2nd Project Advanced Computer Architectures

João Ramiro and Miguel Pinho, Instituto Superior Técnico

Abstract—This project report compares the strong and weak points of the in-order and out-of-order CPU, trying to achieve a maximum and minimum speed-up by exploiting such characteristics to increase or decrease Instruction Level parallelism. This project will be using the gem5 simulator tool to do so.

Index Terms—Computer Architectures, Pipeline, in-order, out-of-order, ILP, IPC, gem5.

1 Introduction

In this project an out-of-order and in-order processors are compared, in regard to their functioning and performance. They are studied using the gem5 simulator which uses a model for each processor and simulates what would happen in a real processor. It takes into account pipeline stages, latencies, cache levels, and many other things. For the in-order processor the model used is the Minor CPU model, and for the out-of-order processor is the DerivO3CPU.

1.1 Processors Structure

The inner structure of each of the used cpu models was studied with the help of the config.ini files generated by the gem5 simulator.

Both system are single core processors and have the same memory hierarchy. They are connected to the same two-level-cache and to same memory bus and main memory. The L1 cache is divided into instruction and data, whereas the L2 cache is unified.

The processors use the same tournament branch predictor. The Branch Target Buffer (BTB) has 4096 entries, with a tag size of 16 bits. There is both a global predictor and, for each branch, a local predictor, and the decision between which to use for each case is selected based on the output of a third tournament predictor (which records which predictor was right more often).

Both processors also use the same x86 Instruction Set Architecture (ISA). All these similarities in the processing system are crucial for the validity of the comparison tests performed, as only the processing core itself is changed.

1.1.1 *Minor CPU model*

This MinorCPU in-order processor is divided into two instruction fetch stages, one decode stage and an execution stage (also responsible for the commit). Fetch1 is responsible for fetching lines of instructions from cache L1 and Fetch2 for decomposing them into instructions. Fetch2 also includes the branch predictor. Decode has an input width of 2 instructions, which are decomposed into micro-operations and sent, also up to 2 at a time, to the Execute stage, which consists of several independent and pipelined Functional Units (FU). Each cycle the Execute is able to issue up to to the FU 2 instructions (in order) and to also commit up to 2 instructions. A scoreboard structure controls the flow of the instructions and stalls the stages whenever required.

1.1.2 *DerivO3CPU model*

The DerivO3CPU out-of-order processor model is divided in Fetch, Decode, Rename, Execute and Commit stages. The issuing and write back are performed in the write back stage. This processor is able to fetch, decode, rename, issue, write back and commit up to 8 simultaneous instructions per cycle.

This processor model contains a Reorder Buffer with size 192, a physical register file (with the same number of integer, floating and vector registers, 256) and a FU pool with 9 different execution ports, consisted of FUs with different latencies that are pipelined (with the exception of integer and floating division operations).

1.2 Differences in execution

The main difference between an in-order processor and an out-of-order is that the latter uses dynamic instruction scheduling to try to execute instructions despite of their original order and only taking into account their dependencies. That is, whereas in the in-order an instruction is stalled while dependencies need to be solved, the out-of-order tries to fill those wasted cycles with other later instructions, exploiting instruction level parallelism (ILP).

Specifically, these two processor will handle differently these situations:

- **RAW hazards** - This is a true dependency so neither processor can continue the execution of that instruction. However, while the in-order stalls waiting for the value to be ready, out-of-order tries to execute other instructions
- **WAR hazards** - this hazard does not happen in an in-order processor, as the operands of an instruction are always read before a later instruction is executed. The same does not happen in the out-of-order, where an operand can be changed before a instruction can be executed. This is solved by register renaming.

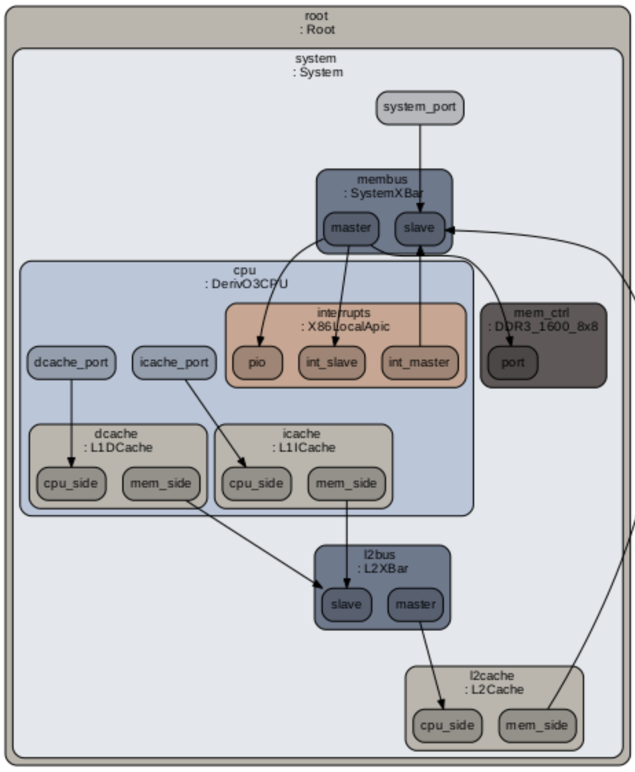

Fig. 1: Processing System for the out-of-order CPU (the in-order is identical)

- **WAW hazards** - in the in-order this is caused by the different latencies of the functional units, and a unit which finishes before time must stall and delay its writing. For the out-of-order this hazard is also a consequence of the instruction re-ordering, but is solved using renaming.
- **Control Hazards** - in case of a wrong speculative execution, the out-of-order will, in general, have allowed the execution of a much higher number instructions before the correct branch result is know, which have to be canceled. The benefit of the out-of-order execution is therefore lost in such a case.
- **Structural hazards** - In both cases they result in an execution stall. In the in-order this can be caused by the functional unit for some operation type being full. In the out-of-order processor this corresponds to the reservations stations of the corresponding execution ports being full. In the out-of-order a structural hazard can also happen when the Reorder Buffer (ROB) is full or the Physical register file (for some data type) is depleted. That is, there is a structural limit to the window of fetched instructions.

1.3 Exploiting performance differences

For the proposed objectives it is crucial to understand theoretically when the speed-up of the out-of-order processor in relation to in-order is maximum and minimum. That is, when dynamic instruction re-ordering is most advantageous and when not.

1.3.1 Maximizing speed-up

The out-of-order execution is advantageous when there are enough dependencies on the result of operations with high latency, typically floating operations (multiplications and division especially) and memory accesses, followed by blocks of instructions that can be executed in parallel (without the same dependencies). The in-order should be made to block frequently its execution, while the out-of-order is allowed to continue as long as it has enough resources to do so. That is, the code should have a high degree of instruction level parallelism combined with well placed dependencies. Preferably the instructions should use different types of FU unit, so that the available resources in the out-of-order are used to the fullest.

1.3.2 Minimizing speed-up

Three situations were identified when the out-of-order execution loses its advantage, tending to a execution time equal to the out-of-order (speed-up of 1). Firstly, when the whole window of instructions loaded to the processor (the ROB is full) there are too many dependencies so that very few instructions can be executed each time. However in practice, it is difficult to avoid even a minimal degree of ILP in a window of 192 instructions. Secondly, when there is a very high percentage of incorrect branch predictions. In this case there is no advantage of executing the later instructions in advance, as they will have to be reverted when the branch result is committed and the execution will restart from then. The problem is that it is not easy to systematically fool a complex branch predictor (tournament branch predictor) without fully understanding its inner workings and without having control of the branch instruction addresses at the assembly code level. Lastly, when the accesses to the memory are so frequent and the caches miss rate is so high that the processor is waiting for memory accesses most of time. The performance bottleneck becomes the memory bus itself, and as the memory hierarchy of both processors is the same, the performances will be identical.

2 Solution

2.1 Program A

The purpose of program A is to maximize the speed up, and two different c programs are presented to achieve this objective. The pseudo-code for program A.1 can be seen on listing 1.

Listing 1: pseudocode for program A.1

```
f0...f9 = 3.1415
vector[0 to 6] = 7.2

for 0 to NUM_ITER
    i1 = 25
    vector[1] = 2.0+3.0*f0;
    vector[1] = 2.0+3.0*f1;
    i6 = i1 ^ 3;
    i2 = i6 - 2;
    vector[2] = 2.0+3.0*f2;
    vector[2] = 2.0+3.0*f3;
    i2 = i6 * 23;
    vector[3] = 2.0+3.0*f4;
    vector[3] = 2.0+3.0*f5;
    i3 = i2 / i1;
    vector[4] = 2.0+3.0*f6;
    vector[4] = 2.0+3.0*f7;
    i4 = i3 - i1;
    vector[5] = 2.0+3.0*f8;
    vector[5] = 2.0+3.0*f1;
    i5 = 3212 % i4;
    vector[6] = 2.0+3.0*j;
    vector[6] = 2.0+3.0*f1/vector[2] - vector[1];
  end
```

This program consists of NUM_ITER iterations with minimal inter-loop dependencies, where each iteration has a mixture of different operation types: floating point, integer and memory (mostly to cache L1, has there are very few addresses); additions, multiplications (and power) and divisions (floating point and integer). There are some dependencies inside the cycle, such as WAW between 2 consecutive floating memory writes and RAW between all integer operations.

Another code that demonstrated a very good speed-up in practice is presented in 2

Listing 2: pseudocode for program A.2

```
for j from 0.0 to NUM_ITER_OUT step 1.0
  for i from 0 to NUM_ITER_IN

  end
end
```

Listing 2 consists in two nested loops, without any code inside. In outer loop has more iterations and uses a float for the loop control. The inner loop has few iterations and uses an integer to iterate. The loop variables were explicitly set to be stored in registers, by using the keyword `register` to warn the compiler. The performance of this second very simple program was unexpected and will try to be explained in the results.

2.2 Program B

The purpose of program B is to achieve a the minimum speed up possible. The pseudo-code can be seen on listing 3.

Listing 3: pseudocode for program B

```
   vetor1[0 to NUM_ITER] = 3.0
   vetor2[0 to NUM_ITER] = 3.0
   vetor3[0 to NUM_ITER] = 3.0
   vetor4[0 to NUM_ITER] = 3.0

  for i from 0 to NUM_ITER
    f0 = vetor1[i];
    f1 = vetor2[i];
    f2 = vetor3[i];
    f3 = vvetor[i];
  end
```

In this listing first 4 floating point vectors were initialized. Then in each loop iteration an element of each vector is read and put on a float variable. The purpose of this code is to maximize the accesses to the memory, so that most of the time the execution is stalled waiting for the memory data.

3 Experimental Methods

The described programs were run in the gem5 simulator (build 20/01/2015) in a machine with operating system CentOS 7.2.1511, CPU Intel® Core™ i7-6700k (4.0GHz), RAM G.Skill DDR4 CL15 4x8GB (2.4GHz) and motherboard Asus Z170-K.

The processing systems described in the Section 1.1 were configured in the gem5 simulator. The code was compiled with the O0 flag, to avoid unintended optimizations. The total simulated time and files with statistics were obtained as output from the simulator.

4 Experimental Results

The results obtained for Program A.1 are presented in Table 1, showing that the out-of-order processor ran 10.70 times faster, and were obtained with 2×10^5 iterations. Care was taken to increase the number of iterations until that speed-up stabilized, for this and the other programs. This is because there is an execution overhead that is not related to the program itself, but is due to initialization code, which should be minimized in comparison to the section being compared. Equation 1 shows how the speed up is calculated,

$$speedup = \frac{\Delta_1 + \Omega_1}{\Delta_2 + \Omega_2}. \tag{1}$$

The Δ_1 and Δ_2 are the execution times of the code presented, for in-order and out-of-order respectively, and Ω_1 and Ω_2 are the respective overheads. If $\Delta_1 \gg \Omega_1$ and $\Delta_1 \gg \Omega_1$, we have $speedup \approx \frac{\Delta_1}{\Delta_2}$, as intended.

TABLE 1: Program A.1 statistics

Processor	timer [ms]	speed-up
in-order	113.9	
out-of-order	10.64	10.70

According to the statistics file, the Instructions per Cycle (IPC) for the in-order and out-of-order processors were respectively 0.250 and 2.67. This shows that the objective of stalling the execution of the first model with enough dependencies, while allowing the out-of-order to extract ILP, was achieved to some extent.

The executed operations are more or less 20 % for memory, 40 % for integer and 40 % for floating, which shows some degree of utilization of the different FUs. However, as discussed with the professors and other groups, there seems to be a error in the counters for the floating operations, as all are classified as additions. The high percentage of floating operations is important for objective A, as the higher latency of this operation highlights the out-of-order advantage.

While tinkering with the code, it was found that the time difference for the out-of-order execution, with and without the integer operations in the cycle is very low, which shows they are mostly hidden under the floating point latency, whereas they increased the time in the in-order. In this case, adding more than one floating division decreased the speed-up, probably because its latency is high in comparison to this cycle's size. In the configurations files it is specified that the floating division unit is not pipelined and has a latency of 12 cycles.

We still think there is a good margin for increase in the IPC of the out-of-order, as it is still quite far from the ideal 8 allowed by the width of the processor. One bottleneck still left to be solved seems to be, according to the statistics file, that the floating additions fail to find an available FU too often (as pointed out, there is likely a bug, and this refers to floating point in general), that is, there seems to still be space for improvement in the usage of the processors resources.

A higher speed-up, of 12.48, was obtained with program A.2, whose results are presented in Table 2. The number of iterations selected for the outer loop were 10^6 and for the inner loop 10. For this result the usage of the `register` keyword for both variables of this program is crucial. This gives an indication to the compiler to use registers for these variables whenever possible. The integer loop being the inner one is also very important, as these operations have very low latency.

TABLE 2: Program A.2 statistics

Processor	timer [ms]	speed-up
in-order	200.1	
out-of-order	16.035	12.48

The higher speed-up of this program is harder to explain, as it has a lower out-of-order IPC, 2.49, and is achieved due to an even lower in-order IPC, 0.200. The code is reduced to the incrementation and control of the cycle. It is noteworthy that the branch predict rate is very high, which shows that it is capable of predicting the periodic fail of the internal cycle. The ”actual” code is therefore sequences of dependent (to start, by RAW hazards) integer operations, periodically interleaved with one floating addition (the floating operations also dependent between them). The increase in the stalling in the in-order is explainable by this high degree of successive dependencies, mostly in the integer additions (although they have very low latency), and seems to be the biggest factor in this speed-up. What we found more interesting is that the out-of-order can handle quite well the high interdependence between the integer operations. A lot of them are issued to the reservation stations each time, but they mostly still execute in order (RAW). Probably the fact that the next one can snoop the next value very fast (maybe directly?) and that the latency of these operations is only one cycle, is crucial to keep a good throughput. We think that in this second program it is the floating point operations that end up hidden under the integer operations latency. All said, we still find the IPC of the out-of-order strangely high.

For the last program, the speed-up was very low, as requested, having been achieved the value of 1.067 (Table 3). As already explained, the program B exploited the fact that both processors have the same caches and memory structure, putting all the pressure in the memory.

Initially, four vectors are created. In every iteration the 4 vectors are read from cache and if they don't generate a hit they fetch the data from lower memory levels. Dependencies were avoided in the solution, as they tend to stall the in-order but not the out-of-order processor, increasing the speed-up.

TABLE 3: Program B statistics

Processor	timer [ms]	speed-up
in-order	69.44	
out-of-order	65.09	1.067

Analyzing the execution statistics, it was verified that the 93.85 % of the time 1 or less commit is performed (69.26 % none is) and we have a quite low number of Instructions per Cycle (IPC), 0.292. This shows that almost no ILP is explored, nullifying the out-of-order advantage. As the addresses are accessed in order the miss rate is quite low, 1.93%, but even then the width of the memory bus is clearly clogged, as the miss latency is enormous (2.95×10^{10}). As for the operations, 60 % are memory and 40 % are integer, this second, which corresponds to the loop control and address calculation, more significant than expected. The high parallelization of this integer part, which is likely the biggest responsible for the still tangential speed-up, is not near enough to mask all the latency in the data access.

5 Conclusion

With this small project we understood that gem5 is a very powerful tool if used correctly. We now also better understand the main benefits of the out-of-order processor relative to the in-order. Although there is still room for improvement in the proposed solutions, this project provided the opportunity to verify the learned theoretical inner workings and performance of a super-scalar in a more realistic setting.

We also concluded that these kind of programs, to exploit the micro-architecture of a processor, past some point must be done in assembly or machine code level, as a higher degree of control over the exact instructions and their order is needed.

As for further work which might have been interesting, it would have been good to also test and benchmark solutions with more realistic code, preferably a variety of different applications, to test how the out-of-order performs in a more realistic setting.

References

[1] Gem5's Minor CPU
`www.gem5.org/docs/html/minor.html`
[2] Gem5's two level cache configuration
`learning.gem5.org/book/part1/cache_config.html`
[3] The gem5 simulator
`pages.cs.wisc.edu/~markhill/cs757/Spring2012/includes/isca_pres_2011.pdf`
[4] Anastasiia Butko et all,
Design Exploration For Next GenerationHigh-Performance Manycore On-chip Systems:Application To big.LITTLE Architectures