دانشگاه تهران
دانشکده‌ی مهندسی برق و کامپیوتر

گزارش نهایی پروژه سیستم‌های سایبر-فیزیکی
طراحی و پیاده‌سازی سیستم ادغام داده بی‌درنگ اطلاعات سنسور لیزری و دوربین استریو در خودروهای خودران (UTFusion)

گروه:

هاتف رضائی و آرین فیروزی و محمد رضا محمد هاشمی و شهزاد ممیز

تاریخ:
۱۴۰۴-۰۵-۲۰

استاد:
دکتر مدرسی

۱۴۰۳-۱۴۰۴

فهرست مطالب

1- مقدمه

- **محدوده پروژه**

این پروژه به عنوان یک ماژول در پروژه ی ماشین خودران UTCar طراحی شده و وظیفه ی دریافت اطلاعات و ادغام سنسور های مختلف را بر عهده دارد. ماژول های ارائه شده در محیط آزمایشگاه تست شده اند و شرایط و محدودیت های اندازه گیری ها در این گزارش ذکر خواهد شد.
پروژه در زمان تقریبی 1.5 ماه (بدون احتساب وقفه ی بین کار) و با هماهنگی اعضای آزمایشگاه دکتر مدرسی و همچنین گروهی که طراحی ماژول پردازش عمق تصویر را بر عهده داشتند انجام شده است.
سخت افزار ها و محیط توسعه در ادامه ی گزارش ذکر شده است.

- **اهداف پروژه**

این پروژه با هدف ترکیب داده‌های سنسور رادار و دوربین استریو در یک سیستم ماشین خودران تعریف شده است. هدف نهایی بهبود دقت در شناسایی اشیا، تابلو ها و تخمین فاصله می باشد که با استفاده از فیوژن داده های چند منبعی انجام می شود تا خطای سنسور دوربین در شرایط مختلف و در صورت وجود نویز های تصویری بهبود یابد. ما تلاش کردیم با ارائه‌ی یک سامانه‌ی فیوژن داده با قابلیت عملکرد بلادرنگ برای افزایش دقت تشخیص موانع به گونه‌ای که داده‌های ترکیب‌شده، در عین حفظ دقت، موجب ایجاد تاخیر محسوس در مسیر پایپلاین پردازش اصلی خودرو نشوند، سیستمی به عنوان بلوک میانی در سامانه‌ی هدایت خودران ارائه کنیم که در افزایش پایداری و دقت مسیریابی کمک کند.
خروجی کلی پروژه تخمین نزدیک فاصله نزدیکترین مانع و یا تابلو های تشخیص داده‌شده توسط اشیا شناسایی شده در فرایند تشخیص عمق در مرحله تشخیص عمق استریو ویژن در پایپ لاین کلی UTCar است.

2- معرفی پلتفرم و ابزارهای استفاده شده

- در این پروژه از پلتفرم ماشین خودران طراحی‌شده توسط دکتر مدرسی استفاده شده است.
- سیستم پردازش مرکزی، یک **Raspberry Pi 5** می‌باشد که به‌عنوان پردازنده ی مرکزی و پردازش داده‌ها عمل می‌کند
- سنسورهای مورد استفاده شامل:
  - **دوربین استریو IMX219** : با بهره‌گیری از دو لنز مجزا و محاسبه اختلاف زاویه دید، امکان ایجاد نقشه عمق و تشخیص اشیاء در محیط فراهم می‌شود.(مانند شناسایی موانع، تخمین عمق و تعیین موقعیت نسبی اشیاء)

- **سنسور TOF**: این سنسور با ارسال پالس لیزری و اندازه‌گیری مدت زمان بازگشت آن، فاصله تا اجسام را با دقت بالا محاسبه می‌کند.

- نرم‌افزار مرکزی ماشین بر پایه **فریم‌ورک Qt** توسعه یافته است و پروژه حاضر به‌عنوان یک زیرسیستم از این نرم‌افزار اصلی عمل می‌کند. این زیرسیستم وظیفه جمع‌آوری داده‌های سنسورها، پردازش اولیه اطلاعات و ارسال نتایج به ماژول تصمیم‌گیری و کنترل حرکت را بر عهده دارد.
- **ابزار ValGrind:** یک فریم ورک instrumentation برای بررسی و تحلیل کد است. ما از ابزار valGrind از زیر مجموعه این فریم ورک برای تحلیل بیشتر کد استفاده کرده ایم.

3- تغییرات نسبت به فاز پروپوزال

چالش‌های پروژه

- از سری چالش های پروژه بی‌درنگ نبودن سیستم عاملی که سیستم ما بر روی آن اجرا می‌شود بود به این ترتیب تضمین دادن زمانی و حافظه ای غیر قابل انجام بود.
- هماهنگی با تیم های دیگری که بر روی پایپلاین UTCar کار میکردند.
- سختی هماهنگ سازی سنسور ها و دیزاین سیستم دو-بافره که یک پترن معروف است

تغییرات نسبت به پروپوزال

- اضافه شدن تیم depthEstimation برای خواندن از بافر دوربین
- تغییر منطق ماژول فیوژن و اجرای ساده تر آن به جای استفاده از روش های ریاضی و بر پایه آموزش[1]

دلایلی که منجر به تغییر در تصمیم‌گیری‌ها شدند

- تصمیمات مطرح شده از تیم ماشین خودران دکتر مدرسی
- کمبود یا عدم توانایی پردازش سخت افزاری برای پردازش های زمان واقعی
- مشترک بودن بخش خواندن از دوربین با تیم depthEstimation

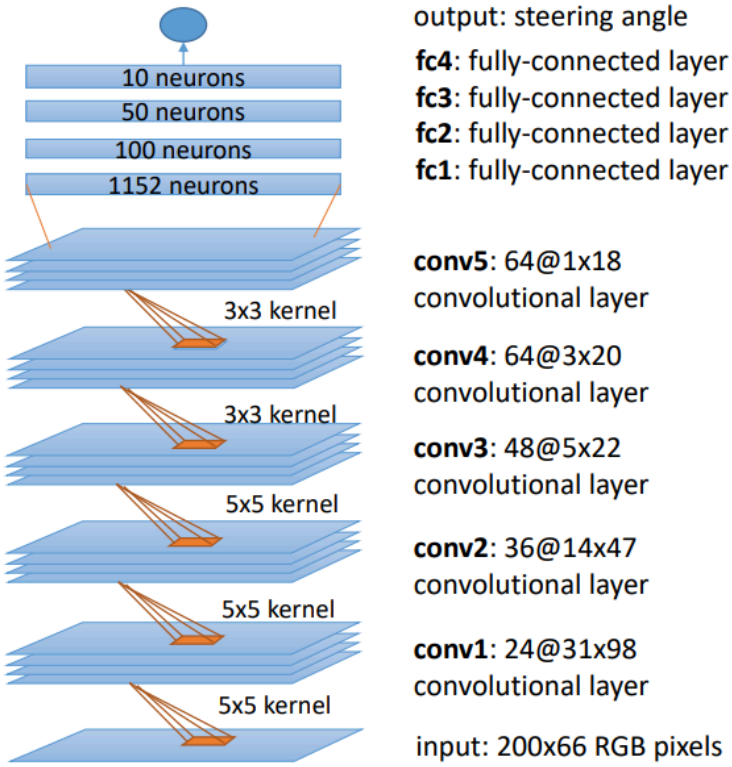
[1] Learning based

- **تجارب ناموفق**

به دلیل نبود سخت افزار مناسب برای اجرای شبکه های عصبی یا روش های ریاضی، فیوژن پیکسل به پیکسل سنسور ها با استفاده از روش های پیشرفته در محدوده زمانی خواسته شده وجود نداشت.
به علت تعطیلی دانشگاه در هفته های پایانی دسترسی کل تیم به سخت افزار ها محدود بود و تست ها به صورت ریموت و با همکاری اعضای آزمایشگاه انجام شد.
به علت آماده نبودن ماژول depthEstimation نتوانستیم تقریب زمانی را با اعداد واقعی بیان کنیم.
همچنین به علت عدم وجود سخت افزار و سیستم عامل real time، زمان پروژه غیر قابل تضمین است.

4- نزدیک‌ترین نمونه‌های مشابه

در حوزه ی سامانه های خودران، یکی از نمونه های مشابه کار انجام شده، DeepPicar است که در ادامه به توضیح آن پرداخته می شود.
معماری سیستم DeepPicar: DeepPicar یک پلتفرم خودروی خودران کم هزینه است که برای شبیه‌سازی معماری و روش کنترل سیستم DAVE-2 شرکت NVIDIA در مقیاس کوچک طراحی شده است. سخت‌افزار این سیستم شامل یک Raspberry Pi 3 به‌عنوان واحد پردازش مرکزی، یک دوربین USB به‌عنوان منبع داده و یک شاسی خودروی RC به‌عنوان بستر حرکتی است. قلب پردازشی سیستم یک شبکه عصبی کانولوشنی ۹ لایه با حدود ۲۷ میلیون اتصال و ۲۵۰ هزار پارامتر است که به‌صورت آفلاین و روی GPU آموزش دیده و قادر است ورودی تصویر خام را به‌صورت مستقیم به زاویه فرمان تبدیل کند.



معماری شبکهٔ عصبی DeepPicar شامل ۹ لایه (۵ لایهٔ کانولوشنی و ۴ لایهٔ فولی کانکتد)، ۲۷ میلیون اتصال و ۲۵۰ هزار پارامتر است. این معماری CNN دقیقاً مشابه معماری استفاده‌شده در خودروی واقعی خودران شرکت NVIDIA است

جریان پردازش و کنترل Real Time: فرآیند کاری DeepPicar با دریافت پیوسته تصاویر از دوربین جلو آغاز می‌شود. هر فریم پس از پیش‌پردازش وارد شبکه عصبی می‌شود تا زاویه فرمان پیش‌بینی شود. این زاویه مستقیماً به سیستم سروو خودروی RC ارسال می‌گردد. حلقه کنترل در زمان واقعی با چرخه‌ای حدود ۲۲٫۸۶ میلی‌ثانیه (نرخ تقریباً ۴۰ هرتز) روی Raspberry Pi 3 اجرا می‌شود که حدود ۸۱٪ زمان چرخه صرف استنتاج شبکه می‌شود. *شبه‌کد صفحه ۴ این مقاله*، نشان می دهد این چرخه شامل چهار مرحله است: (۱) دریافت ورودی حسگر، (۲) پردازش توسط شبکه کانوولوشنی، (۳) ارسال فرمان به عملگرها، و (۴) همگام‌سازی زمان برای حفظ نرخ نمونه‌برداری.

```
while True:
  # 1. read from the forward camera
  frame = camera.read()
  # 2. convert to 200x66 rgb pixels
  frame = preprocess(frame)
  # 3. perform inferencing operation
  angle = DNN_inferencing(frame)
  # 4. motor control
  steering_motor_control(angle)
  # 5. wait till next period begins
  wait_till_next_period()
```

لوپ کنترل

تحلیل عملکرد و محدودیت‌ها: تحقیقات DeepPicar علاوه بر نمایش قابلیت عملکردی، ارزیابی دقیقی از تنگناهای سیستمی در شرایط بلادرنگ ارائه می‌دهد. آزمایش‌ها شامل بررسی مقیاس‌پذیری چندهسته‌ای، تأخیر ناشی از رقابت بر سر پهنای باند حافظه، و اثرات پارتیشن‌بندی کش هستند. نتایج نشان می‌دهند که مقیاس‌پذیری بیش از سه هسته محدود است و تحت بار نوشتن سنگین در حافظه، زمان استنتاج ممکن است تا ۱۱٫۶ برابر افزایش یابد. آزمایش‌های ایزوله‌سازی منابع حاکی از آن است که پارتیشن‌بندی کش تأثیر معناداری ندارد اما محدود سازی پهنای باند حافظه با ابزارهایی مانند MemGuard می‌تواند تداخل را کاهش دهد و عملکرد ایده‌آل به پهنای باند حداقل ۴۰۰MB/s نیاز دارد.

در حالی که DeepPicar نمونه‌ای از یک سیستم کنترل مبتنی بر بینایی و یادگیری عمیق end-to-end است که تنها از یک منبع حسگر (دوربین) استفاده می‌کند، UTFusion رویکردی مبتنی بر ادغام داده‌های چند حسگر اتخاذ کرده است. UTFusion ورودی‌های دوربین استریو و سنسور لیزری TOF را در یک مسیر مشترک پردازشی شامل ماژول‌های DataContainer، Buffer و PerformFusion ادغام می‌کند تا فاصله دقیق موانع یا تابلوها را تعیین نماید. این طراحی روی Raspberry Pi 5 اجرا می‌شود و با تاخیر حدود ۱۲ میلی‌ثانیه، تمرکز آن بر همگام‌سازی دقیق داده‌ها و کاهش خطای تخمین فاصله است. DeepPicar به‌طور ویژه چالش‌های اجرای شبکه‌های عصبی روی سخت‌افزار محدود و مدیریت منابع را بررسی می‌کند، در حالی که UTFusion بر قابلیت اطمینان داده، افزونگی سنسوری و منطق ادغام برای افزایش دقت ادراک محیط تمرکز دارد.

5- مبانی فنی پروژه

- **ارائه راه‌حل(ها) با جزییات**

در این پروژه ما با سه مشکل اصلی مواجه بودیم:

1. هماهنگی بین سنسور ها و صحت زمانی داده های پردازش شده
2. نحوه ی پردازش اطلاعات دریافتی از ماژول های خارج از پروژه
3. تضمین زمانی عملکرد مستقل از ماژول های خارجی

که راه حل های ارائه شده برای برخی کامل و برای برخی ناکافی یا ناقص ارزیابی شدند. در ادامه راه حل های ممکن و دلیل انتخاب شرح داده شده اند.

1. هماهنگی بین سنسور ها و صحت زمانی داده های پردازش شده

هدف این بخش توضیح شفاف سازوکاری است که با آن داده‌های دوربین و رادار با حداقل تصادم، به‌صورت همگام و قابل اتکا ذخیره و مصرف می‌شوند. رویکرد ما ترکیبی از بافر حلقوی برای هر سنسور و یک الگوی دو-بافره در لایه‌ی جمع آوری اطلاعات است تا منطقه بحرانی کوتاه، تأخیر کم، و همگام‌سازی زمانی پایدار حاصل شود. به این ترتیب که یکی از بافر ها برای خواندن و دیگری برای نوشتن است، در صورتی که دیتاهای نوشته روی یک بافر اختلاف زمانی کمتر از مقدار تعریف شده برای Domain پروژه باشد(این مقدار با توجه به حداکثر سرعت ماشین و فاصله ی زمانی تا 0 شدن سرعت ماشین از 100% آن است) این دو بافر در یک بازه ی critical کوتاه جا به جا می شوند تا بافر مربوط به خواندن دیتا ی جدید تری را داشته باشد. همچنین موقع نوشتن ماژولی که در حال خواندن اطلاعات است برای خواندن پشت یک قفل معطل نشود.

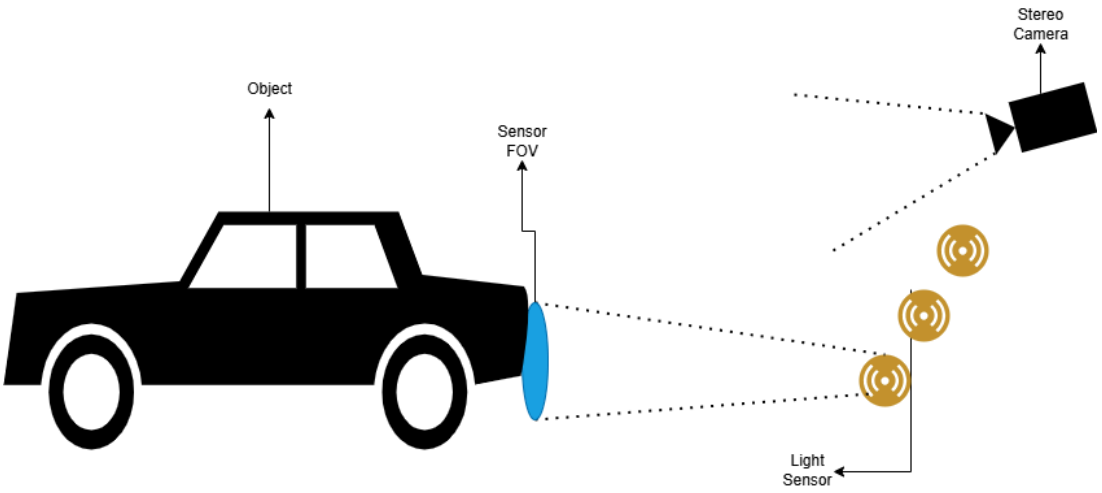
2. نحوه ی پردازش اطلاعات دریافتی از ماژول های خارج از پروژه

اصلی ترین بخش این پروژه مربوط به راه حل این بخش است. کار Fusion معمولا چالشی است که با استفاده از پردازش های پیشرفته و روش های بر پایه آموزش (learning based) یا ریاضی حل میشود. در مورد پروژه ما چنین امکانی وجود نداشت و ما مجبور بودیم روش ساده و قابل پیشبینی ارائه بدهیم. با توجه به این که در نوع داده (رادار و دوربین) به ما ارائه شده بود و هدف اصلی تلفیق دقت رادار با توانایی تشخیص اشیای دوربین بود؛ ما به جای پردازش پیکسل کل تصویر تنها اشیا را ملاک قرار دادیم و با استفاده از ارقام رادار، اقدام به اصلاح دقت دوربین کردیم.

قبل از ارائه ی راه حل، ما نیاز به فرموله کردن فضای حالات ماشین داریم. هوش مصنوعی به کار رفته در ماشین (اعم از سیستم های یادگیری تقویتی، ماژول های CNN یا روش های rule-based) نیاز به یک سری داده از دنیای واقعی دارد. این داده ها در یک دستگاه مختصات سه بعدی قرار دارند که توسط

سیستم قابل تشخیص است. دستگاه مختصات به کار رفته در سیستم تصمیم گیرنده، سیستم دریافت اطلاعات و سیستم پردازش تصویر باید قابل تبدیل به همدیگر باشد تا بتوانیم از صحت داده ها اطمینان حاصل کنیم. به این منظور، سیستم دریافت اطلاعات باید داده های سنسور ها مختلف را از یک نقطه ی مبدا در محور مختصات ارائه کند، سیستم پردازش به درستی این اطلاعات را از فضای دو بعدی تصاویر به فضای سه بعدی تلفیقی تبدیل کند تا اطلاعات قابل ارائه به سیستم تصمیم گیرنده باشد.

بخش اول این کار، یعنی تبدیل فضای دو بعدی دوربین به سه بعد توسط دو ماژول خارجی انجام میگیرد که با استفاده از دوربین استریو را ابتدا اشیا را تشخیص داده و سپس فاصله تخمینی آن ها را به صورت پیکسلی ارائه میکنند. در ادامه ی این داکیومنت، بدون توجه به مکانیزم داخلی به کار رفته، این دو ماژول به صورت black box تحلیل خواهند شد و از این دو ماژول به اسامی ماژول تشخیص اشیا (ObjectDetector) و استخراج فواصل نسبی (DistanceEst) یاد خواهد شد. تنها اطلاعات لازم این است که این دو ماژول به صورت مستقل اما متوالی کار میکنند و ورودی آنها تصویر دوربین و خروجی آن گروهی از اشیا با دقت پیکسلی در مختصات سه بعدی با مبدا مختصات دوربین فیزیکی است.



حال در صورتی که خروجی مورد نظر را داشته باشیم، این مختصات باید با مختصات رادار ها تطبیق داده شود. این تبدیل توسط یک ماژول به صورت غیر خطی و با پارامتر های از پیش اندازه گیری شده انجام میشود و به دو روش قابل انجام است:

الف) تبدیل رادار به دوربین: با توجه به نحوه قرار گیری رادار ها، هر رادار یک قوس بسته در فضای دوربین را میتواند تشخیص دهد. این قوس بسته به نحوه قرار گیری رادار و دوربین، فاصله اشیا در لحظه ی ثبت داده ها و نحوه پردازش تصویر متغیر است و بخشی از این تبدیل باید به صورت دینامیک در هر فریم از تصاویر انجام شود. محاسبه پیکسل های مربوط به رادار در هر تصویر زمانبر است و پیچیدگی های مربوط به ایجاد یک قوس و تبدیل آن به پیکسل های در بر گرفته شده توسط قوس جزو سختی های اسن راه حل است. در مقابل، اگر بتوانیم این تبدیل را محاسبه کنیم ادامه ی محاسبات و اثبات صحت فیوژن آسانتر است. به تعبیری این ادغام interpretability بیشتری خواهد داشت اما هزینه محاسباتی آن نیز بیشتر است.

ب) تبدیل پیکسل ها به رادار: در این روش بر عکس روش الف از پیکسل به رادار یا رادارهای مربوطه میرسیم. روشی که برای این کار در نظر گرفتیم، به جای استفاده از محاسبات قوس و فضای پیوسته، به صورت حریصانه نزدیکترین رادار های موجود را انتخاب میکند و با توجه به مکانیزم هایی که در بخش پیاده سازی شرح خواهیم داد، مشکلات مربوط به mask شدن شی توسط شی دیگر و نقاط کور رادار ها در زمان خطی حل می شوند.
با توجه به دلایل شرح داده شده، ما راه حل دوم را برای انجام ادغام انتخاب کردیم.

3. تضمین زمانی عملکرد مستقل از ماژول های خارجی

یکی از پیچیدگی های طرح پروژه، تضمین انجام خط لوله در کنار ماژول های خارجی دیگر بود که به توجه به محدودیت های سخت افزاری، سیستم عامل غیر بی درنگ و عدم اطمینان موجود در ماژول های دیگر، محاسبه ی زمان دقیق ادغام را ناممکن میکند. برای فائق آمدن بر چنین مشکلاتی، ما پایپلاین خود را در دو ریسه و بدون در نظر گرفتن preemption از طریق پروسه های دیگر تحلیل کردیم و فرض بر این است که دو ریسه از 4 هسته ی موجود در سخت افزار به این پایپلاین اختصاص داده شود. همچنین در کد از کتابخانه ها استفاده ی محدود و حساب شده ای انجام گرفته و از استفاده از همزمانی (به جز در سنسور ها که بخش اساسی کارشان است) خودداری شده است. با وجود این، به علت وجود cache و سیستم عامل نامناسب، اعدادی که در این سند اعلام شده اند قطعی نیستند و تنها تقریبی از بیشترین زمان هستند. تحلیل دقیق نیازمند سخت افزار و سیستم عامل سازگار بوده و از توان نویسندگان خارج است.

- **نحوه تحلیل راه‌حل و اثبات کارایی (مثلا زمان تاخیر، مصرف حافظه و ...)**

1. تحلیل زمان و حافظه

برای تحلیل زمانی بدیهی است که تشخیص دقیق و قابل اطمینان زمان، نیازمند یک تیم متخصص و با تجربه است و آنچه در این بخش ارائه شده، تنها تخمینی از بدترین حالت موجود میباشد.
در گام نخست، ما توابع مورد استفاده را در طی پروژه به دست آوردیم و تلاش بر این بود که تا حد امکان از توابع قابل اطمینان با کمترین نایقینی ممکن استفاده شود. به این منظور تا جای ممکن از موازی سازی خودداری شد و توابع در ساده ترین حالت ممکن با کمترین دستورات شرطی پیاده سازی شدند. تمامی توابع استفاده شده به استثنای توابع دریافت اطلاعات از سنسور ها، به صورت sequential نوشته شده اند و از توابع موازی، بازگشتی و مکانیزم سیگنالینگ در هیچکدام از اینها استفاده نشده است. توابع مربوط به دریافت اطلاعات از سنسور ها بر اساس نحوه ی کار سنسور ها، به صورت جداگانه تحلیل شدند و با اینکه اظهار نظر دقیق در مورد زمان قابل انجام نیست، تحت شرایطی که ذکر خواهد شد تخمین قابل قبولی در مورد آن داریم.

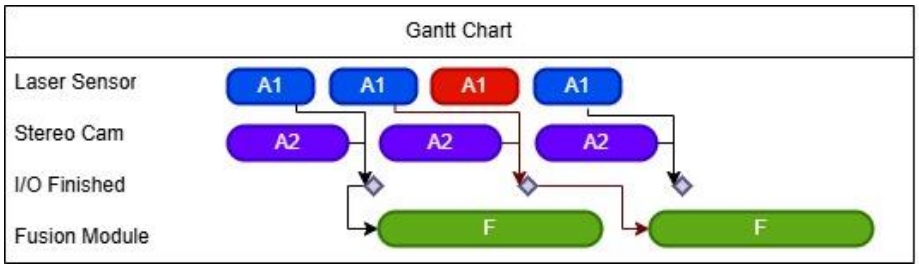
در ماژول های دریافت اطلاعات، به منظور کاهش فاصله بین ثبت واقعه از سنسور تا خروجی مدل (زمان انتها به انتهای کل سیستم) از روش بافرینگ استفاده شده و دو بخش سیستم را به طور ناهمگام اجرا میشود. روش کار به این شکل است که ابتدا دو نوع سنسور ورودی به صورت همزمان شروع به کار کرده، اطلاعات را میخوانند و در بافری با اندازه یک مینویسند (در این حالت اگر اطلاعات جدیدی وارد شود، اطلاعات قبلی دیگر معتبر نیست و پاک میشود). برای این کار نیازی به برچسب زمانی جدا نیست، زیرا اطلاعات موجود در بافرها مربوط به آخرین داده ی سنسور لیزری و آخرین داده ی دوربین است. میتوان نشان داد که فاصله ی زمانی این دو در بدترین حالت به صورت بیشینه A1 و A2 است. در این حالت، می توان گفت که قدیمی ترین داده ی موجود در بافر زمان خوانده شدن توسط مدل ترکیب، در بدترین حالت برابر A1+A2 است و این مقدار با فرض اینکه سنسور ها 50 و 30 فریم بر ثانیه میخوانند (طبق اطلاعات دیتاشیت)، برابر با 0.053 ثانیه است. این مقدار را A می نامیم.

در مورد مدل ترکیب با توجه به اینکه عملیات ثابت و مکان داده ها نیز ثابتند، نه به صورت ریاضی ولی با استفاده از ابزار های profiling میتوان تقریب خوبی از میزان زمان بدترین حالت استخراج کرد. این مقدار را F مینامیم.

هنگام استفاده مشترک از این دو ماژول، بدترین حالت در گانت چارت تصویر 2 با فلش قرمز نشان داده شده و مشاهده میکنیم که حداکثر فاصله جواب خروجی از زمان واقعی ثبت میتواند به صورت زیر باشد:

$$\text{WCE} = 2A + F + buffer\ I/O$$

البته این 2A هیچگاه اتفاق نمی افتد ولی به علت سختی محاسبه دقیق و عدم اطمینان از اعداد همین مقدار گرفته شد.

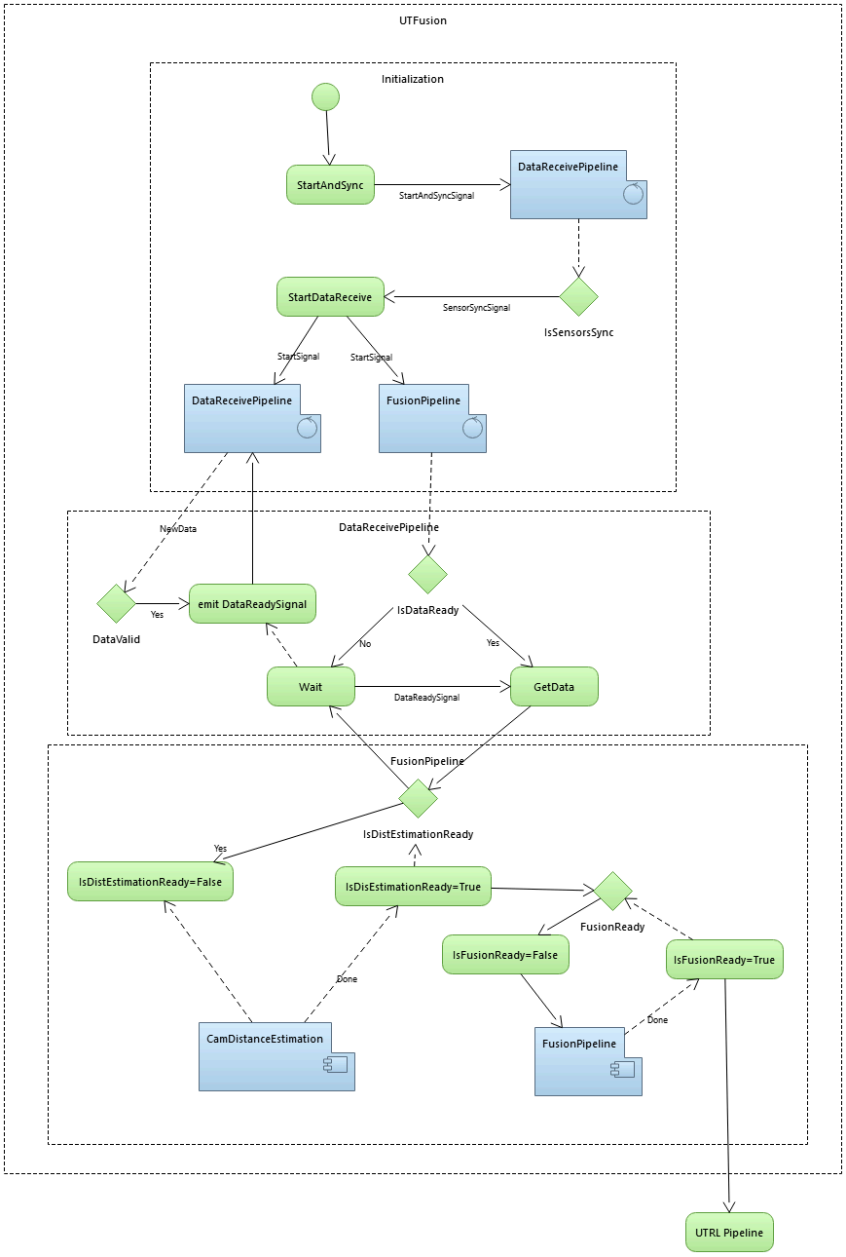


نحوه کار بافر برای جلوگیری از تصادم به این شکل تضمین میشود که در واقع دو بافر، یکی برای نوشتن و دیگری برای خواندن خواهیم داشت. هر یک از سنسور ها در یک بافر، در دو خانه ی مختلف می نویسند. زمانی که سنسوری در خانه ی مربوط به خودش مینویسد، چک میکند که آیا داده ی سنسور دیگر در خانه ی دیگر وجود دارد یا خیر. در صورتی که وجود داشته باشد و برچسب زمانی آن داده، فاصله ای کمتر از کمینه زمان فاصله قابل قبول با داده ی سنسور داشته باشد (در واقع داده ی دیگر قابل ترکیب باشد)، وضعیت دو بافر را جابجا میکند (بافر نوشتن تبدیل به بافر خواندن میشود و بالعکس). در

این حالت می توان گفت که هر موقع ماژول ترکیب بخواهد داده ای را بخواند، حتما یک داده در بافر خواندن وجود دارد که حداکثر فاصله زمانی آن از واقعیت برابر 2A محاسبه شده است و تصادم رخ نمیدهد.

برای توابع sequential در فیوژن، ما هر تابع را به تنهایی و سپس تمام توابع را با همدیگر بررسی کردیم و داده های واقعی در محیط اجرا جمع آوری شد. تست ها به گونه ای انجام شدند که دادگان و مسیر های طی شده در توابع در حالتی نزدیک به کمترین حالت خود قرار داشته باشند و در صورتی که سربار cache را در نظر نگیریم، زمان های به دست آمده به بدترین حالت نزدیکند.

نمودار flow کار در شکل زیر نشان داده شده است. بخش initialization در محاسبات لحاظ نشده و



فرض بر این است که این فاز قبل از حرکت ماشین رخ میدهد. بخش data receive پیش تر توضیح داده شد و بخش نهایی یعنی پایپلاین فیوژن در قسمت تست تشریح خواهد شد.

دو پایپلاین آخر که در طول اجرا فعالند، به صورت مستقل ولی تنیده به همدیگر اجرا میشوند، به این شکل که هر کدام در ریسه ی مربوط به خودشان اجرا می شوند و تاخیر نهایی مجموع بدترین حالت این دو است. به شکل دقیق تر داریم:

$$MaxDelay = DM + DR + UTFusion + ST$$

$$PipeLen = DM + DR + UTFusion$$
$$UTFusion = F + DE + BD$$
$$BD = 2A + I/O$$
$$A = A1 + A2$$

که در آن DM معادل WCET ماژول تصمیم‌گیری، DR معادل تأخیر بین تصمیم‌گیری و واکنش، ST معادل زمان توقف خودرو از Vmax ، متغیر F معادل ماژول فیوژن، DE معادل Depth Estimation WCET، BD معادل سن[2] داده‌های بافر شده و A1 و A2 معادل حداکثر سن داده‌ها هنگام دریافت از یک حسگر هستند. توضیحات بیشتر در مورد دلیل محاسبه BD در بخش دو-بافر توضیح داده شده است و توضیحات مفصل در مورد این محاسبات در پیوست فنی محاسبه بدترین حالت موجود است. همچنین در اینجا DE مجموع پایپلاین yolo و depth estimation است و تفاوتی بین آنها قائل نشدیم ولی در پروژه اصلی این دو ماژول جدا هستند و زمان محاسبه آنها باید جداگانه حساب شود.

از جایی که سایر متغیرها یا ماژول های خارجی هستند و یا مربوط به سنسور ها، ما بخش F را مفصل تر بررسی خواهیم کرد. ماژول فیوژن، یا به طور دقیق تر تابع performFusion متشکل از مراحل زیر است:

1. **تابع fill از std:** که تابع stable و ساده ای است و به مدت زمان $O(N_{image\ size})$ طول میکشد.
2. حلقه بر روی آبجکت ها: تعداد آبجکت ها حداکثر 80 است اما به دلیل زمان ما این تعداد را مشروط بر اندازه اشیا در نظر میگیریم.
3. **حلقه بر روی پیکسل های اشیا:** ما حد بالایی بر تعداد پیکسل هر شی در نظر نگرفتیم، ولی برای اینکه زمان های ارائه شده صحیح باشند، باید مطمئن شد که مجموع تعداد پیکسل های اشیا از اندازه عکس بیشتر نباشد. به این منظور ما از اجرا حلقه بیش از یک بار بر روی هر پیکسل خودداری می کنیم و پیکسل هایی که قبلا دیده شده اند در یک آرایه نگهداری میشوند. خود این آرایه و خواندن از آن ممکن است نایقینی (به علت cache) ایجاد میکند که همانطور که ذکر شد ما نایقینی کش را در محاسباتمان حساب نکردیم. همچنین skip کردن حلقه ها (continue) نیز سرباری دارد که چون تنها یک عملیات goto است آن را ناچیز در نظر میگیریم.
4. **تابع pixelToWorld:** خود این تابع از 18 عملیات ضرب اعشاری، 5 تقسیم اعشاری، 13 عملیات تخصیص حافظه و تعدادی جمع و تفریق تشکیل شده است. باز هم در غیاب کش، این عملیات با سرعت نسبتا stable ای اجرا میشود که در تست ها لحاظ شده، تنها نا یقینی موجود به جز cache، بهینه سازی های سخت افزاری برخی روابط ریاضی است که از توان تحلیل گروه خارج است. در نتیجه سرعت این تابع را نیز ثابت در نظر میگیریم.
5. **شرط:** یک شرط با خروجی تابع findBracketingRadars داریم:

[2] age

a. ***تابع findBracketingRadars :*** یک حلقه بر روی رادار ها دارد و از مرتبه $O(N_{Radar\ Len})$ است.

b. ***findClosestRadar:*** در صورت عدم برقراری شرط، مسیر طولانی تر میشود و تابع findClosestRadar صدا زده میشود. این تابع نیز از مرتبه زمانی $O(N_{Radar\ Len})$ است.

6. **حلقه بر روی رادار های نزدیک:** این حلقه در صورت اجرای مسیر b یک بار و در صورت اجرای مسیر a دوبار تکرار میشود. شامل توابع distance (که خود داری یک سری ضرب و جمع و یک تابع sqrt از std اس) و تابع abs از std و تعدادی عملیات تخصیص حافظه و جمع و تفریق و عملیات های مقایسه ای است. تمام این توابع در مرتبه زمانی ثابت انجام می شوند جز sqrt که از مرتبه $O(Nlog^2N)$ بوده و N طول عدد float است که آن هم در طول پروژه ثابت می ماند. پس این بخش نیز stable است.

با در نظر گیری این تحلیل، مرتبه زمانی به این صورت است:

$$O(F) = O(Nimglen * (O(pixel2World) + 2Nradarlen + Nradarlen * O(logNfloatlen) + O(Nimglen)$$

نتیجه آن که این مرتبه زمانی تقریبا stable است و با ورودی تغییر چشمگیری نمیکند. پس با ساده سازی های در نظر گرفته شده، با تست این تابع در سخت افزار از مسیر های a و b طولانی ترین مسیر را با تقریب قابل قبولی به دست می آوریم.

برای تحلیل حافظه مورد نیاز، ما ابتدا به صورت استاتیک و با استفاده از متغیرهای استفاده شده، میزان حافظه در هر تابع را به دست آوردیم و سپس با اجرای پروژه در یک ریسه بدون preemption تست واقعی گرفتیم و حافظه ی استفاده شده قبل و بعد از اجرای توابع را به دست آوردیم که نتایج آن در قسمت تست ها مشخص شده است. همانطور که در آن بخش نیز ذکر خواهد شد، پایپلاین مصرف حافظه ناچیزی دارد و تنها بخش پرمصرف حافظه محل نگهداری اشیا و عکس است که قابل بهینه سازی نیست.

2. تحلیل صحت اجرا

به دلیل تنیده بودن این موضوع با نحوه ی پیاده سازی، تحلیل صحت در بخش پیاده سازی توضیح داده شده است.

6- جزئیات پیاده‌سازی

- **شکست کار بین اعضای تیم**

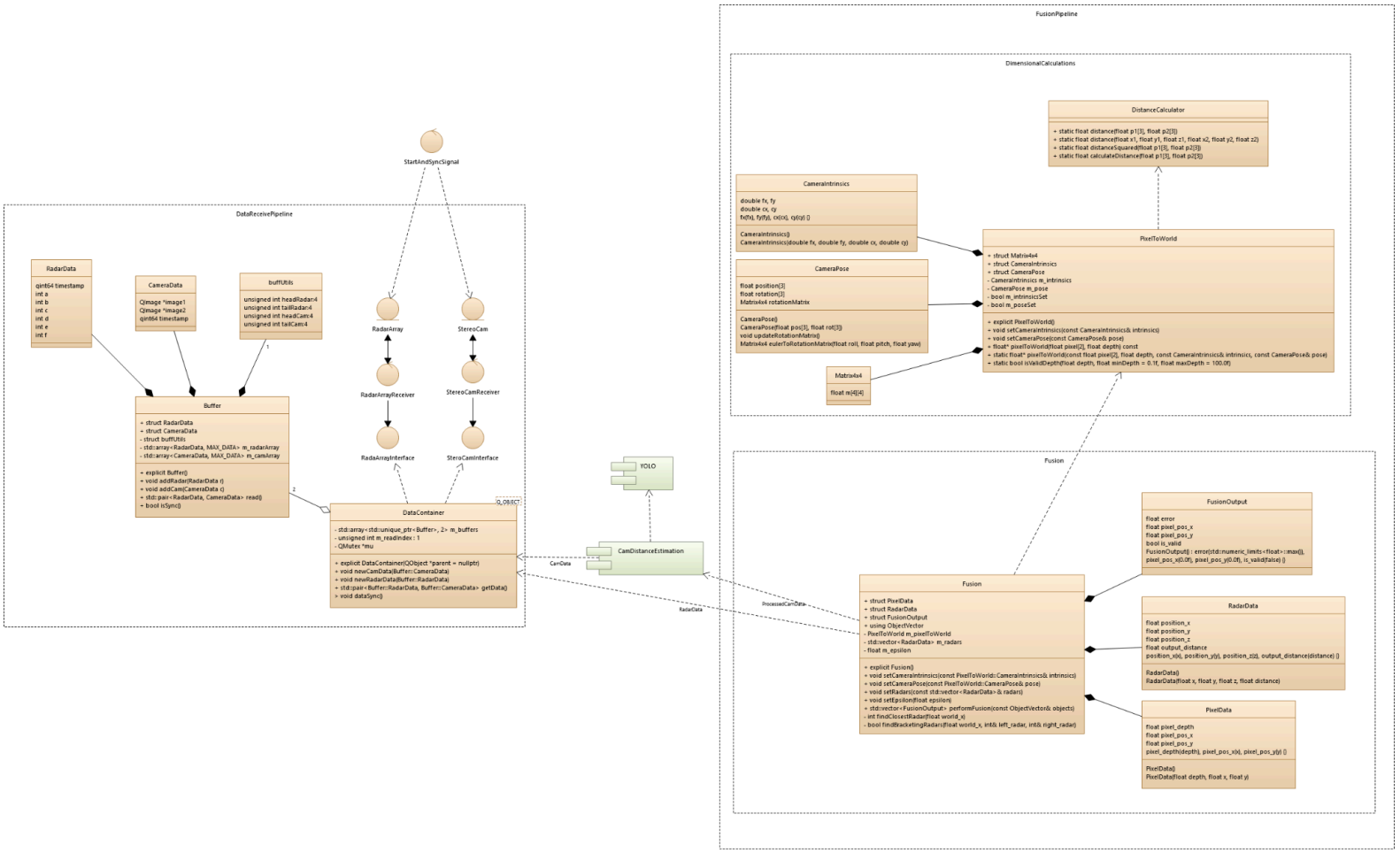
Index	Task	Dependency	Assignee(s)
0	Buffer	-	Hatef - Arian
1	Data Container	0	Hatef
2	Distance Transformations	-	Mohammadreza
3	WCET Calculation-Approximation Methods	1	Arian
4	Time Stamping from Sensors	-	Shahzad
5	Fusion	0, 1, 2, 4	Mohammadreza, Arian(minor modifications only)
6	Dedicated Cores in RaspberryPi	-	Hatef
7	Overall Test	0-6	Shahzad
8	Integration with Depth Estimation	0, 1, 3	Undone (the depth estimation module was not presented to us at the time)

9	Integration with Sensors	0, 1	Shahzad
10	Code Analysis	all	Arian
11	Final WCET approximation	all	Arian
12	Final Report	all	All
13	Implementation of mock data	all	Hatef

- **مشخصات محیط توسعه**
  - محیط توسعه ی این پروژه، پلتفرم ماشین خودران دکتر مدرسی (UTCar) می باشد. برای توسعه ی این پروژه از Qt به عنوان سیستم اصلی استفاده شده است و طراحی ماژولار پروژه باعث اضافه شدن ماژول های مختلف از جمله سیستم مارا دارد. تنها لازم است سیگنال ها و اسلات های پروژه ی ما که رابط خروجی برنامه ما هستند (API) به درستی به دیتا های واقعی ماشین متصل شده و اجرا شوند.

- **تشریح پیاده‌سازی**

نمودار UML پروژه:



ماژول بافر

برای ذخیره کردن داده های دوربین و رادار به طور جداگانه بافر حلقوی پیاده سازی شد. که داده های این دو را در دو آرایه جداگانه m_radarArray و m_camArray ذخیره میشوند. tailRadar و tailCam برای مشخص کردن آخرین موقعیت درج داده ها در هر بافر استفاده می شود.

همچنین از Size% برای مدیریت کردن چرخش ایندکس ها در بافر استفاده می شود. علاوه بر این اگر قدر مطلق اختلاف زمانی بین داده های دوربین و رادار کمتر از TIMESTAMP_MAX_DRIFT اشد، داده ها همزمان در نظر گرفته میشوند.

این کلاس درنهایت تضمین می کند داده ها به صورت همزمان و بدون تصادم ذخیره و خوانده می شوند.

ماژول نگهداری اطلاعات (DataContainer)

این کلاس از دو بافر منطقی تشکیل شده است: یکی مخصوص «خواندن» و دیگری مخصوص «نوشتن». در هر لحظه:

- **بافر خواندنی** حاوی جفت‌داده‌های همگام (دوربین/رادار) است که ابتدای آن نزدیک‌ترین اختلاف زمانی را دارد و برای مصرف‌کننده آماده است.
- **بافر نوشتنی** در پس‌زمینه با داده‌های تازه پر می‌شود تا قفل‌گیری و سربار به حداقل برسد.

در بازه‌های زمانی کوتاه و فقط پس از اطمینان از آماده‌بودن داده‌ها، اشاره‌گرها به‌صورت اتمیک جابه‌جا می‌شوند تا با کمترین منطقه ی بحرانی، بافر نوشتنی به خواندنی تبدیل شود. با جداکردن مسیر خواندن و نوشتن، تصادم قفل‌ها کاهش یافته و مصرف‌کننده همواره به مجموعه‌ای از داده‌های ازپیش همگام شده دسترسی دارد.

کلاس Buffer (بافر حلقوی برای هر سنسور): برای هر سنسور یک بافر حلقوی مجزا پیاده‌سازی شده است؛ داده‌های دوربین در m_camArray و داده‌های رادار در m_radarArray نگه‌داری می‌شوند. متغیرهای tailCam و tailRadar آخرین موقعیت درج در هر بافر را مشخص می‌کنند. با استفاده از عملگر باقیمانده (اندیس % اندازه‌ی بافر)، چرخش ایندکس‌ها به‌صورت کارا و بدون جابه‌جایی داده مدیریت می‌شود. درج هر سنسور مستقل از دیگری انجام می‌شود؛ در نتیجه خواندن با یک «عکس لحظه‌ای» از اندیس‌ها، از ناسازگاری حین خواندن جلوگیری می‌کند.

ماژول های ماک دیتای رادار و دوربین

این کلاس وظیفه ی انجام عملیات دادن دیتای ماک رادار و دوربین را به ورودی برنامه برای انجام عملیات های Integration Test را دارد؛ این کلاس ها همان‌طور که در ادامه به صورت دقیق تر توضیح داده خواهد شد یک Web Server هستند که دیتا را از یک کلاینت ساده دریافت و وارد برنامه می کنند به این ترتیب می توان نویز محیط فیزیکی را با ماهیت unreliability وب سرور و کانکشن tcp مدل شود.

همچنین نکته دیگر این است که با این کار هر دوی داده ها در واقع از دو سورس جدا می آیند و می توان حالات عدم هماهنگی دیتای رادار و دوربین را مدل کرد. این کلاس بعد از پارس کردن دیتای ورودی یک سیگنال Date Ready را Emit می کند.

ماژول فیوژن

این کلاس وظیفه ی انجام عملیات فیوژن را بر عهده دارد که شامل یک اینستنس از کلاس PixelToWorld است که بر اساس کالیبریشن دوربین و موقعیت دوربین مقدار دهی شده و آماده استفاده است. همچنین موقعیت رادار ها را دارد و فاصله ارائه شده توسط رادار ها در هر سمپل گیری

را به عنوان ورودی متد performFusion دریافت میکند همچنین ورودی دیگر این متود وکتوری از آبجکت ها است که هر آبجکت متشکل از مقدار تخمینی عمق تصویر توسط ماژول تخمین زننده و پوزیشن پیکسلی هر پیکسل است.

تا پیش از این فرایند اطلاعات موجود اطلاعات بازگشتی رادار ها به همراه موقعیتشان و اطلاعات تمامی پیکسل های هر وسیله به فرمی که شامل موقعیت پیکلی هر پیکسل و عمق تخمین زده شده توسط فرایند تخمین عمق است که با توجه به انجام این عمل به کمک عکس های دو دوربین Stereo vision انجام شده که به دلیل Distortion موجود در عکس و محدودیت دقت اندازه گیری و محاسبات دارای خطای اندکی میباشد.

با توجه به آنکه موقعیت دوربین و سپر (خط قرارگیری رادار ها) لزوما در موقعیت یکسانی نیست در نتیجه لزوما نزدیکترین پیکسل هر وسیله بر اساس عمق تخمین زده شده از عکس نزدیک ترین موقعیت آن وسیله به سپر نیست در نتیجه لازم است تا ابتدا موقعیت مکانی سه بعدی هر پیکسل را محاسبه کنیم تا نزدیکترین موقعیت هر آبجکت به سپر را بتوانیم به دست بیاوریم. و همچنین خطای تخمین عمق عکس را نیز تا حدودی مدیریت کنیم و موقعیت صحیح را به دست بیاوریم. جهت انجام این هدف پس از تخمین موقعیت حدودی هر پیکسل تصویر به کمک کلاس PixelToWorld که توضیحات مرتبط با آن در پایین تر ذکر خواهد شد، موقعیت مکانی سه بعدی هر پیکسل را به دست می آوریم. همچنین یک فرض اساسی در محاسبات مربوطه آن است که سپر (خط متصل کننده رادار ها به یکدیگر) هم راستای محور x دستگاه مختصات نسبی ما است (با در نظر گرفتن یک نقطه مشخص از ماشین فرضا نقطه میان رادار ها یا نقطه ای از ماشین و محور x بیان شده با حرکت و چرخش ماشین متناظرا مبدا مختصات و محور های دستگاه مختصات همراه ماشین چرخیده تا تمامی مختصات ها نسبت با ماشین حساب شوند و نه بر اساس یک دستگاه مختصات متناظر با ناظر دیگر). با توجه به فرض محور x در نتیجه جهت پیدا کردن نزدیک ترین رادار های به هر موقعیت مکانی کافیست تا بر اساس محور x پیدا کنیم که بین کدام دو رادار یا از کدام سمت خارج از مجموعه یه رادار ها است زیرا تفاوت هر موقعیت در محور y و z برای تمامی رادار ها یکسان خواهد بود و تنها عامل تفاوت x است زیرا رادار ها در خطی موازی محور x هستند و در نتیجه y و z یکسانی دارند. در نتیجه اگر موقعیت آن پیکسل میان دو رادار بود فرایند فیوژن احتمالی را برای آن دو رادار و اگر بیرون از فضای رو به روی رادار ها بود برای نزدیک ترین رادار انجام می دهیم زیرا اگر یک وسیله برای رادار نزدیکش در هر سمت مسدود شده باشد برای تمامی رادار های دور تر آن سمت نیز بلاک شده است و همچنین اگر یک وسیله نزدیکترین وسیله به رادار های دور تر هر سمت باشد قطعا نزدیک ترین وسیله به نزدیک ترین رادار آن سمت نیز هست و با انتخاب کوچکترین فاصله گزارش شده رادار ها تاثیر خطای احتمالی رادار ها که با افزایش فاصله از رادار افزایش میابد نیز محدود میکنیم.

خروجی فیوژن یک آرایه است که موقعیت پیکسلی (قابل تغییر به مختصات سه بعدی در دستگاه ذکر شده یا فاصله آن آبجکت با سپر بدون افزایش پیچیدگی محاسباتی و در O(1) و ارور محاسبه شده برای

تخمین عمق تصویر است که هر Index آرایه بیانگر رادار مربوطه است. جهت محاسبه این خروجی برای هر پیکسل هنگامی که مختصات آن محاسبه شد فاصله اقلیدسی اش با هر رادار متناظرش محاسبه شده و تعریف میشود $error = |D - relevantRadarData|$ و ξ بیانگر حداکثر خطای قابل پذیرش در فرایند فیوژن است که از قبل بر اساس ارزیابی های انجام شده بر روی خطای تخمین عمق مشخص میشود سپس برای هر پیکسل و هر رادار مرتبط با آن

اگر رابطه $error < \varepsilon \; and \; error < current\_relevant\_radar\_output\_error$ برقرار باشد عنصر مرتبط با آن رادار در آرایه را به اطلاعات این پیکسل بروزرسانی میکنیم که با توجه به آنکه این فرایند برای تمامی پیکسل های تمامی وسیله ها انجام میشود در نتیجه حتما نزدیکترین موقعیت به خروجی رادار پیدا می شود. همچنین این تصمیم طراحی که به جای انتخاب نزدیکترین موقعیت به رادار موقعیت با کمترین error را انتخاب میکنیم به این علت اتخاذ شده است که نسبت به خطای تخمین عمق Robust تر باشیم زیرا دقت رادار به مراتب بالاتر از دقت تخمین عمق تصویر است و به دلیل ماهیت local ارور های ایجاد شده در تخمین عمق این رویه خطاهای غیر منطقی تخمین عمق را کاملا مدیریت میکند و به افزایش ثبات عملکرد سیستم کمک میکند.

در ادامه به توضیح ریاضیات استفاده شده در تبدیلات می پردازیم.

ماژول محاسبه موقعیت مکانی پیکسل های دوربین

در این ماژول، هدف تبدیل مختصات یک نقطه پیکسلی از تصویر به مختصات سه‌بعدی آن در فضای دنیای واقعی است، با استفاده از اطلاعات کالیبراسیون دوربین، موقعیت و زاویه‌ی دوربین و مقدار عمق نقطه‌ی مربوطه.

تابع euler_angles_to_rotation_matrix: این تابع یک ماتریس چرخش سه‌درسه را بر اساس زاویه‌های اویلر (Roll، Pitch و Yaw) تولید می‌کند. این زاویه‌ها جهت‌گیری دوربین را نسبت به فضای دنیای واقعی مشخص می‌کنند.

ترتیب چرخش‌ها به صورت چرخش ذاتی است:

ابتدا چرخش حول محور$X(roll)$، سپس حول محور$Y(pitch)$ و در نهایت حول محور$Z(yaw)$.

ماتریس نهایی چرخش به‌صورت زیر محاسبه می‌شود:

$$Rx(roll).Ry(pitch).Rz(yaw) = R$$

تابع pixel_to_world: این تابع مختصات یک نقطه در تصویر (u, v) را به مختصات آن در فضای جهانی (World Frame) تبدیل می‌کند، به شرط آنکه مقدار عمق آن (Z) مشخص باشد. مراحل انجام کار به شرح زیر است:

1. ابتدا نقطه تصویری از اعوجاج اصلاح می‌شود (با استفاده از پارامترهای درونی دوربین و ضرایب اعوجاج توسط تابع undistortPoints).
2. مختصات نرمال‌شده‌ی نقطه با استفاده از ماتریس درونی دوربین به شرح زیر محاسبه می‌شود:

ماتریس دوربین K برای نگاشت یک نقطه سه‌بعدی در دستگاه دوربین به تصویر دوبعدی استفاده می‌شود. ساختار آن به صورت زیر است:

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

- f_x,f_y:فاصله کانونی دوربین به پیکسل (در راستای محور x و y) – معمولاً برابر نیستند.
- c_x,c_y:مختصات نقطه اصلی (Principal Point) تصویر – معمولاً نزدیک مرکز تصویر.
- این ماتریس رابطه بین مختصات سه‌بعدی در دستگاه دوربین و مختصات دو بعدی پیکسلی را برقرار می‌کند.

جهت نرمال سازی مختصات ها از فرمول های زیر استفاده شده است :

$\frac{V'-Cy}{Fy} = y$, $\frac{U'-Cx}{Fx} = x$

3. سپس این مختصات به مختصات دوربین (Camera Frame) ضربدر مقدار عمق تبدیل می‌گردد.

$$\mathbf{P}_{\text{cam}} = Z \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot Z \\ y \cdot Z \\ Z \end{bmatrix}$$

4. در نهایت با استفاده از ماتریس چرخش و بردار انتقال (موقعیت مکانی دوربین)، مختصات دوربین به مختصات دنیای واقعی تبدیل می‌شود.

ماتریس چرخش به کمک تابع euler_angles_to_rotation_matrix محاسبه شده و به کمک رابطه زیر خروجی نهایی مختصات خروجی محاسبه می‌شود.

$$\mathbf{P}_{\text{world}} = \mathbf{R}_{cw} \cdot \mathbf{P}_{\text{cam}} + \mathbf{t}_{cw}$$

ورودی‌ها:

- مختصات پیکسلی (u, v)

- مقدار عمق نقطه (Z)
- ماتریس درونی دوربین (Intrinsic Matrix)
- ضرایب اعوجاج لنز
- زاویه‌های اویلر دوربین (Roll, Pitch, Yaw)
- موقعیت مکانی دوربین در فضای جهانی (X, Y, Z)

ماژول محاسبه فواصل

DistanceCalculator : یک کلاس ایستا (static class) در زبان C++ است که برای محاسبه فاصله و مجذور فاصله بین دو نقطه سه‌بعدی کاربرد دارد. که به کمک رابطه ی $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$ محاسبه می شود.

- **تغییرات اعمال شده در سطح ابزارها، راه‌حل‌ها و (در صورت نیاز) محیط‌های توسعه**

برای ساده سازی تحلیل تصویر، آرایه راداری جلوی ماشین به صورت خطی تغییر داده شد و از حالت قوس خارج شد. رادار های اطراف ماشین به علت خارج بودن از دید دوربین لحاظ نشدند. تغییر دیگری در محیط توسعه و برنامه نویسی اعمال نشد.

7- آزمون، ارزیابی و مقایسه عملکرد

- **طرح آزمون**

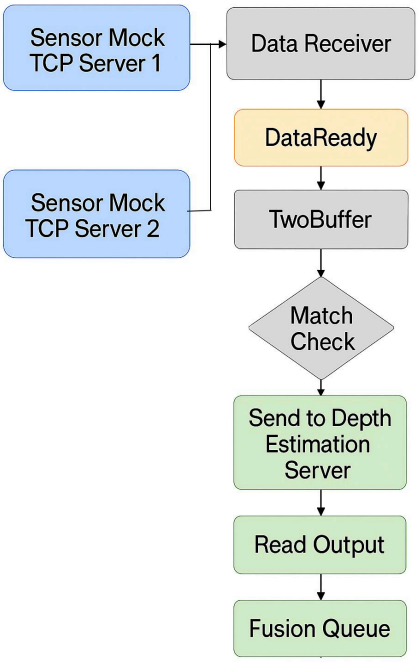
1. مراحل آزمون

a. شبیه‌سازی تاخیرهای ناگهانی و handshake
   i. استفاده از یک وب‌سرور مبتنی بر TCP connection برای مدل‌سازی ارسال داده‌ها.
   ii. این وب‌سرور رفتار handshake مشابه پروتکل‌های I2C و همچنین تاخیرهای تصادفی را پیاده‌سازی می‌کند.
   iii. نرخ ارسال داده‌ها در هر ثانیه ثابت نیست تا شرایط محیط فیزیکی واقعی (مثل نویز یا اختلال) شبیه‌سازی شود.

b. پیکربندی چند سرور محلی
   i. دو سرور محلی (Localhost) روی پورت‌های متفاوت اجرا می‌شوند.
   ii. هر سرور نقش یک سنسور متفاوت (مثل رادار و دوربین) را بر عهده دارد.

iii. سیستم ارسال داده (Data Sender) به صورت local داده‌ها را به endpoint هر سرور ارسال می‌کند.
iv. این طراحی باعث می‌شود بی‌نظمی‌های ارتباط فیزیکی و شبکه در ارسال داده‌ها شبیه‌سازی شوند.

c. 3. دریافت داده و ارسال سیگنال DataReady
i. در سمت دریافت‌کننده، هنگام دریافت داده mock از سرور، یک سیگنال با نام DataReady از داخل تابع دریافت داده emit می‌شود.
ii. این سیگنال به تابع اضافه‌کردن داده به ساختار TwoBuffer متصل است.
iii. TwoBuffer داده‌ها را برای ادامه پردازش ماژول‌های بعدی آماده می‌کند.



d. همگام‌سازی و مقایسه داده رادار و دوربین
i. وقتی داده‌های دریافت‌شده از رادار و دوربین سازگار باشند، یک سیگنال جدید emit می‌شود.
ii. این سیگنال، داده دوربین به همراه داده رادار را به ماژول پردازش عمق تصویر ارسال می‌کند.

e. 5. دریافت خروجی و ورود به ماژول فیوژن
i. خروجی تولید شده توسط ماژول پردازش عمق تصویر خوانده می‌شود.
ii. این داده‌ها وارد صف ورودی ماژول Fusion می‌شوند.
iii. به دلیل عدم دسترسی مستقیم به ماژول پردازش عمق تصویر، این بخش نیز به صورت وب‌سرور شبیه‌سازی شده تا در آینده قابلیت Integration واقعی داشته باشد.

2. طرح آزمون صحت

برای آزمون‌های صحت، در ابتدا نقاط حساس جریان ورودی را استخراج کردیم و سپس همان قسمت ها را مورد تست قرار دادیم. در اینجا معیار ما درستی خروجی مورد انتظار (expected) است که آن را با خروجی داده شده مقایسه می کنیم. در صورتی که این دو خروجی مطابقت داشته باشند تست PASS و در غیر این صورت تست FAIL می شود. برای هر کلاس یک کلاس تست جداگانه ساختیم و نقاط حساس آن ها را مورد تست و بررسی قرار دادیم. برای

ورودی‌های شبکه، تولیدکننده‌های داده (کلاینت‌های تست) دقیقا طبق پروتکل هر ماژول پیام‌سازی می‌کنند و ورودی اولیه و آبجکت های اولیه را می سازند.
طرح «آزمون صحت» در این پروژه برای هر ماژول، دقیقا روی ورودیِ تعریف‌شده، خروجیِ مورد انتظار و رفتارهای مرزی آن متمرکز است. در لایه‌ی ورودی‌های شبکه: برای **MockRadarData** پیام‌های JSON معتبر (با `values` و `timestamp`) و همچنین حالت‌های ناقص/نامعتبر ارسال می‌شود تا تطابق دقیق مقادیرِ سیگنال و نیز رفتار پیش‌فرض در نبود `timestamp` سنجیده شود. برای **MockImageReceiver،** فریم مطابق پروتکل دوتصویری ایجاد می‌شود تا «عدم انتشار در حالت ناقص» و «انتشار دقیق پس از تکمیل» صحت سنجی شود. برای **ErfanMocker**، ساختار پیکسل‌ها و `radarData` همراه با `timestamp` در سناریوهای معتبر و مرزی بررسی می‌شود (از جمله نبود `timestamp`، لیست پیکسلِ خالی، و طول‌های نامعتبر برای `radarData`) تا صحت پارس و بار معنایی سیگنال تثبیت گردد. **Buffer** با سه محور اصلی سنجیده می‌شود—«خواندن خالی» (اکسپشن)، «همگامی در بازه‌ی مجاز» (بازگشت زوج درست و `isSync()==true`) و «عدم همگامی فراتر از آستانه (حد مجاز) »، به‌اضافه‌ی تست مرزهای آستانه و چرخش حلقه‌ای هنگام سرریز. **DataContainer** صحت زوج‌سازیِ رادار/دوربین را در هر دو ترتیب ورود، در چند زوجِ پشت‌سرهم و (در صورت استفاده) زمان/تعداد انتشار سیگنال همگامی بررسی می‌کند. **Fuse** نیز با شمارش `OperationDone` در فراخوانی‌های پشت سر هم و سناریوهای ورودیِ غیرتهی/مرزی، از درست‌بودن مسیر عادی و نبود بن‌بست اطمینان می‌دهد. همه ی این تست ها به صورتی انجام می شوند که نتیجه ی تست (PASS/FAIL) در کنسول چاپ شود.

3. طرح آزمون حافظه و زمان

برای این آزمون ها، توابع مورد استفاده در طول پروژه استخراج شد. توابع مورد نیاز برای حافظه شامل init های هر کلاس و تابع performFusion با تمام توابع صدا زده شده داخل آنها بودند و برای زمان همین توابع به جز init ها استخراج شدند و همانطور که قبلا ذکر کردیم ما فرض را بر آن میگیریم که قبل از حرکت initialization تمام شده است و هرگز آبجکت ها دوباره ساخته نمیشوند.
برای این آزمون ها یک تمپلیت آماده کردیم و برای هر کلاس یک test suite نوشتیم. هر test suite شامل تعدادی تابع برای ایجاد یک آبجکت یا صدا زدن تابعی از آن آبجکت است و هر یک از توابع به طول اختصاصی یا فقط function call دارند یا فقط متغییر می سازند، در نتیجه برای صدا زدن یک فانکشن زمان اتلافی جز سربار صدا زدن آن تابع نداریم.

تمپلیت مورد بحث یک تابع است که زمان اجرا و حافظه مصرف شده را در محیط ویندوز یا لینوکس می سنجد و اطلاعات نهایی را بر میگرداند. هر test suite یک تابع run tests دارد که تمام تست های موجود را اجرا کرده و در نهایت رشته ای شامل اطلاعات به دست آمده را برمیگرداند. تست ها به صورت repeatable طراحی شده اند و با پارامتری می توانیم چندین بار یک تست را اجرا کنیم و نتایج را با هم مقایسه کنیم. از جایی که ما مصرف حافظه توسط ریسه را میسنجیم و حافظه ی مصرف شده بعد از تمام شدن اجرای تابع از بین نمیرود، مصرف حافظه فقط در بار اول صحیح است و بعد از آن صفر میشود. در نتیجه برای تست های حافظه چندین بار کل تست ها را اجرا کردیم اما تغییری در مصرف دیده نشد بنابراین به نتایج اطمینان داریم.

همچنین تحلیل های استاتیک با آنالیز کد نیز انجام شده اند که در بخش نتایج به آن پرداختیم.

4. **ابزار های پروفایلینگ**

به منظور بررسی دقیقتر برنامه، از ابزار ValGrind استفاده شده و نرم افزار از جهت ,call stack control flow و مموری بررسی شده است.

- **نحوه اجرای آزمون (پیاده سازی)**

1. **تست های صحت**

هر ماژول یک باینری تست چاپی (PASS/FAIL) دارد؛ ابتدا یک‌بار روی Windows و پردازنده Core i7 اجرا شد. ورودی‌ها دقیقاً مطابق پروتکل تولید می شوند. همچنین در نهایت در محیط آزمایشگاه بر روی یک core در raspberry pi 5 با سیستم عامل kubuntu اجرا شده اند.

2. **تست های حافظه و زمان**

تمام این تست ها یک بار در محیط ویندوز و بر روی یک پردازنده Core I5-11 اجرا شده اند (برای تست صحت و تقریب زمان نسبی پردازه ها نسبت به یکدیگر) و در نهایت در محیط آزمایشگاه بر روی یک core در raspberry pi 5 با سیستم عامل kubuntu اجرا شده اند. مشخصا ما نتایج دومی را گزارش کردیم.

همچنین یک تست doNothing برای تست زمانی فانکشن کال اضافه ی تحمیل شده توسط تست نوشته شد و زمان های به دست آمده را بدون این زمان اضافی حساب کردیم.

3. **پروفایلینگ**

نرم افزار در محیط لینوکس و با داده های mock شده پروفایل شد.

- **نتایج آزمون‌های انجام شده**

1. تست صحت

- Buffer Tests:

طرح آزمون:

اطمینان از اینکه خواندن از بافر خالی exception می‌دهد .

بررسی هم‌زمانی (sync) فریم رادار و دوربین وقتی اختلاف زمانی داخل تلورانس TIMESTAMP_MAX_DRIFT باشد.

اطمینان از اینکه اگر اختلاف زمانی از حد مجاز بیشتر شود، هم‌زمانی برقرار اعلام نمی‌شود.

گزارش تست:

← testEmptyReadThrows یک بافر خالی می سازد.و انتظار می رود read()، اکسپشن std::runtime_error را بدهد. در غیر اینصورت Fail می دهد.

← testAddAndReadSync رادار با timestamp=1000 و دوربین با timestamp=1000+ TIMESTAMP_MAX_DRIFT/2 اضافه می‌کند؛ انتظار می‌رود isSync()==true و read() همان زمان‌ها را برگرداند؛ در غیراینصورت Fail می‌دهد.

← testNotSyncBeyondDrift رادار با timestamp=2000 و دوربین با timestamp=2000+TIMESTAMP_MAX_DRIFT+1 اضافه می‌کند؛ انتظار می‌رود isSync()==false و هم‌زمانی برقرار اعلام نشود؛ در غیراینصورت Fail می‌دهد.

- DataContainer Tests:

طرح آزمون:

اطمینان از این‌که getData قبل از ورود هر داده خطا دهد/غیرممکن باشد.

بررسی هم‌زمانی در دو ترتیب «Radar→Camera» و «Camera→Radar» با اختلاف زمانی داخل تلورانس.

اطمینان از بازگشتِ چند زوج پشت‌سرهم بدون قاطی‌شدن.

گزارش تست:

← testGetDataBeforeAny: تا وقتی هیچ داده‌ای وارد نشده، getData() باید اکسپشن بدهد.

← testSyncAfterRadarThenCam: صحت هم‌زمان‌سازی وقتی ابتدا داده‌ی رادار و بعد داده‌ی دوربین با اختلاف زمانی «داخل تلورانس» می‌رسند.

← testSyncAfterCamThenRadar: همان آزمون قبلی ولی با ترتیب معکوسِ رسیدن داده‌ها (ابتدا دوربین، سپس رادار) تا ترتیب‌ناپذیری سیستم تأیید شود.

- Fuse Tests:

طرح آزمون:

اطمینان از این‌که فراخوانی dataRecieve منجر به انتشار OperationDone شود.

بررسی دو فراخوانی پشت‌سرهم و انتظار دو بار انتشار OperationDone.

اطمینان از عدم کرش با ورودی خالی/حداقلی.

گزارش تست:

← testOperationDoneOnce: یک بار `dataRecieve` را فراخوانی می‌کند و با لامبدا به سیگنال وصل می‌شود تا شمارش کند؛ انتظار: count==1. هدف: تضمین یک سیگنال برای هر پردازش

← testOperationDoneTwice: همان سناریو با دو فراخوانی؛ انتظار: count==2. هدف: نبودِ side-effectهای ناخواسته بین فراخوانی‌ها

- MockImageReceiver Tests:

طرح آزمون:

بررسی ارسال تکه‌تکه روی همان سوکت؛ قبل از تکمیل فریم نباید سیگنال منتشر شود.

اطمینان از این‌که بسته‌ی ناقص (فقط هدر/سایز) سیگنال منتشر نکند.

گزارش تست:

← testIgnoreIncompletePacket فقط هدر را می‌فرستد و انتظار می‌رود هیچ سیگنالی منتشر نشود؛ در غیر این صورت Fail می‌دهد.

- ErfanMocker Tests:

طرح آزمون:

اطمینان از پارس صحیح پیام معتبر و انتشار سیگنال با ابعاد درست و RadarData صحیح.

گزارش تست:

← test_singleValidMessage پیام معتبر ارسال می‌کند و انتظار می‌رود اندازه‌ی لیست‌ها، فیلدهای RadarData و timestamp دقیقاً مطابق ورودی باشند؛ در غیر این صورت Fail می‌دهد.

← testMissingTimestamp پیام بدون `timestamp` می‌فرستد؛ انتظار می‌رود `RadarData.timestamp==0`، `got==true` و مقادیر `a==10` و `f==60` باشند؛ در غیراینصورت Fail می‌دهد.

← testEmptyPixelsAllowed پیامی با `values:[{pixels:[]}]` می‌فرستد؛ انتظار می‌رود سیگنال دریافت شود، `gotValues.size()==1` و `gotValues[0].size()==0` باشد؛ در غیراینصورت Fail می‌دهد.

File Name	Test Name	Status
buffer_test	testEmptyReadThrows	PASS
buffer_test	testAddAndReadSync	PASS
buffer_test	testNotSyncBeyondDrift	PASS
datacontainer_test	testGetDataBeforeAny	PASS
datacontainer_test	testSyncAfterRadarThenCam	PASS
datacontainer_test	testSyncAfterCamThenRadar	PASS
erfanmocker_test	testSingleValidMessage	PASS
erfanmocker_test	testMissingTimestamp	PASS
erfanmocker_test	testEmptyPixelsAllowed	PASS
fuse_test	testOperationDoneOnce	PASS
fuse_test	testOperationDoneTwice	PASS
mockimagereciever_test	testIgnoreIncompletePacket	PASS
mockradardata_test	testSingleMessage	PASS
mockradardata_test	testMissingTimestamp	PASS

همانطور که مشاهده می شود، تمامی تست ها در این پروژه PASS شدند.

2. تست زمان

تست های زمان اندازه گیری شده به هیچ عنوان بدترین زمان ممکن را ارائه نمیکنند و تنها اندازه گیری هایی از دنیای واقعی هستند. تخمین زمان همانطور که در بخش تحلیل ذکر شد، در حدود مرتبه زمانی زیر میباشد:

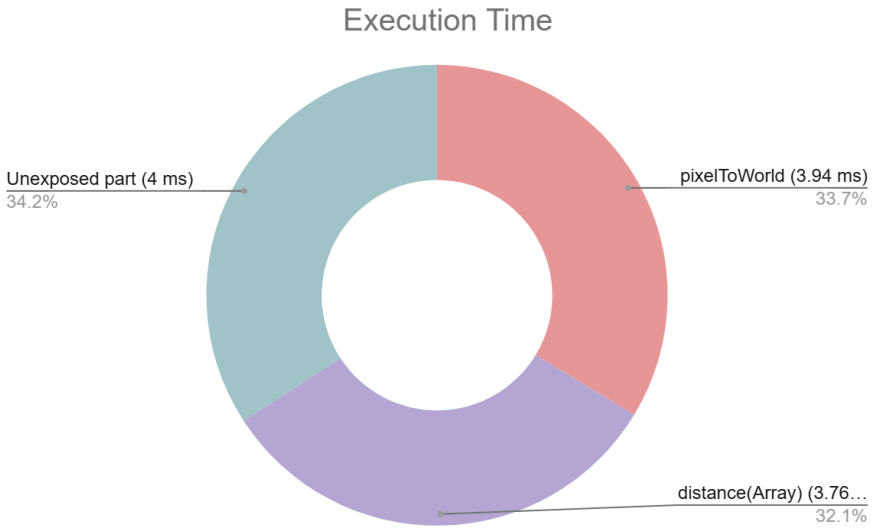
$$O(F) = O(Nimglen * (O(pixel2World) + 2Nradarlen + Nradarlen * O(logNfloatlen) + O(Nimglen)$$

زمان های اندازه گیری شده بر روی سخت افزار Raspberry Pi 5 و سیستم عامل Kubuntu به صورت زیر میباشد. تمامی محاسبات برای عکس 10000 پیکسلی اندازه گیری شده اند.

Function Name	Time(ns)	Time - doNothing(ms)
Fusion Setup	6433095	6.432082
performFusion	11704954	11.703941
distance(float)	10574	0.009561
distance(Array)	1389	0.000376
distanceSquared	1370	0.000357
calculatedDistance	1296	0.000283
pixel2World	1407	0.000394
pixel2World(static)	1297	0.000284
isValidDepth	1278	0.000265
updateRotationMatrix	1426	0.000413
eulerToRotationMatrix	1389	0.000376
Buffer Setup	1944365	1.943352
Data Container Setup	1471663	1.47065
isValid(Buffer)	1296	0.000283
getData(container)	1759	0.000746
doNothing	1,013	0
Total		11.703941

همانطور که مشاهده شد، زمان نهایی performFusion که با ناچیز در نظر گرفتن latency جابجایی اطلاعات تمام چیزیست که در طول پروژه اجرا میشود، حدود 12 میلی ثانیه طول میکشد. این زمان با

توجه به این که توابع pixelToWorld و distance به تعداد تعداد پیکسل های یکتای موجود در اشیا اجرا می‌شوند، طبق شکل زیر بین توابع مختلف پراکنده شده است:



زمان کل پروژه، همانطور که در تحلیل ها ارائه شد حاصل جمع حداکثر تاخیر سنسور ها، زمان اجرای فیوژن و زمان اجرای YOLO+DistanceEst است. برای زمان تاخیر سنسور ها، ماژول رادار VL531X در فرکانس 50 هرتز عمل می‌کند و فاصله 135 سانتی‌متر را در نور شدید محیط در حالت مسافت کوتاه اندازه‌گیری می‌کند و دوربین استریو IMX219-83 طبق مستندات رسمی در فرکانس 60 هرتز عمل می‌کند. با این حال، فاصله واقعی رادار در محیط آزمایشگاهی 400 سانتی‌متر اندازه‌گیری شده است. ما ترجیح می‌دهیم از این اندازه‌گیری ها به جای بدترین حالت استفاده کنیم، زیرا پروژه فقط در محیط آزمایشگاهی کنترل‌شده اجرا خواهد شد. بنابراین، طبق محاسبات توضیح داده شده در مبانی فنی پروژه و محاسبات پیوست فنی الف، حداکثر سن داده ها هنگام رسیدن به ماژول فیوژن 37 میلی ثانیه است. بنابراین اگر زمان تاخیر YOLO و DepthEst را $T_{DepthEst}$ در نظر بگیریم، زمان تاخیر کلی قبل از رسیدن به بخش تصمیم گیرنده بدون احتساب زمان مورد نیاز برای تبادل اطلاعات بین ماژول ها، حدود 49 + $T_{DepthEs}$ میلی ثانیه خواهد بود.

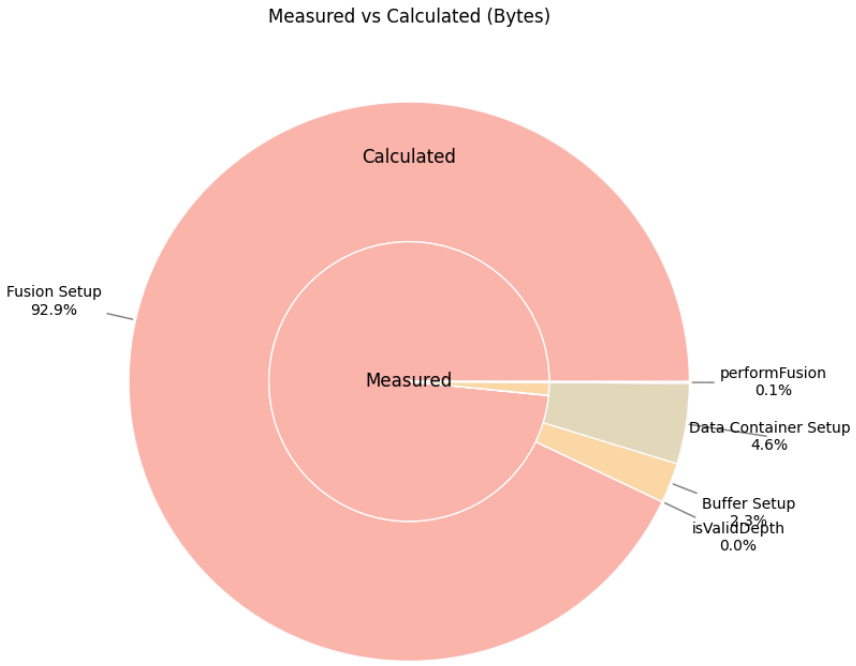
3. تست حافظه

نتایج تست حافظه در جدول زیر نشان داده شده است.

Name	Measured (Bytes)	Calculated (Bytes)	Variable Count
Fusion Setup	9,777,152	9,600,108	4 float(interinsics),

			16 float(radar), 6 float(pose), 1 float(eps), 2400000 float (objects)
isValidDepth	4,096	12	3 float
Buffer Setup	143,360	240,040	6 int(radars), 2 int64(timestamp), 40000 int + 20000 uint (images)
Data Container Setup	12,288	480,080	2 buffers
performFusion	0	10,028	12 float (output), 10000 bool (is_pixel_seen), 16 float (internal variables), 4 int (internal variables)
DO NOTHING	8,192	0	-
Total	9,932,800	10,090,228	Fusion Setup + Data Container + performFusion

در تست های حافظه مشاهده میکنیم که در بسیاری از موارد نتیجه ی اندازه گیری شده با نتایج تحلیل مغایرت دارد (گرچه جمع فرق تغییر چندانی ندارد). این مغایرت به دلایل بهینه سازی های نرم افزاری در کتابخانه ها و اشتباه اندازه گیری به علت وجود پروسه های دیگر و عدم اطمینان از تخصیص کل حافظه به پروسه ی ما است. در نهایت ما تحلیل استاتیک را اولویت می‌دهیم و نزدیکی ارقام اندازه گیری total به تحلیل را به عنوان سندی برای صحت تحلیل ارائه میکنیم. میزان حافظه استفاده شده توسط بخش های مختلف پروژه در نمودار زیر نشان داده شده است:

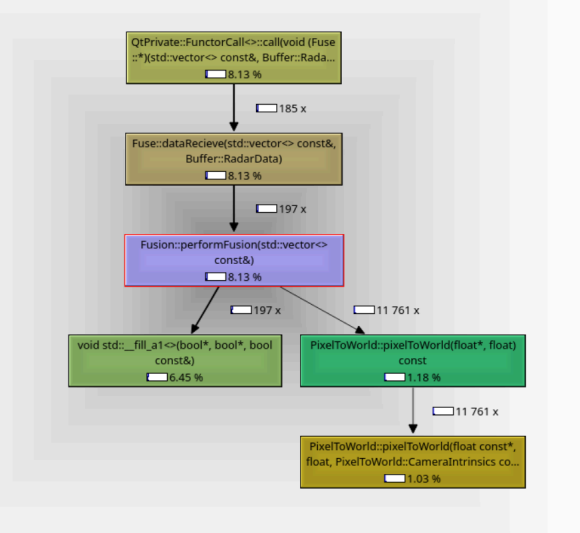


در حالت کلی، میزان حافظه استفاده شده توسط ماژول ها کمی بیشتر از 10MB بوده و بالای 90 درصد آن مربوط به نگهداری اشیا است.

4. پروفایلینگ

در طی پروفایل کردن پروژه، تمام پروژه به مدت محدود توسط داده های ماک اجرا شد و داده های لازم با استفاده از ابزار callGrind در valGrind پروفایل شدند. خروجی مورد نظر توسط kcachegrind ویژوالایز شد و آن را بررسی کردیم. نتایج برخی بررسی ها در اینجا آمده و نتایج کامل تر در پیوست فنی ب موجود است.

تصویر زیر تعداد صدا کردن فانکشن performFusion را نشان میدهد (درصد ها از کل هزینه سی پی یو برنامه حساب شده اند):



با بررسی این مورد می بینیم که تابع fill میزان زیادی زمان اجرا را در بر میگیرد. در محاسبات ما، این عدد قابل قبول است چون از میزان بسیار بیشتری از استفاده سی پیو توسط تحلیل پیکسل های متعدد عکس جلوگیری میکند، اما این عدد با تغییر اندازه عکس متغییر است و در صورت تغییر اندازه عکس توصیه میکنیم این tradeoff دوباره در نظر گرفته شود.

نمودار کل برنامه در پیوست ب قرار دارد که نشان میدهد اپلیکیشن تقریبا به صورت بالانس شده زمان قابل قبولی در هر مرحله سپری می‌کند و تعداد فانکشن کال ها به میزان خواسته شده است و فانکشن کال اضافی نداریم. در مورد QJson که حدود 40 درصد از زمان را اشغال کرده نیز باید گفت که این تنها در فاز ماک و برای ایجاد عکس ها استفاده شده و این سربار در اپلیکیشن اصلی وجود نخواهد داشت.

همچنینی control flow فرایند performFusion در پیوست ب وجود دارد که شامل کد های اسمبلی برنامه نیز هست. در صورتی که تحلیل بیشتر و دقیق تر این برنامه نیاز باشد، میتوان کد های مذکور را بررسی و تحلیل کرد.

- **تحلیل نتایج و مقایسه با نمونه‌های پیشفرض یا مشابه**
- **UTFusion:**

  - توانایی تشخیص مسیر و موانع در محیط تست
  - حفظ کنترل خودرو حتی در پیچ‌ها و تغییرات نور
  - محدودیت در سرعت پردازش هنگام پیچیدگی زیاد صحنه یا نور نامناسب
  - بهبود عملکرد با بهینه‌سازی کد و کاهش حجم پردازش تصویر

- **DeepPicar:**

  - فرکانس کنترل تا ~40Hz روی Pi 3 (در حالت ایده آل)
  - کاهش زمان اجرای حلقه کنترل با چند هسته (ولی مقیاس‌پذیری کامل حاصل نشد)
  - عملکرد حساس به تداخل پهنای‌باند حافظه
  - ایزولاسیون منابع با محدودسازی پهنای‌باند حافظه مؤثرتر از پارتیشن‌بندی کش

در پروژه UTFusion تمرکز بر پایداری کنترل در شرایط محیطی متنوع است، در حالی که DeepPicar روی تحلیل دقیق محدودیت‌های پردازشی متمرکز است.

-

ویژگی	UTFusion	DeepPicar
هدف پروژه	طراحی و ساخت خودروی خودران کوچک با قابلیت تشخیص مسیر و موانع در محیط واقعی	شبیه‌سازی سیستم NVIDIA DAVE-2 روی Raspberry Pi 3 برای بررسی کارایی CNN در کنترل زمان واقعی
سخت‌افزار اصلی	Raspberry Pi 5، موتورها، حسگر اولتراسونیک، دوربین	Raspberry Pi 3، دوربین USB ساده
ورودی‌ها و حسگرها	دوربین + حسگر اولتراسونیک	فقط ورودی تصویری

تمرکز نرم‌افزاری	پردازش تصویر + کنترل موتور + الگوریتم‌های تشخیص موانع	تحلیل کارایی CNN، چند هسته‌ای، تداخل منابع حافظه
عملکرد کلیدی	حرکت پایدار در مسیر، واکنش به موانع، مقاومت نسبی در برابر تغییر نور	کنترل 40Hz روی Pi 3، مقیاس‌پذیری محدود با چند هسته، حساس به پهنای‌باند حافظه
محدودیت‌ها	کندی در پردازش در صحنه‌های پیچیده یا نور کم	افت عملکرد شدید با وظایف پرمصرف حافظه
راهکارهای بهبود	بهینه‌سازی کد، استفاده از مدل‌های سبک‌تر CNN، بهبود داده آموزشی	محدودسازی پهنای‌باند حافظه، خنک‌کننده و منبع تغذیه قوی، داده آموزشی متنوع

شباهت‌ها: هر دو پروژه به اهمیت داده آموزشی و مدیریت منابع سخت‌افزاری تأکید دارند.

تفاوت‌ها: پروژه UTFusion راهکارهای مهندسی و بهینه‌سازی کد را بیشتر مورد توجه قرار داده، ولی DeepPicar بیشتر از منظر سیستم‌عامل و سخت‌افزار تحلیلی کار کرده است.

8- پیوست‌های فنی

پیوست الف) تحلیل زمانی پروژه (متن ترجمه شده)

پیش‌تر، ما پایپ لاینی را معرفی کردیم به نام UTFusion که به عنوان یک ماژول میانه بین داده‌های حسگر و ماژول تصمیم‌گیری در پروژه UTCar عمل می‌کرد. همچنین سیستم را به زیرماژول‌هایی تقسیم کردیم که در این سند به تفصیل توضیح داده خواهند شد. هدف کلی معماری این است که داده‌های دقیق و بلادرنگ را به ماژول تصمیم‌گیری ارائه دهد تا به موقع عمل کرده و به مهلت‌های مورد نیاز برای سیستم خودران قابل اعتماد برسد.

ملاحظات و محدودیت‌ها

هنگامی که در حال طراحی پایپ لاین بودیم، به داده‌های دنیای واقعی از دوره‌های حسگر و WCET ماژول‌های مختلف در خط لوله دسترسی نداشتیم زیرا مدل دقیق حسگرها نهایی نشده بود و ماژول‌ها به سادگی وجود نداشتند و بنابراین نمی‌توانستند تجزیه و تحلیل شوند. بنابراین تصمیم گرفتیم که رویکرد پارامتریک را اتخاذ کنیم و یک چارچوب طراحی کنیم که بتواند با انواع مختلف ماژول‌ها و حسگرها کار کند، به شرطی که دستورالعمل‌های زیر رعایت شوند:

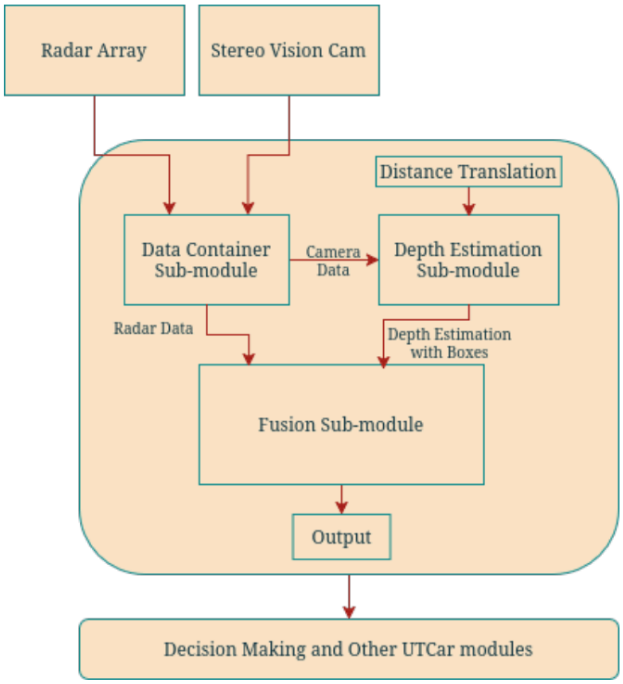
A. در محاسبات خود فرض کردیم که ماژول Fusion همیشه زمان اجرایش بیشتر از سیستم تحویل داده سنسور است. محاسبات ما ممکن است تحت هر فرضیه دیگری صحیح نباشد.

B. سیستم خود را برای سخت‌افزار غیر پیشگیرنده بدون کش طراحی کردیم. در صورت افزودن کش به سیستم، WCET به عنوان یک حد بالا عمل می‌کند اما محاسبات بیشتر ممکن است منجر به WCET کمتری شوند. در صورت استفاده از سیستم پیشگیرنده، هیچ حد بالایی یا پایینی برای تأخیر تضمین نمی‌کنیم.

C. ما مسئولیتی در قبال ماژول تخمین عمق یا هر ماژول دیگری که توسط گروه دیگری طراحی شده است نداریم، ما فقط محاسباتی را ارائه می‌دهیم که در آن صحت WCET داده‌شده توسط افرادی که ماژول را طراحی کرده‌اند درست فرض شده.

D. ما انتظار داریم که تمام اشیاء در جاده برای هر دو حسگر دوربین و رادار قابل مشاهده باشند و حداقل فاصله‌ای از یکدیگر داشته باشند تا به عنوان اشیاء مجزا در نظر گرفته شوند. این حداقل فاصله در فاز بعدی مورد بحث قرار خواهد گرفت.

E. ما انتظار داریم که داده‌های رادار دقیق‌تر از دوربین استریو باشند، که این حالت در بیشتر سنسور ها صادق است.



معماری کلی UTFusion

معماری UTFusion:

همان‌طور که در شکل 1 نشان داده شده است، معماری کلی UTFusion شامل چندین زیر ماژول است که عبارتند از: Distance Translation، Sensors، Fusion، Data Container و Depth Estimation. داده‌ها به‌طور دوره‌ای از رادار و دوربین استریو وارد می‌شوند و به‌طور مستقل در Data Container نوشته می‌شوند. بسته به زمان ورود داده‌ها، داده‌ها توسط زیرماژول‌ها یا از طریق Polling یا Signaling مصرف می‌شوند. زیر ماژول Fusion منتظر می‌ماند تا زیر ماژول Depth Estimation کار خود را تمام کند (برای اطلاعات دقیق‌تر به پروژه Depth Estimation using Stereo Cameras CPS مراجعه کنید) و سپس داده‌های آرایه رادار را با جعبه‌هایی که از زیر ماژول Depth Estimation به‌دست آمده است، ترکیب می‌کند. سپس خروجی در یک بافر ذخیره شده و فرایند به‌طور دوره‌ای اجرا می‌شود. لازم به ذکر است که بخش Sensor-Data Container سیستم و ماژول‌های Depth Estimation و Fusion می‌توانند و باید در حلقه‌های رویداد مختلف اجرا شوند و انتقال داده‌ها بدون قفل (lock-free) خواهد بود.

DataContainer

از دو بافر تشکیل شده است که هر کدام دو بخش دارند تا داده‌های مربوط به حسگرهای دوربین و رادار را ذخیره کنند. هر کدام از این بافرها بافرهای دایره‌ای با طول ثابت هستند و تمامی داده‌ها به محض ورود زمان‌دار می‌شوند. یک اشاره‌گر اضافی برای اشاره به یکی از بافرها استفاده می‌شود که نشان‌دهنده بافر قابل خواندن است. هر بار که یکی از حسگرها داده‌ای را تولید می‌کند که به دلیل تفاوت زمان با داده‌های جدیدترین نوشته شده توسط حسگر دیگر معتبر نیست، اشاره‌گر تغییر می‌کند و بنابراین ما اطمینان حاصل می‌کنیم که در هر زمان مشخص پس از اولین جفت داده معتبر، کانتینر می‌تواند داده‌های معتبر را به دیگر زیرماژول‌ها ارائه دهد. این معماری همچنین بخش بحرانی عملیات I/O را به تغییر اشاره‌گر محدود می‌کند.

Distance Translation

آرایه رادار و دوربین استریو هر دو در بخش جلویی وسیله نقلیه قرار دارند، اما حسگرها اختلاف کوچکی در ارتفاع و فاصله عمودی از اشیاء دارند. این امر منجر به خطای جزئی در تخمین فاصله شیء می‌شود و ممکن است تاثیر زیادی بر زاویه شناسایی شده شیء نسبت به جهت حرکت وسیله نقلیه داشته باشد. علاوه بر این، آرایه رادار به صورت یک خط مستقیم قرار نگرفته است، بلکه به صورت منحنی است که باید به تصویر دو بعدی ترجمه شود. انباشت این خطاها در فرایند ادغام همراه با خطای تخمین فاصله مورد انتظار از هر دو حسگر دوربین استریو و رادار می‌تواند منجر به نتایج فاجعه‌آمیز شود، چرا که بسیاری از مدل‌های یادگیری تقویتی (که کاندیداهای قدرتمندی برای تصمیم‌گیری در خودران‌ها هستند) به تغییرات کوچک در ورودی حساس هستند. بنابراین، ما یک زیر ماژول ترجمه فاصله را پیشنهاد می‌دهیم که یک تبدیل بر روی داده‌ها اعمال می‌کند و تمامی فاصله‌ها را از یک نقطه مرجع محاسبه شده تبدیل می‌کند. این translate از قبل محاسبه شده است و بنابراین بخشی ایستا از سیستم است.

تخمین عمق

ما زیر ماژول تخمین عمق را به عنوان یک جعبه سیاه تحلیل می‌کنیم (اگرچه دانش کافی از معماری کلی زیر ماژول برای برآورد زمان‌ها داریم). این ماژول داده‌های دوربین و ترجمه فاصله‌ها را به عنوان ورودی دریافت می‌کند و جعبه‌های محدود کننده اشیاء قابل مشاهده داخل تصویر را تولید می‌کند. پس از آن، فاصله هر جعبه محدود کننده از وسیله نقلیه تنها با استفاده از داده‌های دوربین استریو محاسبه شده و به ماژول ادغام به عنوان ورودی داده می‌شود.

ادغام

در بخش ادغام، ما جعبه‌های محدود کننده را دریافت می‌کنیم و تعیین می‌کنیم که کدام حسگرهای رادار از آرایه با هر پیکسل تصویر مطابق هستند بر اساس زاویه دید (FOV) حسگرها. نزدیک‌ترین شیء در هر زاویه دید با استفاده از داده‌های فاصله جعبه محدود کننده تخمین عمق انتخاب می‌شود و این

فاصله‌ها با استفاده از داده‌های رادار اصلاح شده و نزدیک‌ترین فاصله جعبه محدود کننده در هر زاویه دید با استفاده از ضریبی که از خروجی‌های رادار استخراج می‌شود مقیاس‌بندی می‌شود. خروجی این ماژول در یک بافر ذخیره شده و کل فرایند به طور دوره‌ای تکرار می‌شود.

زمان بدترین حالت اجرا (WCET)

زمان پاسخ استاندارد

استاندارد واحدی برای زمان پاسخ وسایل نقلیه خودران وجود ندارد و این زمان بسته به مدل تصمیم‌گیری و سناریوهای مختلف متغیر است. زمان پاسخ متوسط یک راننده انسانی بین 1.5 تا 2 ثانیه اندازه‌گیری می‌شود و بسیاری از وسایل نقلیه خودران کمتر از 100 میلی‌ثانیه پاسخ می‌دهند. از آنجا که مهلت مورد نیاز نامشخص است، ما زمان پاسخ و فاصله پاسخ بدترین حالت را محاسبه می‌کنیم و به جای تضمین یک زمان پاسخ مشخص، این مقادیر را در چارچوب محاسبه می‌کنیم. مهلت می‌تواند در یک مورد استفاده خاص پیکربندی شود. همچنین هدف ما این است که خروجی سیستم به طور کلی با نرخ 30 فریم در ثانیه (FPS) باشد، اما این محدودیت زمانی به شدت به ماژول تصمیم‌گیری و زیر ماژول تخمین عمق وابسته است.

زمان پاسخ UTFusion

ما بدترین حالت را به صورت زیر تعریف می‌کنیم:

- وسیله نقلیه باید با سرعت کامل Vmax حرکت کند.
- یک شیء که قبلاً شناسایی نشده است در فاصله D بر سر راه ظاهر می‌شود.
- تمامی ماژول‌ها در خط لوله بدترین زمان اجرا (WCET) خود را تجربه می‌کنند و ادغام نیز در WCET انجام می‌شود.

و انتظار داریم که خط لوله ما به موقع پاسخ دهد و وسیله نقلیه را قبل از برخورد متوقف کند. اگر شرایط برآورده نشود، پیشنهاد می‌کنیم که Vmax را کاهش دهیم تا زمان پاسخ بدترین حالت منجر به برخورد نشود.

طبق معماری نشان داده شده در شکل 1 و محاسبات قبلاً تعریف‌شده در پیشنهاد پروژه، بدترین حالت به شرح زیر محاسبه می‌شود:

$$MaxDelay = DM + DR + UTFusion + ST \div PipeLen = DM + DR + UTFusion$$

$$UTFusion = F + DE + BD$$

$$BD = 2A + I/O$$

$$A = A1 + A2$$

که در آن DM WCET ماژول تصمیم‌گیری است، DR تاخیر بین تصمیم‌گیری و واکنش است، ST زمان توقف وسیله نقلیه از Vmax است، F ماژول ادغام است، DE WCET تخمین عمق است، BD سن داده‌های بافر شده است و A1 و A2 حداکثر سن داده‌ها هنگام رسیدن از حسگر هستند. توضیحات دقیق‌تر در مورد دلیل محاسبه BD در پیشنهاد پروژه آمده است.

علاوه بر این، می‌توانیم حداکثر تفاوت مسافتی که وسیله نقلیه قبل از توقف طی می‌کند را محاسبه کنیم و عبارت زیر باید درست باشد:

$$PipeLen * V\,max \; + \; \frac{(V\,max)^2}{2\mu g} \leq D$$

که در آن g شتاب گرانشی و μ ضریب اصطکاک است.

در حالت ما، ماژول راداری VL531X با فرکانس 50 هرتز در حالت نور محیطی قوی و فاصله کوتاه 135 سانتی‌متر کار می‌کند و دوربین استریو IMX219-83 با فرکانس 60 هرتز طبق مستندات رسمی عمل می‌کند. با این حال، فاصله واقعی رادار در محیط آزمایشگاهی 400 سانتی‌متر اندازه‌گیری شد و ما ترجیح می‌دهیم از این اندازه‌گیری به جای سناریوی بدترین حالت استفاده کنیم زیرا پروژه فقط در محیط آزمایشگاهی کنترل شده اجرا خواهد شد. بنابراین، می‌توانیم D را با 400 سانتی‌متر جایگزین کنیم و A را با 37 میلی‌ثانیه. همچنین Vmax وسیله نقلیه UTCar را 5 متر بر ثانیه و ST را 50 سانتی‌متر اندازه‌گیری کردیم.

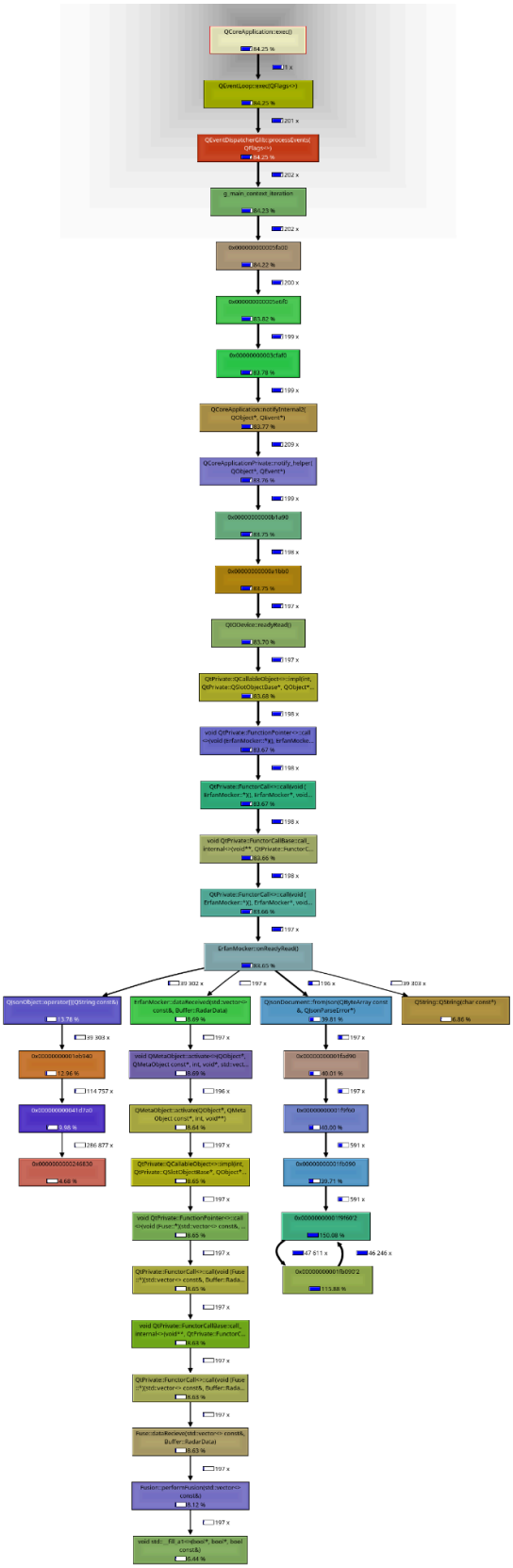
طبق داده‌های زیر، کل خط لوله باید یک چرخه کامل را در 150 میلی‌ثانیه تکمیل کند تا مهلت را رعایت کند و تضمین می‌شود که اشیاء را در فاصله 4 متر شناسایی می‌کند. در این 150 میلی‌ثانیه، 37 میلی‌ثانیه تأخیر داده‌ها است، بنابراین باید نابرابری زیر رعایت شود:

$$DM + DR + F + DE < 112\ ms$$

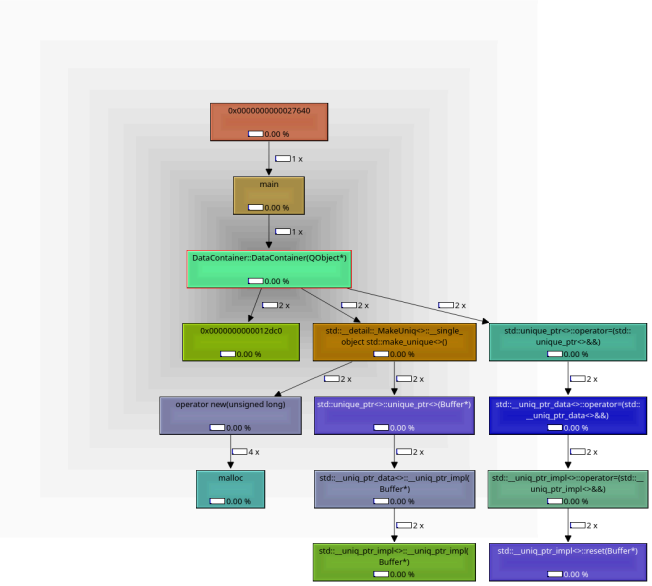
زمان I/O به دلیل معماری خاص واحد Data Container نسبت به سایر قسمت‌های سیستم بی‌اهمیت است و بنابراین نادیده گرفته می‌شود. ما مسئولیتی در قبال DM، DR یا DE نداریم، اما WCET(F) را در گزارش پروژه تخمین زدیم و بقیه محاسبات را به اعضای آزمایشگاه واگذار میکنیم.

پیوست ب) پروفایلینگ توسط ValGrind

1. نمودار QTCoreCallMap:

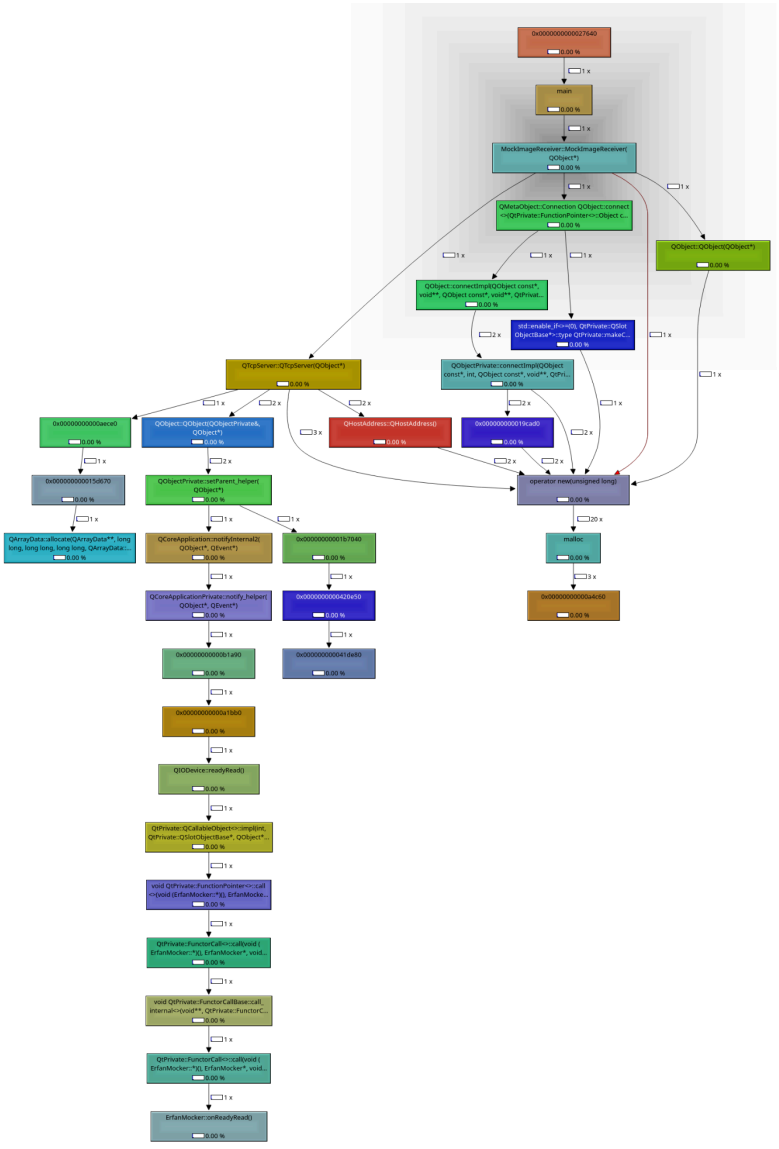


2. نمودار dataContainer:



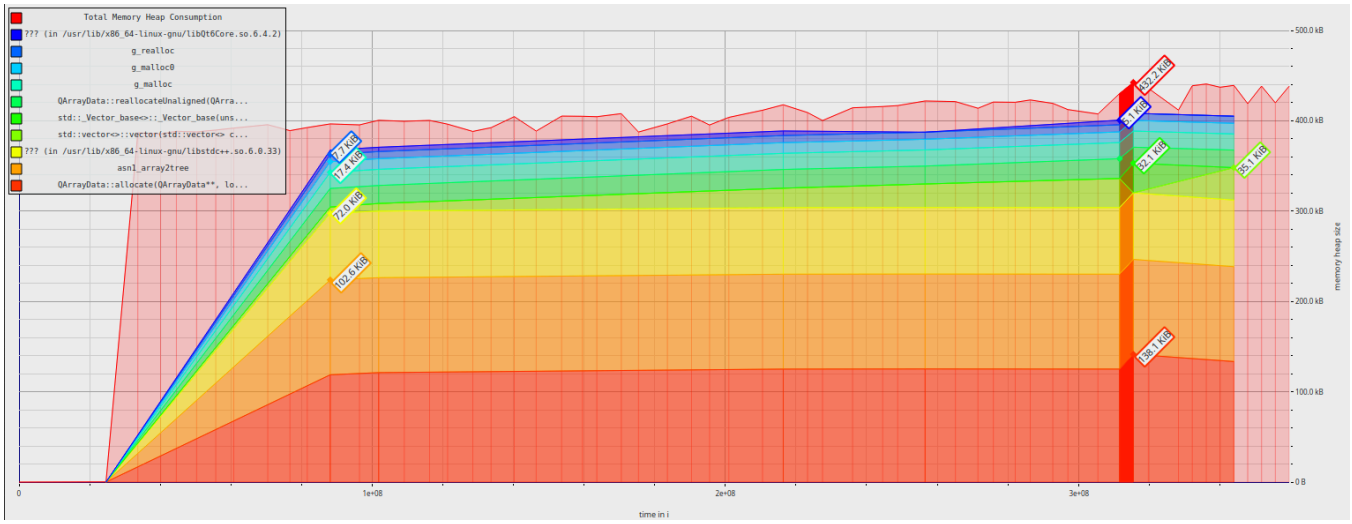
3. نمودار radarMock:

4. نمودار imageMock:



5. کنترل فلو performFusion:

6. میزان حافظه هیپ استفاده شده:



9- مراجع

۱. هارتلی، ریچارد و زیسرمن، اندرو. هندسه دید چندگانه در بینایی ماشین، ترجمه نشده. نسخه اصلی: Hartley, R., & Zisserman, A. (2003). Multiple View Geometry in Computer Vision. Cambridge" University ".Press

۲. سزلیسکی، ریچارد. بینایی ماشین: الگوریتم‌ها و کاربردها، ترجمه نشده. نسخه اصلی: "Szeliski, R. (2010). ".Computer Vision Algorithms and Applications. Springer

۳. مستندات رسمی کتابخانه OpenCV (نسخه ۴ به بعد), 2024. OpenCV Documentation.
https://docs.opencv.org/4.x/ به‌ویژه برای تابع cv2.undistortPoints و مدل اعوجاج.

۴. مستندات رسمی محیط QT6
https://doc.qt.io/qt-6/

۵. دیتاشیت سنسور رادار (نوری)
https://www.st.com/en/imaging-and-photonics-solutions/vl53l1x.html

۶. مستندات رسمی سنسور دوربین استریو
https://www.waveshare.com/wiki/IMX219-83_Stereo_Camera

۷. مستندات رسمی Raspberry Pi 5
https://www.raspberrypi.com/documentation/

۸. ریپازیتوری گیتهاب DeepPiCar
https://github.com/dctian/DeepPiCar

۹. مقاله DeepPiCar
https://arxiv.org/abs/1712.08644

۱۰. ابزار ValGrind
https://valgrind.org/