UNIVERSITY OF AMSTERDAM
Institute of Informatics
Computer Systems Architecture

Dr Andy Pimentel
Jun Xiao
Simon Polstra
August 31, 2017

Parallel System Architectures 2017

— Lab Assignment 2: Branch Prediction —

Introduction

In this assignment you will use the `C` programming language to build a simulator for several branch predictors and evaluate their performance. The simulator will be driven using trace files which will be provided. All documents and files for this assignment can be found at the website of the lab.
The framework can be downloaded from:

`http://staff.fnwi.uva.nl/s.polstra/psa2017/`

This framework contains all documentation, the helper library with supporting functions for managing trace files and statistics, the trace files, and a Makefile to automatically compile and run your assignment.

Submission & Assessment

For assessment you must:

1. Demonstrate and present your solutions and code to one of the lab assistants during one of the lab sessions before each deadline.
2. You cannot assume unlimited resources in your branch predictor. Make the number of resources (i.e. table size of the branch predictor) configurable at runtime.
3. Write a report with your results, explanation of the results and a comparison between the different branch predictors.
4. Submit your solution including report before the deadline on blackboard, packed in tar/gz format.

Submission Requirements:

1. Your source code should compile and run on the Linux lab machines with the command `make` without any modifications to the submitted code, either using the provided Makefile or your own, which has to be included.
2. The files `framework.c` and `framework.h` may not be modified.
3. Your code has to be properly formatted and documented.
4. The programming assignment and report should be written individually.
5. The tarball you submit must be created with the `make tarball` command we provided.
6. Reports must be in PDF format.

Additional Information: Even though attending the lab sessions is not compulsory, we would suggest you to come in regularly. This is because the lab assistants will have the most time for answering your questions during the sessions. If you want to work from home or your own machine, test if your code is working properly on the Linux lab machines before you submit it.
Questions: You can ask questions during the lab sessions. If you have attended the lab sessions and have additional questions you can ask them via email (`J.Xiao@uva.nl` and/or `S.Polstra@uva.nl`) or drop by in our office **C3.101** or **C3.159**. If there are general questions or if the assignment is unclear and your fellow students can also benefit from the answer, please post to the psa2017 mailing list `psa2017@list.uva.nl`. You can (should) register for this list at `https://list.uva.nl/mailman/listinfo/psa2017`.

Trace files

There are different kinds of trace files available to you. We have provided branch traces of the following benchmarks:

- N-Queens: A chessboard of $N \times N$ tiles and N queens. The queens must be positioned in such a way that no two queens are on the same horizontal, vertical or diagonal line.
- Fibonacci: Calculate the N-th Fibonacci number recursively.
- Matmul: Do a series of matrix multiplications in different ways
- Ray Tracing: A simple ray tracer.

There are benchmarks of different sizes as can be seen in Table 1.

Benchmark	Total Branches	Unique Branches
8-queens	13.212	743
9-queens	36.429	742
10-queens	119.355	762
11-queens	581.263	769
12-queens	2.727.424	768
fib(5)	1.476	485
fib(10)	1.849	491
fib(20)	36.203	494
fib(30)	4.241.389	491
matmul	838.802	1001
matmul (no print)	22.622	660
ray tracing	42.400.131	797

Table 1: Statistics for different benchmarks

Framework functions

The framework, as specified and documented in `framework.h` provides you with the following functions:

- `int predictor_setup(const char *filename, const char *predictor_name)`
  Sets the environment and opens `filename` for reading the trace. The `predictor_name` is only used in statistics output. Returns `0` on success, `-1` otherwise.
- `int predictor_getNextBranch(uint32_t *addr)`
  Provides you with a branch address. Returns `0` on success, `-1` in case of invalid state.

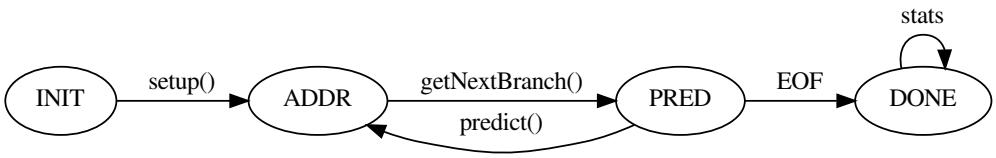

Figure 1: Framework state machine

- `int predictor_predict(bool prediction, bool *result)`
  Takes your `prediction` (boolean, `true` equals taken) for the branch associated with the last branch address returned by `predictor_getNextBranch()` and provides the actual `result` (`true` = taken) of the branch.
- `state predictor_getState()`
  Returns the current state of the statemachine (`INIT`, `ADDR`, `PRED`, `DONE`), in case you are lost...
- `void predictor_printScore()`
  Only outputs the tracefile name, predictor name and percentage of well-predicted branches.
- `void predictor_printBasicStatistics(int csv)`
  Outputs some more statistics in a table (`csv=0`) or csv (`csv=1`) format.
- `void predictor_printAdvancedStatistics(int csv)`
  Outputs detailed statistics per branch in a table (`csv=0`) or csv (`csv=1`) format.
- `void predictor_cleanup()`
  Cleans up environment and frees memory.

Makefile

The provided `Makefile` can be used to compile your program, but also to create (test)runs for the predictors. For example the command `make runtest` can be used to perform a test run on the `sample` (small) tracefile. You can change the tracefile and arguments yourself: `make runtest TRACE=8queens ARGS='-t -s'`. `make runall` will execute a script that runs all predictors with all possible output statistics and create a directory `results` where all statistics will be stored in csv format. Use `make clean` to clear all temporary files binaries created by the framework.

Use `STUDID=<your student id> make tarball` to create a tarball suitable for submission of your work.

Assignments

For the assignments you can not assume unlimited resources in the system. The number of history tables etc. should be configurable at runtime. We provide skeleton functions for you to implement. If you add options in *main.c* do not forget to document those in your report.

For your report you should experiment with the different parameters to see how the branch predictors perform. And try to explain why.

As always properly documented code and data structures are required.

Assignment 0: Getting to know the framework

Familiarise and experiment with the prediction framework and test the pre-implemented branch predictors. Explain briefly in your report how this works and what your experiences are.

Assignment 1: A simple branch predictor

Implement the simple branch predictor in Figure 2. Keep only one global history vector of two elements.

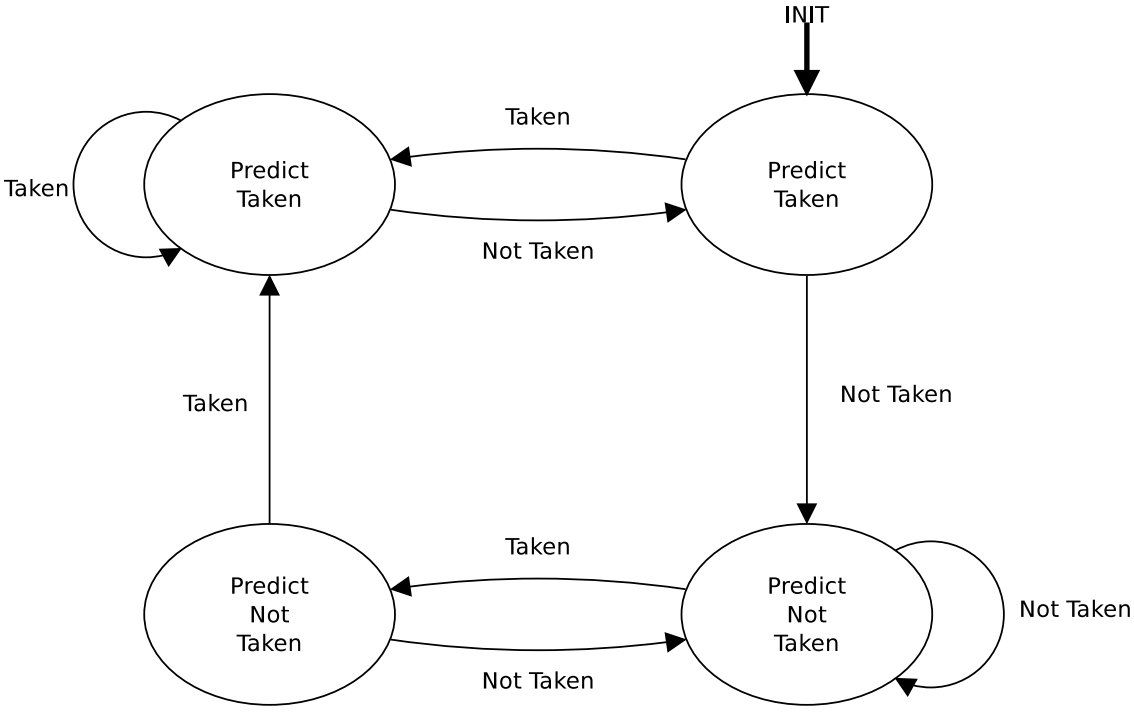

Figure 2: State machine for a very simple branch predictor

Assignment 2: Implement the GAg branch predictor

Implement branch prediction algorithm *GAg* from [1]. Write a section in your report on this branch predictor and how you have implemented this. What are the advantages and disadvantages of this branch predictor, etc. The number of history elements are provided by the *-k* option.

Submission due date: to be announced

Assignment 3: Implement the SAs branch predictor

Implement branch prediction algorithm *SAs* from [1]. Write a section in your report on this branch predictor and how you have implemented this. What are the advantages and disadvantages of this branch predictor, etc. Again the number of history elements are provided by the *-k* option, the *-n* option specifies the number of sets.

Assignment 4: Implement your own branch predictor

Implement your own-very-fancy-branch-predictor. Your implementation should take the hardware costs into account and if you use some kind of table to store state information, the maximum size of this

table must be limited with a runtime option. Compare the hardware costs for the different branch predictors in your report. Also include at least one literature reference to the hardware complexity of branch predictors and discuss this briefly in a quarter to half a page.

In this framework, we only provide you the branch address. For most predictors this is sufficient, but you may require more information to give a better prediction. Describe in your report which additional information may be useful and how you could obtain this (i.e. from the instruction stream: the exact type of the branch instruction, the branch target etc.). Can the compiler play a role in predicting branches? How?

Submission due date: to be announced

Grading

Part	Points
Submission	1.0
Task 1	0.5
Task 2	1.0
Task 3	1.0
Task 4	2.5
Report	4.0
Total	10.0

References

[1] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 257–266, New York, NY, USA, 1993. ACM.