Cardano 92 Performance Regression Follow-up

IOG DevX

December 4, 2023

Contents

1 Executive Summary

- From inspecting the 92 regression the DevX team made several recommendations for code changes to `cardano-ledger-core`.
- This document tests two of these changes with `beacon` and `db-analyzer` on GHC-8.10.7, GHC-9.2, and GHC-9.6.2, and is only concerned with total execution time (wall time).
- Our findings are:
  - `beacon` and `db-analyzer` do observe changes in `cardano-core-ledger`.
  - This method does independently observe the GHC-9.2 regression.

  - Removing `FailT` has a negative impact on performance. It is not worth further testing.
  - Splitting `UMElem` to take better advantage of pointer tagging improves performance by 3% on GHC-9.6.2. Compared to the GHC-8.10.7 baseline, the GHC-9.6.2 `SplitUMElem` branch improves performance by ~5% for 70% of the data; for the very slowest slots it slightly regresses the baseline on GHC-9.6.2.
  - For the slowest slots in the dataset, GHC-9.6.2 outperforms GHC-8.10.7 with a 13% improvement.

2 Background

2.1 General Goal

- The larger goal is to be able to predict the performance of the ledger operations before shipping the code.
- `db-analyzer` and `beacon` are attempts to do this by replaying the state changes that took place. They replay the ledger state by using ledger operations to make queries on the ledger state.

2.2 `db-analyzer` and `beacon`

- db-analyzer is the tool that is doing the work
- beacon is just a convenient wrapper for data generation and plotting.

3 Methodology

- This data is generated by db-analyzer and beacon. Using a handcrafted chainDB from the P&T team. The handcrafted chainDB is constructed to be more dense (a `k` value of 3) than `mainnet` and thus should be a stress test of the ledger ops.
- The handcrafted chainDB has two blocks, which yields

  ```
  nrow(df810_baseline)
  ```

  122 observations per run.

  - Furthermore, each observation belongs to a specific slot id, which means we have statistically *paired* data. Paired data occurs when an observation is not independent between runs of an experiment. For example, if one was testing a diet pill on 10 mice, the dataset will have 10 observations before the diet pill, and 10 after, for a total of

20 observations. Each observation belongs to a unique mouse and the experiment is interested in how the pill changed each mouse individually, e.g., how did mouse 1 change. Thus comparing mouse 1 before the pill to mouse 4 after the pill is not a meaningful comparison. Similarly, comparing mouse 1 before the pill to mouse 3 before the pill is not meaningful.

We are dealing with a similar kind of data. Each observation belongs to a unique and not-independently sampled slot id. For example, slot id 63 might always be more performance intensive than slot 73, and so it would be incorrect to use statistical tests that assume a random sample. Thus we must use statistical tests that account for paired data.

- Next this data is not normally distributed:

```
shapiro.test(df810_baseline$totalTime)
```

```

Shapiro-Wilk normality test

data:  df810_baseline$totalTime
W = 0.42007, p-value < 0.00000000000000022
```

The baseline data for `totalTime` fails the normality test with a p-value of 2.2e−15 (this is the relevant bit: `p-value < 0.00000000000000022`) Even if we sample the data to exploit the central limit theorem we get a non-normal sample:

```
## Here I take 50 samples from the data set
shapiro.test(sample_n(df810_baseline, 50)$totalTime)
```

```

Shapiro-Wilk normality test

data:  sample_n(df810_baseline, 50)$totalTime
W = 0.53227, p-value = 0.00000000002173
```

Thus we must use a kruskall-wallace test to test if a change has a statistically significant effect and a pairwise-wilcox-test to determine which change had which effect and in what direction (slower or faster). These are the *non-parametric* versions of the regular t-test and ANOVA analyses.

4 Introduction

We'll be comparing three ghc versions: GHC-8.10.7, GHC-9.2, and GHC-9.6.2; across three branches: the baseline, split UMElem, and removing the FailT library.

4.1 The baseline

The baseline branch is set to ouroboros-consensus commit `e3917f684e8b60e7bfc453d6d8114b800bdf167d`, which is the release for `node-8.5`.

4.2 Split UMElem

The ledger uses a map data structure called `UMap` whose range is represented by a type called `UMElem` which looks like this:

```
-- So,
-- TEEEE means none of the components are present,
-- TFEEE means only the reward-deposit pair is present,
-- TEFEE means only the set of pointers is present,
-- TEEFE means only the stake pool id is present. etc.
-- TEEEF means only the voting delegatee id is present, and
--
-- The pattern 'UMElem' will correctly use the optimal constructor.
data UMElem c
  = TEEEE
  | TEEEF !(DRep c)
  | TEEFE !(KeyHash 'StakePool c)
  | TEEFF !(KeyHash 'StakePool c) !(DRep c)
  | TEFEE !(Set Ptr)
  | TEFEF !(Set Ptr) !(DRep c)
  | TEFFE !(Set Ptr) !(KeyHash 'StakePool c)
  | TEFFF !(Set Ptr) !(KeyHash 'StakePool c) !(DRep c)
  | TFEEE {-# UNPACK #-} !RDPair
  | TFEEF {-# UNPACK #-} !RDPair !(DRep c)
  | TFEFE {-# UNPACK #-} !RDPair !(KeyHash 'StakePool c)
  | TFEFF {-# UNPACK #-} !RDPair !(KeyHash 'StakePool c) !(DRep c)
  | TFFEE {-# UNPACK #-} !RDPair !(Set Ptr)
  | TFFEF {-# UNPACK #-} !RDPair !(Set Ptr) !(DRep c)
  | TFFFE {-# UNPACK #-} !RDPair !(Set Ptr) !(KeyHash 'StakePool c)
  | TFFFF {-# UNPACK #-} !RDPair !(Set Ptr) !(KeyHash 'StakePool c) !(DRep
  ↪  c)
  deriving (Eq, Ord, Generic, NoThunks, NFData)
```

Notice that this data type has 16 constructors. The idea behind this branch is to split this data type into two types each with 8 constructors. GHC uses three bits to tag pointers with `000` reserved to check for Thunks. Thus for a given data type GHC will check the pointer tag for the first 7 constructors; which is the case for the original version. By splitting `UMElem` into two data types GHC which check the pointer tag for the first 14 constructors, while constructor 15 and 16 will be scrutinized by looking up the constructor information in the heap objects' info-table. This should be much faster than the 16 constructor version, even at the cost of two extra machine words (one word each to differentiate between the two new data types). You can find the patch here.

4.3 Removing FailT

The idea behind this patch is remove the polymorphism in `Cardano.Ledger.Address`. This comes straight from the DevX analysis on the GHC-9.2 regression which found that a major difference on GHC-9.2 was a lack of specialization. `FailT` frequently showed up in that analysis and so removing it should pay off *if* the specialization was a contributing factor to the regression. This is especially the case because the code in `Cardano.Ledger.Address` uses a `NOINLINE` pragma for its `fail` function, which is known to prevent specialization. You can find the patch here.

5 Analysis

This analysis was done in R version:

```
R.version.string
```

```
[1] "R version 4.3.1 (2023-06-16)"
```

and is written in a literate programming style with inline R. All data was collected on a machine running:

```
neofetch --stdout --color_blocks off
```

```
doyougnu@7thChamber
-------------------
OS: NixOS 23.05.20231105.aeefe20 (Stoat) x86_64
Host: ASUSTeK COMPUTER INC. PRIME X470-PRO
Kernel: 6.5.9-xanmod1
Uptime: 14 days, 23 hours, 8 mins
Packages: 928 (nix-system), 2241 (nix-user), 8 (nix-default)
Shell: fish 3.6.1
Resolution: 1920x1080, 1080x1920
WM: xmonad
Theme: Breeze-Dark [GTK2/3]
Icons: breeze [GTK2/3]
Terminal: .emacs-29.1-wra
CPU: AMD Ryzen 7 2700X (16) @ 3.700GHz
GPU: NVIDIA GeForce GTX 1080 Ti
Memory: 7850MiB / 64218MiB
```

==

5.1 Loading and preparing the data

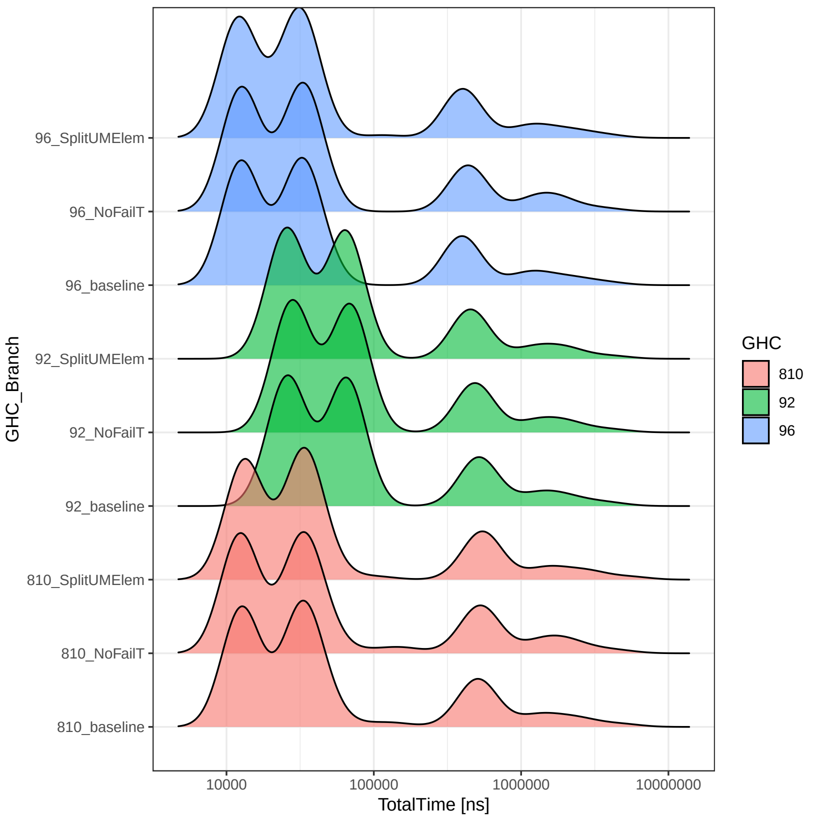
Feel free to skip this section if you are not interested in the R code.

```r
library("ggridges")
library("tidyverse")
library("rstatix")
library("tables")

options(scipen = 999)

data_dir <- "./data/"

load_data <- function(filename, ghc, branch) {
  read_tsv(paste(data_dir, filename, sep = "")) %>%
    mutate(GHC = as.factor(ghc), Branch = as.factor(branch))
}

## time units are nanoseconds
df810_baseline <- load_data("ledger-ops-cost-e3917f684e8b60e7bfc453d6d8114b800⌋
↪  bdf167d-haskell810-from-63-nr-blocks-100000.csv", 810,
↪  "baseline")
df92_baseline  <- load_data("ledger-ops-cost-e3917f684e8b60e7bfc453d6d8114b800⌋
↪  bdf167d-haskell-from-63-nr-blocks-100000.csv", 92,
↪  "baseline")
df96_baseline  <- load_data("ledger-ops-cost-e3917f684e8b60e7bfc453d6d8114b800⌋
↪  bdf167d-haskell96-from-63-nr-blocks-100000.csv", 96,
↪  "baseline")

df810Split_umelem <- load_data("ledger-ops-cost-a929cd7616668b61bea38486b1641d⌋
↪  5d45f13442-haskell810-from-63-nr-blocks-100000.csv", 810,
↪  "SplitUMElem")
df92Split_umelem  <- load_data("ledger-ops-cost-a929cd7616668b61bea38486b1641d⌋
↪  5d45f13442-haskell-from-63-nr-blocks-100000.csv", 92,
↪  "SplitUMElem")
df96Split_umelem  <- load_data("ledger-ops-cost-a929cd7616668b61bea38486b1641d⌋
↪  5d45f13442-haskell96-from-63-nr-blocks-100000.csv", 96,
↪  "SplitUMElem")

df810_noFailT <- load_data("ledger-ops-cost-6dc508fd5c0ddb73e4a5e01877dfcd698b⌋
↪  1c1bd0-haskell810-from-63-nr-blocks-100000.csv", 810,
↪  "NoFailT")
df92_noFailT  <- load_data("ledger-ops-cost-6dc508fd5c0ddb73e4a5e01877dfcd698b⌋
↪  1c1bd0-haskell-from-63-nr-blocks-100000.csv", 92,
↪  "NoFailT")
df96_noFailT  <- load_data("ledger-ops-cost-6dc508fd5c0ddb73e4a5e01877dfcd698b⌋
↪  1c1bd0-haskell96-from-63-nr-blocks-100000.csv", 96,
↪  "NoFailT")

df <- bind_rows(
  df810_baseline, df92_baseline, df96_baseline,
  df810Split_umelem, df92Split_umelem, df96Split_umelem,
  df810_noFailT, df92_noFailT, df96_noFailT
) %>%
  mutate(TestCase = paste(GHC, Branch, sep = "_")) %>%
  arrange(slot)
```

5.2 A first look at the data

Now we have our dataset, let's plot the distribution of `totalTime` for each ghc and branch. I'll use a ridgeline plot to observe changes in the distributions. Note that the x-axis is `log10` because we have an exponential distribution:

```r
p <- ggplot(df, aes(totalTime,
                    y = TestCase,
                    fill = GHC)) +
    geom_density_ridges(alpha = .6) +
    scale_x_log10() +
    xlab("TotalTime [ns]") +
    ylab("GHC_Branch") +
    theme_bw()
p
```



Each plot is a kernel density plot which shows the shape and relative position of the distribution of `totalTime` for each GHC and each branch. With this plot we are simply trying to visualize the distribution of the `totalTime` date. We see that the distributions all have three distinct clusters and are similar; the branches and GHC versions have not fundamentally changed the distribution of `totalTime` . GHC-9.2 shifts towards higher `totalTime` while GHC-9.6.2 looks similar to GHC-8.10.7. Differences between branches are too close to observe with the default density smoothing (the default smoothing is for univariate data which is the kind of data we are dealing with).

5.3 Are the versions significant

First let's check that there is a difference between GHC versions:

```r
kruskal.test(totalTime ~ GHC, data = df)
```

```

Kruskal-Wallis rank sum test

data:  totalTime by GHC
Kruskal-Wallis chi-squared = 70.109, df = 2, p-value =
0.0000000000000005969
```

We find a p-value of 5.9e−15 meaning that GHC version has a statistically meaningful impact on `totalTime`. Now to check if the branches have had a statistically meaningful impact while controlling for the GHC version:

- GHC-9.6.2

  ```r
      kruskal.test(totalTime ~ Branch, data = df %>% filter(GHC == 96))
  ```

  ```

  Kruskal-Wallis rank sum test

  data:  totalTime by Branch
  Kruskal-Wallis chi-squared = 12.293, df = 2, p-value = 0.00214
  ```

- GHC-9.2

  ```r
      kruskal.test(totalTime ~ Branch, data = df %>% filter(GHC == 92))
  ```

  ```

  Kruskal-Wallis rank sum test

  data:  totalTime by Branch
  Kruskal-Wallis chi-squared = 14.716, df = 2, p-value = 0.0006376
  ```

- GHC-8.10.7

```
    kruskal.test(totalTime ~ Branch, data = df %>% filter(GHC == 810))
```

```
    Kruskal-Wallis rank sum test

    data:  totalTime by Branch
    Kruskal-Wallis chi-squared = 7.9877, df = 2, p-value = 0.01843
```

For each version of GHC, we find p-values of less than 0.05 meaning that the branches have had a statistically significant impact on `totalTime`.

5.4 How are the branches significant

Now we'll use a pairwise wilcox to check which branches differ from the baseline. We'll just test with GHC-9.6.2 for now and return to the other GHC versions:

```
pairwise.wilcox.test(df$totalTime, filter(df,GHC == 96)$Branch,
↪   p.adjust.method = "holm", paired = TRUE)
```

```
Pairwise comparisons using Wilcoxon signed rank test with continuity correction

data:  df$totalTime and filter(df, GHC == 96)$Branch

            baseline            SplitUMElem
SplitUMElem 0.000000023         -
NoFailT     < 0.0000000000000002 < 0.0000000000000002

P value adjustment method: holm
```

The first column compares the branches `SplitUMElem` and `NoFailT` to the `baseline`, we find that both have a p-value less than 0.05 meaning that both branches are statistically different from the baseline for GHC-9.6.2. Now we'll compare the branches for each ghc version explicitly:

```
pairwise.wilcox.test(df$totalTime, filter(df,GHC == 92)$Branch,
↪   p.adjust.method = "holm", paired = TRUE)
```

```
          Pairwise comparisons using Wilcoxon signed rank test with continuity correction

      data:  df$totalTime and filter(df, GHC == 92)$Branch

                  baseline            SplitUMElem
      SplitUMElem 0.000000023          -
      NoFailT     < 0.0000000000000002 < 0.0000000000000002

      P value adjustment method: holm
```

```
pairwise.wilcox.test(df$totalTime, filter(df,GHC == 810)$Branch,
↪  p.adjust.method = "holm", paired = TRUE)
```

```
Pairwise comparisons using Wilcoxon signed rank test with continuity correction

data:  df$totalTime and filter(df, GHC == 810)$Branch

        baseline             SplitUMElem
SplitUMElem 0.000000023          -
NoFailT     < 0.0000000000000002 < 0.0000000000000002

P value adjustment method: holm
```

And we can see that both branches are meaningfully different from the baseline for all versions of GHC.

Now we'll see *how* they differ, we'll calculate the median `totalTime` and interquartile range by GHC version and branch to observe how each branch has impacted `totalTime` (note that we use the median because we have an exponential distribution, thus the mean would be heavily skewed by the extreme outliers in the dataset):

```
df %>%
group_by(GHC,Branch) %>%
select(totalTime) %>%
get_summary_stats(type = "median_iqr")
```

```
Adding missing grouping variables: `GHC`, `Branch`
# A tibble: 9 × 6
  GHC   Branch      variable      n median    iqr
  <fct> <fct>       <fct>     <dbl>  <dbl>  <dbl>
1 810   baseline    totalTime   122 32200. 37113
2 810   SplitUMElem totalTime   122 33083  31973.
3 810   NoFailT     totalTime   122 32521  71903.
4 92    baseline    totalTime   122 65250. 39085.
5 92    SplitUMElem totalTime   122 64412. 38234.
6 92    NoFailT     totalTime   122 68834. 41404.
7 96    baseline    totalTime   122 32088. 28964.
8 96    SplitUMElem totalTime   122 30942. 27022.
9 96    NoFailT     totalTime   122 32738  28118.
```

Let's begin with GHC-9.6.2; the last three rows. We can see that SplitUMElem median execution time is 30942 nanoseconds, compared to the baseline median of 32088, a difference of 1146 nanoseconds or 1 millisecond (an improvement of 3%). Similarly we can see that the inter-quartile range of SplitUMElem has reduced by 1942 nanoseconds or (2 ms). This means that the SplitUMElem distribution is tighter than the baseline and consequently the performance has become more precise. Let's check the distributions outside of the interquartile range to observe the best and worst performing slots:

```r
df %>%
group_by(GHC,Branch) %>%
reframe(enframe(quantile(totalTime, c(0.05,0.1,0.5,0.9,0.95)), "quantile",
↪  "totalTime")) %>%
pivot_wider(names_from = quantile, values_from = totalTime)
```

```
# A tibble: 9 × 7
  GHC   Branch        `5%`  `10%`  `50%`   `90%`    `95%`
  <fct> <fct>        <dbl>  <dbl>  <dbl>   <dbl>    <dbl>
1 810   baseline    12490. 12521. 32200. 516207  1288644.
2 810   SplitUMElem 12901. 12982. 33083  564792. 1485672.
3 810   NoFailT     12181. 12260. 32521  553308. 1617108.
4 92    baseline    24130. 24190. 65250. 532394. 1323829.
5 92    SplitUMElem 23914. 24007. 64412. 460273. 1149916.
6 92    NoFailT     26324. 26399. 68834. 497695. 1238604.
7 96    baseline    12225. 12261. 32088. 407903. 1122081.
8 96    SplitUMElem 11842. 11890. 30942. 414497  1131364.
9 96    NoFailT     12291. 12328. 32738  455974. 1440405.
```

In this table we have the 5th, 10th, 50th (median), 90th, and 95th percentile by GHC version and branch. There are several notable things:

- GHC-9.6.2 SplitUMElem is consistently better than baseline *until* the 90th percentile.
- NoFailT consistently grows more rapidly than baseline *except* on GHC-9.2. It's likely that the signal is obscured by something else an GHC-9.2 given that all data on GHC-9.2 shifts regardless of branch.
- baseline is consistently the best performing branch on GHC-8.10.7.
- The median values between GHC-8.10.7 and GHC-9.6.2 are basically identical (except SplitUMElem), but the top end of the distribution (i.e. the slowest slots): 90th percentile and above show a drastic improvement with GHC-9.6.2 compared to GHC-8.10.7. For example, the 95th percentile for baseline on GHC-9.6.2 is 1122081 compared to 1288644, an improvement of 13%.

The speedup at the upper tail of the distribution is interesting. Let's calculate the speedup of the distribution for each GHC version and branch and plot them:

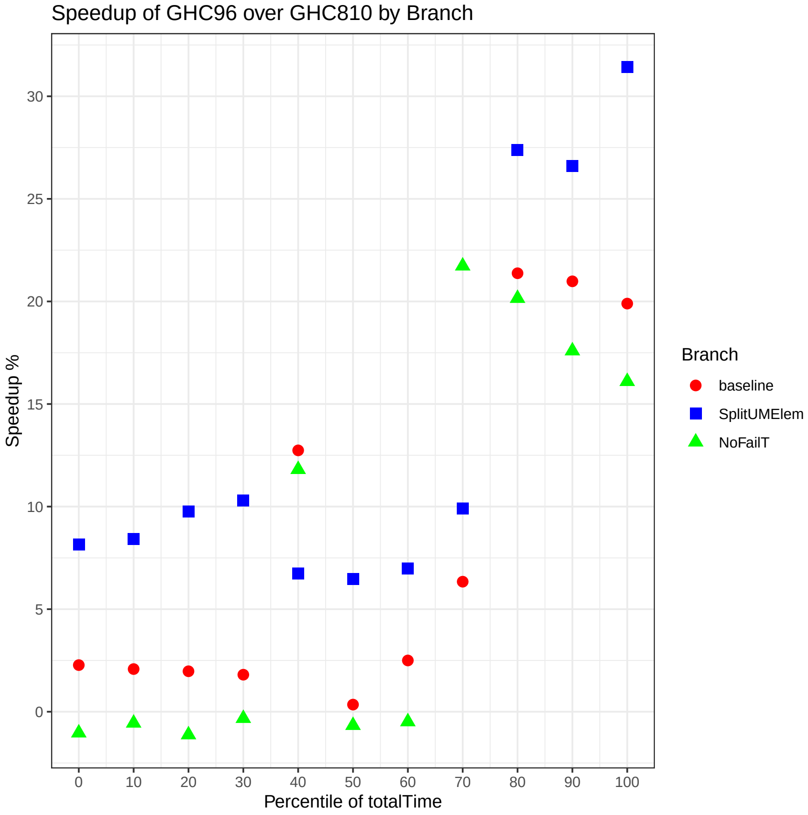
```
speedup_df <- df %>%
    group_by(GHC,Branch) %>%
    reframe(enframe(quantile(totalTime, seq(0,1,0.1)), "quantile",
    ↪  "totalTime")) %>%
    pivot_wider(names_from = GHC, values_from = totalTime, names_prefix =
    ↪  "GHC") %>%
    mutate(speedup96 = ((GHC810 - GHC96) / GHC810) * 100
         ,speedup92 = ((GHC810 - GHC92) / GHC810) * 100
         ,percentile = as.numeric(substr(quantile,1, nchar(quantile)-1)))

speedup_df
```

```
# A tibble: 33 × 8
   Branch   quantile  GHC810   GHC92   GHC96 speedup96 speedup92 percentile
   <fct>    <chr>      <dbl>   <dbl>   <dbl>     <dbl>     <dbl>      <dbl>
 1 baseline 0%         12359   23862   12078      2.27     -93.1          0
 2 baseline 10%        12521.  24190.  12261.     2.08     -93.2         10
 3 baseline 20%        12602.  24615.  12354.     1.97     -95.3         20
 4 baseline 30%        12832.  27776   12600.     1.80    -116.          30
 5 baseline 40%        31815.  46769.  27762.    12.7      -47.0         40
 6 baseline 50%        32200.  65250.  32088.     0.348   -103.          50
 7 baseline 60%        33190.  65774   32360.     2.50     -98.2         60
 8 baseline 70%        35470.  66135.  33221      6.34     -86.5         70
 9 baseline 80%       482283. 494415. 379200.    21.4       -2.52        80
10 baseline 90%       516207  532394. 407903.    21.0       -3.14        90
#  23 more rows
#  Use `print(n = ...)` to see more rows
```

and now to plot, we'll only focus on GHC-9.6.2 because GHC-9.2 clearly regresses:

```
p <- ggplot(speedup_df %>%
              select(!speedup92) %>%
              pivot_longer(cols = starts_with("speedup"),names_to =
              ↪  "comparison", values_to = "speedup")
          , aes(x = percentile, y = speedup, color = Branch, shape = Branch))
          ↪  +
  geom_point(size = 3) +
  scale_shape_manual(values=c(16,15,17)) +
  scale_color_manual(values=c("red", "blue", "green")) +
  scale_y_continuous(breaks = seq(0,35,5)) +
  scale_x_continuous(breaks = seq(0,100,10)) +
  ylab("Speedup %") +
  xlab("Percentile of totalTime") +
  ggtitle("Speedup of GHC96 over GHC810 by Branch") +
  theme_bw()
```

```
p
```



This plot shows the speedup of GHC-9.6.2 compared to GHC-8.10.7 for all branches at each 10th percentile of the `totalTime` distribution. For example, at the median (50th percentile) we see `baseline` with a speedup of 0% while `SplitUMElem` shows a speedup of 7% at the median. This means that at the median of the `totalTime` distribution the `baseline` did not improve *on GHC-9.6.2* while `SplitUMElem` did by 7%. Note that a negative value indicates a slowdown. We see that each branch, even `baseline` experience a speedup of GHC-9.6.2 over GHC-8.10.7.

The takeaway from this plot is that the upper tail of the distribution, that is, the slowest slots in the dataset, experience the largest improvement on GHC-9.6.2 over GHC-8.10.7. Furthermore `SplitUMElem` is particularly sensitive showing an improvement of 27% at the 80th percentile and 7-10% improvement for the rest of the distribution (compared to 0-2% improvement for the `baseline`). This implies that GHC-9.6.2 better optimizes the `SplitUMElem` branch.

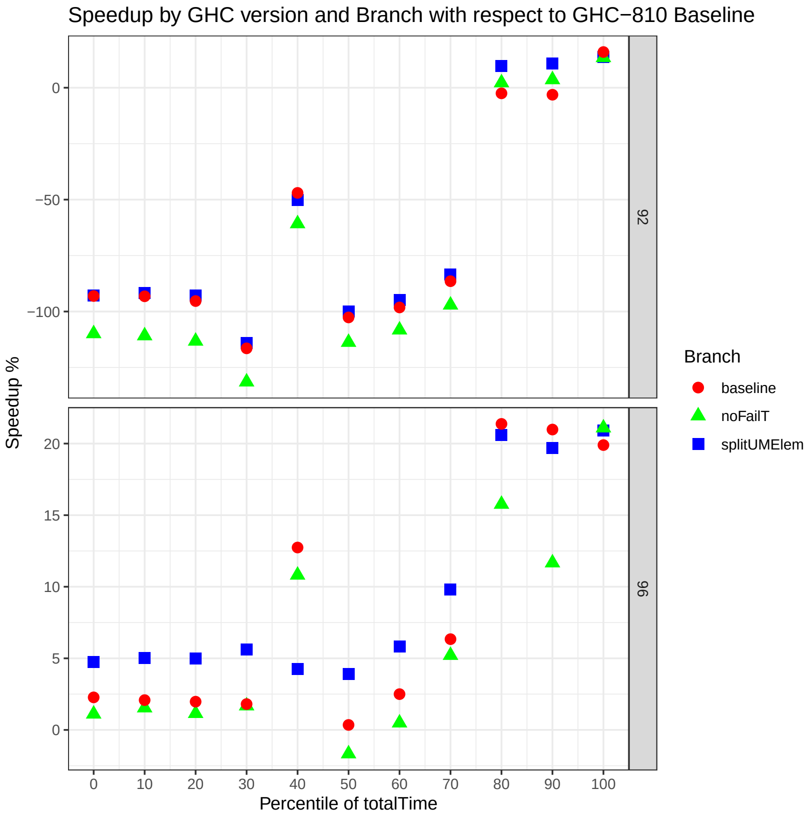
To wrap up, we'll create the same speedup plot but instead of showing the speedup of each branch on GHC-9.6.2 compared to GHC-8.10.7, we'll compare each branch on GHC-9.6.2 to only the baseline of GHC-8.10.7:

```
calc_speedup <- function(baseline,branch) {
  ((baseline - branch) / baseline) * 100
}

speedup_baseline <- df %>%
  group_by(GHC,Branch) %>%
  reframe(enframe(quantile(totalTime, seq(0,1,0.1)), "quantile", "totalTime"))
  ↪  %>%
  pivot_wider(names_from = c(Branch,GHC), values_from = totalTime, names_sep =
  ↪  "") %>%
  mutate(baseline_92    = calc_speedup(baseline810,baseline92)
       , baseline_96    = calc_speedup(baseline810,baseline96)
       , splitUMElem_92 = calc_speedup(baseline810,SplitUMElem92)
       , splitUMElem_96 = calc_speedup(baseline810,SplitUMElem96)
       , noFailT_92     = calc_speedup(baseline810,NoFailT92)
       , noFailT_96     = calc_speedup(baseline810,NoFailT96)
       , percentile = as.numeric(substr(quantile,1, nchar(quantile)-1))
       ) %>%
  select(percentile,contains("_")) %>%
  pivot_longer(cols = contains("_"), names_to = c("Branch", "GHC"), values_to
  ↪  = "speedup", names_sep="_") %>%
  mutate(Branch = as.factor(Branch)
         , GHC  = as.factor(GHC))

speedup_baseline
```

```
# A tibble: 66 × 4
   percentile Branch      GHC   speedup
        <dbl> <fct>       <fct>   <dbl>
 1          0 baseline    92     -93.1
 2          0 baseline    96       2.27
 3          0 splitUMElem 92     -92.8
 4          0 splitUMElem 96       4.76
 5          0 noFailT     92    -110.
 6          0 noFailT     96       1.11
 7         10 baseline    92     -93.2
 8         10 baseline    96       2.08
 9         10 splitUMElem 92     -91.7
10         10 splitUMElem 96       5.04
#  56 more rows
#  Use `print(n = ...)` to see more rows
```

and now the plot:

```
p <- speedup_baseline %>%
              pivot_longer(cols = contains("Speedup"),names_to = "comparison",
              ↪  values_to = "speedup") %>%
              arrange(desc(Branch)) %>%
  ggplot(aes(x = percentile, y = speedup, color = Branch, shape = Branch)) +
  geom_point(size = 3) +
  facet_grid(GHC ~ ., scales = "free_y") +
  scale_shape_manual(values=c(16,17,15)) +
  scale_color_manual(values=c("red", "green", "blue")) +
  scale_x_continuous(breaks = seq(0,100,10)) +
  ylab("Speedup %") +
  xlab("Percentile of totalTime") +
  ggtitle("Speedup by GHC version and Branch with respect to GHC-810
  ↪  Baseline") +
  theme_bw()

p
```



This is a faceted plot, the top subplot shows the speedup relative to the baseline of GHC-8.10.7 for GHC-9.2, notice that the y-axis is negative, i.e., GHC-9.2 regresses. The bottom subplot shows the same speedup for GHC-9.6.2. We see that `SplitUMElem` consistently shows more speedup over the baseline of GHC-8.10.7 except at the 40th percentile and above the 80th percentile where it matches the GHC-9.6.2 baseline. Note the subtle difference in this plot versus the last plot. In this plot we compare SplitUMElem on GHC-9.6.2 to the GHC-8.10.7 baseline, whereas in the last plot we compared SplitUMElem on GHC-9.6.2 against SplitUMElem on GHC-8.10.7. Thus we have two conclusions: first SplitUMElem experiences a larger speedup from GHC-9.6.2 than other branches; and second, that SplitUMElem performs better than both the GHC-9.6.2 and GHC-8.10.7 baseline until top 20 percent of the totalTime distribution.

Therefore, whether to use SplitUMElem or not is a tradeoff: gain a 5% performance bump for the majority of slots in the sample at the cost of a slight regression for the absolutely slowest slots.