CS4202 – CA CW1: Branch Prediction

This aim of this practical was to produce a set of virtual branch predictors, which could be used to analyse the efficacy of different branch predictors. The predictors are run against tracefiles produced using Intel’s PIN tool and the supplied branchtrace.cpp.

Predictors implemented

Predictors were implemented as an interface that was implemented by structs. These structs kept the internal state of each predictor, with the minimum of number of correct vs incorrect predictions (implemented in the struct “BasePredictor”, which all other predictors embedded).

An interface was required, with methods to obtain the correct and incorrect prediction counts, and to make a new prediction (and so updating those counts in the process).

These were then specialised in various ways:

Always Taken

Trivial.

Never Taken

Trivial.

2-bit predictor, in 512, 1024, 2048 and 4096 sized tables

Table was given as an array of 4096 ints. These ints were updated according to the state changes detailed in the book and lectures, using a switch statement.

When the predict statement is given an integer for the branch program counter, then this is and’ed with a bitmask to use the last n bits, where 2^n bits is the size of the table to be used. This n-bit-wide number is used to index the table, updating the relevant int as the state in the state machine.

Correlated 2-bit predictor, also in 512, 1024, 2048 and 4096 sized tables

An extension to the practical was to provide more predictors, so a Correlated 2-bit predictor was added to make the statistical analysis more interesting. This works on the assumption that the previous two branches taken affect the likelihood of a branch being taken.

Similar to the 2 bit predictor, an array of ints represent states — however, there are now 4 such arrays (in a 2-dimensional array). Each time after a prediction is made, the history of the last two branches is updated by shifting an int right, and using a bitwise or with 2 if the state was taken this time around. This is then used to index which table to use.

Profiled

This is largely theoretical, in the case of running a program through once already, a map of branch locations to their likelihood of being taken is made, and this probability is then used with a random number generator to make the prediction of a branch being taken.

Tests

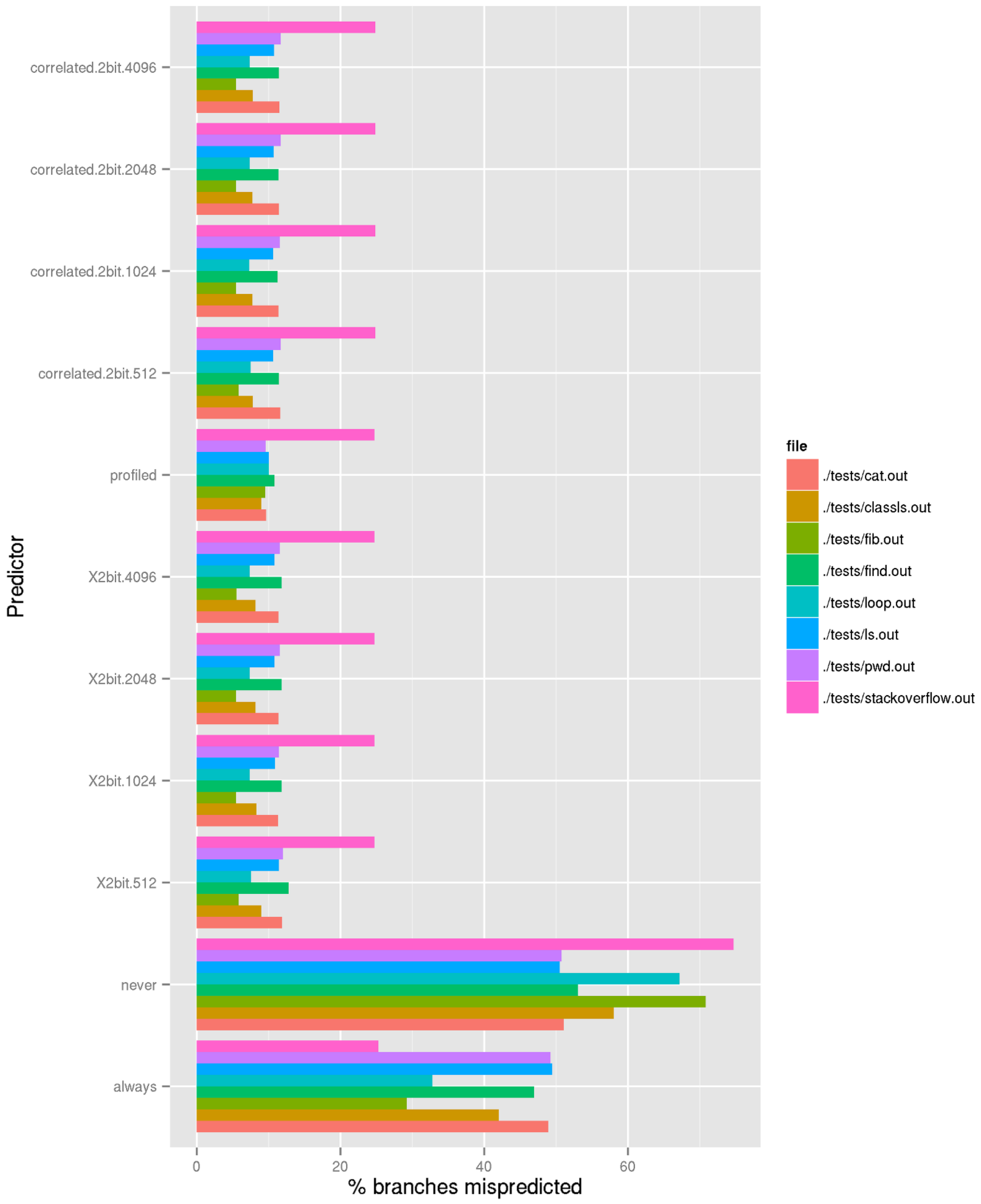
The programs profiled and used in the test case were a mixture of especially compiled ones and system tools. The system tools traced were:

- ls .
- find .
- pwd
- cat ls.out > /dev/null
- and a trace of ls provided amongst students in the class

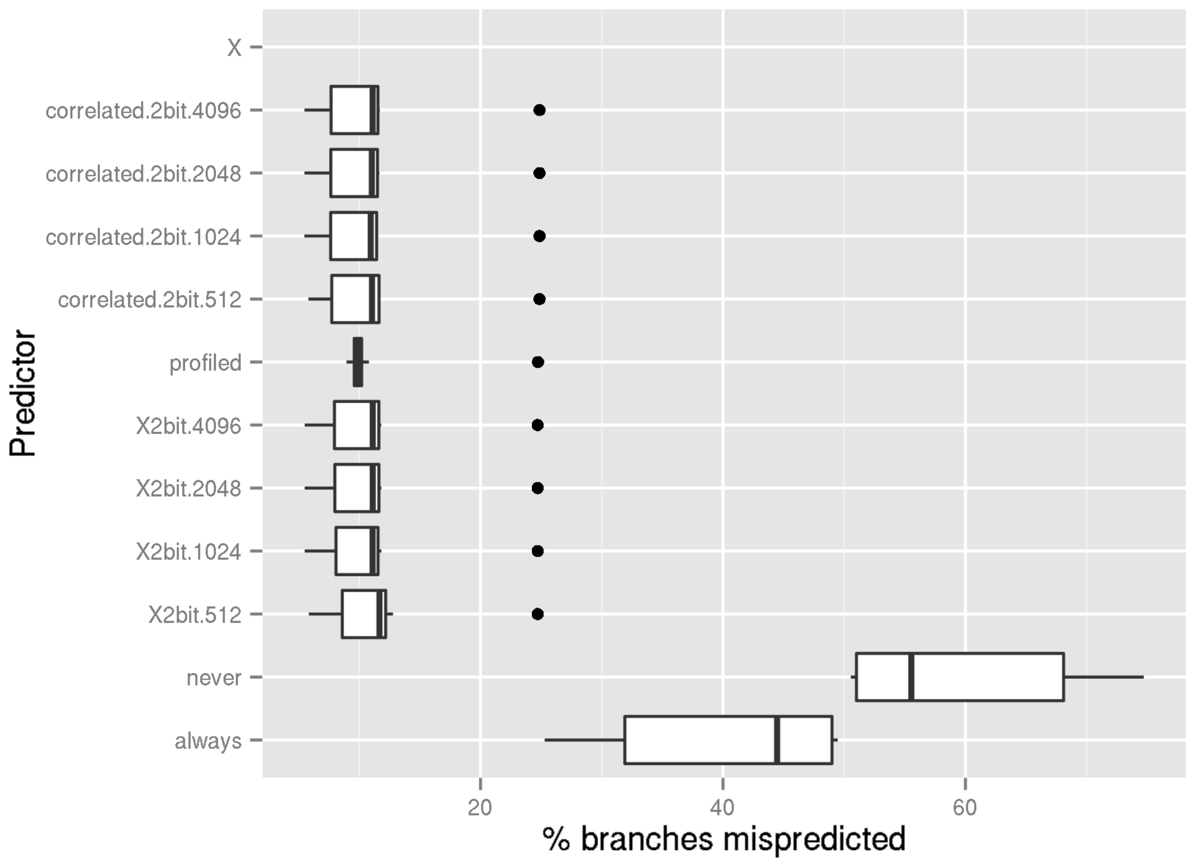
The especially coded ones included the program mentioned in this stackoverflow: http://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-an-unsorted-array with the sorting (and timing) lines removed. Also coded were a fibonacci program to see if recursion affected things (difficult to say, probably not), and a small loop that updated two variables in succession, chosen by an if statement.

Results

The metric decided to be tested was the percentage of mistaken branches. This decision was mostly arbitrary, and inconsequential, and can be seen for a branch trace when the program is passed the -csv flag. As such, the overall results looked as follows:



As can be seen, the fibonacci had generally the last amount of branches mispredicted, with the stackoverflow example generally with the largest amounts. Also, even at the best cases (with the stackoverflow program in the “always” case), the always taken and never taken predictors seem to fare worse than just about every other predictor — unsurprisingly. In a slightly easier to compare form:

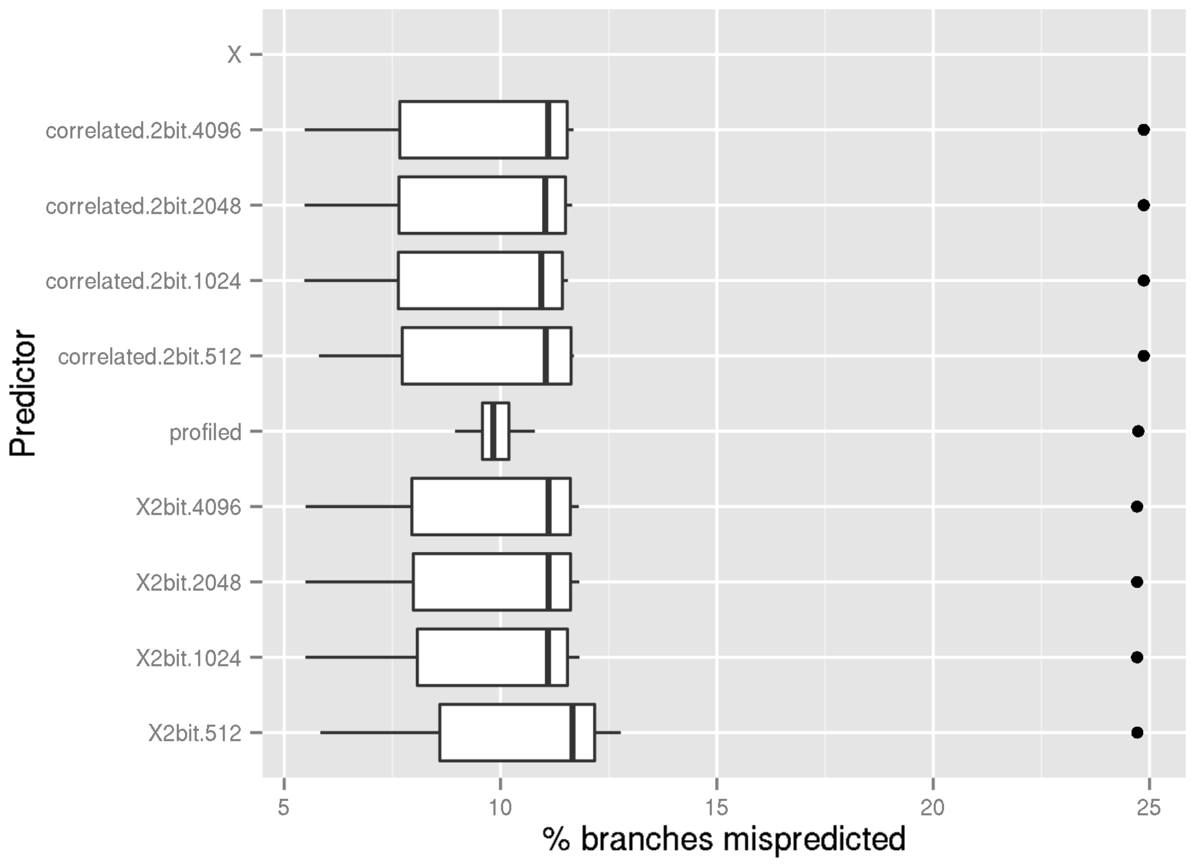


Again, it can be seen that the distribution of never and always taken is pretty dire, and mirrors each other (as one might expect, when they are literally opposite predictors of each other).

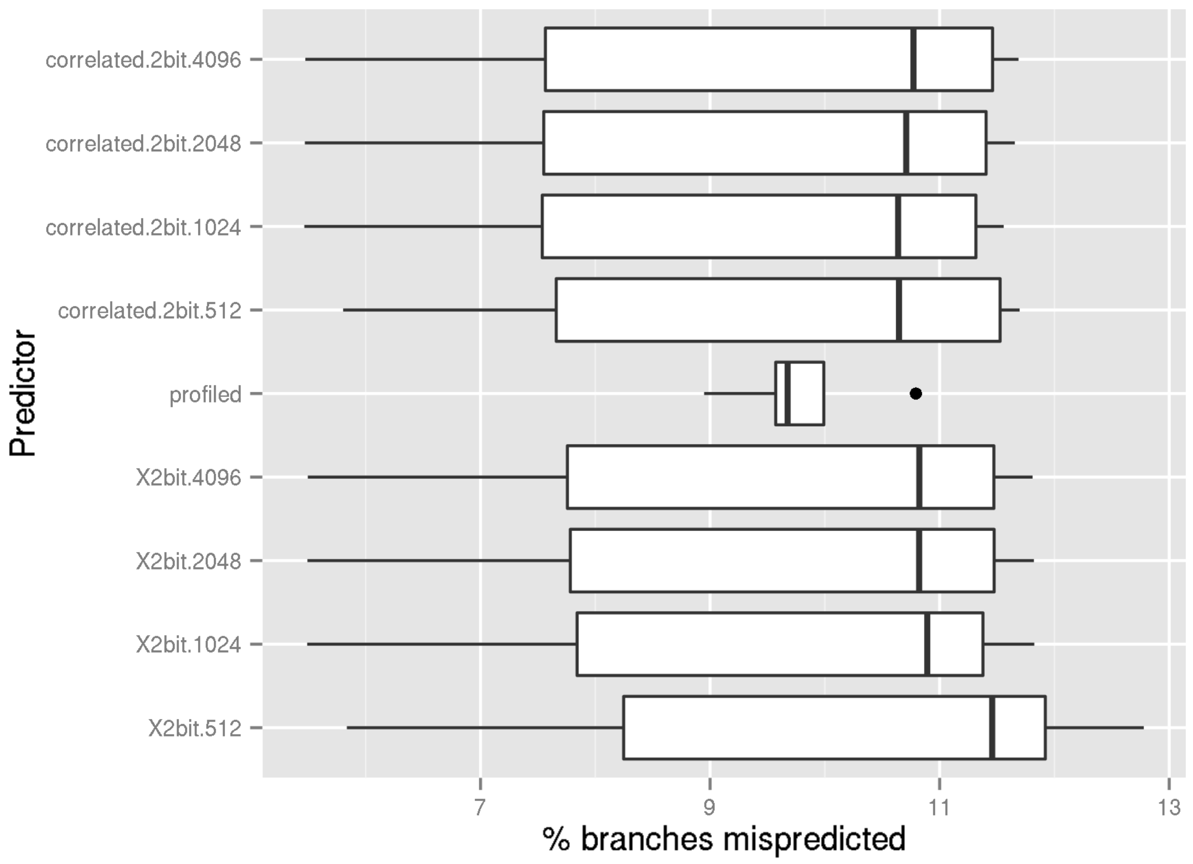
If we take the null hypothesis that the never taken has a true location shift of 0 (that points are around the same points), we can use a Wilcoxon Rank Sum Test to compare the values to, say, the Correlated 2-bit predictor with a 4096 entry table. In this case, the p-value is 0.000154, so we can say with some certainty that this null hypothesis is dismissed. This is repeated for the always taken, and the same value is found.

Given this, we can regard the never taken and always taken are dominated by the other predictors, and so can be discarded.

The other predictors can be now be compared easier on a graph:



The points near the 25% mark are the stackoverflow tracefile. It seems that this is somewhat pathological for all of the remaining predictors (and so difficult to compare the remaining predictors easily). Removing it produces this:



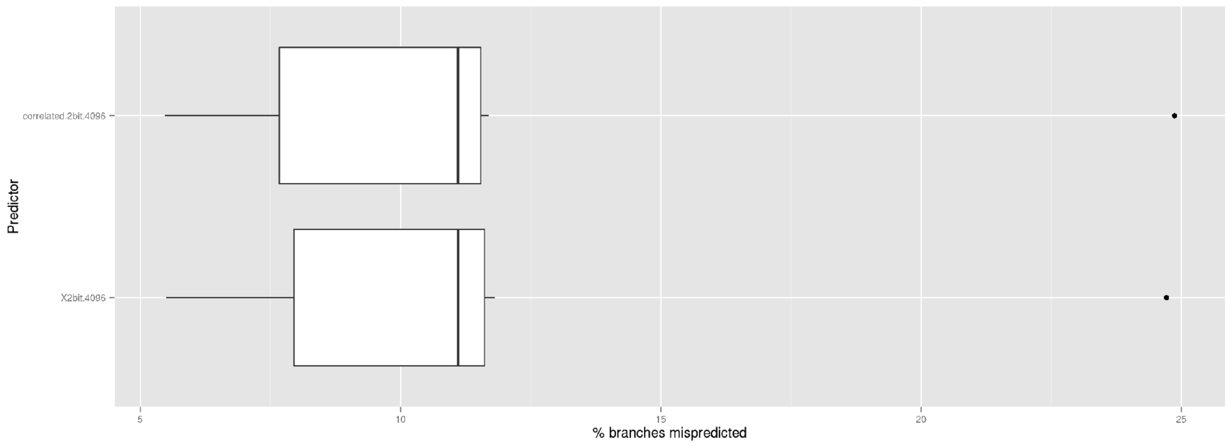
Now, the first noticeable difference is the distribution of results for the profiled results shows a much lower median than the various 2-bit predictors. Indeed, it's median is 9.832, compared to say, the 512 entry 2-bit predictor, which is 11.66. Its spread of results are also less wide than 2-bit predictors, which is difficult to qualify as a good or a bad thing — with a 2-bit predictor sometimes the results will be faster, and sometimes they will be slower than the profiled.

A Wilcoxon Signed Rank Test with the same null hypothesis as before (from hereon denoted as "the WSRF"), comparing the profiled predictor with the 2-bit 512 entry table then the p-value is 0.7984, a much higher number, and thus not as easy to dismiss, though with obvious differences in the range.

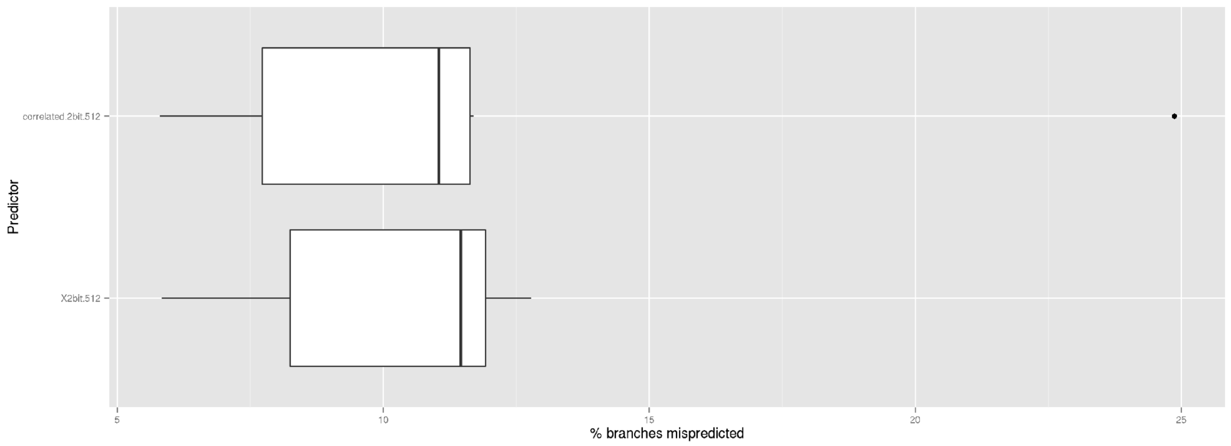
Comparing the 2-bit predictors is a little harder. It is clear that the 512-bit predictors fare worse than the higher tables, and there is a small trend in the median percentage of branches mispredicted decreasing for the plain 2-bit test (11.663 → 11.419 → 11.420 → 11.412 for 512→4096; there is a slight rise between 1024 and 2048, though this is probably more to do with the clashings in the table than anything else) but not for the correlated 2-bit test (11.046 → 10.942 → 11.034 → 11.101; no obvious pattern can be seen, but the 1024 entry appears to be faring the best). It's difficult to make any wide ranging statements with such little changes in data, but the p-value given by the WSRF between Correlated 2-bit 1024 (C2B-1024) and C2B-4096 is

0.6454, a definite change in data that might make us tentatively suggest 1024 as the better option, but not conclusively showing much without more data being collected.

The other main thing to check out is whether a correlated 2-bit predictor is any better than a normal one. Checking our results graph for the C2B-4096 and 2B-4096 is not very conclusive:



The correlated predictor has a slightly lower skewed range for mispredictions, but otherwise seem very similar. The WSRF between the two is 0.8785, so again, more data is needed to show an overall benefit. A wider difference is seen between the two 512 bit predictors:



Here, the WSRF p-value is much lower, at 0.5054. And with an obvious change in the graph, we can say with a little more certainty that the correlated branch predictor shows an improvement. However, more data is yet again needed.

Conclusion and Reflection

The data collected show a definite sign that there may be improvements made by the branch predictors shown. Correlating 2-bit predictors appear to be better than 2-bit predictors, and it is unclear what the advantages are with a 2-bit predictor vs a profiled predictor in terms of results. Though, it is questionable which situations you get to do a profile of a program's run time, and creating the random number to do the branch prediction would have to be built into the hardware of the CPU.

More data is needed to clearly distinguish the above, predictions, and also whether there is a difference in selecting for 1024 entry tables versus other entries — it may have been a quirk of the tracefiles given that lead to more evictions in the higher tables. However, it is difficult to obtain testing programs without access to something like a test suite, and a lot more time to do the testing in.
Whilst it would be possible to run the same programs in different states (say, using ls or find in different directories), to then make a call on the differences with these repeated programs may be making overstating the differences; the benefits of using C2B-1024 may be because of the repeated use of ls as a training source in this data, and less visible for other forms of programs.