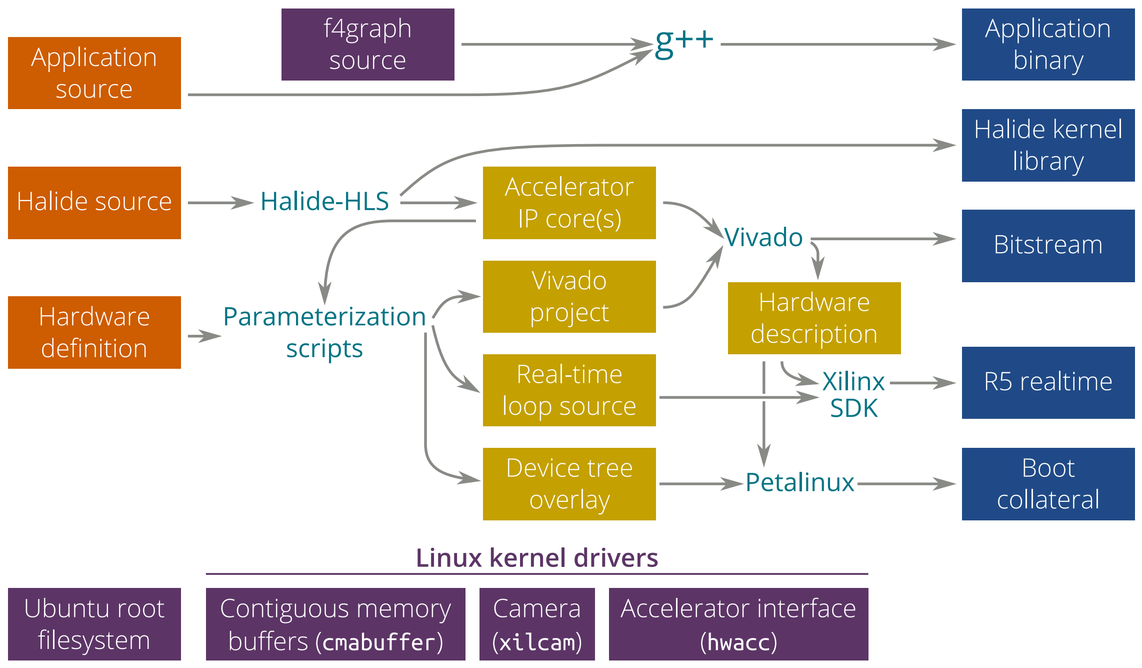


Figure 4.12: Overview of the complete build system. From left to right, user-supplied objects are in orange, intermediate artifacts in yellow, and outputs files in blue. Static parts of the system are in purple boxes, and tools are shown in green.

automatically generated with additional logic in the Halide compiler, but for now they are written by hand.

- A hardware definition file. As described in the previous section, the hardware definition is a YAML file naming the hardware modules and their interconnections, and providing basic board and project configuration.
- F4graph application source code, which is a C++ userspace application with the necessary calls to the F4graph library.

Additionally, there are a set of "static" components which are not design dependent, but must be compiled or installed at least once. The root filesystem for the Linux system is created automatically by a script, and can then be further customized by booting the system and using standard Linux tools. Our script uses a root filesystem based on Ubuntu Core rather than one generated from Petalinux, which means that the runtime software environment is essentially the same as our desktop development environments. A full-fledged shell is available, and useful tools and packages are easy to install via `apt-get`.

The F4graph library source is built as a static library which is later linked into the final application binary. One major advantage of using an Ubuntu filesystem is that F4graph code can be compiled on the Zynq itself. This means that all of the dependencies (particularly Halide-HLS and Google or-tools) can be installed using their standard build scripts and sub-dependencies can be pulled in with `apt` rather than requiring a cumbersome cross-compilation toolchain.

The three kernel drivers are likewise compiled on the Zynq device. For ease of distribution and deployment it would better to include these in the Petalinux build flow, but for rapid iterative development it is more convenient to compile them separately. Petalinux can take several minutes to compile kernel modules, versus a few seconds with the Linux build system.

4.4.2 Build and development process

No development cycle is linear, much less one with as many components as our camera system. Nonetheless, it helps to think of the development cycle and build process in three main phases: Halide development, FPGA design, and F4graph application development.

Halide development

A natural first step is to write the Halide code which will be used in the accelerator. Because Halide separates the algorithm and schedule, the algorithm can be developed and tested in a software-only environment before moving to the hardware. One way to do this is to write an F4graph application which uses webcams or video files as input and passes them to a software implementation of the algorithm. Once the algorithm has been verified, it can be scheduled for the FPGA and exported.

FPGA design

Once the hardware modules have been generated from Halide, the user can move into the FPGA development phase. This mostly consists of writing[10] the hardware definition YAML file, and then running all of the build tools to generate the outputs. In theory this process could be automated as a single "one-button flow", but in practice prototype development is not linear and it is helpful to re-run the tools individually as changes are made. Rather than a monolithic build, we use a collection of scripts to set up each of the three major Xilinx tools (Vivado, Petalinux, and Xilinx SDK) and then run them manually.

These scripts first read the hardware definition and inspect the referenced Halide modules to obtain details about the inputs and output of each module. Then they use this information with a set of templates to construct the inputs to the particular tools. We use the Mako template engine for this task because of its simple API and clean integration with Python. While Mako was developed with HTML generation

[10]Where "writing" usually means "copying an example and editing it".

in mind, it is sufficiently general-purpose that it works with all of the text formats we need to generate (Tcl, device tree text, and C code).

Vivado already has an extensive scripting interface which can be used to create and build FPGA projects, so we generate a Tcl script for Vivado that creates a new project and compiles it into a bitstream. Petalinux does not have a scripting interface, so instead we generate a complete project directory tree using the template and call `petalinux-build` to compile it. The R5 realtime build is nothing more than cross-compiled C/C++ code and could easily be automated with a Makefile. However, the XSDK project creation process is quick and simple, so for the purposes of prototype development we found it more convenient to run XSDK and take advantage of its IDE features. The templating script simply generates some of the C code files.

The result of running Vivado, XSDK, and Petalinux is a set of files which can be copied to the boot partition of the SD card. If the SD card is new, the user must also run the root filesystem creation script to initialize and configure the Ubuntu root filesystem. Once this is done, the UltraZed can boot off of the card to an Ubuntu login prompt.

Application development

With the system booted and running, the complete F4graph application can be developed and tested. If this is a new setup, the kernel drivers and F4graph dependencies must first be compiled; otherwise the existing installations can be reused without modification. The software components of the Halide modules must be rebuilt for the aarch64 architecture, and the kernel definition files placed in an accessible location. Again, a CMake build script automates this process. The F4graph application is a normal C++ application, and can include whatever code and libraries are necessary. A CMake build script handles compilation and linking with the F4graph library.

4.4.3 Launch process

All of these components are initialized and launched in several phases. When the system powers on, the bitstream is loaded into the FPGA before anything else happens.

Then the Linux kernel boots, using the SD card as its root filesystem. The kernel drivers are loaded and configured based on the device tree, and their handles appear in `/dev` once the system finishes booting.

Prior to launching the application, a short script is executed which enables memory coherency between the CPU and FPGA (via a single register write) and loads and launches the realtime code on the R5 cores using the `remoteproc` sysfs handle (part of Xilinx OpenAMP). Finally, the application itself is launched like any other Linux program. It opens the driver handles, loads the shared libraries containing the Halide kernels, and begins executing.

Chapter 5

System evaluation

This thesis has so far described a set of interfaces which enable userspace code to execute and respond to camera-related actions with microsecond accuracy and to interface with FPGA accelerators while hiding the peculiarities of the supporting hardware. We have argued that a practical implementation of these interfaces requires a real-time coprocessor, since a standard multi-core application processor running Linux is unable able to meet the desired timing specifications. This chapter provides experimental data to support these assertions, ranging from micro-benchmarks that measure system overheads to complete applications that demonstrate the performance and usability of the proposed API and our prototype system.

The following section begins with a series of micro-benchmarks which quantify the timing precision possible on both the APU and RPU. We find that although the APU can achieve very good timing precision, doing so reliably essentially requires reserving a CPU core for time-critical tasks. Given that a CPU must be reserved, it makes sense to use a Zynq RPU core rather than one of the full-featured application processors. Building on these results, the next section steps up the stack and explores the overheads that are incurred in our FPGA prototype. These include launching FPGA operations, synchronizing between the userspace process and the real-time core, and running the user-level software framework. While most of these overheads are minor, we find that the cost of initiating FPGA operations is substantial, and discuss how future systems could reduce this overhead. The final section moves up one more conceptual level to measure the runtime behavior of several complete applications built with F4graph.

5.1 Timing

This section evaluates the timing performance of our prototype camera system. First we review the timing constraints for peripherals on our prototype platform. Then we benchmark the timing performance of the A53 running Linux against the RPU running our bare-metal control loop, testing their accuracy both for initiating and responding to events.

5.1.1 Timing constraints

Several pieces of hardware impose timing constraints on the system, the most obvious of which is the image sensor. The IMX 219 latches its settings with a vertical-sync signal once per frame, so the "configuration window" for each frame includes the entire previous frame. Given a maximum frame rate of 180 FPS, the configuration window is 5.5 ms. As another point of reference, the On Semiconductor AR1011HS uses a similar latching scheme but is capable of framerates up to 1200 FPS, for a minimum configuration window of 830 µs.

Inertial measurement units require similar timing precision, with typical sample rates between 200 and 1000 Hz[49]. Units such as the Invensense MPU-9250 include a small "digital motion processor" which handles task of sampling the sensors at a high rate and performing sensor fusion, specifically so that the host processor can sample the orientation at a lower rate. However, using the IMU for optical image stabilization or rolling shutter correction requires multiple orientation samples per frame, pushing the host processor timing requirements back into the 3-5 ms range.

Far more stringent than either of these requirements is the flash. With a rolling-shutter camera, the timing of a flash will be visible within one or two lines of the image. For the IMX 219 operating at full resolution with 2 data lanes, one line of the image corresponds to about 19 µs; at reduced resolution and 180 FPS, this drops to

9 µs. Said another way, the sensor settings must be configured once per frame, the IMU read several times per frame, and the flash driven several hundred times per frame.

5.1.2 Interrupt latency

Given these timing requirements, we can evaluate the timing precision both the APU running Linux and the R5 RPU running our bare-metal dispatch loop. There are several scenarios which must be considered as part of a complete understanding of “timing precision.” For timestamping events such as receiving the beginning of a frame or a shutter-button press, we need to examine the external interrupt latency — the interval between the event occurring and the processor reacting to record the timestamp.

On the RPU, the interrupt latency is easy to measure: the IRQ is configured to set an output pin high as soon as the interrupt occurs. The interrupt is triggered by external hardware, and a logic analyzer measures the interval between the interrupt signal and the response pin. Over 10,000 trials, the latency from the interrupt pin being driven high to the response pin going high was approximately uniformly distributed between 550 ns and 785 ns, meaning that any timestamp has a potential error of ±235 ns.

The interrupt latency on the APU is slightly more complex. Following good design practice, our kernel driver interrupt handlers are split into two parts, known in Linux terminology as the “top half” and “bottom half” [71]. The top half is called immediately in response to the interrupt, but runs in a restricted context where it cannot do anything that might block or fail (at the risk of a kernel panic). As a result, the top half’s only job in our implementation is to queue the bottom half for execution and to mark the interrupt as handled. The bottom half does the actual work of the interrupt handler. As with the R5, we trigger the interrupt with an external pin, and each half of the handler sets an output pin high. The test was run both with the Zynq APU cores idle and at full load. The results are summarized in Figure 5.1 along with the R5 results.

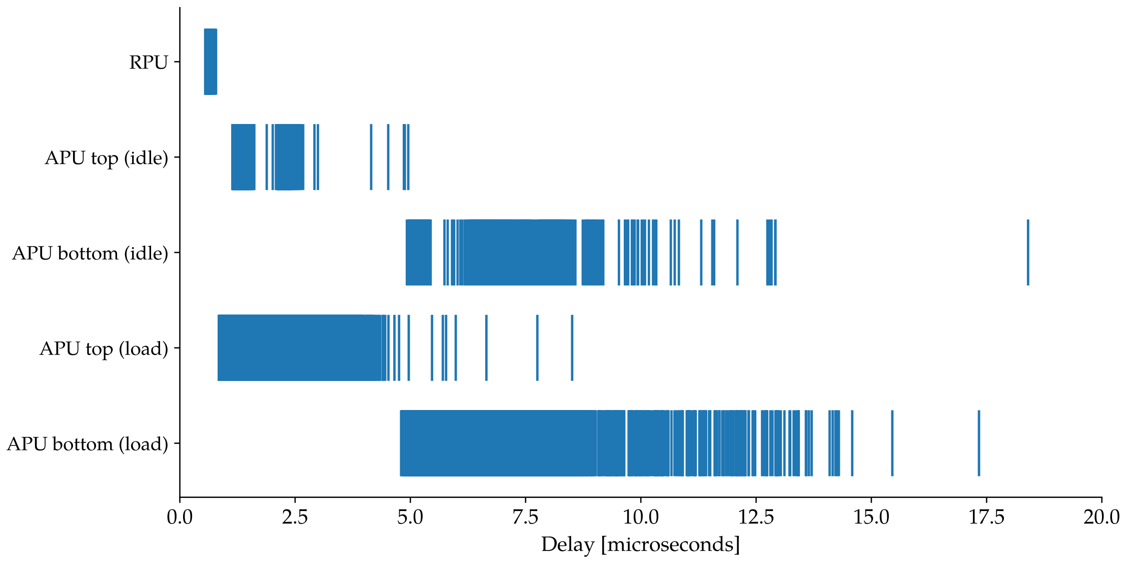


Figure 5.1: Interrupt latency for RPU (running a bare metal ISR), and APU (captured in a Linux kernel driver). The kernel driver is split into a top and bottom half, and behaves differently under load, so all four permutations are shown here.

In general, the APU responds within the timeframes outlined in the previous section — on average, executing the top-half handler within 3 µs of the input event and the bottom half within 6 µs. However, there is a long tail of much slower responses. In the samples recorded here, the timing uncertainty is as large as 13 µs, and even longer intervals may occur on rare occasions. This is acceptable for frame-level or even sub-frame timing, but falls short for recording or synchronizing events at the scanline level.

5.1.3 Dispatch latency

To quantify the precision with which the system can launch actions, we must examine the time to execute each iteration of the dispatch loop on the RPU. The current implementation of the loop is not strictly deterministic — the time for each iteration depends on what actions (if any) are being processed. When idle, the loop runs in 2.7 µs. The longest delay is incurred when initiating the I2C transaction to reconfigure

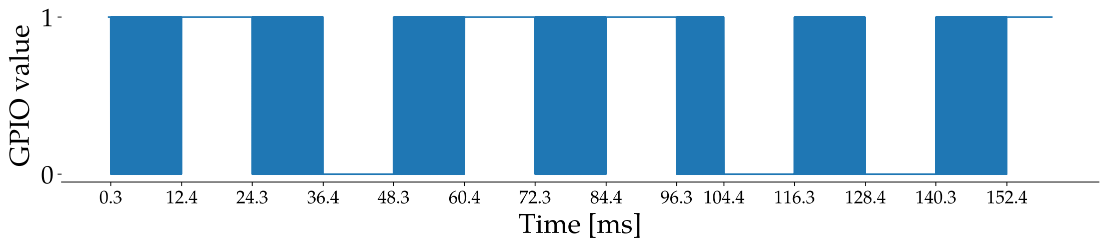


Figure 5.2: Output captured by a logic analyzer while rapidly toggling a GPIO pin from a userspace thread with the system under heavy load. The solid blue regions indicate when the task is running. The task is switched out at intervals of 8-12 milliseconds, severely limiting the timing precision of a userspace thread.

the image sensor for the next frame, which stretches the loop time to 14 µs. Only one long-delay action is allowed to run per loop iteration, guaranteeing that a single flash request will execute within 14 µs (about one scanline) of the scheduled time.

To maintain this guarantee, a production system would need to prevent many time-sensitive requests from stacking up at the same time. This could easily be achieved by applying system-level scheduling constraints (e.g., any two flash requests must be 3 µs apart). Applying these constraints has the added benefit of making this low-level system limitation visible to the user rather than silently introducing unexpected behavior.

The APU cores are more powerful processors running at a higher peak clock rate (1.1 GHz vs 500 MHz), so an equivalent loop on the main processor would run faster. However, we also have to account for preemption due to running within a multitasking OS. To measure preemption delays, we created a userspace program that drives a square wave on an output pin at 1 MHz and monitored the output with a logic analyzer. This test was run using the default scheduler in the Petalinux 2017.2 kernel (Linux 4.9.0), and with the task at the default priority level. With the system idling or running light tasks, the process runs continuously without interruptions. Since the Zynq has four APU cores, one core can continue uninterrupted while the other three handle intermittent operating system tasks. Under heavy load, the task begins to be switched out for intervals of about 12 ms, as illustrated in Figure 5.2.

Of course, it is possible to tune the scheduling parameters of the Linux kernel and configure the priority and scheduling behavior of the task to minimize the possibility and duration of these interruptions. If we take this approach to its logical end, we are dedicating a processor core to this real-time task, which is exactly what we are advocating. However, there is no need for a large and power-hungry application processor; a low-power, low-speed processor is sufficient. The real-time dispatch loop does not require intense processing; it simply needs continuous attention.

5.1.4 Camera dispatch

The image sensors have their own timing qualities, since they are clocked independently of the main processor. As described earlier, the sensors are configured during a window of several milliseconds while the previous frame is being captured. The actual timing of the image capture is controlled by inserting dummy frames with a variable exposure, and we can analyze this to obtain a quantitative error bound on the timing error between two frames. The exposure for the IMX 219 is set in increments of the row readout time, which means that our overall timing granularity is determined by the row readout time. Given a random target time for the beginning of the frame, we can achieve the nearest integer multiple of the row time, leaving a random residual error up to half of the row time.

To confirm this, we ran a simple experiment: a test program running on the RPU generated random target timestamps for a series of 1000 frames, and attempted to capture frames at those times. A histogram of the timing error (the actual frame timestamp minus the target) is shown in Figure 5.3. As anticipated, the error is a uniform random distribution from $-t_{row}/2$ to $+t_{row}/2$. In this case, $t_{row} = 18.8\,\mu s$ and the timestamps are only taken to the nearest microsecond, giving an error range from $-9\,\mu s$ to $10\,\mu s$. With two or more cameras, the error between any pair becomes a symmetric triangular distribution centered at 0 and ranging from $-t_{row}$ to $+t_{row}$. This means that two cameras capturing side-by-side will be synchronized within one scanline.

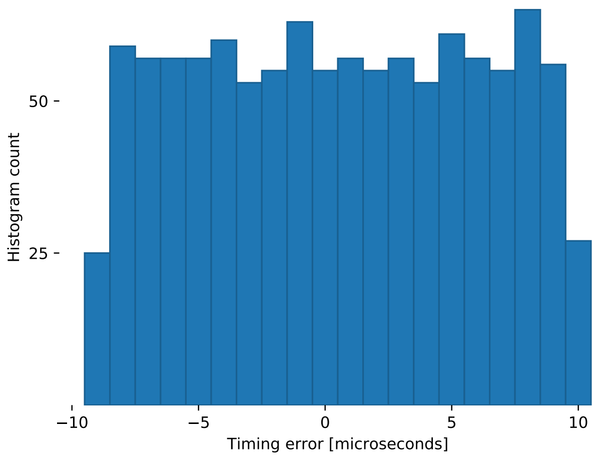


Figure 5.3: Histogram of the timing error for 1000 frames scheduled for random timepoints. The system is able to reliably hit the timepoint within 10 µs, half of the the image sensor's row readout time.

5.1.5 Timestamping

Finally, the timing precision is potentially affected by the time required to read from the 100 MHz global clock.[1] To measure this, we executed a test on both the RPU and the APU from userspace: A function executes a loop 100,000 times, reading the clock twice in sequence and calculating the time difference between the two reads. This effectively calculates the time required by an entire read operation, which is an upper bound on the timing error due to this function. Histograms of the results from both the APU and RPU are shown in Figure 5.4.

The time to read the clock from the RPU is typically 0.56 µs, with a maximum time of 0.58 µs in our test. On the APU, the typical read time is 0.64 µs, with a very few ($< 0.03\%$) taking longer than this. No reads took longer than 10 µs. The timing under heavy CPU load is essentially the same, except for occasional outliers when the thread is switched out for 8 ms or more.

[1]This is slightly complicated by the fact that each clock read actually requires two reads and a simple calculation to create a 48-bit clock from the two synchronized 32-bit clocks which the hardware provides. At 100 MHz, a 32-bit unsigned clock will roll over every 43 seconds.

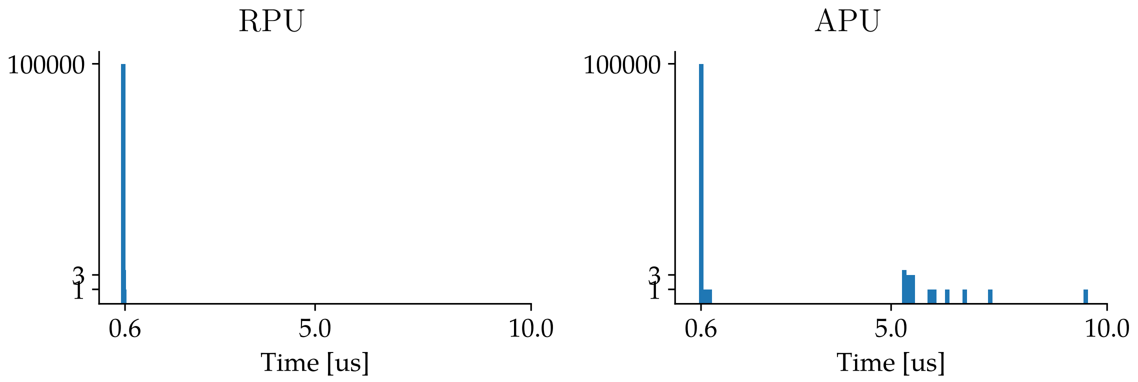


Figure 5.4: Histogram of time taken read the system clock on the APU and RPU. Any variation introduces uncertainty into the timestamps, but both APU and RPU reliably read the sensor in less than 1 µs. Even the handful of outliers on the APU are less than 10 µs. Note the log scale on the vertical axis.

5.1.6 Real-time cores for real-time threads

These micro-benchmarks demonstrate the value of the Zynq RPU and of small co-processors more generally. When the system is idle, the APU is easily capable of timing precision on the order of 10 µs, and even userspace applications can operate this precisely without any special tricks. However, this degrades precipitously to about 10 ms/100 Hz when the system reaches full load. Additionally, the timing behavior on the APU the behavior is stochastic, which is problematic if the application cannot tolerate failures.

Of course, many knobs can be tweaked to adjust timing response of the system — increasing the thread priority, changing the kernel task scheduler, or even designating an entire core for a task. Each of these tweaks effectively reserves all or part of a CPU core for a specific thread, generally at the cost of overall throughput. We satisfy this need by using the Zynq RPU for high-precision camera tasks, since simple processors are cheap and in the case of the Zynq Ultrascale+ chip, already built in. In this way, the timing-critical thread runs uninterrupted, and applications access it through a limited and structured API. User applications can specify the timing constraints which matter and allow the system to handle the rest, maximizing flexibility and overall throughput while meeting the needs of the application. We now turn to examine the cost of this allocation and API.

5.2 Runtime performance and overhead

The previous section demonstrated the value of a real-time coprocessor for achieving precise timing. But the benefits are not free: sharing work between the APU and RPU takes some processor time, creating “overhead”. This is true more generally; the abstractions proposed in this thesis all require resources to implement. On the hardware side we must consider the cost of delegating work to the FPGA accelerator and to the real-time core; on the software side the request scheduler and runtime framework both introduce overhead. The rest of this section examines the cost of implementing these abstractions.

5.2.1 FPGA Acceleration

Although the purpose of an FPGA accelerator is to reduce the work done by the host CPU, a substantial amount of work is required to configure and launch the accelerator and to synchronize with the hardware again when it completes. Depending on what task is being accelerated and how, the time taken to manage the accelerator can be larger than the computation time that was saved. This section details each of the subtasks that contribute to this overhead, explaining why they exist and what can be done in future systems to reduce, amortize, or completely eliminate them.

First, as described in Section 4.3.3, our FPGA accelerators work best with special contiguous memory buffers which are allocated by a kernel driver, and allocating these buffers takes a significant amount of time. While running a benchmark application with 72×72 RGB image tiles (15kB per tile), the driver spent an average of 120 µs to allocate each pair of input and output buffers — larger than the amount of time spent actually processing the image. By profiling the benchmark application with the Linux `perf` utility, we find that the majority (82%) of the time time taken by the `GET_BUFFER` call is spent inside the Linux kernel function `dma_alloc_coherent`. Kernel functions make up a similar fraction of time spent on the `FREE_BUFFER` call.

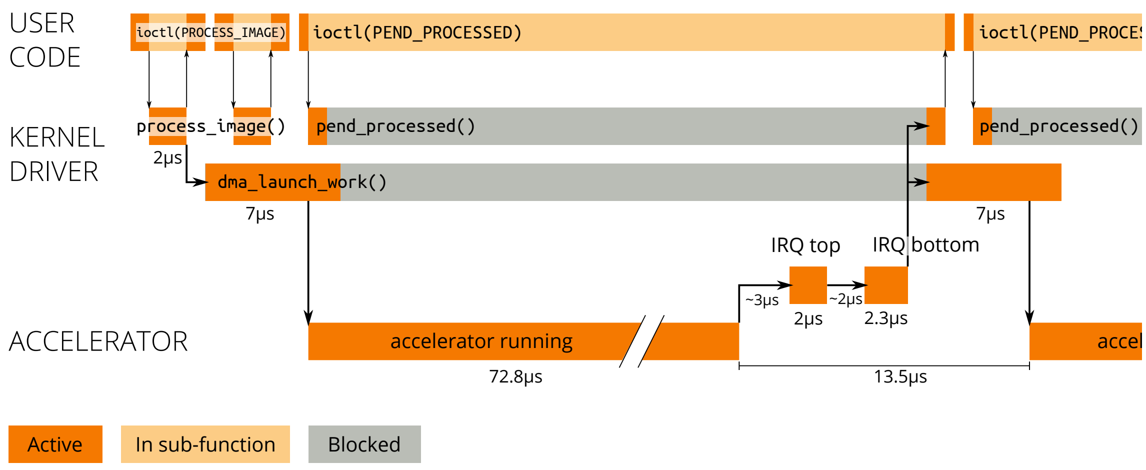


Figure 5.5: Timeline of hardware and software operations while processing two image tiles in sequence, showing the sequence of calls and the overheads incurred. As long as multiple tiles have been queued up, the critical path between completing one tile and launching the next takes about 13.5 µs. The remaining software overhead overlaps with the accelerator run and does not reduce the overall processing throughput.

There is no way around the basic need for contiguous buffers: the other option is to identify all of the pages which compose a userspace buffer and write this into a scatter-gather table for the DMA engine, which demands even more from the kernel. However, it is not necessary to allocate a new buffer for every image tile, or even to make a kernel request for every allocation. If the amount of memory is known (or even estimated) beforehand, a pool of buffers can be preallocated at the start and this time can be amortized over the entire runtime of the application. When our driver is configured to do this, claiming a slab of pre-allocated memory takes a mere 2 µs, and releasing it is similarly fast.

In addition to memory allocation, the kernel drivers take time responding to interrupts, switching between multiple threads, and actually configuring the accelerator for each processing run. Figure 5.5 shows a representative timeline of the hardware and software activity while processing two image tiles in sequence. The CPU-based work can be broken into two categories based on whether it lowers the overall throughput of the system or merely consumes CPU cycles while the accelerator is running. The first category consists of the driver's critical path from receiving the "accelerator finished" interrupt through setting the accelerator running on the next image tile (inside `dma_launch_work`). To measure the hardware downtime due to this software overhead, we used the Xilinx integrated logic analyzer (ILA) to capture the cycle-by-cycle operation of an accelerator while the aforementioned benchmark application was running. The benchmark application queued up multiple image tiles so that the kernel driver was always supplied with image data to process. For our benchmark accelerator working on 72×72 RGB tiles at 150 MHz, a single processing run takes 72.8 ± 0.3µs. There is a gap of 10-35 µs between runs while the software re-launches the accelerator. The first re-launch can be slow (around 30 µs) but subsequent intervals average 13.5 µs[2].

This critical path for re-launching the accelerator can be broken into three pieces: First is the latency from the time the hardware raises the interrupt signal until the interrupt handler begins running. As detailed in Section 5.1.2, this time is on the order of 2-3 µs. Second is the execution of the top and bottom halves of the interrupt handler. The time here is mostly spent acquiring a mutex to safely move the completed buffer between queues and switching threads between the top and bottom. Finally, the `dma_launch_work` thread is woken up and it proceeds to write the registers of the DMA engine and accelerator. This routine completes in about 7 µs, most of which is actually consumed by the register writes themselves.

The second category of overhead includes processes which execute while the accelerator is running rather than during the reconfiguration interval. While these operations do represent CPU cycles which do not directly contribute to processing image data, they overlap with the accelerator run and do not reduce the overall throughput. Using `perf`, we measured the overall CPU utilization while continuously processing tiles was 43% of one CPU core,[3] versus $< 1\%$ when the system is idle. Said another way, 37 µs worth of CPU cycles are consumed for each run of the accelerator.

Neither of these categories can be eliminated, but they can both be reduced in several ways. First, the re-launch time is essentially constant and can easily be amortized by using larger tiles or otherwise doing more computation per accelerator

[2]Figure 5.5 shows the typical "warm" timing representative of processing many tiles in sequence. An actual run with only two tiles would take slightly longer.

[3]In practice, the work is distributed across two or three cores, using a smaller fraction of each.

invocation. Given the line-buffered pipeline architecture of the accelerator, taller tiles do not require any more internal storage and are essentially free to implement. Wider tiles do require more intermediate storage, but may also be an option to further amortize the overhead.

Another option is to make the hardware more intelligent such that it can perform tiling on its own. We make the case later for an improved DMA engine; adding multi-tile processing would be a natural extension. A final option is to delegate the tiling and accelerator orchestration to a small CPU core like the Zynq RPU, much as we have already done for camera tasks. With such an arrangement, the APU could delegate an entire image for processing with a single command and leave the coprocessor to complete the work. Compared to doing tiling in hardware, this approach is more flexible (and can even be reconfigured on the fly), while maintaining fast and deterministic interrupt responses and eliminating the need for thread switching.

5.2.2 RPU offload

Another overhead is the cost of transferring work from the APU to RPU via the shared memory interface. To quantify this, we created a benchmark application where the APU sets an external pin high and then notifies the RPU by writing to the shared memory mailbox. When the RPU receives the message, it sets the pin low again. We executed this transaction 10,000 times and monitored the output pin with a logic analyzer.

The shared memory interface itself is extremely fast (approximately 250 ns per transaction), so the overhead is essentially determined by how frequently the RPU checks the mailbox. Our implementation polls the mailbox once per iteration of the processing loop, resulting in an average response time under 3 µs. While it would be possible to go slightly faster by using interrupts to notify the processor of new messages, the benefit would be negligible given that the RPU cannot dispatch events faster than the processing loop can run.

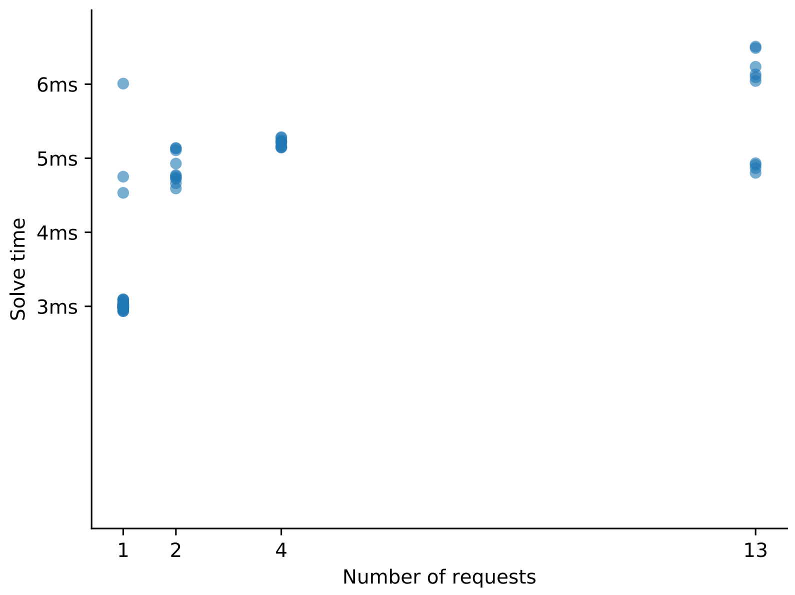


Figure 5.6: Time taken by the APU to schedule graphs of various sizes while running the complete applications described in this chapter. In general, the number of constraints is roughly equal to the number of requests.

5.2.3 Scheduler

The hardware action scheduler is at the heart of our method for separating the user from hardware timing concerns, but it too incurs some overhead. Section 4.3.4 described the scheduler implementation and compared the performance of various constraint solvers on the scheduling problem; here we examine that overhead in the context of the prototype hardware system.

Figure 5.6 shows the time spent by the APU to schedule graphs of various sizes while running the applications described in this chapter (viewfinder, stereo, flashflag, and others). Applications which schedule only one or two requests at a time are dominated by the startup time of the scheduling library, and typically execute in 3-5 ms. Additional requests increase the scheduling time only slightly, and scheduling up to 13 requests still completes in around 6 ms.

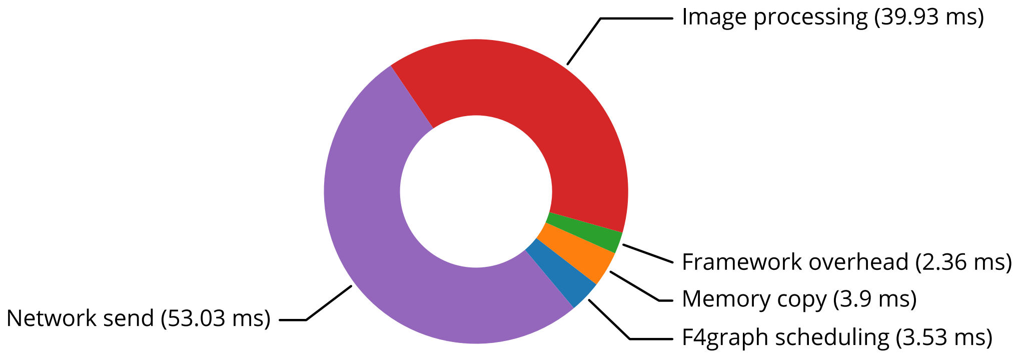


Figure 5.7: Breakdown of the CPU time spent by the APU (milliseconds per frame) while running a streaming application which captures images, processes them in software with Halide, and sends the result over the network to a remote computer. Note that the labeled times represent work done across multiple cores, so the wall time elapsed per frame is lower than their sum.

5.2.4 Graph execution framework

The final component which introduces overhead is the userspace runtime that runs the complete processing graph. One way to measure this is to run a complete F4graph application while monitoring the system with `perf`. Figure 5.7 shows a breakdown of the CPU execution time while running a simple application which captures images from a single camera, processes them, and streams them over the network.

The measurable overhead of the framework is small, since the framework does very little besides chaining image processing operations together. In this example, more than 90% of the CPU time is spent inside the processing nodes. Of the remaining time spent running the application, 3.5 ms per frame (3.4%) is spent scheduling the graph (discussed in the previous section) and 3.9 ms (3.8%) is spent copying data from a contiguous buffer to a normal userspace buffer for downstream processing. The rest of the framework takes less than 3% of the overall runtime.

The real opportunity to improve performance is not by reducing the CPU footprint of the graph framework code, but rather by using the graph framework to fully utilize the computing resources. In our prototype system, the user can explicitly create threads within F4graph as well as split image processing kernels across multiple threads using Halide, but it is challenging to achieve optimal parallelism this way without extensive experimentation. Possible ways to improve this are discussed in Chapter 6.

5.3 Applications

The previous sections examined the performance of various components of the system; this section looks at the overall system behavior with three sets of applications. We begin with a fairly typical viewfinder and capture application. This shows how the API works in a simple application and illustrates how the system allows multiple graphs to share resources. This is followed by a set of applications which demonstrates the timing precision of the end-to-end system in terms of its ability to synchronize events or launch actions with exact timepoints. The final group of applications illustrate how the system performs under load and with errors.

5.3.1 Viewfinders and basic capture

The examples in Chapter 3 illustrated a number of simple graphs which stream or capture shots. In this section, we combine them into a complete viewfinder/snapshot application. The application should behave like a typical camera app: a live view of the camera is shown on the screen, possibly with a set of viewfinder aids. When the shutter button is pressed, a single shot is captured and saved. This demonstrates the ability to run graphs either once or continuously, to execute multiple graphs simultaneously, and to accelerate image processing on the FPGA.

Since this application requires actions at two different rates — the viewfinder runs continuously at a "live" framerate, and the capture runs just once — we break it into two separate graphs. The first executes repeatedly to capture the stream of images for the viewfinder while the second runs when the shutter button is pushed to grab a single snapshot. We'll refer to these as the "viewfinder graph" and the "capture

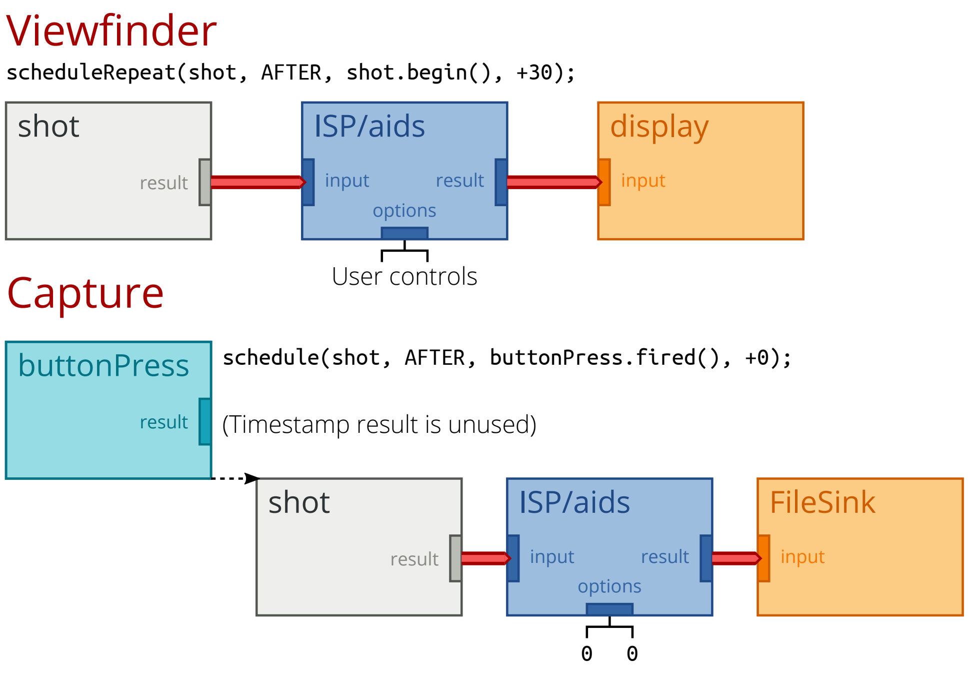


Figure 5.8: Parallel graphs for the basic camera application. The viewfinder (top) captures frames continuously and routes them to a display, while the capture graph (bottom) captures a single shot when the shutter is pressed.

graph" respectively. Diagrams of the two graphs are shown in Figure 5.8.

In the viewfinder graph, a single camera captures a stream of images, which are processed by a `HalideNode` and then sent to the display. The processing node also accepts a parameter bundle which controls two simple viewfinder aids: a zebra pattern that highlights overexposed (clipped) areas of the image, and a Sobel filter which highlights in-focus areas of the image. Application code can toggle these aids on the fly.

In the capture graph, a ButtonPush request waits until the button is pushed and then fires its event and triggers a `Shot`. The resulting image is passed through the same processing (and on the FPGA, the same hardware) as the viewfinder, but with the viewfinder aids turned off.

Figure 5.9 shows a timeline of the viewfinder process and the interruption to capture a frame when the button is pressed. Even with an instantaneous reschedule

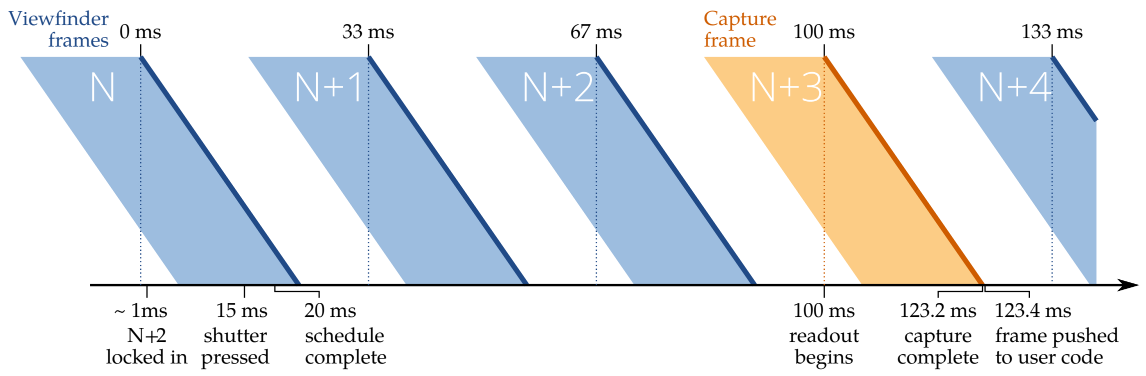


Figure 5.9: Timeline of actions for the viewfinder application when the shutter button is pressed.

time, there will be some shutter lag because of the pipeline required to configure the sensor settings and read out the data. A solution to this (known as “zero shutter lag”, or ZSL capture) works by continuously capturing at full resolution while watching the shutter button. When the button is pressed, the frame captured closest to the shutter time is kept as the final picture.

To implement this, the system must accurately timestamp the images and button-press event, and must route the captured images both to the viewfinder and to the user code for buffering. Both of these are straightforward in F4graph. Like the traditional capture/viewfinder, we split the application into two graphs which run concurrently, shown in Figure 5.10. The first drives the image capture, processing, and display, while the second merely watches for the button press and returns a timestamp. The application receives the completed frames in parallel with the display, and buffers the most recent one. When the application receives the button event, it examines the timestamp and chooses the nearest frame in time to save to disk while the viewfinder continues running.

To confirm the timestamp accuracy in the ZSL application, we wired a strip of LED lights to the shutter button. If the exposure time is short, the rolling shutter will cause a clear horizontal line in the output image when the flash turns on, as illustrated in Figure 5.11. Since the timestamps are accurate to within a few microseconds, we can use them not only to select the correct frame, but also to predict exactly where the

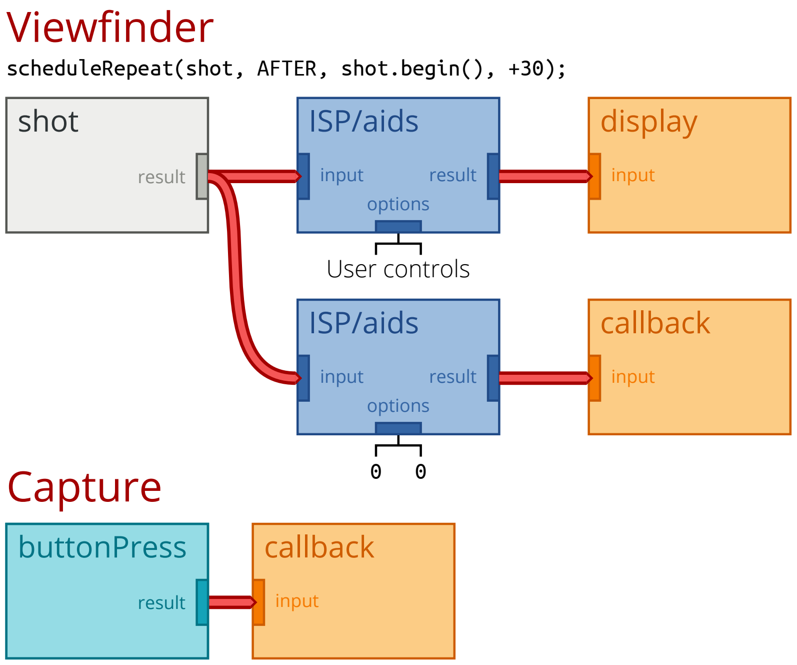


Figure 5.10: Parallel graphs for the ZSL camera application.

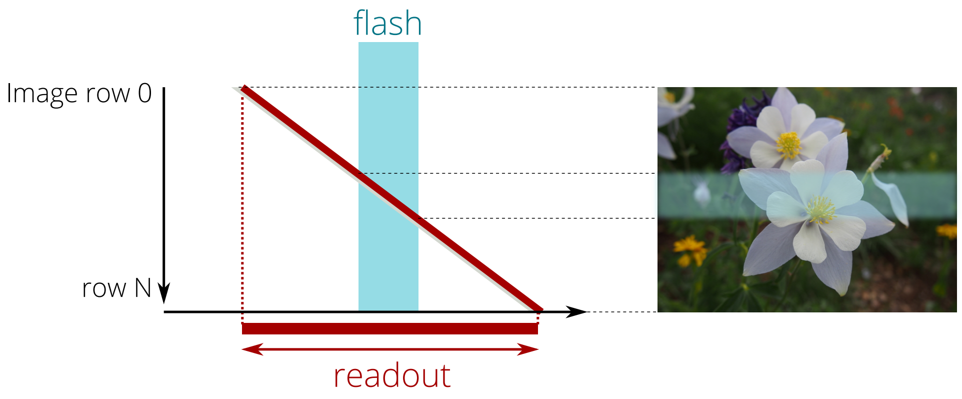


Figure 5.11: A brief flash during a rolling-shutter readout will appear as a bright horizontal band in the output image.

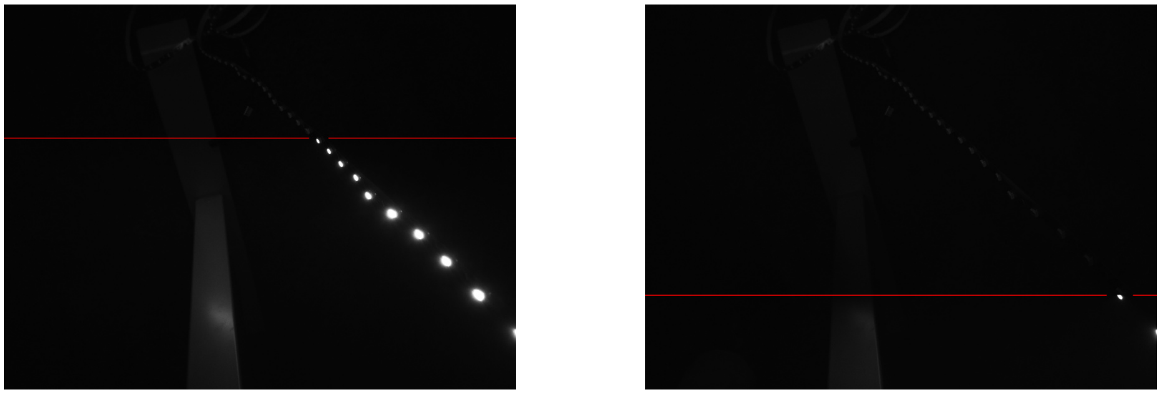


Figure 5.12: Example shots of a strip of LED lights captured with zero shutter lag using a very short exposure. The lights were wired to an external switch which also served as the shutter button. The red line highlights the scanlines of the image which were exposing when the “shutter” was first pressed according to the event timestamps, and this precisely matches the first rows of pixels to see the LEDs turn on.

flash line will occur in the image[4]. Figure 5.12 shows two example shots captured with this setup annotated with the edge of the flash band calculated from the timestamps.

5.3.2 Precise timing and synchronization

The zero-shutter-lag application demonstrates the ability of F4graph to accurately timestamp events. In this section, we show several applications which test the ability to synchronize requests in various ways. The first, which we have already discussed briefly, is to capture a stereo pair at exactly the same time for depth estimation via disparity. To do this with F4graph, we simply schedule one `Shot` to occur at the same time as the other:

```
Shot* left = new Shot(leftCam);
Shot* right = new Shot(rightCam);
g.addRequest(left, "left");
g.addRequest(right, "right");

g.schedule(left, AT, right.start, +0);
```

[4]Specifically, image row = (button timestamp − frame start timestamp)/18.903 μs

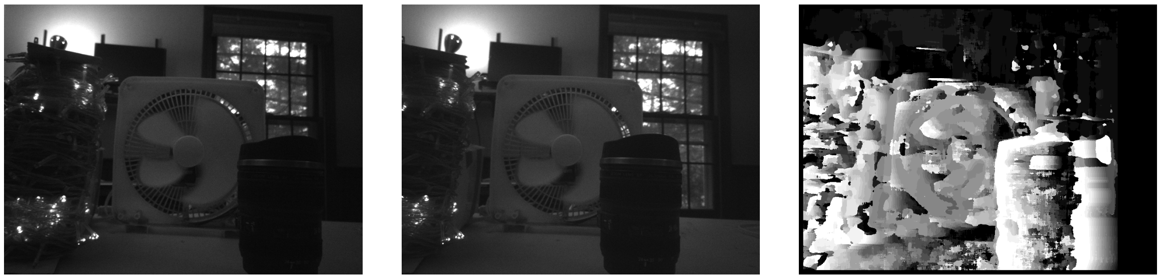


Figure 5.13: Demonstration of top-of-frame synchronization, and output of stereo pipeline (where brighter pixels indicate larger disparity).

The corresponding processing graph was introduced as an API example in Figure 3.8. The left and right images are fed into a single processing block which produces the stereo result.

To perform depth processing on the FPGA, we adapt the algorithm used by Pu et al. [47]. The two input images are assumed to be rectified such that the epipolar line is horizontal,[5] and the image streams coming from the cameras are passed through a simplified demosaicker which produces a grayscale image. Then, a 9×9-pixel patch surrounding each pixel in the left image is compared with patches along the epipolar line in the right image using the sum of absolute differences (SAD) metric. The offset producing the closest match is assumed to be the correct disparity.

Figure 5.13 shows a pair of images of a spinning fan and a PWM-dimmed light string captured with the synchronized cameras. The rolling shutter artifacts are clearly visible in the distortion of the fan blades and the alternating horizontal bands where the lights (which appear to the naked eye to be steadily on) are turned on and off. Because the cameras are synchronized, these artifacts are identical across the two cameras and the stereo disparity estimation is not compromised.

Just for fun, we can use carefully-timed flashes synchronized to a rolling-shutter exposure to produce pictures. Figure 5.15 shows a simple setup to capture the American flag, using a total of eleven requests on three separate flashes. Programmatically creating requests to make the stripes is easy with a `for` loop:

[5]In our test setup, the images straight from the camera have some misalignment and distortion and would benefit from proper calibration and rectification.

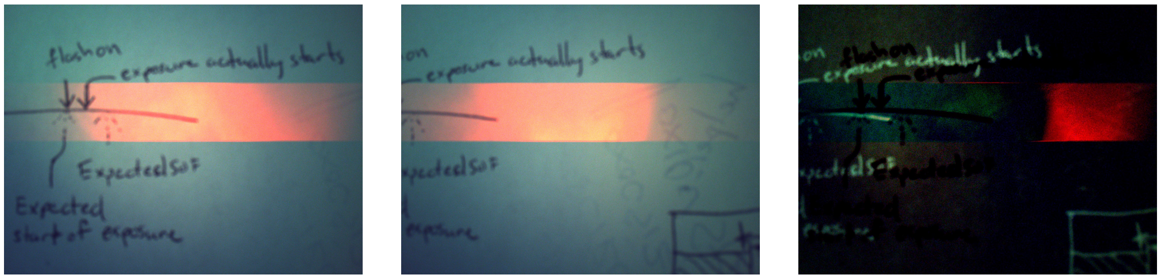


Figure 5.14: Demonstration of top-of-frame synchronization using a flash. The first two images are a left/right pair, and the third is an amplified difference image. The flash brightens the same horizontal slice of both images, indicating that the exposures began at the same time.

```
float stripetime = IMAGE_HEIGHT / 13 * LINE_TIME;

FlashRequest* stripesRight[7];
FlashRequest* stripesLeft[3];
for(int i = 0; i < 7; i++){
  stripesRight[i] = new FlashRequest(red1);
  stripesRight[i]->flashDuration = (unsigned int)stripetime;
  g.addRequest(stripesRight[i], "stripe");
  g.schedule(stripesRight[i], AT, s0->start(), i*2*stripetime/1000);

  if(i > 3){
    stripesLeft[i] = new FlashRequest(red2);
    stripesLeft[i]->flashDuration = (unsigned int)stripetime;
    g.addRequest(stripesLeft[i-4], "stripe");
    g.schedule(stripesLeft[i-4], AT, s0->start(), i*2*stripetime/1000);
  }
}
```

5.3.3 Exception handling

Finally, we turn to a couple of applications that demonstrate how the system handles exceptions. The first is metering for a single shot

The scheduling graph for this application is shown in Figure 5.16.

Provided the computation finishes quickly, this executes like any other graph. However, because the final capture depends on the results of the computation, this

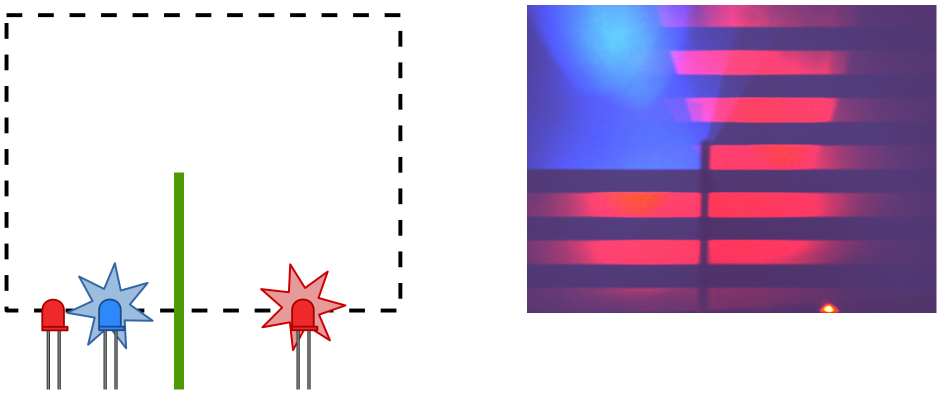
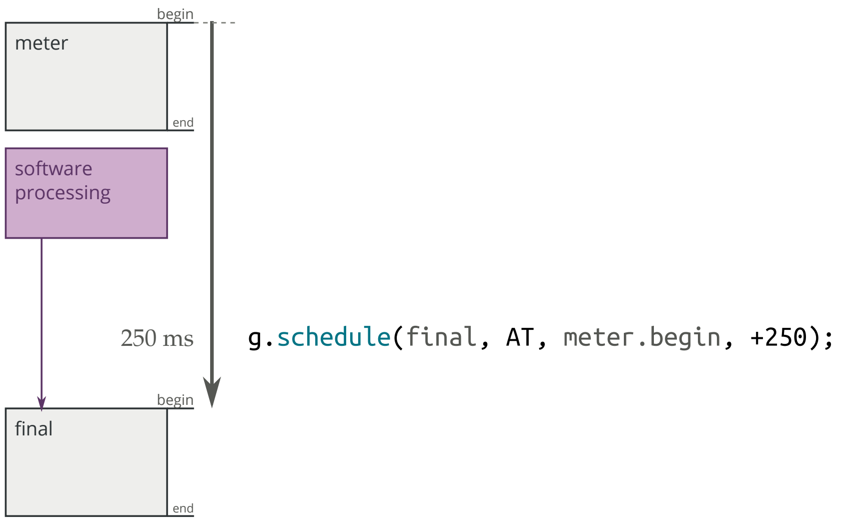


Figure 5.15: Setup to capture the American flag image, and the result. Three flashes (two red, one blue) make the stripes, while an opaque barrier prevents the flashes on each side from flooding the other half.



```
g.schedule(final, AT, meter.begin, +250);
```

Figure 5.16: Scheduling graph for single-shot metering. A single image is captured and processed in software to determine the parameters for another shot, captured 250 ms later.

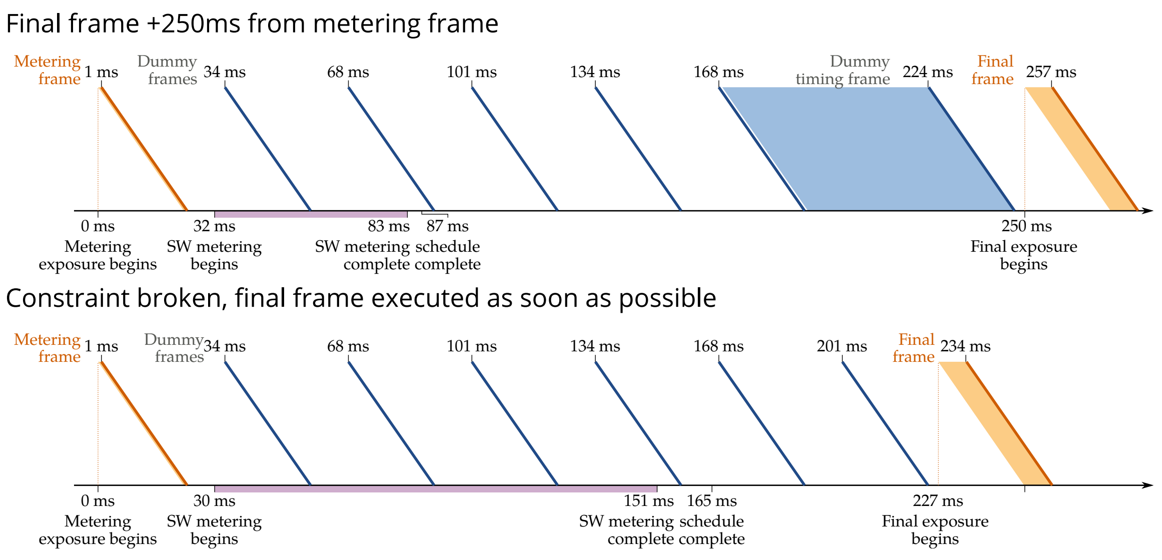


Figure 5.17: Actual execution timelines for two representative runs of the metering application. In the top timeline, the metering finishes in time for the shot to be scheduled as normal, and the shot executes exactly 250 ms after the metering shot. In the bottom timeline, the software metering was artifically prolonged to 120 ms. By the time the scheduler runs, there is insufficient time to insert a dummy timing frame to satisfy the AT constraint, so this constraint is broken and the frame is simply executed as soon as possible. Depending on the exact timing, this may result in the frame being captured up to one frame-time early, as shown in this example.

creates an implicit AFTER constraint to a dynamic event. Only the first shot is scheduled initially; the second shot must wait for the computation results. As described in Section 3.3.2, if the computation does not finish in time the AT constraint will be broken. Figure 5.17 shows two sample timelines from executing this graph on the prototype system; one where the processing finishes in time, and one where it does not.

Unlike the metering example, auto-white-balance (AWB) runs continuously on a viewfinder or video stream. For some cameras and some applications AWB can be applied after the image is analyzed, but if the processing pipeline operates in a streaming fashion, then most of the image has already been fully processed by the time the complete image statistics are known. In these cases, an effective solution is to use the statistics from the previous frame to derive the white balance parameters

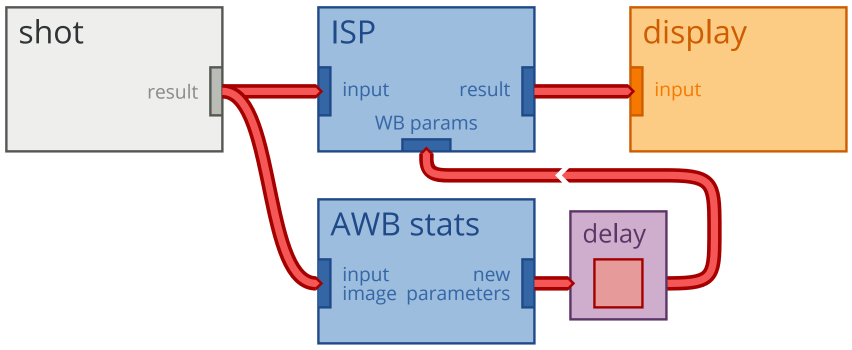


Figure 5.18: Processing graph for the continuous AWB application.

for the current frame.

Figure 5.18 shows the processing graph for such a streaming AWB application. The image sensor produces a single stream of frames, which are processed to produce final white-balanced image as well new parameters for the next iteration. A feedback path with a fixed latency of one frame pipes the calculated results back as the input parameters for the next image.

Like the flash metering example, the analysis and white-balancing must complete within a brief time window. However, here is the issue is not that capture constraints will immediately fail, but instead that frames will back up in the pipeline until the system runs out of resources. We can simulate this case simply by introducing an artificial delay into the processing code such that each frame takes 66 ms (15 Hz to process while the camera produces frames at twice that rate (33 ms / 30 Hz).

Within one second, the pipeline is 15 frames behind and has consumed all of the free buffers allocated by the framework. Because the framework requires that resources be allocated before committing a graph for execution, this depletion causes the pipeline to stall and the repeat timing constraint to be broken. The user code is notified of this condition via a callback.

Chapter 6

Conclusion

The previous chapters described the F4graph API and presented the performance of our prototype platform. A key theme has been that latency-critical tasks are best delegated to dedicated hardware where they can run without causing conflict and interference with the main processing tasks. Latency-critical tasks are generally very simple, so the required hardware is very modest. However, including these additional hardware resources increases the software complexity of the overall system. This chapter explores this trade-off, and provides suggestions for future researchers working in this area.

6.1 Delegating tasks for high performance

The results presented in the previous chapter validate our basic claim that precise timing requires a dedicated processor core. Most of the system is not especially latency-sensitive — at least no more so than any other application with a live user interface and real-time graphics — and works well with a multitasking OS running on a multicore processor with a GPU. However, the image timestamping and peripheral control require precision that cannot reliably be obtained with the stock Linux kernel. As demonstrated in a number of the timing benchmarks, an application processor running Linux exhibits response times with a "long tail." That is, the system responds very quickly on average, but it does not make any guarantees and a small number of events have unacceptably long responses.

While it is possible to tweak the system for better timing precision (the PREEMPT_RT real-time patch for the Linux kernel has existed for a decade and is close to being merged into mainline [72]), performance issues on a multithreading OS remain extremely complex and intertwine many elements of the system. Moreover, the Linux scheduler has been designed and tuned to give good performance for general-purpose computing, and tweaks to the scheduler will likely be at the cost of overall performance.

Instead, a more effective solution is to isolate the timing-critical pieces of the system and execute them on their own dedicated processor. This dedicated processor need not be powerful to be effective: the timing-critical tasks are very lightweight, and therefore can and should run on a lightweight core.

We can take the isolation of timing-critical tasks one step further by building custom hardware. For example, our custom CSI receivers include features which reduce the timing burden on the CPU and increase its effective precision. Because the receiver generates an interrupt at the start of each frame, the processor can determine the frame timestamp without repeated polling. Likewise, because the configuration registers are saved immediately but do not take effect until the next frame begins, the processor can simply "set and forget" the configuration once per frame.

As with dedicated processor cores, the custom hardware need not be complex to be effective. A trivial module that triggers an interrupt may be sufficient to release a CPU from spending all of its time polling a register, or a module which triggers an action on a particular clock cycle may relax the timing constraints on the processor by an order of magnitude or more. For reference, our CSI receivers are implemented with only 370 LUTs each, which is a minuscule 0.5% of the overall fabric.

If there were even stricter timing constraints to be met, or dozens more devices to control, the natural solution would be to build small hardware units to handle individual tasks with the needed precision. For example, flash control units could be designed with registers for the start and end time, and the unit could watch the clock and fire the flash at the programmed interval. The CPU could configure these units ahead of time on a less stringent schedule, and let them complete their tasks independently. Likewise, the Xilinx I2C controller could be replaced by a custom design that can handle an entire transaction without intervention.[1]

6.2 Managing complexity

Delegating timing-critical tasks to dedicated processor cores and custom hardware is essential for performance, but the expanded capability and additional hardware both introduce complexity. This raises two issues. First, how should an application developer ("the user") access the capabilities of the system without needing to understand the implementation? Second, adding another heterogeneous processor means that there is another code base, another build system, and another thread to debug. This becomes even more complex when custom hardware is added because its thread of control cannot be accessed by a traditional software debugger. We examine each of these in turn.

6.2.1 Harnessing capability

The F4graph API is designed to expose the rich capture capability of host camera platform without requiring detailed knowledge of the hardware. It does this by allowing a user to specify a set of requests on the hardware and timing relationships between them, and the system assumes responsibility for executing the requests such that the desired timing is preserved. Since certain timing relationships may be invalid or impossible to achieve, every collection of requests and timing relationships is analyzed to find a feasible schedule before being accepted for execution. If a schedule is found, then the system guarantees that the requests will execute and all statically analyzable relations will be maintained. Constraints relative to non-deterministic events (whether an external signal or the completion of a task on the multi-tasking CPU) are kept on a best-effort basis.

[1]I2C is particularly slow, since the bus itself runs at a mere 400 kHz and a simple 8-bit read transaction can take 100 µs or more. Our prototype system uses an interrupt-based interface to avoid blocking for this entire time, but the I2C driver configuration is still the slowest task the R5 is responsible for.

This constraint model allows the user to specify only the timing relationships that matter, leaving flexibility for the schedule to be fit on various platforms or runtime contexts. Further, it allows the system to make meaningful guarantees while providing a fallback to "best-effort" in cases where a strict guarantee is infeasible. With our prototype system and the demonstration programs described in the previous chapter, we are able to schedule and execute Requests with a temporal accuracy matching individual scanlines and to timestamp events even more precisely.

6.2.2 Build and debug tools

The build system for our demonstration platform manages complexity by automating most of the build process based on a straightforward system-wide configuration. As described in in Section 4.4, the developer specifies the design using a simple configuration language and includes references to hardware modules created with Halide. The rest of the build is automatically built and packaged based on this configuration, including the FPGA bitstream, the Zynq hardware configuration, the bootloaders, device tree, and Linux kernel.

As mentioned in Chapter 4, the Xilinx DMA engines were one of the most troublesome parts of the system. The exact conditions necessary to complete a transaction and generate an interrupt were not well-defined and misconfigurations could lock up the hardware, necessitating a driver reset or complete reboot. If we had the project to do over, we would design a DMA engine from scratch, optimized for streaming images with minimal CPU intervention and built for debugging.

A redesigned DMA engine would have the ability to operate on a circular buffer of 2D images (i.e., defined by width, height, and stride), relaxing the level of synchronization required between the software and hardware. A dozen or so image buffers could be programmed directly into configuration registers, eliminating the need to fetch scatter-gather descriptors from memory and sidestepping the corresponding coherency issues. The engine would use a simple and well-defined completion and error model: a "complete" interrupt is raised when the last byte of a frame is received and the sender simultaneously asserts `TLAST`. If the sender asserts `TLAST` early, or the

DMA reaches the end of the buffer without receiving `TLAST`, an "error" interrupt is raised and the engine halts.[2]

For debugging purposes, we would include the ability to "single-step" the DMA operation, transferring exactly one beat of data to or from the stream with each register write. New debugging registers would mirror the present stream values, allowing them to be read directly. Finally, the ability to discard or inject exact amounts of data would make it straightforward to debug issues where a component is expecting more (or less) data than is actually produced, causing the pipeline to stall.

Together with appropriate software, these features would turn the DMA engine into a software-controlled AXI-stream logic analyzer, making it simple to investigate coherency issues or to check the inputs and outputs of processing blocks. If the hardware pipeline execution is considered a thread of control — it is launched, works independently, and then is "joined" back to a CPU thread — these debugging features would function much as a software debugger does, allowing breakpoints, single-stepping, and inspection or modification of values. A software library on the host would manage this hardware-based debugger, making it possible to automate hardware tests and easily debug drivers or other code that interacts with the hardware. The Xilinx FPGA integrated logic analyzer is ill-suited for these tasks, since it must be compiled into the design ahead of time, has a narrow capture window,[3] and cannot be controlled by software running on the device. In the same way that our custom CSI receiver implemented simple hardware features that made software easier, a customized DMA engine would simplify the software design and greatly accelerate debugging.

[2]The `TLAST` signal indicates that the sender has transmitted the final beat of data. Since the image size is known, the buffer can be made exactly the right size. Together, these semantics simply ensure that the received data matches what was expected. There is no need to handle cases where too much or too little data is received; these can be flagged as errors.

[3]An integrated logic analyzer can capture tens of thousands of samples, but this is insufficient when processing images with millions of pixels.

6.3 Directions for future work

While the F4graph API is general and capable of describing and controlling many potential processing graphs, the current prototype has several limitations. First, as described in Section 4.2.4, FPGA designs are currently limited to one of two basic design patterns with no option to chain multiple operations without going back to memory or to conditionally execute part of a pipeline.

A more flexible FPGA architecture would make it possible to pack multiple accelerators into one FPGA design, and to stream data between them without going back to main memory — or, with the flip of a configuration bit, to skip over an accelerator or stream the intermediate results to memory. To do this, the system would need more capable DMA engines and reconfigurable AXI-stream routers, and the design specification and build system would need to be extended to support these additional components. More sophisticated kernel drivers would also be needed to manage the run-time hardware configuration.

Going even further, the prototype system could make use of the partial reconfiguration facility of the FPGA to dynamically load accelerators as needed by different applications. Or the fixed-function pipelines could be replaced by other reconfigurable fabrics or programmable image processing hardware. The graph processing API itself is agnostic to the specifics of the hardware accelerator, and could work with any of these.

A second improvement which would streamline application development and improve performance is to integrate F4graph more deeply with Halide. The system currently treats Halide as a sort of black-box plugin, where monolithic chunks of Halide code (which may or may not make use of hardware accelerators) can be executed as graph nodes. However (as we discussed in Chapter 2) a string of optimized kernels does not necessarily produce an optimal implementation of a complete application, so for best performance developers must manually merge and optimize blocks of Halide code before including them in F4graph. If a future system were able to understand and operate directly on Halide code, it would be possible to automatically merge and optimize blocks of processing code, improving modularity and reuse and saving developers time.

Depending on the sophistication of the system, this could go beyond merging sequential blocks of Halide code and allow whole sections of the processing graph to be compiled and optimized together. Consider for example the auto-white-balance algorithm which takes statistics generated from one frame and uses them to compute color-correction parameters for the next. If this algorithm were coded as a Halide node along with the statistics and color-correction nodes, then an improved F4graph could compile and optimize this feedback loop as a complete unit and implement it on whatever piece of the hardware was most effective. In the best case, the calls to and from the graph framework could be elided entirely and the algorithm could run without intervention from the APU. While the overhead of the graph framework is small, eliminating it would allow the best possible performance for applications with tight feedback loops.

These remain active areas of research, and we look forward to the innovations and improvements developed in future systems.

Bibliography

[1] Gartner, Inc, "Gartner says worldwide sales of smartphones recorded first ever decline during the fourth quarter of 2017." https://www.gartner.com/newsroom/id/3859963, February 2018.

[2] Camera and Imaging Products Association, "Total shipments of digital still cameras." http://www.cipa.jp/stats/documents/common/cr300.pdf, Mar. 2018.

[3] Flickr, "Camera finder." https://www.flickr.com/cameras, Mar. 2018.

[4] Apple, Inc, "iPhone 8 - Technical Specifications." https://www.apple.com/iphone-8/specs/, March 2018.

[5] Nikon, "D500." https://www.nikonusa.com/en/nikon-products/product/dslr-cameras/1559/d500.html, March 2018.

[6] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian, "Image denoising with block-matching and 3D filtering," in *Image processing: algorithms and systems, neural networks, and machine learning*, vol. 6064, pp. 354–365, SPIE, 2006.

[7] S. W. Hasinoff, D. Sharlet, R. Geiss, A. Adams, J. T. Barron, F. Kainz, J. Chen, and M. Levoy, "Burst photography for high dynamic range and low-light imaging on mobile cameras," *ACM Transactions on Graphics*, vol. 35, pp. 192:1–192:12, Nov. 2016.

[8] J. T. Barron, A. Adams, Y. Shih, and C. Hernández, "Fast bilateral-space stereo for synthetic defocus," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4466–4474, 2015.

[9] N. Wadhwa, R. Garg, D. E. Jacobs, B. E. Feldman, N. Kanazawa, R. Carroll, Y. Movshovitz-Attias, J. T. Barron, Y. Pritch, and M. Levoy, "Synthetic depth-of-field with a single-camera mobile phone," *ACM Transactions on Graphics*, vol. 37, no. 4, pp. 1–13, 2018.

[10] Y. Xiong and K. Pulli, "Fast panorama stitching for high-quality panoramic images on mobile phones," *IEEE Transactions on Consumer Electronics*, vol. 56, no. 2, pp. 298–306, 2010.

[11] O. Liba, K. Murthy, Y.-T. Tsai, T. Brooks, T. Xue, N. Karnad, Q. He, J. T. Barron, D. Sharlet, R. Geiss, S. W. Hasinoff, Y. Pritch, and M. Levoy, "Handheld mobile photography in very low light," *ACM Transactions on Graphics*, vol. 38, Nov. 2019.

[12] Olympus, "OPC hack & make project." http://web.archive.org/web/20151209025122/https://opc.olympus-imaging.com/en_sdkdocs/, Dec. 2015.

[13] Canon, "Camera SDK overview." https://developers.canon-europe.com/s/camera, July 2020.

[14] Y. Yamada, T. Sano, Y. Tanabe, Y. Ishigaki, S. Hosoda, F. Hyuga, A. Moriya, R. Hada, A. Masuda, M. Uchiyama, M. Jobashi, T. Koizumi, T. Tamai, N. Sato, J. Tanabe, K. Kimura, Y. Ojima, R. Murakami, and T. Yoshikawa, "A 20.5 TOPS multicore SoC with DNN accelerator and image signal processor for automotive applications," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 1, pp. 120–132, 2020.

[15] O. Shacham, "Pixel Visual Core: image processing and machine learning on Pixel 2." https://www.blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2/, Oct. 2017.

[16] B. Barry, C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O'Riordan, and V. Toma, "Always-on vision processing unit for mobile applications," *IEEE Micro*, vol. 35, no. 2, pp. 56–66, 2015.

[17] G. C. Holst, *CMOS/CCD sensors and camera systems.* SPIE monograph; PM208, second edition. ed., 2011.

[18] Sony Semiconductor Solutions, "IMX-219PQ." `https://www.sony-semicon.co.jp/products_en/new_pro/april_2014/imx219_e.html`, 2018.

[19] ON Semiconductor, "AR1011HS 1-Inch 10MP CMOS Digital Image Sensor," 2012.

[20] A. Adams, E.-V. Talvala, S. H. Park, D. E. Jacobs, B. Ajdin, N. Gelfand, J. Dolson, D. Vaquero, J. Baek, M. Tico, H. P. A. Lensch, W. Matusik, K. Pulli, M. Horowitz, and M. Levoy, "The frankencamera: An experimental platform for computational photography," *ACM Transactions on Graphics*, vol. 29, pp. 29:1–29:12, July 2010.

[21] E.-V. Talvala, *The Frankencamera : building a programmable camera for computational photography.* PhD thesis, Stanford University, 2011.

[22] FCam authors, "FCam API documentation." `http://fcam.garage.maemo.org/apiDocs.html`, 2011.

[23] A. Troccoli, D. Pajak, and K. Pulli, "Fcam for multiple cameras," in *Multimedia on Mobile Devices 2012; and Multimedia Content Access: Algorithms and Systems VI*, vol. 8304, p. 830404, International Society for Optics and Photonics, 2012.

[24] Google, "Camera2 API." `https://developer.android.com/reference/android/hardware/camera2/package-summary`, Mar. 2018.

[25] Google, "CaptureRequest — Android Developers." `https://developer.android.com/reference/android/hardware/camera2/CaptureRequest#FLASH_MODE`, Mar. 2018.

[26] Google, "Android 9 features and APIs." https://developer.android.com/about/versions/pie/android-9.0#camera, Mar. 2018.

[27] O. Wahltinez, "Getting The Most From The New Multi-Camera API." https://medium.com/androiddevelopers/getting-the-most-from-the-new-multi-camera-api-5155fb3d77d9, Nov. 2018.

[28] Microsoft, "DirectShow." https://docs.microsoft.com/en-us/windows/win32/directshow/directshow.

[29] Apple, "AVFoundation Framework." https://developer.apple.com/documentation/avfoundation.

[30] Google, "Media — Android Open Source Project." https://source.android.com/devices/media.

[31] GStreamer authors, "GStreamer open source multimedia framework." https://gstreamer.freedesktop.org/.

[32] Khronos Group, "OpenMAX." https://www.khronos.org/openmax, 2011.

[33] A. François, "A hybrid architectural style for distributed parallel processing of generic data streams," in *Proceedings. 26th International Conference on Software Engineering*, pp. 367–376, May 2004.

[34] A. François, "An architectural framework for the design, analysis and implementation of interactive systems," *The Computer Journal*, vol. 54, pp. 1188–1204, July 2011.

[35] Khronos Group, "OpenVX." https://www.khronos.org/openvx, 2019.

[36] Khronos Group, "Conformant products." https://www.khronos.org/conformance/adopters/conformant-products/openvx, 2018.

[37] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, “Making OpenVX really ‘real time’,” in *2018 IEEE Real-Time Systems Symposium (RTSS)*, pp. 80–93, IEEE, 2018.

[38] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, “Adrenaline: An OpenVX environment to optimize embedded vision applications on many-core accelerators,” in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pp. 289–296, IEEE, 2015.

[39] H. Omidian and G. Lemieux, “Janus: A compilation system for balancing parallelism and performance in OpenVX,” in *Journal of Physics: Conference Series*, vol. 1004, p. 012011, 2018.

[40] S. Taheri, P. Behnam, E. Bozorgzadeh, A. Veidenbaum, and A. Nicolau, “Affix: Automatic acceleration framework for FPGA implementation of OpenVX vision algorithms,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 252–261, 2019.

[41] M. A. Özkan, B. Ok, B. Qiao, J. Teich, and F. Hannig, “HipaccVX: Wedding of OpenVX and DSL-based code generation,” 2020.

[42] Khronos Group, “OpenKCam.” `https://www.khronos.org/openkcam`, 2013.

[43] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, “Decoupling algorithms from schedules for easy optimization of image processing pipelines,” *ACM Transactions on Graphics*, vol. 31, pp. 32:1–32:12, July 2012.

[44] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’13, (New York, NY, USA), pp. 519–530, ACM, 2013.

[45] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, “Automatically scheduling halide image processing pipelines,” *ACM Transactions on Graphics*, vol. 35, pp. 83:1–83:11, July 2016.

[46] Halide contributors, "Halide." https://github.com/halide/Halide.

[47] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing DSL," *ACM Transactions on Architecture and Code Optimization*, vol. 14, pp. 26:1–26:25, Aug. 2017.

[48] J. Pu, *Programming heterogeneous systems from an image processing domain specific language*. PhD thesis, Stanford University, 2017.

[49] TDK Invensense, "ICM-20948." https://www.invensense.com/products/motion-tracking/9-axis/icm-20948/, 2018.

[50] RISC-V International, "Available Hardware." https://riscv.org/risc-v-cores/.

[51] OpenCores Project, "Projects::Processors." https://opencores.org/projects?expanded=Processor.

[52] SiFive, "E20 processor." https://www.sifive.com/cores/e20, 2019.

[53] T. Insights, "Samsung Galaxy S9 Teardown." https://www.techinsights.com/blog/samsung-galaxy-s9-teardown, 2018.

[54] Xilinx, "Zynq UltraScale+ MPSoC." https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[55] Avnet, "UltraZed-EG." http://zedboard.org/product/ultrazed-EG, 2019.

[56] R. P. Foundation, "Camera Module V2." https://www.raspberrypi.org/products/camera-module-v2/.

[57] DrYerzinia, "Raspberry Pi Camera v2.1 Reversed." https://hackaday.io/project/19480-raspberry-pi-camera-v21-reversed.

[58] Broadcom, "Sony IMX219 sensor driver." https://android.googlesource.com/kernel/bcm/+/android-bcm-tetra-3.10-lollipop-wear-release/drivers/media/video/imx219.c, 2013.

[59] S. Munaut, "Raspberry Pi interface prototype." https://github.com/parallella/parallella-examples/tree/master/rpi-camera.

[60] Kimovil, "Smartphones with IMX219 Exmor RS lens model." https://www.kimovil.com/en/list-smartphones-by-lens-model/sony-imx219-exmor-rs.

[61] Digilent, "FMC Pcam Adapter." https://store.digilentinc.com/fmc-pcam-adapter/.

[62] Xilinx, "MIPI D-PHY v4.0 LogiCORE IP Product Guide." https://www.xilinx.com/support/documentation/ip_documentation/mipi_dphy/v4_0/pg202-mipi-dphy.pdf, 2017.

[63] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 369–383, Association for Computing Machinery, 2020.

[64] Xilinx, "libmetal." https://github.com/Xilinx/libmetal, 2018.

[65] S. Ansari, N. Wadhwa, R. Garg, and J. Chen, "Wireless software synchronization of multiple distributed cameras," in *2019 IEEE International Conference on Computational Photography (ICCP)*, pp. 1–9, IEEE, 2019.

[66] B. Dirks, M. H. Schimek, H. Verkuil, M. Rubli, A. Walls, M. Karicheri, M. C. Chehab, P. Osciak, S. Ailus, A. Palosaari, and T. Figa, "Video For Linux API." https://www.linuxtv.org/downloads/v4l-dvb-apis-new/userspace-api/v4l/v4l2.html.

[67] L. Gurobi Optimization, "Gurobi optimizer reference manual." "http://www.gurobi.com", 2018.

[68] A. Gleixner, L. Eifler, T. Gally, G. Gamrath, P. Gemander, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, F. Serrano, Y. Shinano, J. M. Viernickel, S. Vigerske, D. Weninger, J. T. Witt, and J. Witzig, "The SCIP Optimization Suite 5.0," technical report, Optimization Online, December 2017.

[69] X. Olive, "Facile constraint programming library." https://facile.readthedocs.io/en/latest/index.html, 2016.

[70] L. Perron and V. Furnon, "Or-tools." https://developers.google.com/optimization/, 2018.

[71] J. Corbet, G. Kroah-Hartman, and A. Rubini, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[72] The Linux Foundation, "Real time linux." https://wiki.linuxfoundation.org/realtime/start, 2021.

[73] Gartner, Inc, "Newsroom (collected press releases)." http://www.gartner.com/newsroom, 2008-2018.

[74] J. S. Brunhaver, *Design and optimization of a stencil engine*. PhD thesis, Stanford University, 2015.

[75] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level image processing code into hardware pipelines.," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 144–1, 2014.

[76] Xilinx, "Zynq Ultrascale+ Devices Register Reference (UG1087 v1.7)." https://www.xilinx.com/html_docs/registers/ug1087/ug1087-zynq-ultrascale-registers.html.

[77] T. Mertens, J. Kautz, and F. Van Reeth, "Exposure fusion," in *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, pp. 382–390, IEEE, 2007.

[78] Avnet, “UltraZed EG PCIe Carrier Card.” `http://microzed.org/product/ultrazed-eg-pcie-carrier-card`.

[79] C. S. Johnson, *Science for the Curious Photographer: An Introduction to the Science of Photography*. Milton: Routledge, 2 ed., 2017.