CA Assignment 3 - BranchPredicion

Orson Peters, s1412728; Micky Faas, s1407937

The Framework

Files are organized in three sections: `main.c`, which parses the commandline arguments and fires up the correct predictor, `framework.h/c` which contain the framework functions and `predictors.h/c` which contain only the specific predictor functions.

The program cycle is as follows: - `main()` calls `predictor_setup()` with the filename and the name of the predictor from the commandline arguments. (The latter is only used for the output of the statistics in the end) - After all is set-up well, `main()` calls the predictor function of the user's choice. - The predictor function now uses the framework to fetch new branches and file predictions. This is essentially where the mainloop takes place, so every predictor-function has its own mainloop. - Every iteration, the predictor calls `predictor_getState()` to check if there are branches remaining - If so, it calls `predictor_getNextBranch()` which returns the address of the branch and advances the pseudo programcounter. This address may or may not be used by the predictor - Then `predictor_predict()` is called with the prediction. It returns the actual outcome (taken or not taken) of the branch to the predictor function. It also maintains the internal statistics. - If no more branches are left, control is transfered back to `main()` which calls one of the statistics-print functions and terminates the program.

The 'Simple' predictor

The Simple Predictor is implemented using a `state` enum which contains the values `TAKEN=0x1` and `CORRECT=0x01`. It can represent the four states of the automaton by bitwise operations, e.g.: `TAKEN | CORRECT`, `TAKEN & ~CORRECT`, `~TAKEN & CORRECT` and `0` meaning respectively taken and correct, taken but not correct, not taken and correct and not taken and not correct.

The GAg predictor

GAg is implemented using a simple bit-queue of k bits for the Branch History Register. A bit queue is of the type `bitQueue_t` which holds at most 64 bits. Only k bits are used. A bit (branch outcome) is added to the register using `pushFront()` by shifting all bits to the left towards the MSB by one step. The LSB is then set to the new bit (the head of the queue). Everything beyond the k-th bit is masked off. The Pattern History Table is an array of 2^k elements and is indexed directly by the integer value obtained by the bit-queue in the Branch History Register. The PHT array holds elements that are 8 bit wide as this is the with closest to 2 bit for the counter. All counters are initialized on 2 (weak taken).

When a prediction is made, the following cycle happens: - The address is fetched and discarded (global prediction) - The counter at `PHT[BHR]` is fetched, if greather than 1, the prediction is taken. - After the prediction is made, `PHT[BHR]` is updated with the actual outcome. - If is was taken, the counter is incremented, otherwise decremeneted - Lastly, the outcome is added to the current pattern, the BHR by doing `pushFront( &branchRegister, k, actual )`

The GAg is a fairly productive predictor. The hitrates range from 70% all the way up to 94% (fib30, k=18). Its biggest advantage is the size and hardware cost of the implementation - which are theoretically the smallest of the predictors described in the article (k + 2(2^k) for a given k). Simplicity is also an advantage: one register and a direct-mapped cache are sufficient for the implementation of GAg. The biggest downside is that due to its global nature, it is less accurate in real-life than its more complex, address- based peers.

The SAs predictor

SAs is easily implemented after GAg. Instead of taking just the global history and using that index a global pattern table to create sets of histories, and sets of patterns based on the address of the branch. We had 4 BHRs in a table, using the 11th and 12th least significant bits from the address to index them. To access the pattern table we concatenated those previous two bits with however many low-order bits of the address are needed to uniquely identify `n_sets`.

This predictor is quite a bit better than GAg in terms of performance, often improving hit rates from 70% to 80%, but also uses a lot more memory. It's probably better suited for a high-end CPU and not for embedded or small CPUs.

The GAg with Adaptive Counter

For our own prediction algorithm we focussed on the way the counter is maintained in the various prediction algorithms. For the sake of simplicity, we took the previously implemented GAg and modified its counter to 'anticipate' on the program flow, hence the name 'Adaptive Counter'. We will briefly lineout how it differs from the standard counter below.

The 2 bit counter is a simple state machine with four states. Two of them result in the prediction 'taken', two in 'not taken'. Based on the actual outcome of the prediction, the counter is either incremented (taken) or decremented (not taken). This gives a rough estimate of the probability the pattern will be taken/not taken again. The range of the counter is very limited though, which means a long repeating streak of the same prediction may influence the counter as much as an erratic pattern. Enlarging the counter gives emphasis on the long repeating streaks, but there is a trade-off here:

- The smaller the counter, the faster it 'reacts' to trends. It also picks up a lot of 'noise' though.
- The bigger the counter, the more a random pattern within a streak is smoothed out. At the cost of the reaction to short patterns and of course at the cost of much

more memory.

The Adaptive Counter splits the counter buffers (here 8 bit) in two: the low-order bits function the same as before, the high-order bits count the hits and misses. In our implementation the 'counter' field is 5 bits [0..31], the 'counter accuracy' field is 3 bits but is stored as a power of two (shifted four bits right) so its range is [0..15].

On every hit, the accuracy counter (M) is increased with its range. For every miss, M is decreased until two (the lowest bit is not stored). The counter (N) is incremented/decremented at the same time to 'scale' its range. The value for M now sets the bounds for N, i.e. the range for N is [0..2M-1]. Every value greater or equal to M means 'taken', otherwise not taken. So the most notable changes as opposed to standard GAg are:

- `prediction = N >= M` instead of `prediction = N >= 2`
- `if( actual && N < 2*M-1 ) N++` instead of `if( actual && N < 3 ) N++`

One can see the value of M is a 'confidence level' of the predictor. If M is high, changes to the pattern will take more time to affect the predictor. This way, noise is smoothed out. On the other hand, if M is small - it will be no smaller than 2 - the predictor is easily affected by the hits and misses.

Evaluation

In almost all cases, the GAg with Adaptive Counter scores better than the original GAg. The differences are not dramatic, but still in the range of 0.5 to 2% more hits (k=8). Unfortunately, the implementation also costs four times as much memory - 2 bits vs. 8 bits. There are cases, however, where implementing an Adaptive Counter gives a larger performance increase than multiplying k by four (example: k=6 and k=8).

A global Adaptive Counter

Implemented in the bonus1() function is also an attempt on a GAg with a Global Adaptive Counter. This takes only half the memory of the above implementation, plus a 3 bit global accuracy counter (M). The counter (N) is 4 bits in this implementation. Tests show that its performance can differ very little from the other Adaptive Counter implementation for certain k (eg. k=8).