# IS2202 - Computer Systems Architecture

## Lab 2

## Micro-architectural exploration

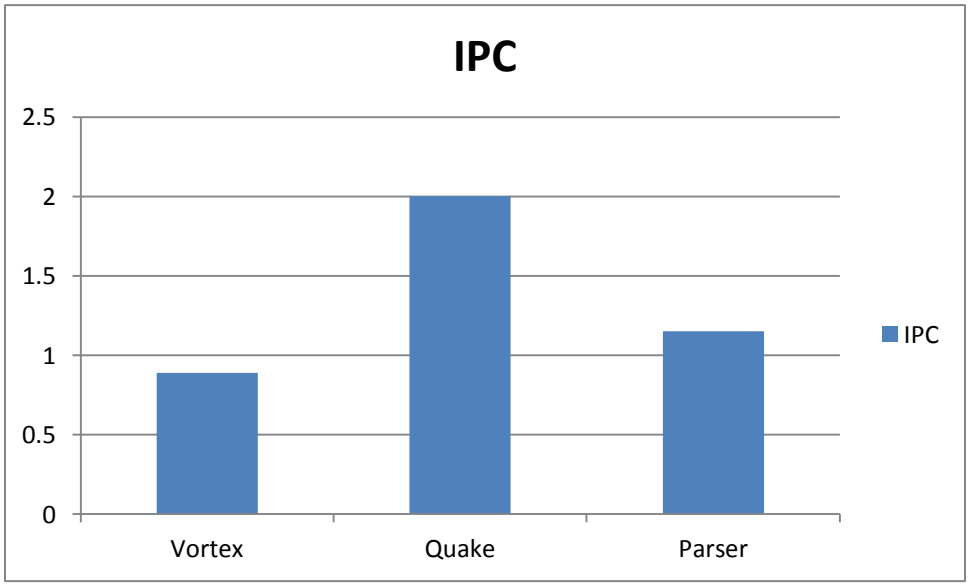
-Submitted by

Tamilselvan Shanmugam tamsha@kth.se

# 2. Directed Portion

### 2.3 Collecting IPC statistics using the MAI

| Benchmark | Cycles | Steps | IPC |
|---|---|---|---|
| Vortex | 9000000 | 8014599 | 0.89 |
| Quake | 9000000 | 17964555 | 2 |
| Parser | 9000000 | 10331390 | 1.15 |




From the simulated results, Quake benchmark surpassed other two benchmarks. Quake completes 2 instructions on average for every single clock cycle. Its quiet good compared to vortex and parser.

As we have seen from Lab-1 simulation results, vortex data and instruction hit ratios were so poor. That behavior influenced in IPC also. Vortex wasted so many cycles to access data & instruction, since they were not present in the cache. This is one of the reasons for worst IPC.

Quake is the best guy ☺ and vortex is the worst guy ☹

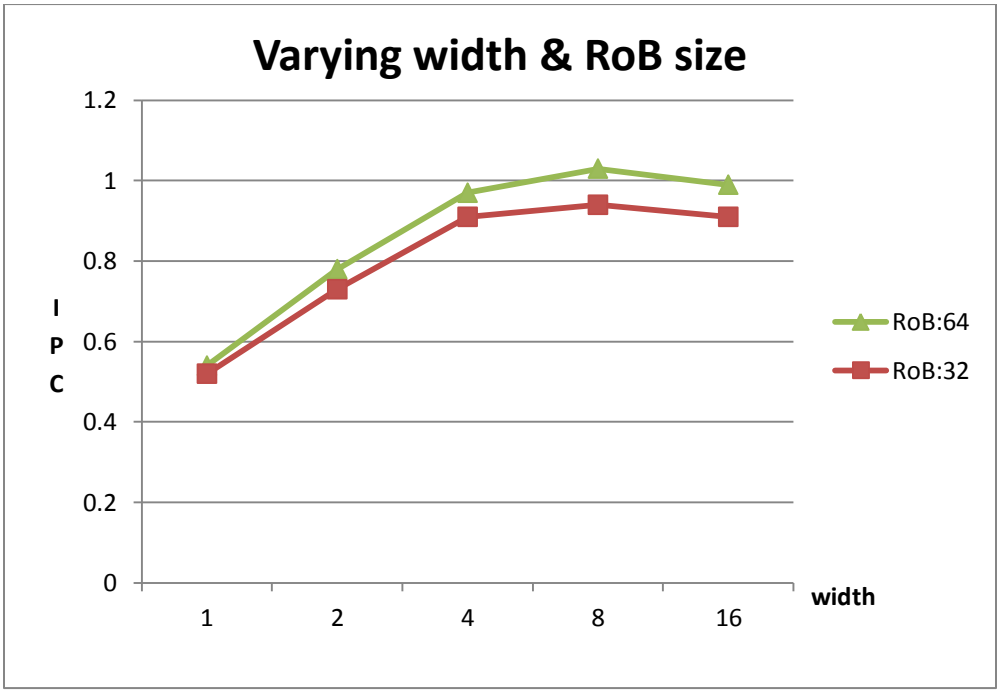
## 2.4 Collecting data about the effect of superscalar pipeline width on IPC

vortex benchmark, RoB Size = 32

| Width | Cycles | Steps | IPC |
|---|---|---|---|
| 1 | 9000000 | 4709600 | 0.52 |
| 2 | 9000000 | 6587157 | 0.73 |
| 4 | 9000000 | 8166696 | 0.91 |
| 8 | 9000000 | 8486077 | 0.94 |
| 16 | 9000000 | 8221357 | 0.91 |

vortex benchmark, RoB Size = 64

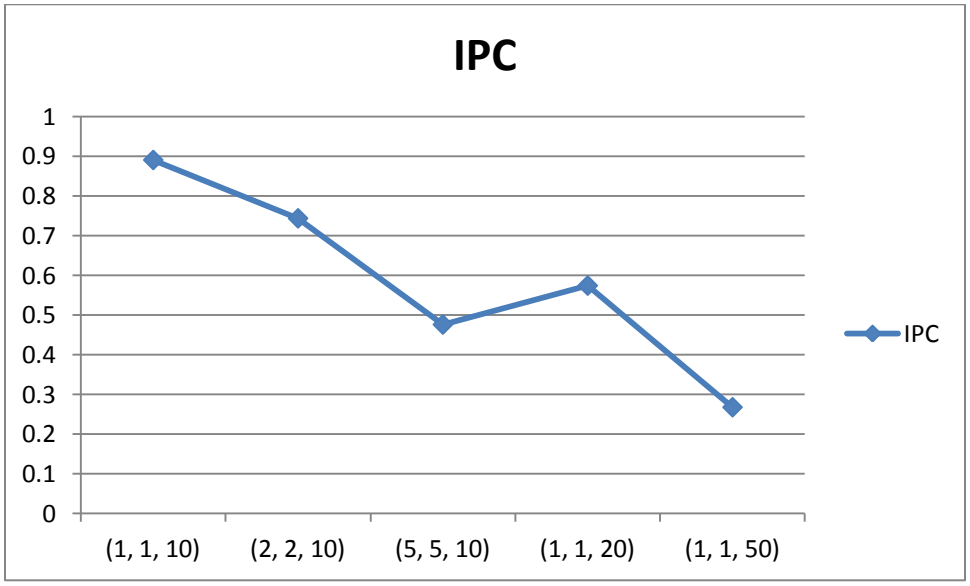
| Width | Cycles | Steps | IPC |
|---|---|---|---|
| 1 | 9000000 | 4855291 | 0.54 |
| 2 | 9000000 | 7064666 | 0.78 |
| 4 | 9000000 | 8702952 | 0.97 |
| 8 | 9000000 | 9269142 | 1.03 |
| 16 | 9000000 | 8929453 | 0.99 |



Increasing width of OoO & buffer size increases IPC count until a threshold value of width = 8. Beyond this value, IPC count started diminishing. This implies that the processor cannot complete higher number of instructions even it has the capacity to complete. Causes for this kind of behavior may be the previous instructions take more cycles to complete than usual or current instruction is data dependent on the previous instructions. In addition branching also affects the performance when scaling.

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

| Memory hierarchy | IPC |
|---|---|
| (1, 1, 10) | 0.891 |
| (2, 2, 10) | 0.743 |
| (5, 5, 10) | 0.476 |
| (1, 1, 20) | 0.574 |
| (1, 1, 50) | 0.268 |



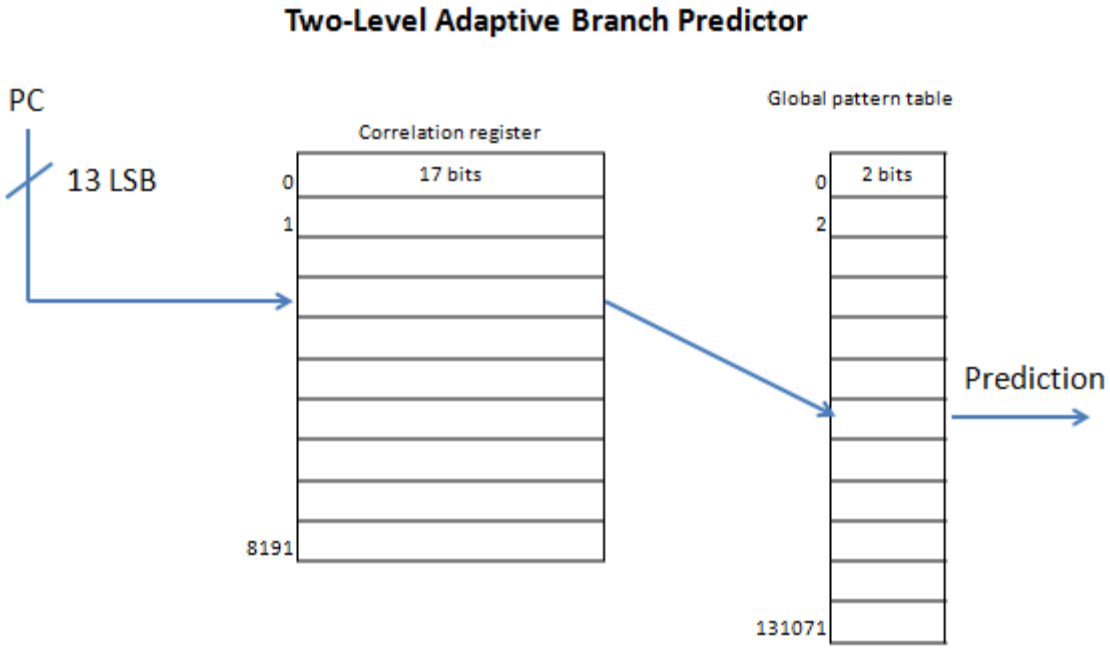
From the above graph, in general increasing latency decreases average IPC.

The first 3-set data (1,1,10 : 2,2,10, 5,5,10) demonstrate the effect of cache access latency versus IPC. Doubling the cache access time didn't decrease the IPC by half. Rather it decreased IPC by 15%. Increasing cache latency 5 times decreased IPC by 42%. This improvement is because of OoO hides the latencies by doing useful work.

Data sets (1,1,10 : 1,1,20 : 1,1,50) picks up the memory access latency to experiment. We can see that increasing latency doesn't decrease the performance linearly. OoO execution gives some performance boost up by hiding memory latencies.

# 3 Open-ended Portion

## 3.1 Branch predictor study



We have implemented "Two level adaptive" branch predictor which keeps track of branch history and global history pattern. 13 LSB bits of PC is used to index the correlation table which stores the past 17 branch decisions. This correlation table value is then indexed in global history pattern of 2-bit size, which predicts, will the branch taken or not for this kind of correlation pattern. Accuracy of prediction improves, as the correlation bits, number of PC bits and global pattern table size increases. Because they avoid aliasing, shorter history problems. Results are compared with the given Gshare predictor.

Total storage budget: 49kB
Total GPT counters: 2^17
Total GPT size = 2^17 * 2 bits/counter = 32kB
CRR size: 2^13 entries * 7 bits/entry = 17kB
Total Size = GPT size + CRR size

| | Mispredictions per 1K Instructions | |
|---|---|---|
| Trace | Gshare | Two Level Adaptive |
| SHORT-INT-1 | 7.347 | 5.552 |
| SHORT-FP-1 | 3.479 | 2.405 |
| LONG-SPEC2K6-03 | 5.658 | 4.854 |
| LONG-SPEC2K6-07 | 14.062 | 18.848 |
| LONG-SPEC2K6-11 | 3.929 | 0.978 |

**Source code:**

Filename: ***predictor.cc***

```
#include "predictor.h"
#define GPT_CTR_MAX  3
#define GPT_CTR_INIT 2
#define CRR_CTR_INIT 65535
#define CRR_CTR_MAX 131071
#define HIST_LEN   17
#define PC_AND 8191
#define CRR_AND 131071

PREDICTOR::PREDICTOR(void){
  historyLength    = HIST_LEN;
  numGptEntries    = (1<<HIST_LEN);
  numCrrEtries          = (1<<PC_LSB_TO_COMPARE);
  gpt = new UINT32[numGptEntries];
  crr = new UINT32[1<<PC_LSB_TO_COMPARE];

  UINT32 ii;
  for(ii=0; ii< numGptEntries; ii++){
    gpt[ii]=GPT_CTR_INIT;
  }

  for(ii=0; ii< numCrrEtries; ii++){
    crr[ii]=CRR_CTR_INIT;
  }
}

bool   PREDICTOR::GetPrediction(UINT32 PC){
  UINT32 gptIndex   = crr[PC&PC_AND] & CRR_AND;  // Index last
13 bits of PC
  UINT32 gptCounter = gpt[gptIndex];

  if(gptCounter > GPT_CTR_MAX/2){
    return TAKEN;
  }else{
    return NOT_TAKEN;
  }
}

void  PREDICTOR::UpdatePredictor(UINT32 PC, bool resolveDir,
bool predDir, UINT32 branchTarget){
  UINT32 gptIndex   = crr[PC&PC_AND] & CRR_AND;
  UINT32 gptCounter = gpt[gptIndex];

  // update the GPT
```

```
  if(resolveDir == TAKEN){
    gpt[gptIndex] = SatIncrement(gptCounter, GPT_CTR_MAX);
  }else{
    gpt[gptIndex] = SatDecrement(gptCounter);
  }

  // update the CRR
  crr[PC&PC_AND] = (gptIndex << 1);
  if(resolveDir == TAKEN){
       crr[PC&PC_AND]++;
  }
}

void    PREDICTOR::TrackOtherInst(UINT32 PC, OpType opType,
UINT32 branchTarget){
  return;
}
```

Filename: ***predictor.h***

```
#ifndef _PREDICTOR_H_
#define _PREDICTOR_H_
#define PC_LSB_TO_COMPARE 13
#include "utils.h"
#include "tracer.h"

class PREDICTOR{
 private:
  UINT32  *crr;           // correlation register table
  UINT32  *gpt;           // Global pattern table
  UINT32  historyLength;      // history length
  UINT32  numGptEntries;      // entries in gpt
  UINT32  numCrrEtries;       // No.of crr entries

 public:
  PREDICTOR(void);
  bool    GetPrediction(UINT32 PC);
  void    UpdatePredictor(UINT32 PC, bool resolveDir, bool
predDir, UINT32 branchTarget);
  void    TrackOtherInst(UINT32 PC, OpType opType, UINT32
branchTarget);
};
#endif
```

### 3.2 Create code that performs no better on an OoO machine

**How to crucify the <code>:**

1. **Data dependency**

   Include read after write dependencies so that each instruction depends on previous instruction. Also modify variables that will be used in next iteration. So loop unrolling will not work.

2. **Control dependency**

   Conditional branches inserts control hazard. Either processor has to wait or speculatively execute the branch instructions.

3. **Cache enemy code**

   Purposefully access the elements such a way that most cache miss occurs. Cache miss has to wait till the data fetched back again.

**Tested configuration**

Cache

Number of cache lines : 128
Cache line size : 32 bytes
Total cache size : 4 kbytes
Associativity : 4

Penalties

cache_cpu0->penalty_read =5
cache_cpu0->penalty_write =5
staller_cpu0->stall_time =50

| Out of Order | In order |
|---|---|
| config<br>ma_cpu0->fetches_per_cycle = 4<br>ma_cpu0->execute_per_cycle = 4<br>ma_cpu0->retires_per_cycle = 4<br>ma_cpu0->commits_per_cycle =4<br>cpu0->reorder_buffer_size =32 | config<br>ma_cpu0->fetches_per_cycle = 1<br>ma_cpu0->execute_per_cycle = 1<br>ma_cpu0->retires_per_cycle = 1<br>ma_cpu0->commits_per_cycle =1<br>cpu0->reorder_buffer_size =1 |
| IPC = 0.3259 | IPC = 0.1457 |

**Source code:**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

/* ripped from simics/src/inclue/magic_instruction.h */
#define MY_MAGIC(n) do {                                        \
        __asm__ __volatile__ ("sethi " #n ", %g0");             \
} while (0)

#define MAGIC_BREAKPOINT MY_MAGIC(0x40000)
#define ARRAY_SIZE 65536

int main(int argc, char* argv[]){
     double A[ARRAY_SIZE], B[ARRAY_SIZE], C[ARRAY_SIZE];
     int i,j,k;
     //fill arrays
       for(i = 0; i < ARRAY_SIZE; i++){
         A[i] = rand();
         B[i] = rand();
         C[i] = rand();
       }

       MAGIC_BREAKPOINT;
     while(1){
       for(i = 3; i < ARRAY_SIZE; i++){
            A[i] = A[i-3] + A[i-2] * A[i-1] + A[i+1];
            B[i] = A[i] * B[i-1] + B[i-2] + B[i-2] + B[i+1];
            C[i] = A[i+1] + B[i] + C[i-1] + C[i-2] + C[i-3] +
C[i+1];
```

```
                if(i%4==0){
                      for(j=6230; j<7255; j=j+33){
                            A[j-2] = B[j*3+i];
                            for(k=j*2;k<j*5;k=k+35){
                                  C[k-j] = C[k+j];
                                  B[k-3] = A[k]+C[k-j];
                            }
                      }
                }

          }
      }
}
```

RAW: Array calculation in first for loop has data dependency. Second instruction waits until A[i] is computed and available. This is holds true for third instruction also. C[i] depends on previous B[i] execution.

Avoid loop unrolling: A[i] is computed with adjacent values of A[i]. This adds more dependency of its own data, thereby trying to avoid loop unrolling.

Control dependency: "**if**" loop introduces control hazard. Processor can speculatively execute the taken path or not taken path and commits only after the branch is resolved.

Cache enemy code: Cache line size is 32 bytes. It can accommodate 4 double variables. In our code, nested for loop access elements that are not present in the same cache line. Bringing in data takes extra penalty cycles as configured. Out of order execution is successful to an extent in hiding latency. But it's not helpful beyond some point. We have utilized this fact to pull down IPC.