Laboratory 2

Yrkkö Äkkijyrkkä yrkko@kth.se Martin Yrjölä yrjola@kth.se

Introduction

We have been collaborating with a group of other students to automate the Simics benchmarking in the git repository published at https://github.com/myrjola/comparch_labs. The data used for our figures and tables and the code to generate them are published in the same repository. The majority of the lab time has been spent developing the automation and doing data gathering. We have discussed our findings actively with the repository collaborators, but also took care to conduct our own benchmarks and write our own analysis.

2.3 Collecting IPC statistics using the MAI

The recorded IPCs of each benchmark is presented in tbl. 1. The equake benchmark has the highest IPC of almost 2 and the vortex benchmark has the worst IPC of under 0.9.

Table 1: Instructions per cycle performance for the benchmarks

Benchmark	Instructions per cycle
vortex	0.8933801
parser	1.1425188
equake	1.9896457

2.5 Collecting data about the effect of memory latency on OoO efficiency

We chose the equake benchmark for this task. The IPC statistics for different penalties is given in tbl. 2. We can clearly see that increasing the penalties will have a negative impact on performance. The impact of increasing cache penalty is higher than that of increasing the memory penalty. The results suggest that out-of-order processing masks small penalties very well because it's likely there is always a couple of instructions not dependent on the load instructions. Once the penalties get long enough the cpu starts stalling to wait on the memory transactions to finish.

Table 2: IPC performance for the equake benchmark with different penalties

Read penalty	Write penalty	Memory penalty	Instructions per cycle
1	1	10	1.9979500
2	2	10	1.6909622
5	5	10	1.2146641
1	1	20	1.8118661
1	1	50	1.4248578

3.1 Branch predictor study

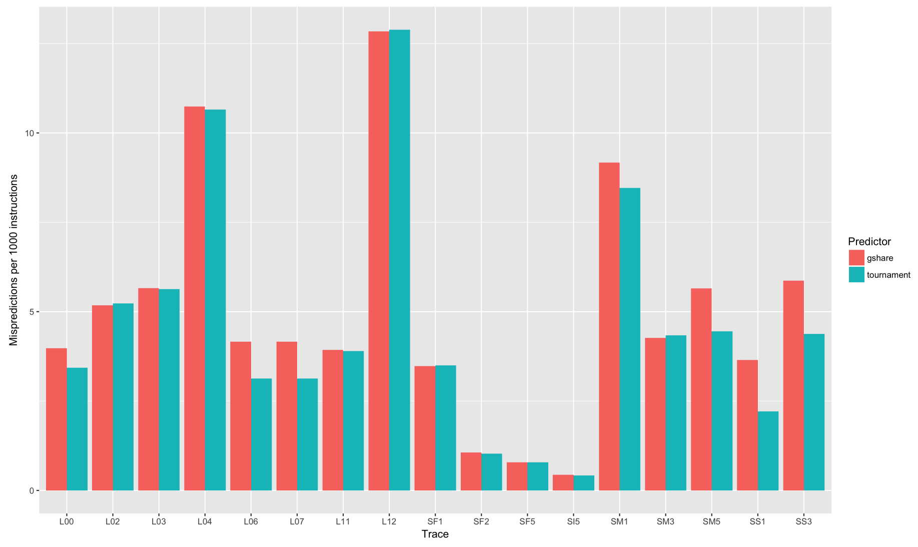

Figure 1: Branch predictor performance

Inspired by the course book chapter 3.3 we implemented a tournament predictor Hennessy and Patterson (2011). We chose to use the already implemented Gshare predictor with a 2-bit local predictor. The performance comparison is presented in fig. 1, the metric used is mispredicts per 1000 instructions. The traces we used were the ones we got working, we also noticed that the arithmetic mean calculated by the perl script was incorrect. The correct arithmetic mean of the mispredicts per 1000 instructions is 5.00 for the Gshare predictor and 4.56 for our tournament predictor. So we only achieved a marginal improvement over the baseline. The source code for our predictor is given in Appendix A.

Appendix A: Code for assignment 3.1

Contents of `predictor.h`

```
#ifndef _PREDICTOR_H_
#define _PREDICTOR_H_

#include "tracer.h"
#include "utils.h"

enum ChosenPredictor {
  GSHARE,
  LOCAL_PREDICTOR
};

class PREDICTOR {

private:
  UINT32 globalHistoryRegister;
  UINT32 *gsharePredictionTable;
  UINT32 *localPredictionTable;
  UINT32 *tournamentPredictionTable;
  UINT32 historyLength;
  UINT32 numEntries;

public:
  // The interface to the four functions below CAN NOT be changed
  PREDICTOR(void);
  bool GetPrediction(UINT32 PC);
  void UpdatePredictor(UINT32 PC, bool resolveDir, bool predDir,
                       UINT32 branchTarget);
  void TrackOtherInst(UINT32 PC, OpType opType, UINT32 branchTarget);

  // Contestants can define their own functions below
  UINT32 localIndex(UINT32 PC);
  UINT32 tournamentIndex(UINT32 PC);
  UINT32 gshareIndex(UINT32 PC);
  enum ChosenPredictor tournamentPrediction(UINT32 PC);
  bool localPrediction(UINT32 PC);
  bool gsharePrediction(UINT32 PC);
};

#endif
```

Contents of predictor.cc

```
#include "predictor.h"

// Unused parameter macros to silence compiler warnings.
// Taken from http://stackoverflow.com/a/12199209
#define MON_Internal_UnusedStringify(macro_arg_string_literal) \
  #macro_arg_string_literal
#define MONUnusedParameter(macro_arg_parameter) \
  _Pragma(MON_Internal_UnusedStringify(unused(macro_arg_parameter)))

enum {
  // We are using a 2-bit counter for all predictors
  GSHARE_CTR_MAX = 3,
  GSHARE_CTR_INIT = 2,
  LOCAL_PREDICTOR_CTR_MAX = 3,
  LOCAL_PREDICTOR_CTR_INIT = 2,
  TOURNAMENT_CTR_MAX = 3,
  TOURNAMENT_CTR_INIT = 2,

  HISTORY_LENGTH = 16,
} Constants;

// STORAGE BUDGET JUSTIFICATION
//
// Total storage budget: 65KB
//
// Counter size: 2^16 * 2-bits
// Number of predictors: 3
// Global history register size: 16 bits
// Total size = 2^16 * 2 * 3 + 16 = 393229 bits = 49KB

// PREDICTOR DESIGN
//
// Tournament predictor with local branch predictor and gshare correlating
// predictor. Used to predict conditional branches. All counters are 2-bits.

PREDICTOR::PREDICTOR(void) {
  historyLength = HISTORY_LENGTH;
  globalHistoryRegister = 0;
  numEntries = (1 << HISTORY_LENGTH);
  gsharePredictionTable = new UINT32[numEntries];
  localPredictionTable = new UINT32[numEntries];
  tournamentPredictionTable = new UINT32[numEntries];

  for (UINT32 i = 0; i < numEntries; i++) {
    gsharePredictionTable[i] = GSHARE_CTR_INIT;
    localPredictionTable[i] = LOCAL_PREDICTOR_CTR_INIT;
    tournamentPredictionTable[i] = TOURNAMENT_CTR_INIT;
  }
}

bool PREDICTOR::GetPrediction(UINT32 PC) {
  enum ChosenPredictor chosenPredictor = tournamentPrediction(PC);

  if (chosenPredictor == GSHARE) {
    return gsharePrediction(PC);
  }
  return localPrediction(PC);
}

void PREDICTOR::UpdatePredictor(UINT32 PC, bool resolveDir, bool predDir,
                                UINT32 branchTarget) {
  MONUnusedParameter(PC);
  MONUnusedParameter(predDir);
  MONUnusedParameter(branchTarget);

  UINT32 tournamentIdx = tournamentIndex(PC);
  UINT32 tournamentCounter = tournamentPredictionTable[tournamentIdx];
  UINT32 gshareIdx = gshareIndex(PC);
  UINT32 gshareCounter = gsharePredictionTable[gshareIdx];
  UINT32 localIdx = localIndex(PC);
  UINT32 localCounter = localPredictionTable[localIdx];

  // Update the tournament predictor.
  bool localPred = localPrediction(PC);
  bool gsharePred = gsharePrediction(PC);
  if (resolveDir == gsharePred && resolveDir != localPred) {
    tournamentPredictionTable[tournamentIdx] = SatIncrement(tournamentCounter,
                                                            TOURNAMENT_CTR_MAX);
  } else if (resolveDir == localPred && resolveDir != gsharePred) {
    tournamentPredictionTable[tournamentIdx] = SatDecrement(tournamentCounter);
  }

  // Update the gshare and local prediction table.
  if (resolveDir == TAKEN) {
    gsharePredictionTable[gshareIdx] = SatIncrement(gshareCounter, GSHARE_CTR_MAX);
    localPredictionTable[localIdx] =
        SatIncrement(localCounter, LOCAL_PREDICTOR_CTR_MAX);
  } else {
    gsharePredictionTable[gshareIdx] = SatDecrement(gshareCounter);
    localPredictionTable[localIdx] = SatDecrement(localCounter);
  }

  // Update the global history register.
  globalHistoryRegister = (globalHistoryRegister << 1);

  if (resolveDir == TAKEN) {
    globalHistoryRegister++;
  }
}

void PREDICTOR::TrackOtherInst(UINT32 PC, OpType opType, UINT32 branchTarget) {
  // This function is called for instructions which are not conditional
  // branches, just in case someone decides to design a predictor that uses
  // information from such instructions. We expect most contestants to leave
  // this function untouched.

  MONUnusedParameter(PC);
  MONUnusedParameter(opType);
  MONUnusedParameter(branchTarget);

  return;
}

UINT32 PREDICTOR::localIndex(UINT32 PC) {
  return PC % numEntries;
}

UINT32 PREDICTOR::tournamentIndex(UINT32 PC) {
  return localIndex(PC);
}

UINT32 PREDICTOR::gshareIndex(UINT32 PC) {
  return (PC ^ globalHistoryRegister) % numEntries;
}

enum ChosenPredictor PREDICTOR::tournamentPrediction(UINT32 PC) {
  UINT32 tournamentCounter = tournamentPredictionTable[localIndex(PC)];

  if (tournamentCounter > TOURNAMENT_CTR_MAX/2) {
    return GSHARE;
  }
  return LOCAL_PREDICTOR;
}

bool PREDICTOR::localPrediction(UINT32 PC) {
  UINT32 localCounter = localPredictionTable[localIndex(PC)];

  if (localCounter > LOCAL_PREDICTOR_CTR_MAX/2) {
    return TAKEN;
  }
  return NOT_TAKEN;
}

bool PREDICTOR::gsharePrediction(UINT32 PC) {
  UINT32 gshareCounter = gsharePredictionTable[gshareIndex(PC)];

  if (gshareCounter > GSHARE_CTR_MAX/2) {
    return TAKEN;
  }
  return NOT_TAKEN;
}
```

Bibliography

Hennessy, John L., and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.