CS4202 P01 - Branch Prediction

150015673

17 October 2018

Contents

1 Task 3

2 Programs Chosen 3

3 Program Input 3

4 Data Generation 3

5 Implementation 4

5.1 Branch Predictors 4

5.2 Data Generator 4

5.3 Plotting Tool 4

6 Results and Analysis 5

7 Conclusion 6

8 Given More Time 6

References 6

A Plots (standard error) 8

A.1 CS2002_P03 8

A.2 ffmpeg 10

A.3 jpegtran 12

B Plots (standard deviation) 14

B.1 CS2002_P03 14

B.2 ffmpeg 16

B.3 jpegtran 18

1 Task

The aim of this practical was to examine various branch-prediction methods and evaluate their performance. This had to be done using Intel's Pin tool[1] to generate branchtraces for various programs and compare these with the branch-prediction methods.

2 Programs Chosen

I chose to use the programs `ffmpeg`, `jpegtran`, and my personal sudoku solver written as part of the CS2002 coursework. I chose `ffmpeg` and `jpegtran` because they are programs which are not 'toy programs', i.e. they do work and are used, they do not simply generate branches. I chose my sudoku solver for a similar reason, the main difference being that it is a naïve solver, i.e. it simply tries all possible numbers. As this is done through a triply-nested for-loop, this will generate a pattern of branches which should be interesting for seeing how the branch-predictors perform. In order to keep this, I modified the `Makefile` to use level-0 compiler-optimisation instead of the original level-3 compiler-optimisation.

3 Program Input

For the sudoku-solver, I chose the first 5^{th}-order (i.e. 5 5×5 grids) sudokus from the given ones for `stacscheck`. These increase in difficulty and hence increase the amount of branching in the solver.

For `ffmpeg`, I chose 10 small `.wav`-files. I then started the `ffmpeg` transcoding to `.flac` to have `ffmpeg` do a conversion, i.e. do work.

For `jpegtran`, I chose 10 pictures of varying size, resolution, and black/white vs. colour. I did this to ensure that most possibilities in terms of image input were covered.

4 Data Generation

I initially tried running Pin until the program I was tracing terminated. However, even with fairly small inputs (couple of second long files for `ffmpeg`, small pictures for `jpegtran`, sudokus with few numbers missing) the branchtraces quickly reached gigabytes in size which was impossible to process in time. Files restricted to 100-150 megabytes still took too long to process, especially when wanting to have multiple runs of the same file/input. Therefore, I ended up with the method in the `gen_input.sh` script: it runs the files in a given program-input folder 10 times through a hard-coded program, lets the program run for 10 seconds (which generates a brachtrace of around 100MB), and then takes a slice of the branchtrace. The slicing cuts the first and last n-thousand lines and then keeps up to the first 500,000 lines of the remainder. I chose 100,000 as the number to remove, in order to remove potential setup-stages of the program. The reason the script then keeps 500,000 is that this was a size which seemed to have a decent slice of the program, whilst still being processable in reasonable time by the branch-predictors.

I ran each input 10 times to reduce the variation in accuracy. Ideally, the same input should not change the branchtrace at all, apart from maybe changing the specific addresses. However, I found that it occasionally did change the accuracy slightly (by around 0.05-0.50%) and as such chose to run each input 10 times just to be sure I had a good average.

[1](*Pin - A Dynamic Binary Instrumentation Tool*, 2018)

5 Implementation

I chose to implement my programs in Python due to its ease of use in terms of handling files and lists.

5.1 Branch Predictors

I implemented the branch predictors such that they could be run as individual programs as well as as part of a general program. For easy data-collection, each predictor can write its results, in csv-format, to a file. If it does not write to a file, the result is pretty-printed to stdout. The accuracy of a predictor is defined as the number of correct predictions compared with the total number of branches. It is output in percent. Because I chose to do multiple runs of each input and store these as separate files, each predictor stores which 'set' the data belongs to, i.e. which input it was part of. There are 10 sets in my data as I had 10 inputs for each program.

I wrote two profiled approaches. The first one records whether a branch is most often taken or not taken. If a branch is most often taken, it will predict always-taken when seeing it in the "real" run and vice-versa if it is most often not-taken. The second profiled approach records the frequency of a branch being taken vs. the number of times the branch is encountered. It then uses this information to predict if that branch will be taken in the real run, predicting 'not-taken' or 'taken' with respect to the recorded probabilities. Similar to the two-bit predictor, I chose to limit the number of branches the profiled approaches could store to a table-size. Since branches are most often taken, the profiled approaches will predict always-taken if the branch is not in the table.

My programs use four table sizes (determined as powers of 2): $512, 1024, 2048$, and 4096. My implementation of *gshare* uses a global history of 8 bits, as described in the original paper[2]. In addition to this, I chose to keep *gshare*'s table-size at 256, i.e. the maximum number of entries indexable by 8 bits.

5.2 Data Generator

The data generator runs each branchtrace through all 5 predictors and records the results in a CSV-file. For the predictors that take a table size, every table size is tried and recorded. To speed up the processing of data, and since the branch-predictors do not modify the trace file, the generator can run the predictors in parallel, through N processes (where N can be specified by the user). By using processes instead of threads, the memory-spaces are different, meaning that they will not share the same branchtrace file representation in memory. The only limitation on the parallelism is that for output-sanity, only one file can write to the output file at a time.

5.3 Plotting Tool

The data is manipulated and plotted through the pandas[3] and matplotlib[4] libraries. By indexing the data by predictor and then by data-/input-set, the mean and standard deviation can be calculated on a per-set and per-predictor basis. The per-set data is interesting to see how the programs vary with the same input, and the per-predictor data is interesting to see how each predictor does. For error bars, I chose to support plotting both standard deviation and standard error.

[2](McFarling, 1993)
[3](McKinney, 2010)
[4](Hunter, 2007)

6 Results and Analysis

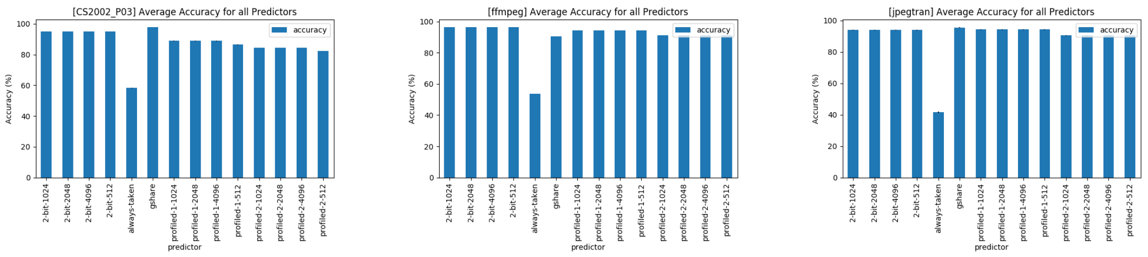


Figure 1: The average accuracy for all predictors across all three programs (with standard error).

As can be seen in the graphs above, always-taken is not very good, achieving a maximum accuracy of around 50%. Always-taken was expected to not perform well, however I was still surprised by how low its accuracy was. That its best average of just under 60% is in the sudoku-solver makes sense due to the large number of repeated loops in the program. Even so, I would have expected that programs took a branch more often in general.

When looking at the set-wise plots, we can see that the sample size of 10 achieves a negligible standard error for most of the predictors apart from *gshare*. The fact that *gshare* has a bigger standard error in the 'professional' programs in each set indicates that the line number could have been higher for more accurate results. However, the standard error is still within ±1% in the worst case scenario (jpegtran, set 6)[5], meaning that our mean is close to the true mean.

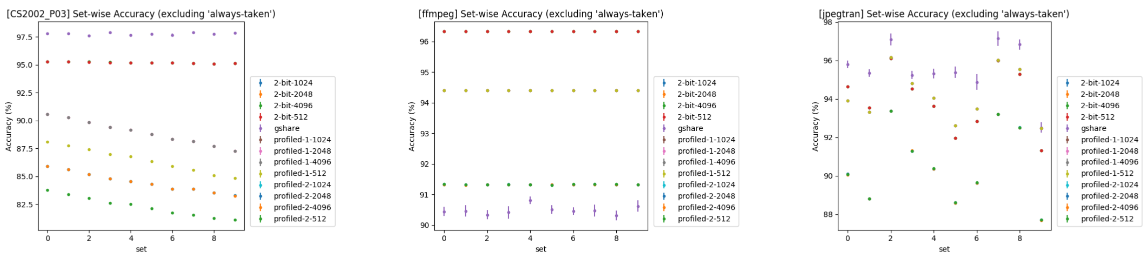


Figure 2: The set-wise accuracy for the predictors across all three programs, excluding always-taken to increase the resolution (with standard error).

The first and last of the three figures are particularly interesting. The first figure shows a decline in the profiled approaches. When comparing this with the figure including *always-taken*[6], we see that *always-taken* actually climbs in performance. This indicates that the increasing difficulty of the sudokus increases the number of branches taken. This makes sense, as the solver would have to iterate more, the more difficult the sudokus get. The profiled approaches' declining accuracy is somewhat odd, especially given that *2-bit* and *gshare* keep up their performance. For the second profiled approach, it could be that the possibility of predicting 'not-taken' works to its disadvantage

[5] B.3
[6] A.1

and it ends up predicting 'not-taken' too often. For the first profiled approach, it could simply be, that counting ends up mispredicting too often, i.e. the program is too varied.

In the last of three figures above, the results are vastly varied from set to set compared to the other two. Within each set, the difference between the various approaches is mostly similar, but apart from that, the results vary a lot. This could be indicative that `jpegtran` behaves significantly differently for different inputs. Set 2 was a screenshot with a lot of white and text, set 7 a picture of the moon with a black sky and stars, and set 8 a picture of some water. That set 2 was as intensive as set 7 and 8 is interesting because the latter were 4K-resolution, whereas the screeshot was 1080p. As set 5 was a 4K scenery-picture of a forest and some mountains, I have no explanation for why the variation is so great. One theory could be that the three pictures with higher prediction accuracy had an encoding further from the one `jpegtran` was trying to transcode to and therefore caused more iterations of something in the program. However, in order to fully understand this, a deeper understanding of `jpegtran` would be necessary.

7 Conclusion

Both *2-bit* and *gshare* perform consistently well. This was to be expected, as they are both real-world predictors compared to the others, which have presumably not been used in a real chip. Both *2-bit* and *gshare* consistently having above 90% accuracy shows why they are used: the speedups that can be gained from being right over 90% of the time are significant enough that you would want to have them.

I have also shown that having seen the program before does not necessarily mean that the prediction can be accurate (at least with limited table-size). Both my profiled approaches performed worse than *gshare* and *2-bit*, and would be more complicated to implement in hardware (with need for more storage, number calculation, generating random numbers, etc.).

Finally, I have appreciated the use of a tool like Pin. Without it, I would have had to write actual branch predictors instead of simulators, and this would have been much more complicated. Having access to a tool like Pin and using it to generate input for simulations, allowed me to much easier write and test new branch-prediction methods.

8 Given More Time

Given more time, I would have liked to include more programs in my tests in order to improve the validity of my conclusion. I would also have liked to look further into `jpegtran`. Finally, I would have liked to examine edge-cases of branch prediction, potentially examining programs which are difficult to branch-predict for.

References

Hunter, J. D. (2007, May). Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, *9*(3), 90-95. doi: 10.1109/MCSE.2007.55

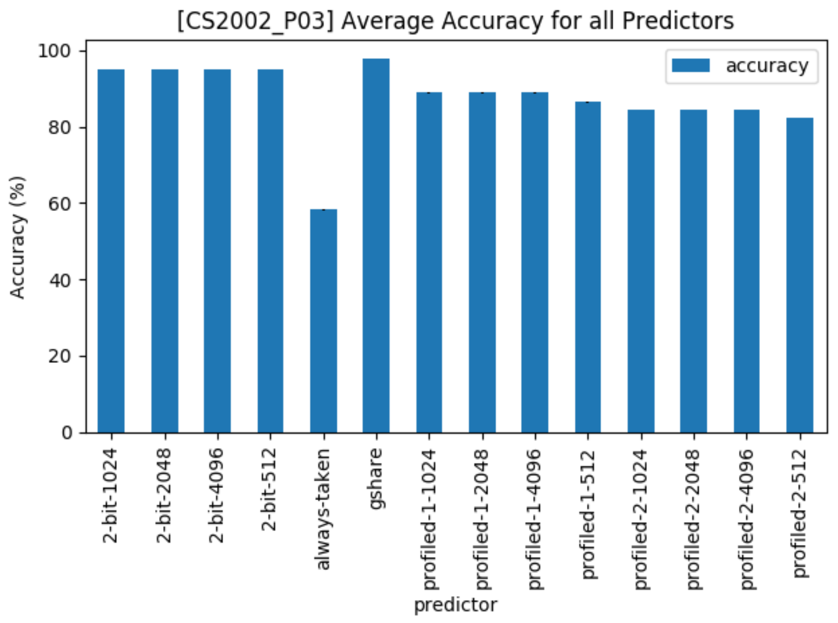
McFarling, S. (1993). *Combining branch predictors.*

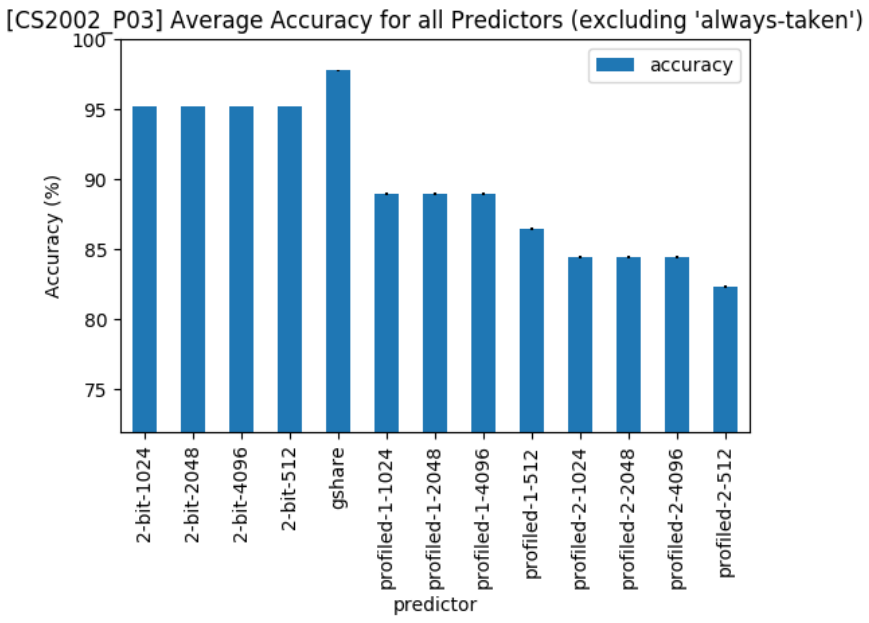
McKinney, W. (2010). Data structures for statistical computing in python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th python in science conference* (p. 51 - 56).

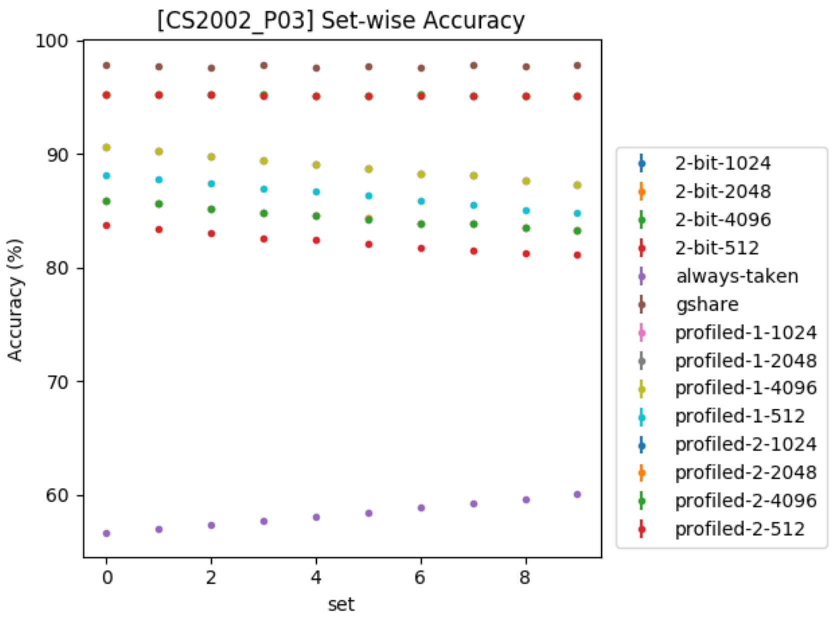
Pin - A Dynamic Binary Instrumentation Tool. (2018, Jun). Retrieved from `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool` ([Online; accessed 17. Oct. 2018])

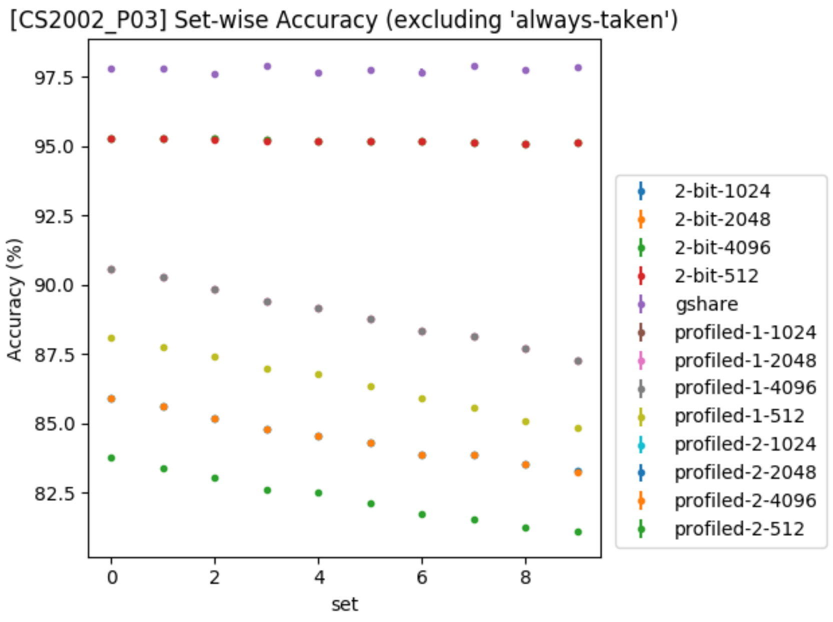
A Plots (standard error)

A.1 CS2002_P03

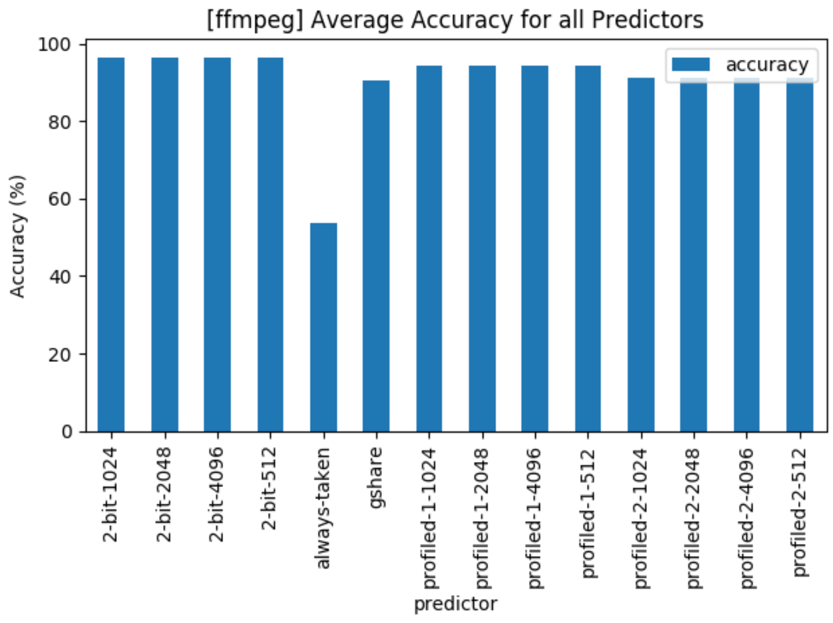


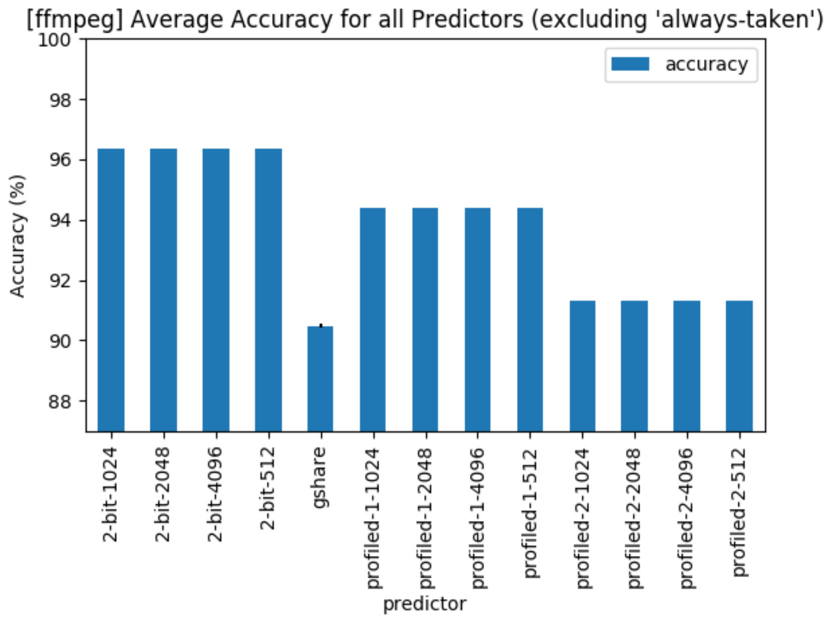


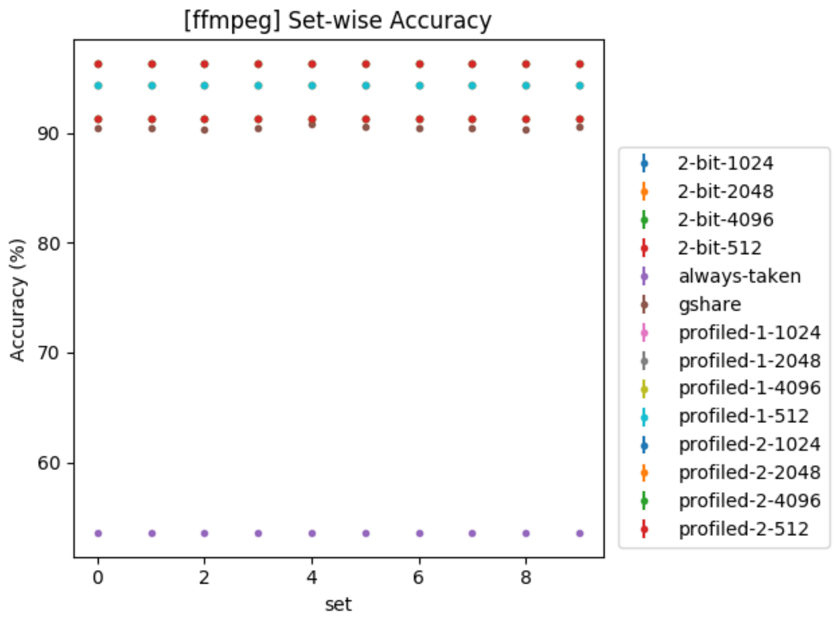

[CS2002_P03] Set-wise Accuracy
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

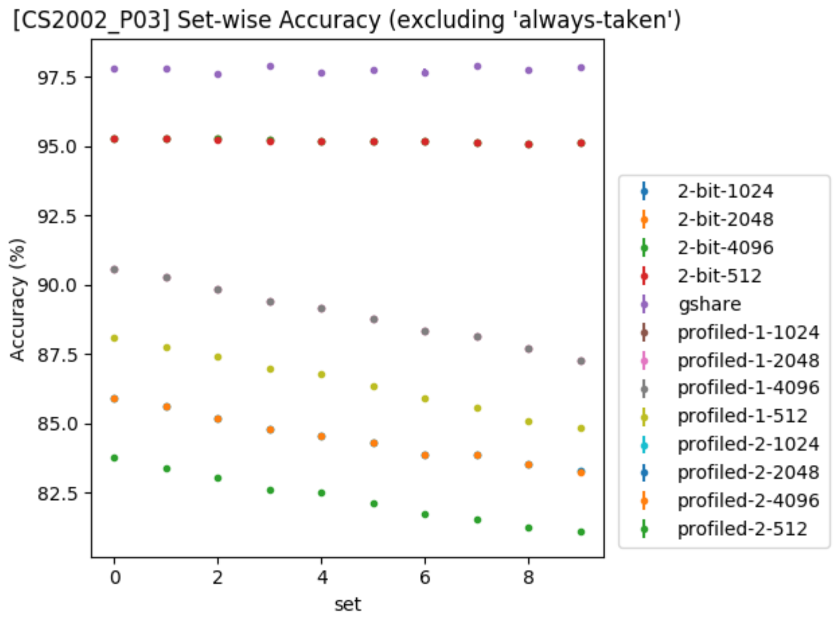

[CS2002_P03] Set-wise Accuracy (excluding 'always-taken')
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

A.2 ffmpeg

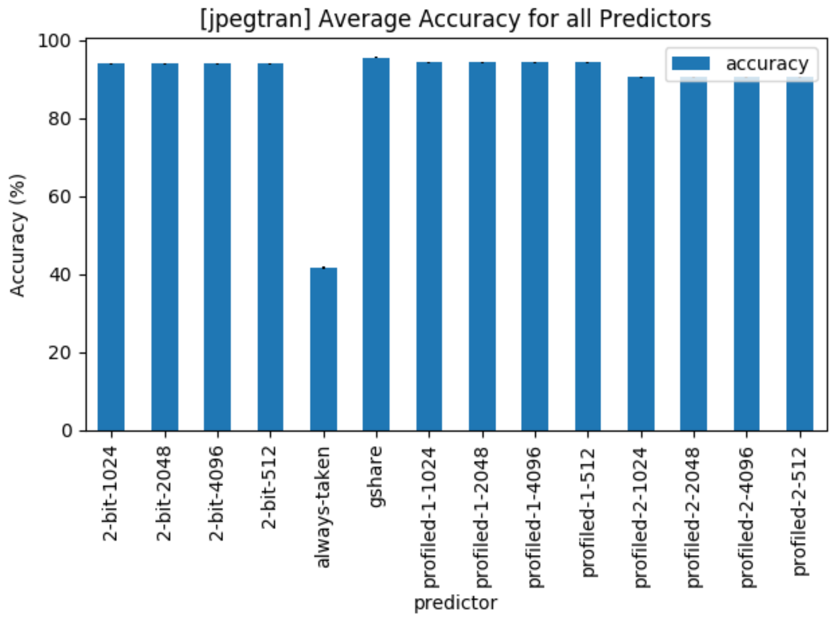

[ffmpeg] Average Accuracy for all Predictors
accuracy
Accuracy (%)
0
20
40
60
80
100
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512
predictor

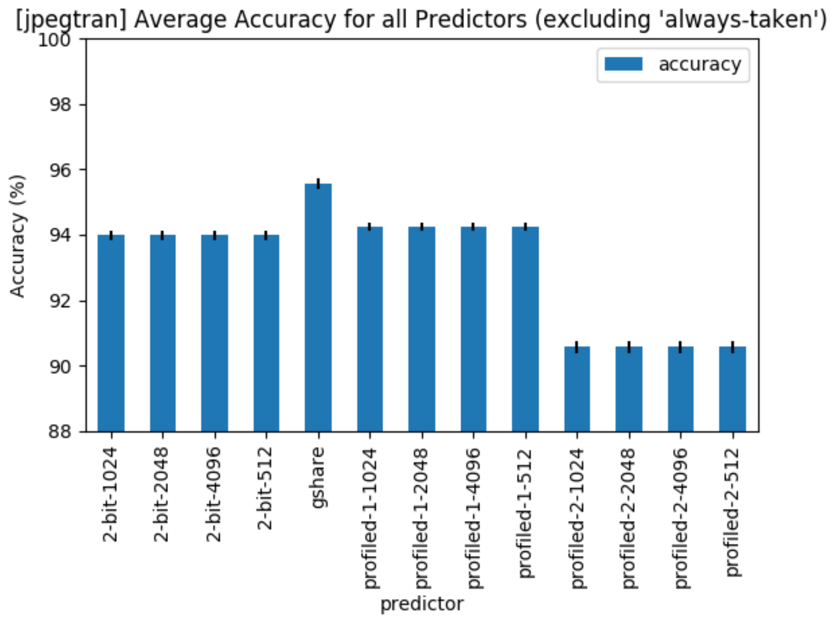

[ffmpeg] Average Accuracy for all Predictors (excluding 'always-taken')
accuracy
Accuracy (%)
88
90
92
94
96
98
100
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512
predictor

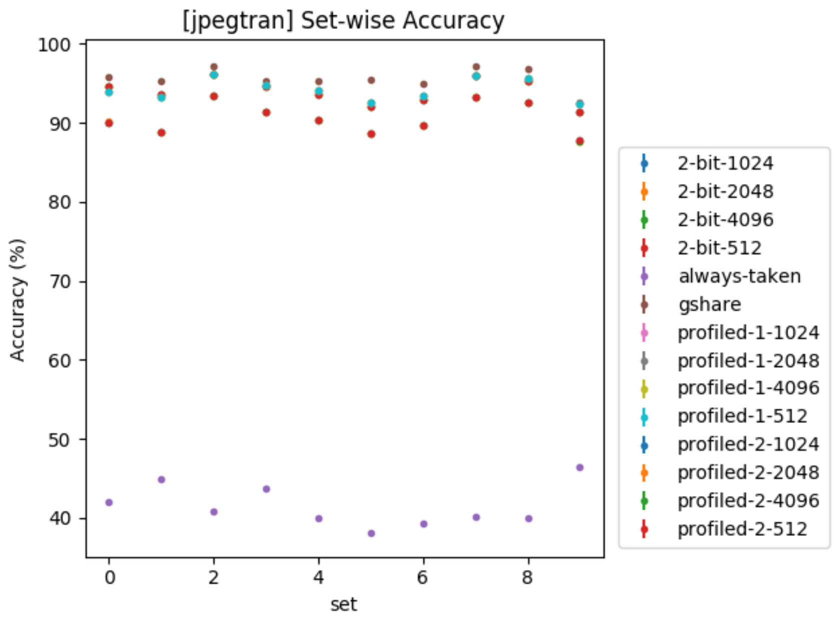

[ffmpeg] Set-wise Accuracy
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

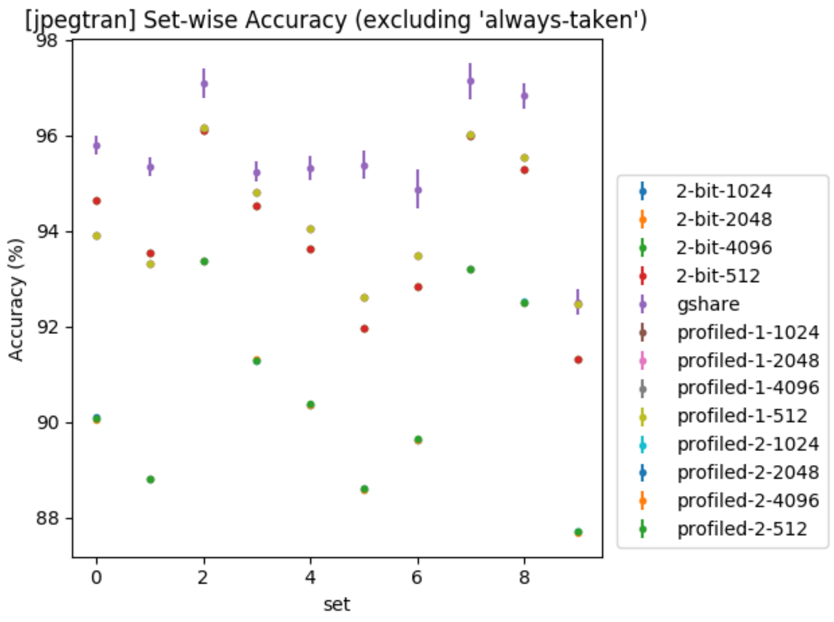

[CS2002_P03] Set-wise Accuracy (excluding 'always-taken')
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

A.3 jpegtran


[jpegtran] Average Accuracy for all Predictors
accuracy
Accuracy (%)
0
20
40
60
80
100
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512
predictor

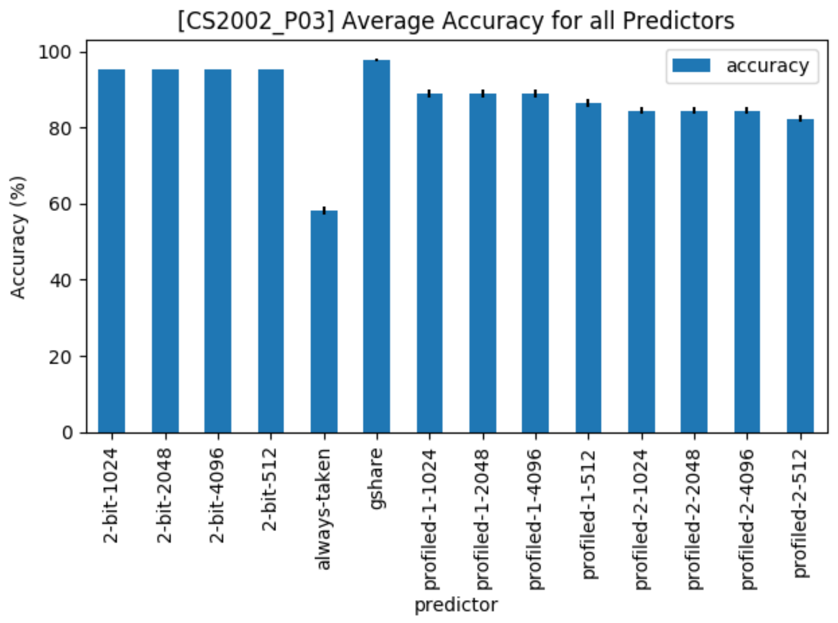

[jpegtran] Average Accuracy for all Predictors (excluding 'always-taken')
accuracy
Accuracy (%)
88
90
92
94
96
98
100
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512
predictor

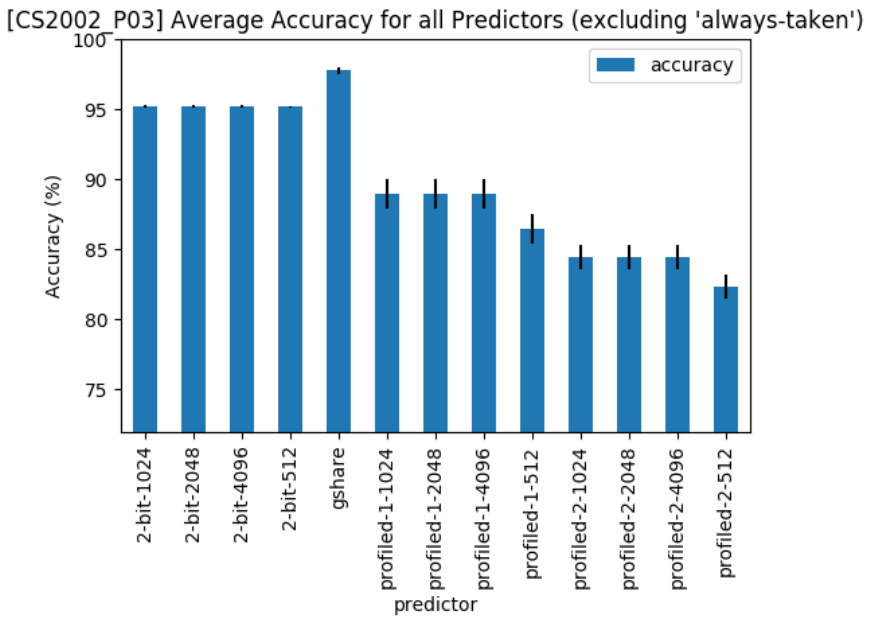

[jpegtran] Set-wise Accuracy
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

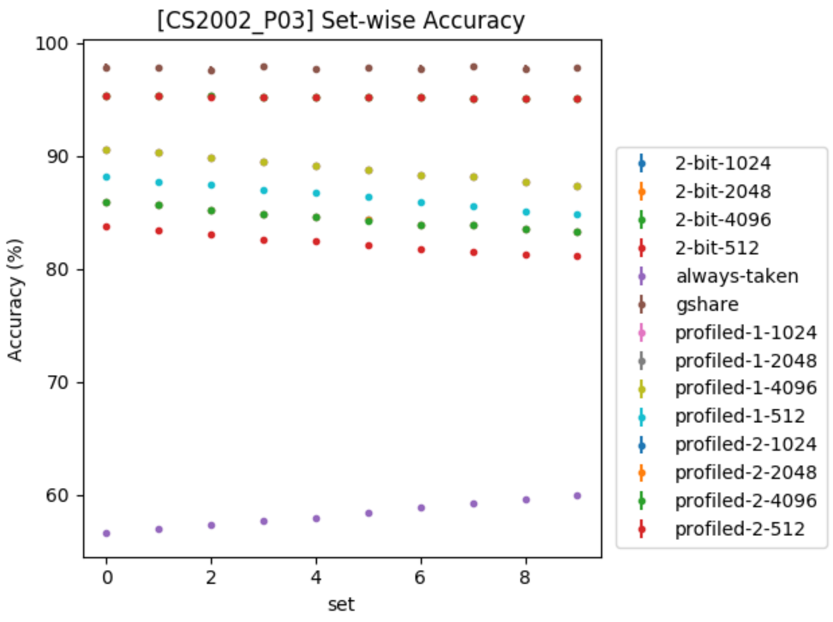

[jpegtran] Set-wise Accuracy (excluding 'always-taken')
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

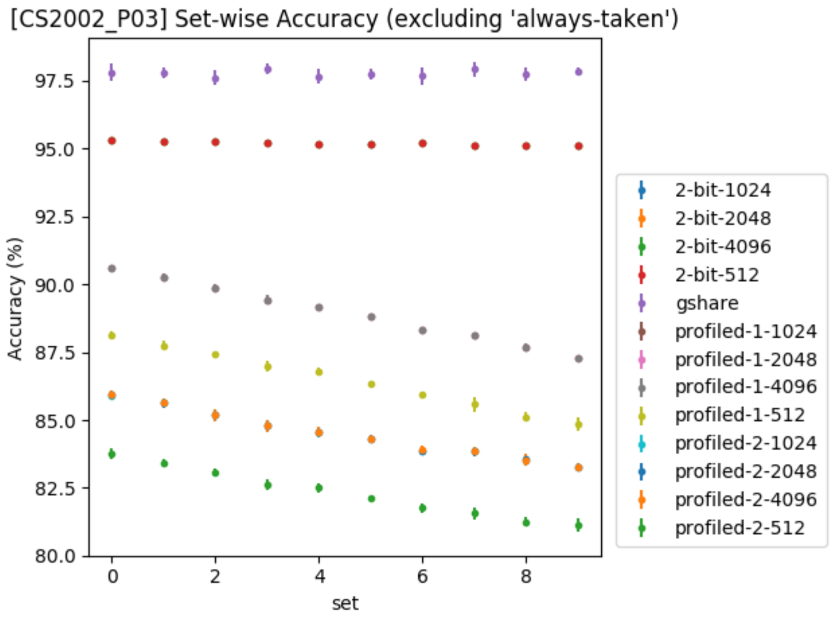
B Plots (standard deviation)

B.1 CS2002_P03

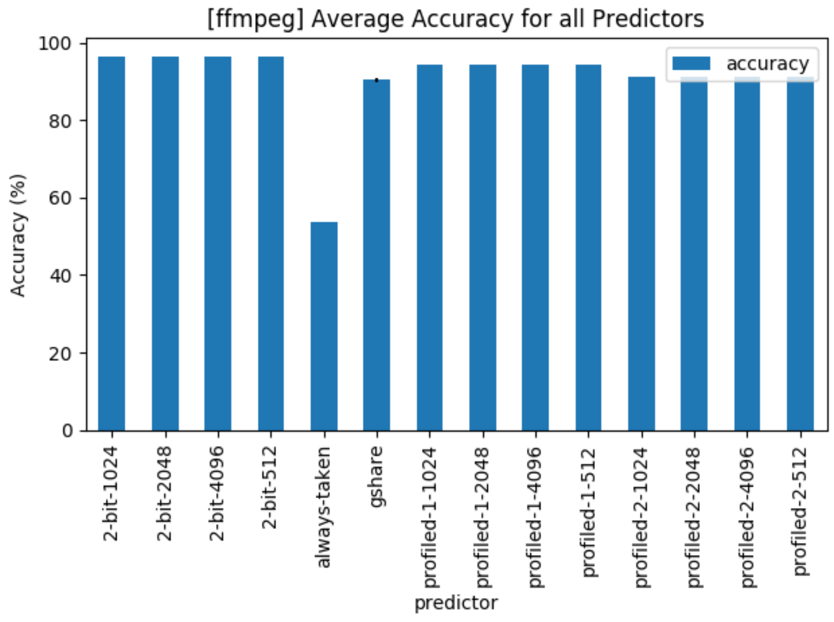


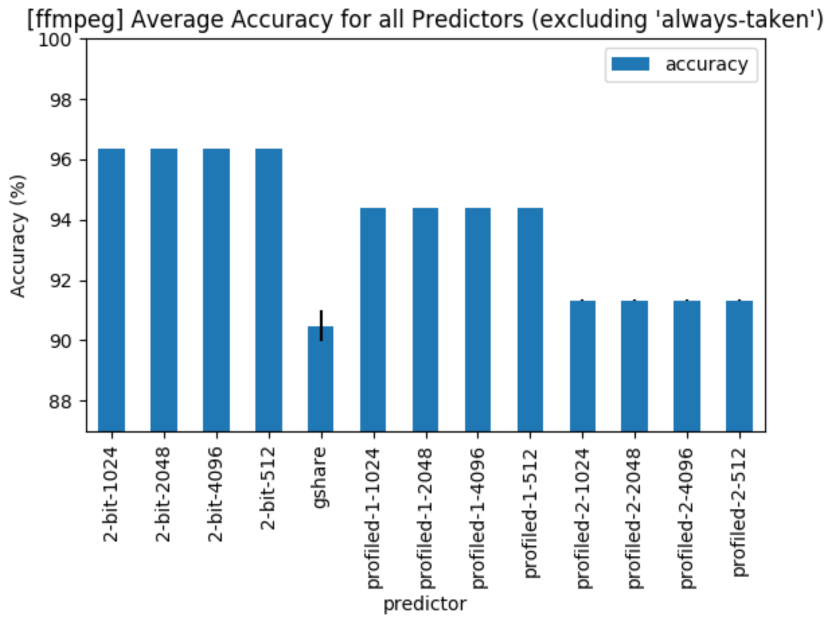


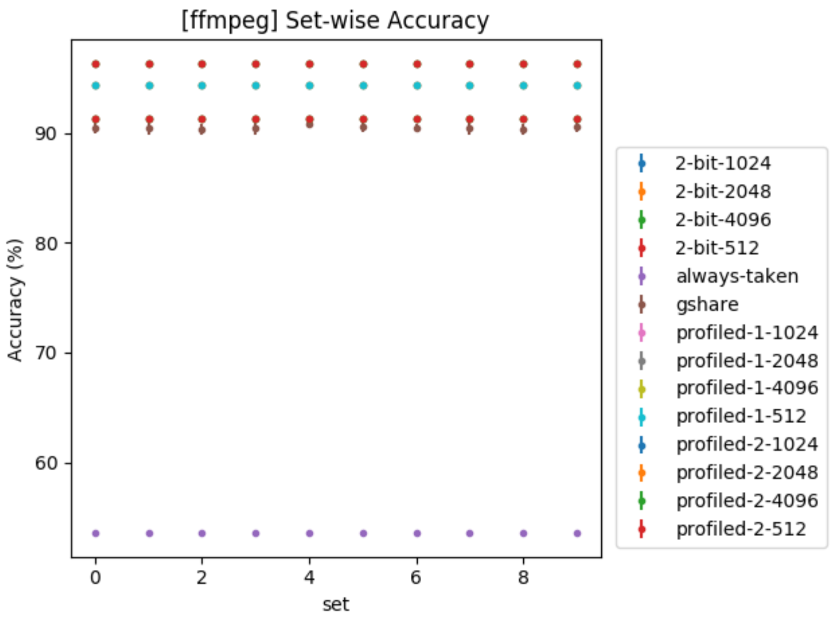

[CS2002_P03] Set-wise Accuracy
Accuracy (%)
set
100
90
80
70
60
0
2
4
6
8
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

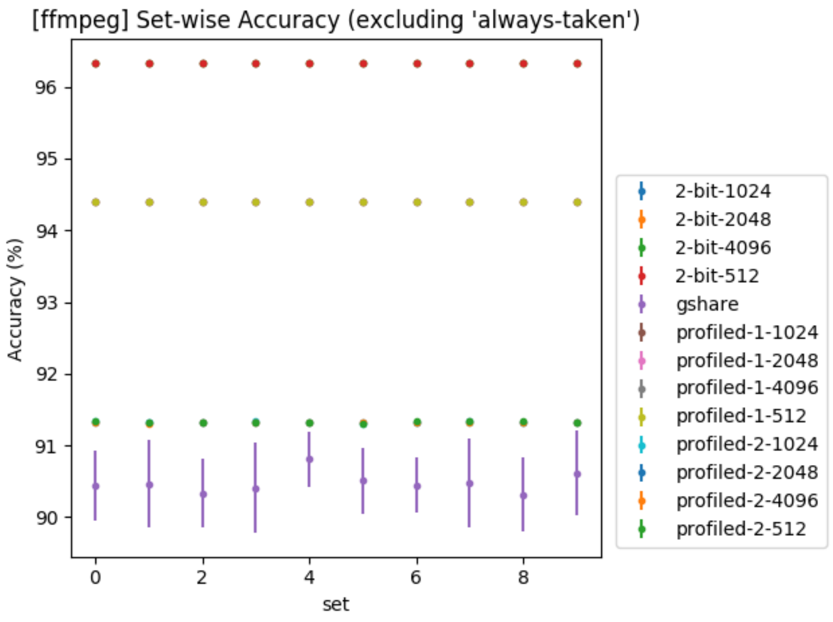

[CS2002_P03] Set-wise Accuracy (excluding 'always-taken')
Accuracy (%)
set
97.5
95.0
92.5
90.0
87.5
85.0
82.5
80.0
0
2
4
6
8
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

B.2 ffmpeg

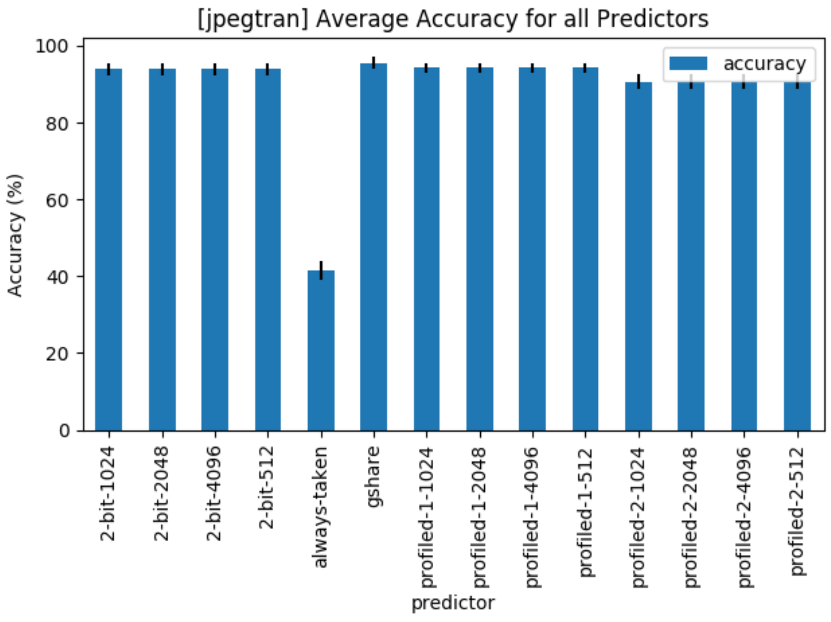

[ffmpeg] Average Accuracy for all Predictors
accuracy
Accuracy (%)
predictor
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

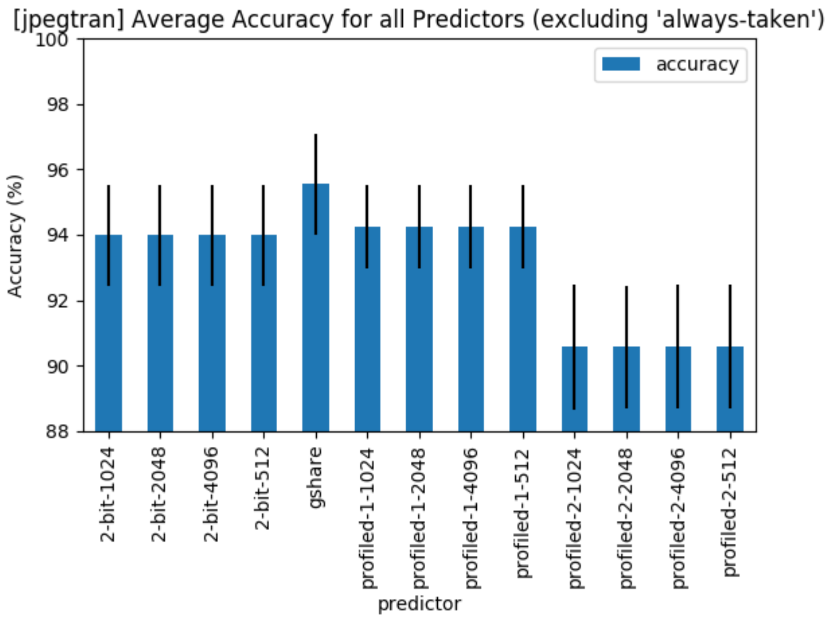

[ffmpeg] Average Accuracy for all Predictors (excluding 'always-taken')
accuracy
Accuracy (%)
predictor
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

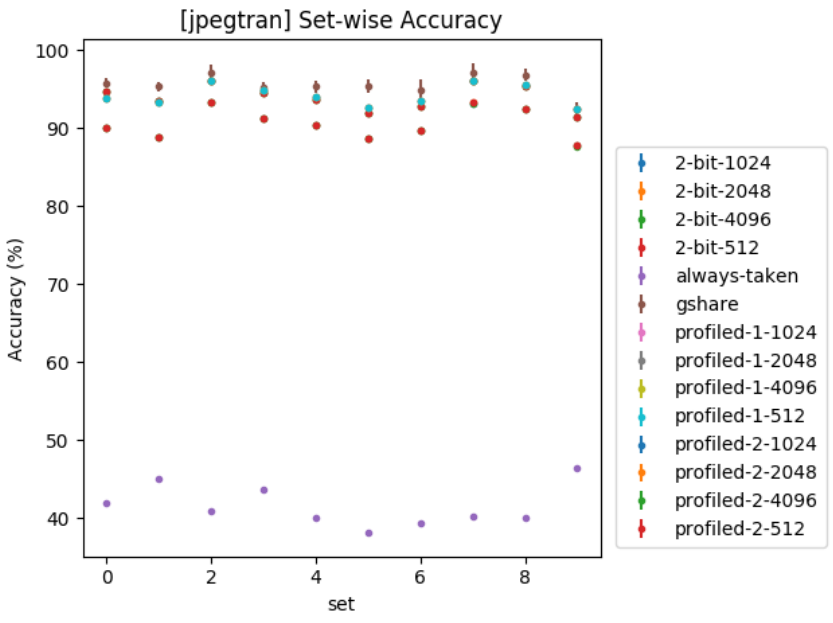

[ffmpeg] Set-wise Accuracy
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

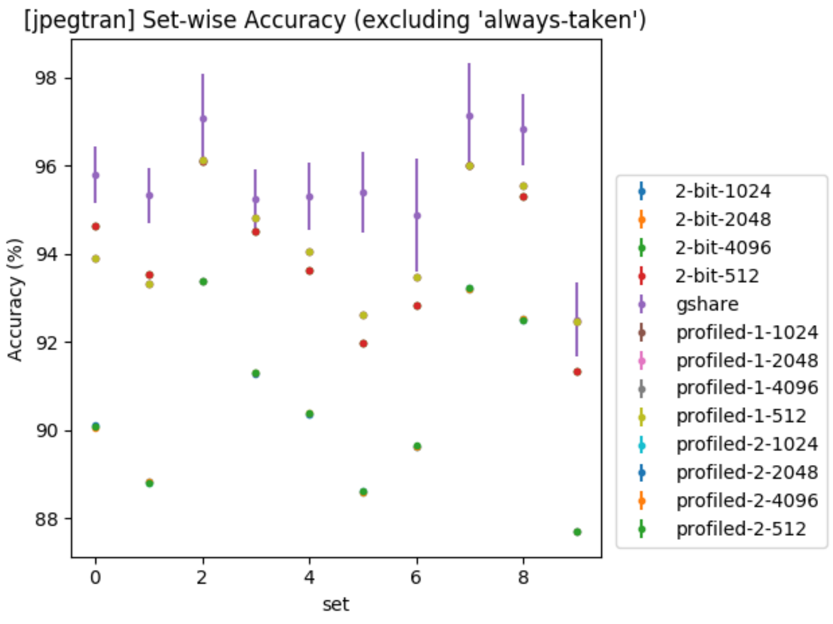

[ffmpeg] Set-wise Accuracy (excluding 'always-taken')
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512

B.3 jpegtran


[jpegtran] Average Accuracy for all Predictors
accuracy
Accuracy (%)
0
20
40
60
80
100
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512
predictor


[jpegtran] Average Accuracy for all Predictors (excluding 'always-taken')
accuracy
Accuracy (%)
88
90
92
94
96
98
100
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512
predictor


[jpegtran] Set-wise Accuracy
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
always-taken
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512


[jpegtran] Set-wise Accuracy (excluding 'always-taken')
Accuracy (%)
set
2-bit-1024
2-bit-2048
2-bit-4096
2-bit-512
gshare
profiled-1-1024
profiled-1-2048
profiled-1-4096
profiled-1-512
profiled-2-1024
profiled-2-2048
profiled-2-4096
profiled-2-512