Case Study: Building a High Quality Video Pipeline Using GStreamer and V4Linux on an i.MX6

Sean Hudson

Embedded Linux Architect & Member of Technical Staff

mentor.com/embedded

Android is a trademark of Google Inc. Use of this trademark is subject to Google Permissions.
Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Who am I?

- Embedded Linux Architect at Mentor Embedded, a division of Mentor Graphics
- Member of the OpenEmbedded Project's board
- Former representative to the Advisory Board for the Yocto Project

What’s this presentation for?

- Intended audience
  - Aimed at those considering a Linux based video project, specifically on an i.MX6 board
- What will be covered?
  - Key technologies used to build the product
  - What we learned (good, bad, & ugly)
- Primary goal of this presentation
  - Help folks get a running start on a similar project

Outline

- Project Background
- Hardware Components
- Software Components
- Final Thoughts

PROJECT BACKGROUND

Project Background

- Services engagement that ran from the end of 2012 – the end of 2013
- Replacing an older, FPGA based design
- A customized, portable design based on the i.MX6
- Device processes and displays video from two independent sensors
- Device output to a built-in OLED display or a connected HDMI monitor
- Displays sensor input either singly or combined
- Sensor 1 input was 1280x1024 at up to 60 FPS
- Sensor 2 input was 640x480 at up to 30 FPS
- Desired latency for all modes was < 100ms at 30 FPS
- Intended as a reusable platform for future products

Starting, Known Project Challenges

- Additional algorithmic processing of images would be required, so CPU utilization should be kept as low as possible
- Customer wanted the complete software stack developed in < 12 months
- Hardware was in development and wasn't scheduled to be available for ~3 months
- Several components were new, including the sensors
- Small team, ~3 software engineers

HARDWARE COMPONENTS

i.MX6 - SOC

- Quad-core Cortex A9
- ARM NEON SIMD with each core
- 3D Graphics Processing Unit (GPU)
- 2x Image Processing Units (IPU)
- Additional device and bus support
- Had existing software support

i.MX6 Hardware Availability Strategy

- In order to account for the initial, scheduled delay of the hardware, we opted to use Sabre Lite boards to begin working on the software stack.
- This proved to be quite valuable due to additional delays in the actual hardware being available.

- **Take Away**: If you are building a custom, i.MX6 design for your video processing product, getting a reference board, e.g. the Sabre-Lite, can allow some software development to continue when the hardware is invariably late.

i.MX6 – IPU

- Expectation:
  - IPU hardware accelerated conversion of sensor input and transfer of data frames
- Outcome:
  - The sensor selections prevented the hardware from being used directly for some important frame conversion operations, specifically, the input frame format was not understood by the IPUs and so a raw data mode was used to transfer data from the bus into the video pipeline, however, frame conversion was required in software that wasn't expected.
  - Worked great for some things like re-sizing frames, but were in limited supply and had to be carefully allocated to ensure that contention over the IPU wouldn't occur

i.MX6 – NEON

- Expectation:
  - NEON instructions would run in parallel with ARM core instructions (offload)
  - NEON would provide significant computational resources that would help fill the requirement for additional algorithms
- Outcome:
  - The ARM core is tightly couple to NEON and can not be utilized separately
  - This reduced the expected parallel computing capabilities of the platform and increased contention for resources
  - We found that the NEON SIMD were well suited to the computations required, however, we ran into issues with contention due to the number of operations required of them
- **Takeaway**
  - **Managing the specific hardware resources assigned to a task becomes critically important to meeting performance targets. (See later slide on how we enforced this)**

Sensors

- Expectation:
  - The sensors were connected via well defined interfaces and would be able to utilize the IPU to transfer the frame, convert the format for internal use, and resize, as necessary.
- Outcome:
  - Both sensors output frames in a format that the IPU could not handle directly.
  - "Raw Mode" transfers enabled the DMA transfer of the frames
  - Additional work was required to convert these frames into the proper format for use in the GStreamer pipeline
  - IPU worked well to resize frames as well, but only after conversion
- **Takeaway**
  - **The frame conversions dominated the work on this project. They also consumed a significant amount of the latency budget.**

i.MX6 Platform – GPU

- Expectation:
  - GPU would provide a raw computation resource
  - openGL support, which was available, would provide efficient access to the GPU processing capabilities
- Outcome:
  - When the IPU frame conversion was not possible, the GPU was selected to perform that operation
  - Sensor 1 output was in a Bayer BGGR format that required a "demosaic" operation to get to a RGB format
  - Initial algorithm selected had a reference GLSL shader implementation for the GPU
  - Unfortunately, the data transfer rates into the GPU were not enough to sustain the target frame rate. We discovered that the frame rate dropped linearly with the size of the data frame transferred
  - With several weeks lost to the effort, we made the decision to move the processing to the NEON processor

SOFTWARE COMPONENTS

Mentor Embedded Linux (MEL)

- MEL is based directly on The Yocto Project
- Provides a reference rootfs image and kernel
- Contains the Freescale BSP bits
- Integrates with additional MGC tools, which became important later
- Seriously, I work for Mentor, is it any surprise that's what we used?

Freescale i.MX6 BSP

- Freescale publishes their BSP via a public, Yocto Project layer
- The BSP contains V4L drivers for their IPU sensor interfaces that include DMA transfer support
- The BSP version used contained kernel 3.0.35
- It also contained GStreamer plugins compatible with GStreamer base 0.10.36.
- One of these plugins, mfw_v4lsrc, provided sensor frames to the GStreamer pipeline

i.MX6 Platform – MEL w/FSL BSP

- Expectation:
  - MEL would provide solid base to begin work and also as basis for future platform
- Outcome:
  - MEL base worked as expected
  - Allowed an update in the middle of the project of the FSL BSP release with minimal effort and impact
  - Allowed work to continue on Sabre-Lite when hardware was delayed
  - Allowed work to begin and complete on the BSP for the new project without impacting work that was underway for the rest of the stack
  - MEL/FSL platform on i.MX6 is now being considered for additional products at that customer

Video4Linux (V4L)

- API and driver framework that is part of the kernel and provides support to video devices
- Provides standard way for video devices to communicate to userspace
- Stable API that has been around for a while

- For more information on V4L:
  - http://linuxtv.org/downloads/v4l-dvb-apis/index.html
  - http://www.linuxtv.org/wiki/index.php/Developer_Section

Video4Linux (V4L)

- The FSL BSP provided V4L drivers that connected directly to the standard sensor inputs of the i.MX6.
- These drivers largely worked as expected and provided the frames, in raw mode, from the sensors with almost no effort on our part.
  - One significant bug was found in a local timer/scheduler when Gstreamer threads were created/destroyed rapidly under system stress
- Hooking these up to the pipeline was accomplished using a plugin provided by the FSL BSP that wrapped the V4L source with some FSL specifics, called mfw_v4lsrc.

GStreamer

- From Gstreamer.freedesktop.org:
  - “GStreamer is a library for constructing graphs of media-handling components.”
- It has a well defined API for the plugins
- It allows for components to be re-ordered, inserted, and dropped, dynamically
- GStreamer was chosen due to the flexibility of the architecture and immediate availability in MEL
- In the end, the GStreamer plugin work took most of the available schedule
- **Takeaway**
  - **Make sure you know what the quality level is of any open plugins that you plan to use. Also, plan to have to re-write or modify some regardless of their quality.**

GStreamer – Good, Bad, & Ugly

- GStreamer plugins are collected into three broad categories:
  - Good - plug-ins that have good quality code, correct functionality, and preferred licensing
  - Bad – plug-ins that aren't up to par compared to the rest
  - Ugly - plug-ins that have good quality and correct functionality, but distributing them might pose problems
- During development a “good” plugin, videomixer, was substantially reworked to enhance stability and performance for merging video streams together
- **Takeaway**
  - **Make sure you know what the quality level is of any open plugins that you plan to use. Also, plan to have to re-write or modify some regardless of their quality.**

GStreamer – Thread control

- Gstreamer, as of the version we used, did not give direct control to the threading
- Threads were created internally to the base framework
- Threads are created/destroyed quickly depending on your system and creates a fair amount of overhead by itself
- We discovered that by inserting a “queue” element into the pipeline, we could force GStreamer to create a new thread

- **Takeaway**
  - **Make sure you know what the quality level is of any open plugins that you plan to use. Also, plan to have to re-write or modify some regardless of their quality.**

GStreamer – Resource Contention

- We found that GStreamer threads would many times starve each other due to unnecessary resource contention
- Our hardware budgeting/allocation needed to have a way to force operations to occur on a specific core
- We modified the “queue” element to accept a CPU affinity parameter that allowed us to accomplish this task

GStreamer – DMA buffers

- The Freescale GStreamer plugins were *very* finicky about using DMA buffers
- We discovered that certain plugin combinations would not use a DMA-able buffer and the final transfer to the display would require an additional copy of the frame
- This *killed* performance quickly
- To overcome this issue, we wrote a new element that “fooled” the elements into using the correct, DMA buffers

GStreamer – gst-launch

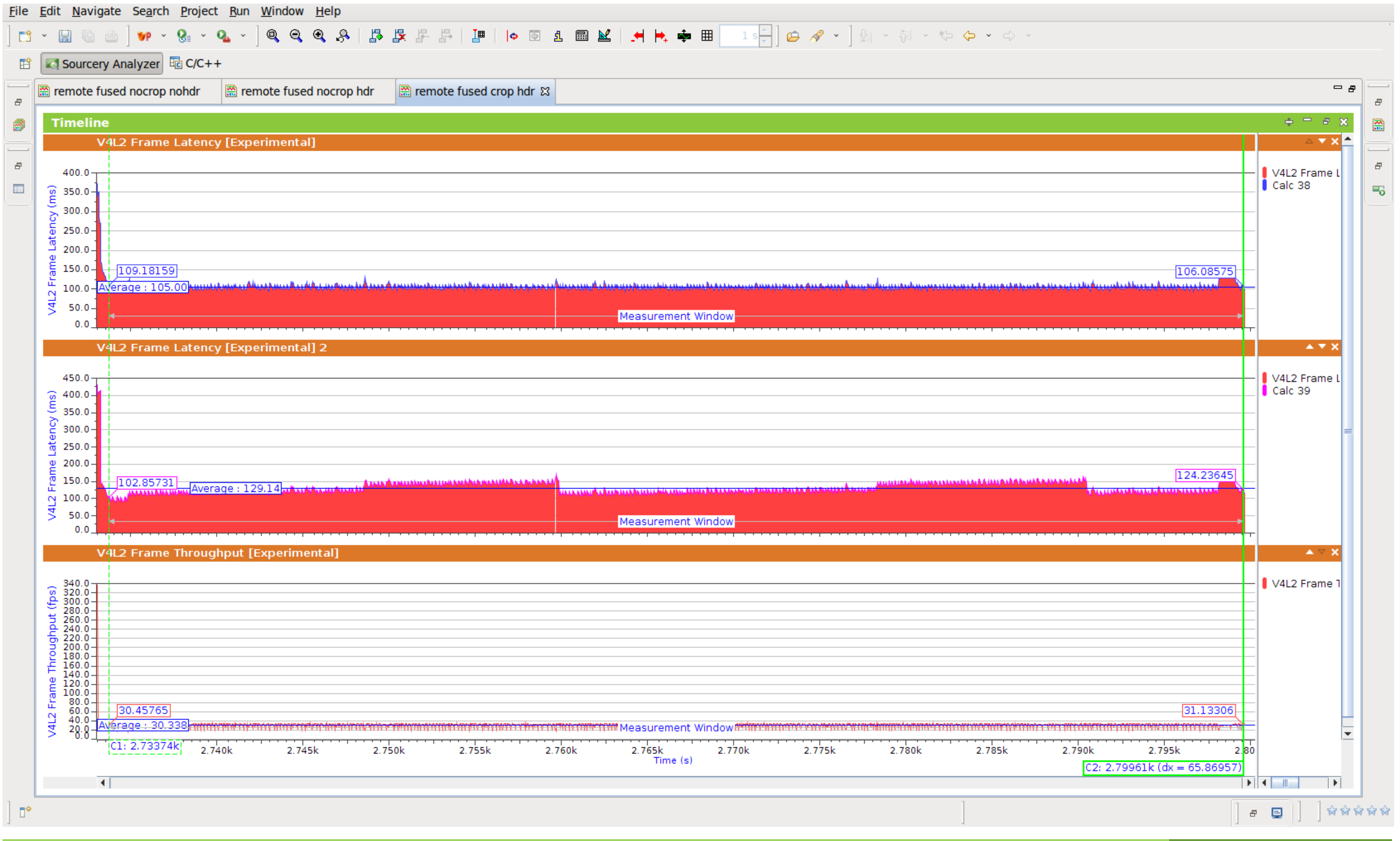
- GStreamer provides a command line tool to launch a pipeline
- While useful for testing, it quickly becomes cumbersome
- Learn how to use it, but be ready to figure out how to create the same thing in code

```
usr/bin/gst-launch --gst-debug-no-color videomixer2 name=mixer background=4 sink_1::alpha=0.5 sink_0::alpha=1 \
! queue name=output cpu=4max-size-buffers=1min-threshold-buffers=1\
! mfw_v4lsink sync=falseqos=false\
mfw_v4lsrc name=TI fps-n=30fps-d=1device=/dev/video0 pixel-format=3cpu=4\ ! 'video/x-raw-
gray,bpp=8,width=640,height=480,framerate=30/1'\
! gray2rgb crop-left=2crop-right=6\
! 'video/x-raw-rgb,bpp=24,depth=24,framerate=30/1'\
! mgc_hwbpool pool-size=8\
! mfw_ipucsc crop-top=8crop-bottom=8\
! 'video/x-raw-rgb,bpp=24,depth=24,width=800,height=600'! mgc_hwbpool pool-size=8\ ! mixer. \
mfw_v4lsrc pixel-format=2device=/dev/video1 fps-n=60fps-d=1cpu=2num-buffers=4000 \
!  'video/x-raw-bayer,format=bggr,width=1280,height=960'\
!  frameavg omit-avg=false\
!  queue name=bayer cpu=3max-size-buffers=1min-threshold-buffers=1\
!  bayerneon2 crop-left=2crop-right=6\
!  'video/x-raw-rgb,bpp=24,depth=24,framerate=30/1'\
!  queue name=qTV cpu=2max-size-buffers=1min-threshold-buffers=1\
!  mgc_hwbpool pool-size=8buffer-size=3686400\
!  mfw_ipucsc crop-top=8crop-bottom=8\
!  'video/x-raw-rgb,bpp=24,depth=24,width=800,height=600'! mgc_hwbpool pool-size=8\ ! mixer.
```

GStreamer – Performance analysis

- Gstreamer can report some statistics, but not all plugins handle that data correctly and not all plugins report that data correctly
- In order to analyze the performance of the pipeline, we instrumented the GStreamer pipeline and put it through a visualization tool to help us identify issues

GStreamer – Performance analysis



mentor embedded

FINAL THOUGHTS

Final Thoughts

- Sensor frame input format took a lot of unexpected work to address, pay close attention to the formats needed to display versus the format the sensors produce
- DMA support is critical for the pipeline to work efficiently (zero-copy). Expect to spend time making elements of a GStreamer video chain handle DMA buffers correctly.
- The FSL BSP has revved since this work, it now contains a 3.10.17 kernel and a GStreamer 1.0 version.
  - Reportedly many improvements in later version of GStreamer around buffer handling and thread control
- Even with the moderately conservative plan I put in place early in the process, we had to push very hard to hit the final milestone on time.

QUESTIONS?