Benchmarks in gem5 of High Performance Computing Architectures

G03: Afonso Alemão (96135), Rui Daniel (96317), Tiago Lourinho (96327)

***Abstract*—We analyze the architecture and the connection with the memory of out-of-order and in-order ARM processors and we use it to create specific benchmarks that stress specific components for the out-of-order: issue stalls, cache misses, and branch misprediction, comparing them with the baseline developed. The in-order processor was used to compare results. We use the gem5 simulator with realistic processor configurations. All the stress tests result in a decrease in performance and in speedups lower than 1 for the out-of-order processor. The baseline reaches 5.2 of IPC but has a poor performance in the in-order due to differences in the architecture. For the issue stalls, we get an average of 0.267 instructions issued per cycle. For the cache misses, we create 3 benchmarks capable of obtaining: 0.99 L1D and 0.02 L2 miss-rate; 0.04 L1D and 1.00 L2 miss-rate; 0.99 L1D and 1.00 L2 miss-rate. Finally, for the final benchmark we reached 0.97 of branch misprediction ratio. The IPC of the out-of-order processor is almost always higher than the IPC of the in-order processor, due to better utilization of available resources.**

***Index Terms*—Benchmarking, Microarchitecture, C language, Assembly, Student experiments**

I. Introduction & Architecture Overview

With the increase in computing power, the importance of testing the performance of the microarchitectures grows in a similar fashion, especially the components that have a big impact on performance. On that note, this work aims to produce a set of benchmarks, using the gem5 simulator [2], to stress 3 different important areas: issue stage, branch predictor, and L1 and L2 data caches. As multiple microarchitectures exist, the work will be focused on Arm A15 (out-of-order) used as the main development and test focus, and on Arm A7 (in-order) used to compare the benchmark results with the out-of-order (OoO) microarchitecture.

A. Arm A15 (out-of-order)

Recurring to the `config.ini` file, the following summary of the OoO architecture was produced and will be referenced later as justifications for the benchmarks developed.

- **Pipeline Stages:** Fetch, Decode, Issue, Dispatch and Commit stages have an instruction width of 8.
- **Functional Units:** Displayed in Table I.
- **Branch Predictor:** Tournament branch predictor with 4096 entries in the BTB and a RAS of size 16.
- **L1 data cache:** 2-way associative cache with a total of 64 KB and 64 bytes per line. Implements a write back/write allocate policies. Additionally, it has an LRU data replacement policy.
- **L2 cache:** 8-way associative cache with a total of 256 KB and 64 bytes per line. Implements a write back/write allocate policies. It also has a LRU data replacement policy.

TABLE I
FUs available ARM A15

Number	FU Name	Operations supported	Latency [Cycles]
6	FUList0	IntAlu	1
2	FUList1	IntMult	3
		IntDiv	20
4	FUList2	FloatAdd	2
		FloatCmp	2
		FloatCvt	2
2	FUList3	FloatMult	4
		FloatMultAcc	5
		FloatMisc	3
		FloatDiv	12
		FloatSqrt	24
4	FUList5	SimdAdd	1
		SimdFloatAlu	1
		...	...
		SimdFloatReduceAdd	1
		SimdFloatReduceCmp	1
1	FUList6	SimdPredAlu	1
4	FUList8	MemRead	1
		MemWrite	1
		FloatMemRead	1
		FloatMemWrite	1
1	FUList9	IprAccess	3

TABLE II
FUs available ARM A7

FU Name	Operations supported	Operation Latency [Cycles]
funcUnits0	IntAlu	3
funcUnits1	IntAlu	3
funcUnits2	IntMult	3
funcUnits3	IntDiv	9
funcUnits4	FloatAdd	6
	FloatMisc	
	FloatSqrt	
	...	
	SimdShaSigma2	
	SimdShaSigma3	
funcUnits5	SimdPredAlu	3
funcUnits6	MemRead	1
	MemWrite	
	FloatMemRead	
	FloatMemWrite	
funcUnits7	IprAccess	1
	InstPrefetch	

B. Arm A7 (in-order)

Recurring to the `config.ini` file, the following summary of the in-order architecture was produced.

- **Functional Units:** Displayed in Table II.
- **Branch Predictor:** Tournament branch predictor with 4096 entries in the BTB and a RAS of size 16.
- **L1 data cache:** 2-way associative cache with a total of 64 KB and 64 bytes per line. Implements a write back/write allocate policies. Additionally, it has a LRU data replacement policy.
- **L2 cache:** 8-way associative cache with a total of 256 KB and 64 bytes per line. Implements a write back/write allocate policies. It also has a LRU data replacement policy.

II. Solution

In the following section, the benchmarks developed will be presented and justified. For each one, we present the pseudo-code that achieved the best results, while referring to other tested methods and the reason for the worse result.

A. Baseline

The benchmark used for the baseline can be seen in Algorithm 1 and is based on the information regarding the available functional units (Table I).

Considering that the issue width is 8, the baseline uses mainly the 6 IntAlu and the 2 FloatMisc by doing basic attributions to the variables created (immediate moves). It should be noted that only 5 integers were used, instead of 6 to account for the update of the loop variable. Finally, the "basic block" of attributions was repeated to mitigate the negative impact of the loop in the IPC (for example, when fetching the 8 coalesced instructions).

Algorithm 1 Baseline Solution

int $i_1, ..., i_5$; **float** f_1, f_2
for `N iterations` **do**
  $i_{1,2,3,4,5} \leftarrow 1$
  $f_{1,2} \leftarrow 1.1$
  ... {Repeat the "basic block" above 32 times}
end for

Other variations were also tested, including:

- **Storing the variables in registers:** Discarded as yielded an experimental lower IPC, explained by the decrease in the usage of the MemRead / MemWrite functional units.
- **Performing another type of operations, for example, float additions:** Discarded as yielded an experimental lower IPC, explained by the increase of the latency, as not only the operations were more costly, but also required the load of another operand.

B. Issue Stalls

The benchmark used for generating stalls during issue can be seen in Algorithm 2 and is based on renaming and on the lack of ROB entries.

Algorithm 2 Issue Stalls Solution

int $x = 1, y = 20, z = 10, k = 3$
for `N iterations` **do**
  $y \leftarrow x/z$
  $x \leftarrow z/y$
  $x \leftarrow k/z$
  $x \leftarrow k/z$
  $x \leftarrow z/y$
  $y \leftarrow x/z$
  ... {Repeat the "basic block" above 32 times}
end for

Given that WAR and WAW hazards can be solved by renaming the dependent register, OoO processors use the ROB ID, instead of the ID of the register. So the association between logical and physical registers is done by assigning a new physical register to the destination in the Register Alias Table (RAT). This process may result in structural hazards.

In order to cause these hazards our algorithm has a lot of WAW and WAR hazards. We also highly occupy the FU unit capable of performing IntDiv giving it excessive work, thus causing stalls. There are only 2 FU units of this kind available and most of our instructions issue that unit, so as the IntDiv operation has 20 latency cycles, the issued instructions have to wait for the FU unit to be available.

Finally, the "basic block" of attributions was repeated to mitigate the negative impact of the loop control instructions that do not contribute to the WAW and WAR hazards nor to the saturation of any of the FU units.

Other variations were also tested, including:

- **Square root operations:** We tried to simultaneously saturate the FU unit that performs FloatSqrt, but this caused the number of issue stalls to decrease. This happens because the two FUs that perform FloatSqrt and IntDiv would decrease their rate of instructions to process.

C. Branch Miss-Prediction

The benchmark that got the best results can be seen in Algorithm 3. Given that the branch predictor assumes that a newly seen branch is taken at first, a brute force approach was used to exploit this. By not using a loop, every branch is a new one so, if the branch is not taken, the predictor will fail every time (not considering the code ran outside the scope of the `main` function).

Algorithm 3 Branch Miss-Prediction Solution

int $miss \leftarrow 0$ {The branch predictor always guesses taken at first.}
if miss **then**
  ...
end if
... {Repeat the "basic block" above 32768 times}

Other benchmarks were also tested, including:

- **Random number**: A local version of the Xorshift algorithm [3] and Linear Feedback Shift Register [1] were used to generate a pseudo-random number and perform a branch depending on its parity. However, as the branch only had 2 outcomes (taken or not), the predictor would guess right about 50% of the time.
- **Array of function pointers:** Creating an array of pointers to functions previously defined and generating a random number to index the array, thus calling a random function and generating a branch miss prediction most of the time (considering a decent number of destiny functions). The problem was that the return address was stored in the RAS (of size 16), so even if the first branch was incorrectly predicted, the return branch was always right, achieving a branch misprediction rate of slightly less than 50%.
- **Using `goto` keyword:** By using the `goto` keyword in C, it was possible to branch to a random label (by indexing an array of label pointers, similar to what was done in the previous method). As the labels needed to be manually inserted in the C code, a python script was created to automate the process. This strategy had the advantage of not including the RAS, thus mispredicting every branch (for a considerable number of labels). However, the problem found was related to the compiler as it didn't map exactly the C code, adding non-conditional branches that were always correctly predicted, thus compensating the previous misprediction, achieving once again a branch misprediction rate of slightly less than 50%.

Note: The methods referred above can also be found in the delivered zip.

D. Cache Misses

The benchmark used for causing misses in both L1D and L2 cache can be seen in Algorithm 4.

Our strategy was to generate L1D and L2 cache misses due to a non-contiguous memory access pattern. Specifically, the algorithm reads and writes to elements in two integer arrays, `A` and `B`, with a stride of 16. This results in each access being to a different 64-byte cache line, causing in both L1D and L2, compulsory and capacity misses, i.e, at the beginning misses are caused due to the first time referencing certain blocks,

Algorithm 4 Cache Misses Solution For Both L1D and L2

int $A\,[65536]$, $B\,[65536]$ {L2 can only store a maximum of 65536 integers; 16 is the number of integers that fit in a cache line}

for `j = 0 ; j < N; j += 16` **do**
  $B\,[j \mod 65536] \leftarrow A\,[j \mod 65535]$
end for

but then this continues to happen since some blocks that are loaded, end up being discarded before their next reference. The LRU policy ensures that the least recently used cache line is the first to be evicted when a cache miss occurs. Both L1D and L2 implement write back/write allocate policies, which can exacerbate the problem as writes to a cache line may cause its eviction and reload from main memory, causing further misses. This algorithm generates writing misses in `B` and load misses in `A`. These misses end up overwriting each other, leading to cache trashing. Given the size of the arrays, the L1D cache is unable to hold all of the data the algorithm needs, leading to cache misses.

Similarly, due to the non-contiguous memory access pattern of the algorithm and the fact that the working set of the program is two times bigger than the L2 working set (2 arrays, each with space equal to the number of integers that L2 can store), the cache lines have to be evicted and replaced frequently, leading to L2 cache misses.

Two additional algorithms have been developed. The first is designed to produce hits in the L1D while inducing misses in the L2. This is accomplished by utilizing the same cycle outlined in Algorithm 4, but with the addition of a block of code within the loop that performs mathematical operations using only three variables: k, v, and w. In the first iteration, these operations within the loop will generate cache misses in both L1D and L2. Subsequent iterations will result in only L1D hits because of the LRU policy. However, the first line of code inside this loop will continue to consistently generate a miss on both caches in every iteration, as previously explained.

In order to produce misses in L1D while generating hits in L2, another algorithm was developed, where a working set larger than the L1D working set but smaller than the L2 working set is used. This follows the same logic as Algorithm 4, but with the size of the arrays `A` and `B` reduced to a value greater than the number of integers that can be stored in the L1D, yet less than the integers that can be stored in L2. As a result, the first few iterations will result in compulsory misses since the blocks are being accessed for the first time. However, due to the limited space available in the L1D, subsequent iterations will also generate cache misses while the L2 won't find this issue.

III. Experimental Results

In this section, we report the experimental results obtained using the latest version of gem5 (version 22.1.0.0), using the ArmO3CPU (OoO) and ArmMinorCPU (in order).

We used the compilation flag $-O0$ with the objective of no compiler optimizations. For each benchmark, we analyze the two models and explain the obtained speedups.

TABLE III
Baseline Results

Architecture	Instructions Issued per Cycle	Branch Miss-Prediction	D-L1 Miss Rate	L2 Miss Rate	IPC
Arm A15	5.205	0.003	0.000	0.980	**5.196**
Arm A7	-	-	0.000	0.960	**0.583**

The main advantage of OoO processors is their ability to exploit instruction-level parallelism (ILP) by executing instructions out of order, which can lead to improved performance.

In-order processors overlap the execution of multiple instructions in the pipeline. However, their performance can be affected by hazards that can cause pipeline stalls. To mitigate these problems, they use techniques such as forwarding and bypassing.

So, as expected, we can verify in the results described below that the IPC of the OoO processor is almost always higher than the IPC of the in-order processor.

The speedups reported in the Tables are obtained by comparing the program IPC against the IPC obtained with that processor in the baseline. For the metrics not available in the `stats.txt` file, it is placed − in the Tables.

A. Baseline

In the baseline benchmark, we aim to obtain maximum performance, measured in IPC. The results obtained are reported in Table III.

We obtain a value of 5.196 for the OoO processor. As the issue width is 8, ideally, the IPC of the baseline would be the same. The loop control instructions and the setup instructions of the program are in conflict with the goal of ideally utilizing the FU units. Furthermore, there are only 6 FU units to perform IntAlu operations ($6 < 8$), and that is the reason that we also use the FloatMisc operation in order to increase the instructions throughput. However, the FloatMisc operation has a latency higher than 1 cycle, compromising the goal of 8 completed instructions per cycle. Compared with the IPC obtained for the in-order processor, we obtained an IPC of 8.9 times higher for the OoO processor, as expected.

Our benchmark is designed in order to improve the IPC of the OoO processor. However, the in-order processor architecture is slightly different, as we can see in Tables I and II. That way, as there is less FU units to perform IntAlu operations (2 in in-order and 6 in OoO) and the latency of that operation is higher (3 cycles in in-order and 1 cycle in OoO), for the in-order processor the IPC will not be ideal. The ideal IPC for the in-order processor would be the width of the decode and execute unit that is equal to 2.

It should also be noted that the high miss rate for the L2 cache is due to the program setup, as the data used in the `main` will remain in the L1D, as seen by its low miss rate.

B. Issue Stalls

In the issue stalls benchmark we aim to generate stalls during the issue stage. The results obtained are reported in Table IV. We obtain for the average instructions issued per cycle a much smaller value than for the baseline, which is reflected in a much lower IPC.

TABLE IV
ISSUE STALLS RESULTS

Architecture	Instructions Issued per Cycle	IPC	Speedup
Arm A15	**0.267**	0.264	0.051
Arm A7	-	0.310	0.532

Ideally, the benchmark would reach around 0 instructions issued per cycle, however, some factors contribute against it. The loop control instructions do not contribute to the WAW and WAR hazards and also decrease the saturation of the FU units that we are over-utilizing. Furthermore, reservation stations and ROB entries have a finite number and the latency of operations is not infinite, so it is not possible to reach an infinite number of instructions trying to saturate a FU during the issue phase of the pipeline.

The issue stalls happen due to structural hazards like lack of ROB entries in the renaming process that is triggered by WAR and WAW hazards. So, they highly affect the program performance, resulting in a decrease of the IPC for both OoO and in-order processors: the speedups are lower than 1.

C. Branch Miss-Prediction Stalls

In the branch miss-prediction stalls benchmark we aim to generate stalls in the branch prediction. The results obtained are reported in Table V, including the benchmark with the best results (brute force) and the other also tested methods. We would ideally obtain a branch misprediction of 1. For the brute force benchmark, we obtain almost this ideal result. The main reason for the result not being ideal is the correctly predicted branches in the program setup. The other tests got about 0.5 misprediction ratio, as explained in the section II-C.

In general, branch misprediction leads to a pipeline flush and a delay in instruction execution. This delay can significantly reduce the IPC, as the processor has to wait for the correct instruction to be fetched and executed. For an OoO processor, if a branch is mispredicted all non-committed ROB entries are removed and that causes a huge overhead in program performance because implies that useless work was performed, occupying the machine resources. So the execution continues only after obtaining the correct branch address in the WB stage. For an in-order processor, the pipeline is simple and follows a strict order of execution. So a branch misprediction can result in a pipeline stall that can also lead to a significant decrease in IPC because the processor has to wait for the branch to be resolved before proceeding with other instructions. We also can observe that for some cases the speedup of the in-order processor is higher than 1 because this processor has a poor performance in the baseline, as we verified in section III-A.

D. Cache Misses

The results obtained for the previously presented cache misses benchmarks are reported in Table VI and for all of them, our goal was almost ideally reached.

In our approach, we focus on capacity misses and compulsory misses. However, the loads and stores of the program setup and the loop control may cause misses and hits in the caches, affecting the results.

TABLE V
BRANCH MISS-PREDICTION RESULTS

Benchmark	Architecture	Branch Miss-Prediction	IPC	Speedup
Brute Force	Arm A15	**0.973**	0.121	0.023
	Arm A7	-	0.081	0.138
Function Calls	Arm A15	0.457	0.726	0.140
	Arm A7	-	0.773	1.326
Goto	Arm A15	0.438	0.789	0.152
	Arm A7	-	0.698	1.197
Random	Arm A15	0.509	1.479	0.285
	Arm A7	-	0.876	1.503

TABLE VI
CACHE MISSES RESULTS

Benchmark	Architecture	D-L1 Miss Rate	L2 Miss Rate	IPC	Speedup
L1D and L2 Misses	Arm A15	**0.986**	**1.000**	0.326	0.063
	Arm A7	**0.986**	**1.000**	0.119	0.204
L1D Hits and L2 Misses	Arm A15	**0.038**	**1.000**	0.626	0.121
	Arm A7	**0.022**	**1.000**	0.460	0.788
L1D Misses and L2 Hits	Arm A15	**0.986**	**0.021**	1.517	0.292
	Arm A7	**0.985**	**0.021**	0.520	0.892

Caches are able to reduce memory access times. When a miss occurs, the program needs to access the next level of memory, spending more time. This will cause a higher number of stalls because the processing of the instruction needs to wait for its operands. Cache misses require data to be fetched, which is slower than accessing data from the caches. That way, the IPC of the program is reduced. So, the performance is highly affected, which results in a really low speedup.

The miss rates are similar for both in-order and OoO processors, but the IPC is higher for the OoO. The speedup is higher for the in-order processor because the baseline in this case has poor performance as we described in section III-A.

IV. CONCLUSIONS

We used the processor description to create specific benchmarks that stress specific components for the OoO. The baseline for the OoO reaches 5.2 of IPC. For the issue stalls, we get 0.27 IPC. Through the sizes of the L1D and L2 we exploit capacity and compulsory misses by creating 3 stress tests and the results are near ideal. For the final benchmark, we reached 0.97 branch miss-prediction.

In conclusion, the IPC of the OoO processor is almost always higher than the IPC of the in-order processor, due to better utilization of available resources. However, in the specific benchmarks their ratio of IPC is lower than in the baseline, making their performance comparable.

Finally, all the objectives were fulfilled as all the stress tests resulted in a decrease in performance and in speedups lower than 1 for the OoO processor.

REFERENCES

[1] *Linear-feedback shift register*. Jan. 2023. URL: https://en.wikipedia.org/wiki/Linear-feedback_shift_register.

[2] Jason Lowe-Power et al. “The gem5 Simulator: Version 20.0+”. In: *ArXiv* abs/2007.03152 (2020).

[3] George Marsaglia. “Xorshift RNGs”. In: *Journal of Statistical Software* 8.14 (2003), pp. 1–6. DOI: 10.18637/jss.v008.i14.