Quick Boot Optimization for Exterior View System in Android

Bo Tong, Open Source Technology Center, Software and Services Group (OTC/SSG)

NOTICE & DISCLAIMER

- Intel technologies’ features and benefits depend on system configuration

and may require enabled hardware, software or service activation.

- Performance varies depending on system configuration.
- Intel, the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.
- *Other names and brands may be claimed as the property of others.

Agenda

- Introduction of Exterior View System (EVS)
  - Why EVS is introduced
  - EVS stack in Android
- EVS Boot Latency Analysis
  - Boot sequence diagram
  - Boot time evaluation
- Quick Boot Optimization
  - Improved by 3 steps
  - Conclusion and suggestion

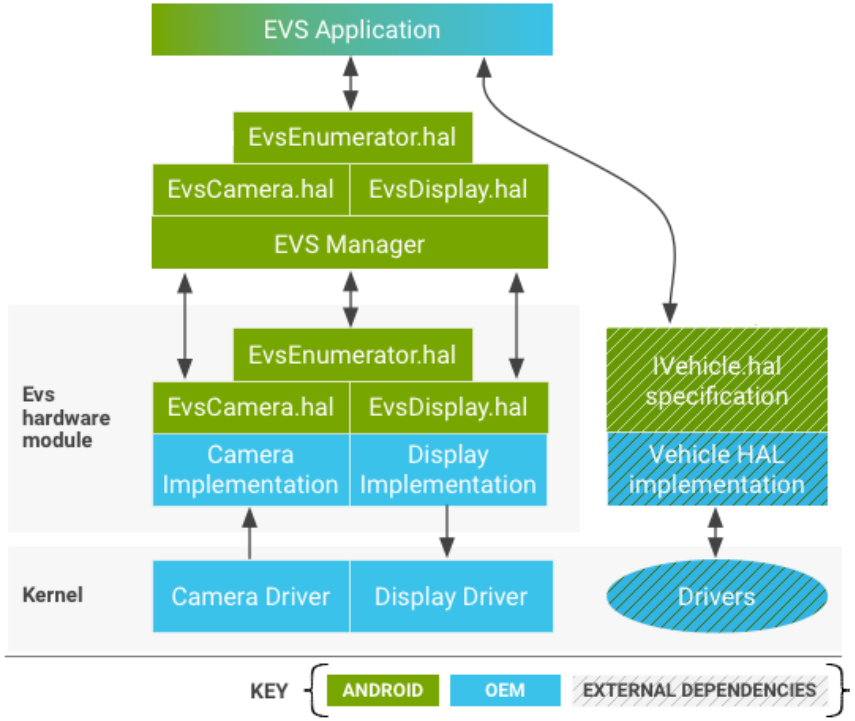
Introduction of Exterior View System (EVS)

Why EVS is introduced

- **SIMPLE**: Support camera and view display with simplified design.
- **EARLY**: Intend to show display very early in the Android boot process.
- **EXTENSIBLE**: Enables advanced features to be implemented in user apps.

EVS stack in Android

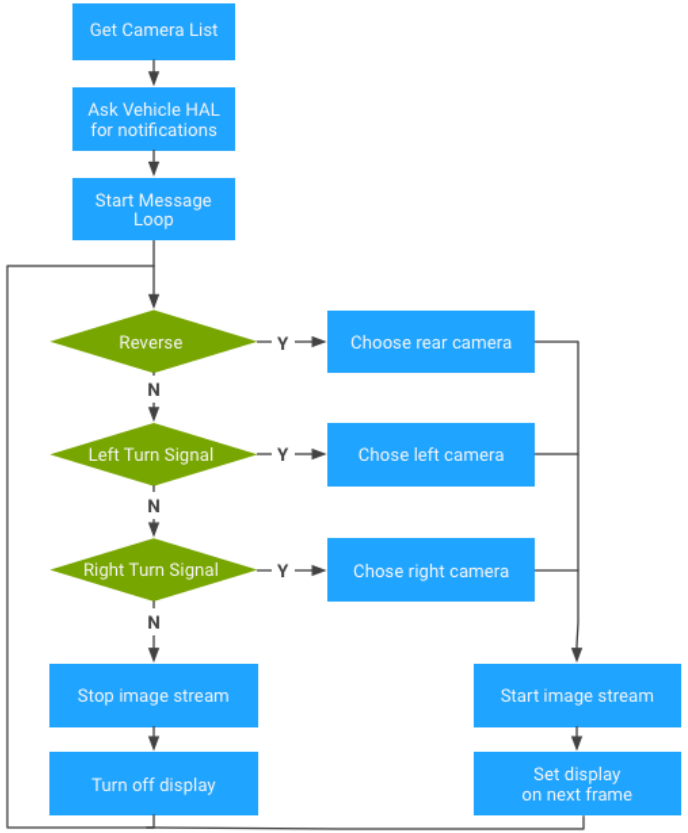
- EVS application
  - Native code started by init.rc
  - Run in background when not in use
- EVS Manager
  - Wrapper between App and HAL
  - Accept multiple concurrent clients
- EVS HAL
  - Depends on SurfaceFlinger
  - H/W independent but not

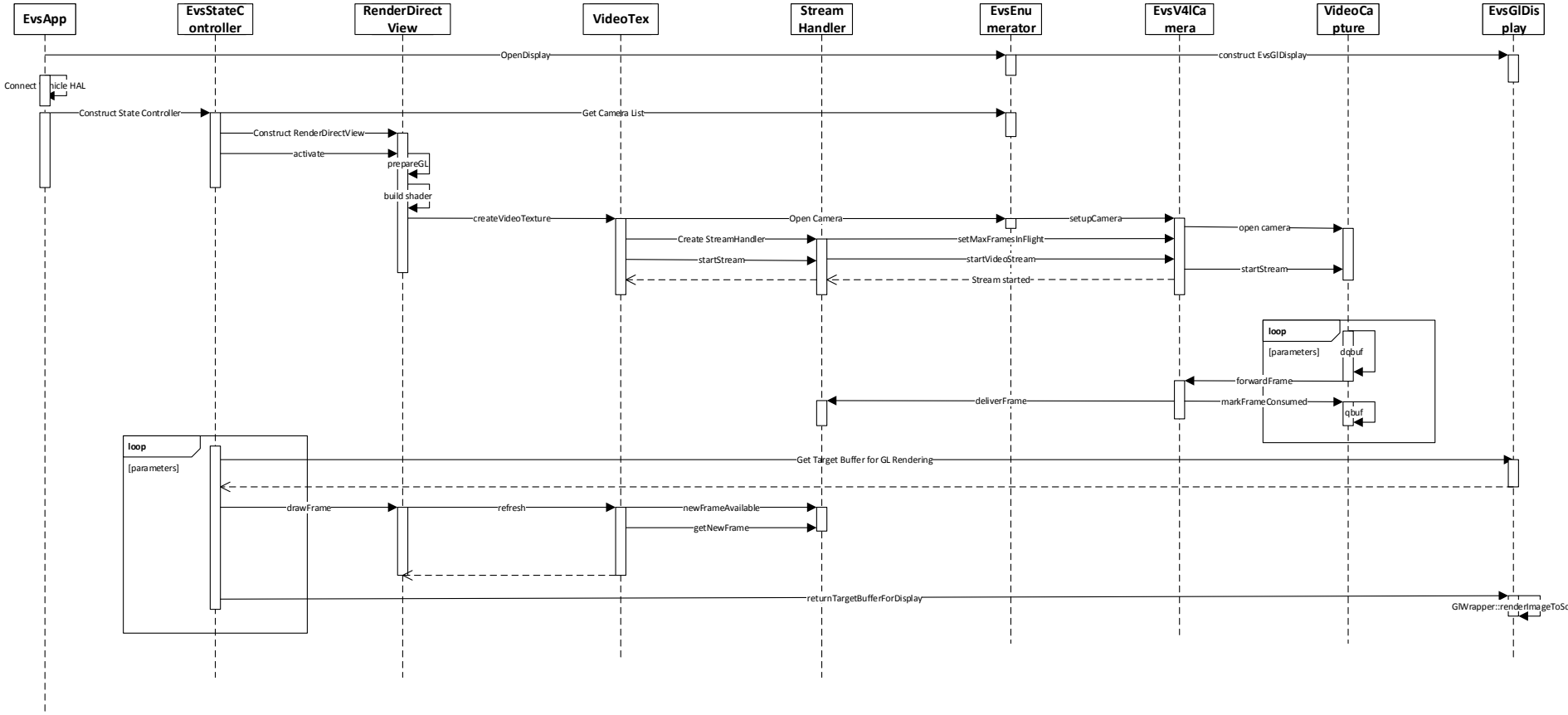


EVS Boot Latency Analysis

Boot sequence diagram

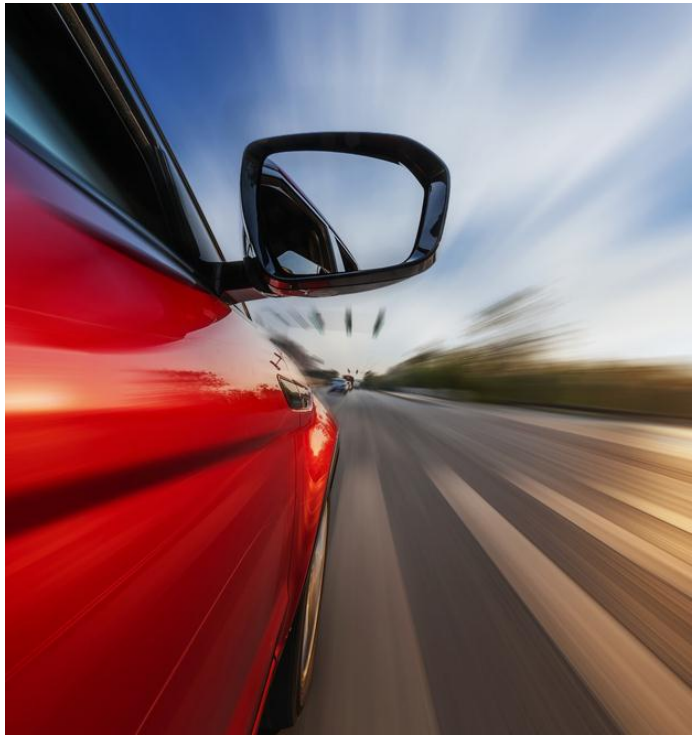
- Communicate with the EVS Manager and the Vehicle HAL
- An infinite loop monitoring camera and gear/turn signal state and reacting respectively
- Use the source image as an OpenGL texture and render a complex scene to the output buffer




EvsApp
EvsStateController
RenderDirectView
VideoTex
StreamHandler
EvsEnumerator
EvsV4lCamera
VideoCapture
EvsGlDisplay
OpenDisplay
construct EvsGlDisplay
Connect hicle HAL
Construct State Controller
Get Camera List
Construct RenderDirectView
activate
prepareGL
build shader
createVideoTexture
Open Camera
setupCamera
open camera
Create StreamHandler
setMaxFramesInFlight
startStream
startVideoStream
startStream
Stream started
loop
[parameters]
dqbuf
forwardFrame
deliverFrame
markFrameConsumed
qbuf
loop
[parameters]
Get Target Buffer for GL Rendering
drawFrame
refresh
newFrameAvailable
getNewFrame
returnTargetBufferForDisplay
GlWrapper::renderImageToScre

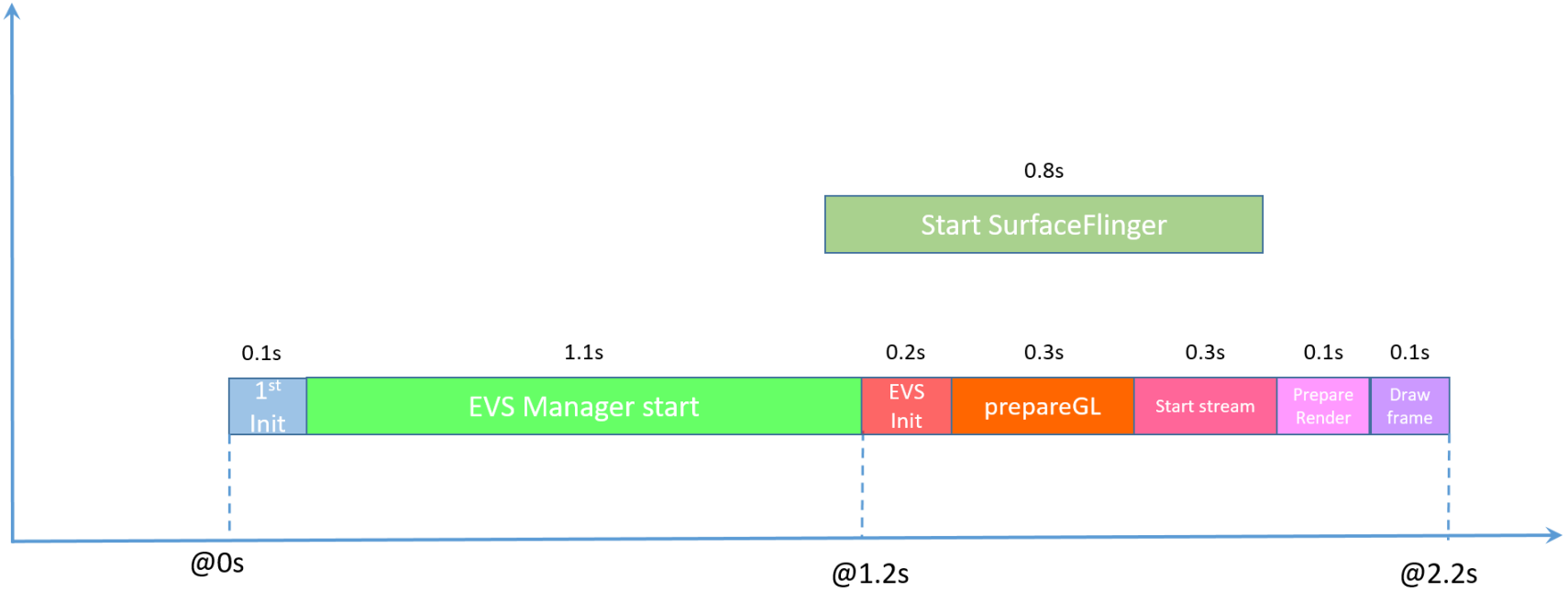
Boot time evaluation

- The application is expected to be started by init as soon as EVS manager and vehicle HAL are available, targeted within 2.0 seconds of power on.



Boot time evaluation

Measured from Android first stage init



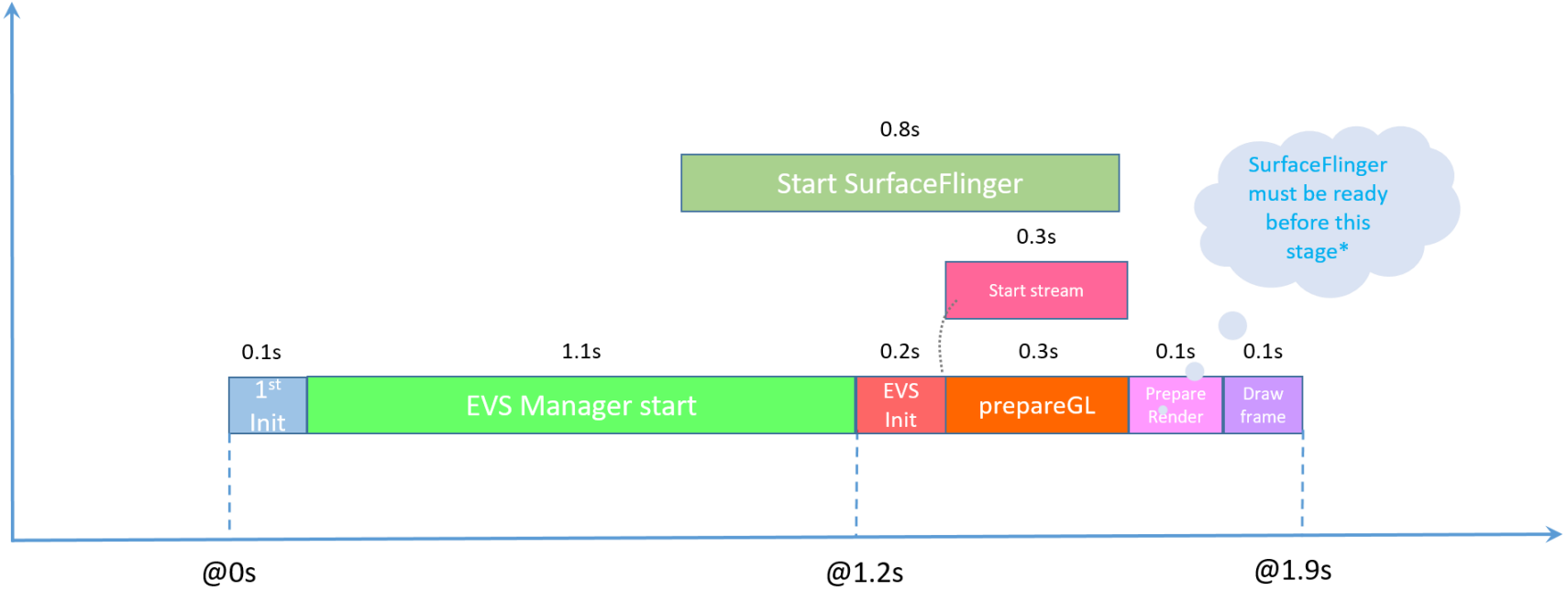
Quick Boot Optimization

Improve by 3 steps

- EVS App: Start Camera Stream with GL preparing concurrently
- EVS HAL: Display frames via composer service before SufaceFlinger is ready
- Android Init: Start EVS related services/HALs earlier (on boot → on early-init)

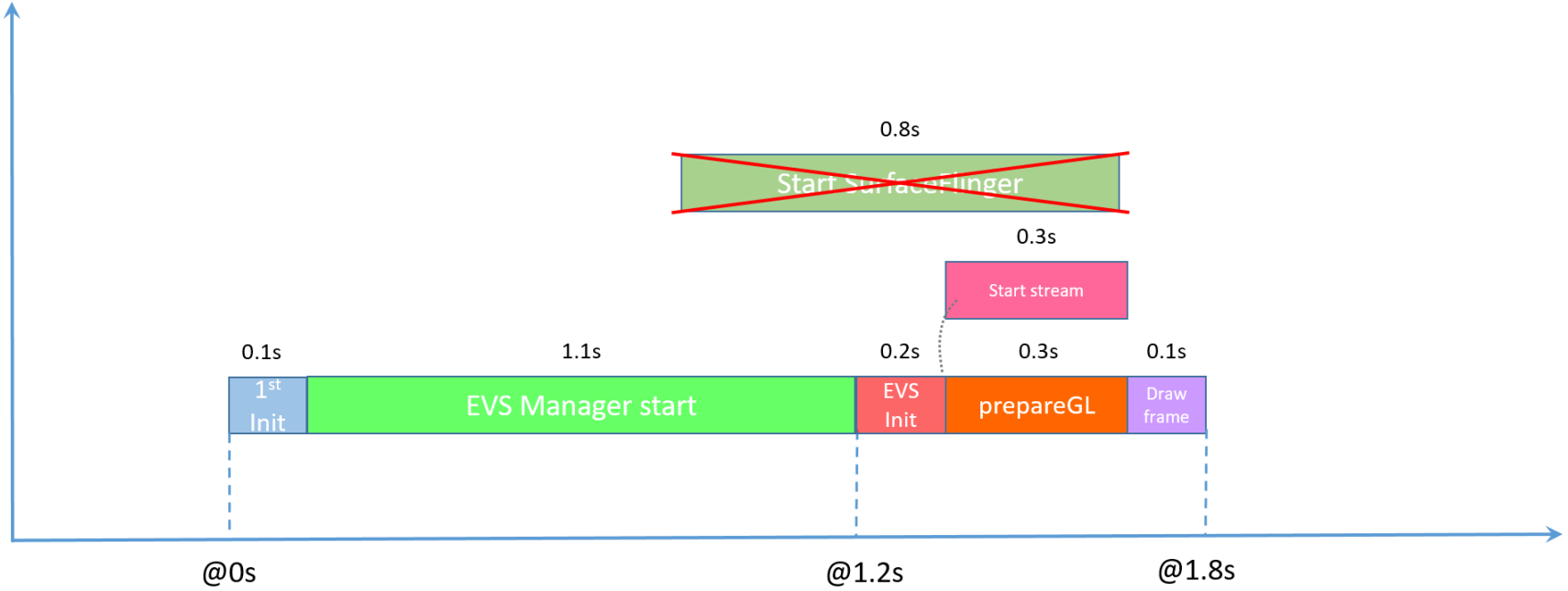
Step 1

EVS App: Start Camera Stream with GL preparing concurrently



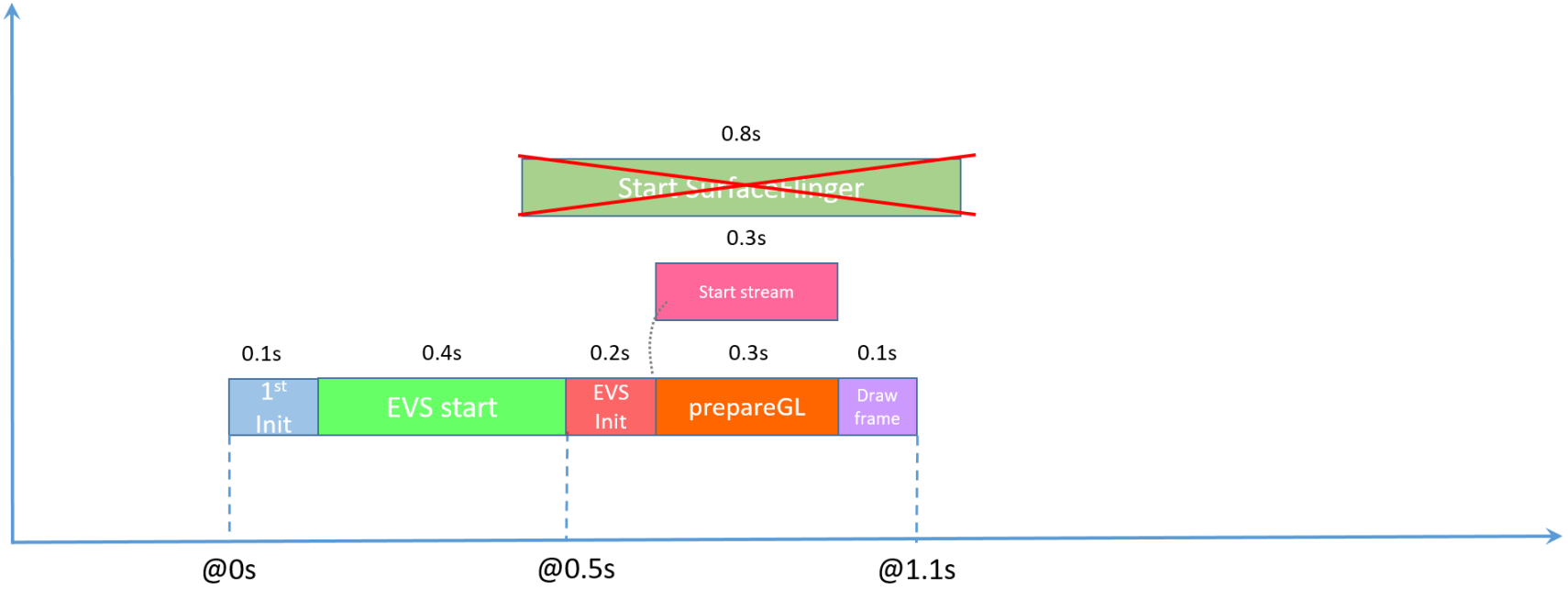
Step 2

EVS HAL: Display frames via composer service before SufaceFlinger is ready



Step 3

Android Init: Start EVS related services/HALs earlier (on boot → on early-init)



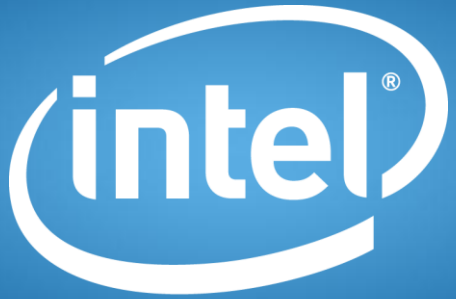
Conclusion

- The optimization can shorten EVS launch time to *1.1s* (from Android first stage init), and the total time is about 3.0s (including bootloader and kernel time) on our hardware development board.
- If we remove GL preparation and texture operations from EVS App, we expect EVS launch time can fall down to 0.7s and the total is 2.6s*.

Suggestion

- TODO: Multiple clients support in Composer Service
  - Composer Service allows only one composer client currently*
  - Bypass SurfaceFlinger to shorten the latency
  - Solve the EVS HAL lib dependency (e.g. libgui is not VNDK lib since P)
  - The temporary solution is to create two composer clients for EVS and SurfaceFlinger successively in Composer HAL, and the EVS HAL should destroy EVS composer client and switch the EVS display to SurfaceFlinger smoothly once it’s ready.

Q & A



Software