High-Performance Computing Architectures Benchmarking for in-order and out-of-order CPU Architectures

Group 14 - João Rodrigues, Martim Bento and Rodrigo Coimbra
Under the orientation of: Prof. Nuno Roma

***Abstract*—In this lab assignment, we analyse in-order and out-of-order processor architectures and their connection with the memory subsystem. To achieve this, we develop C programs that perform different stress tests to analyse how some different design parameters affect them. Both processors are simulated with the state-of-the-art gem5 simulator, with realistic processor configurations.**

First, two benchmarks, *BenchMax* and *BenchMin*, are created to maximize and minimize, respectively, the differences in execution time between both processors. Both goals are achieved through various hardware exploits, resulting in a maximum speedup of $8.827\times$ and a minimum below $1\times$.

Afterwards, a new benchmark, *BenchX*, is developed (matrix multiplication) and subsequently optimized to maximize its performance on each processor separately, using the knowledge acquired in the previous benchmarks. Finally, one final version of the benchmark is created, aiming for a best performance overall.

***Index Terms*—Lab assignment, in-order processor, out-of-order processor, gem5 simulator.**

I. Introduction

A. Architectural Features

The gem5 documentation of the BaseCPU, MinorCPU and O3CPU CPUs is bare bones, providing a small amount of useful specifications for these Architectures. To fill this gap we explored the configuration files for the gem5 simulator[1]. From these files we mostly get information from the CPU caches both with:

- L1 Instruction Cache size of 16kB
- L1 Data Cache size of 64kB
- Cache line size of 64B
- Two way associativity
- Both caches have a latency of 2 cycles

From the source files for the CPUs architectures[2] we build a more detailed notion of the CPU's architecture:

- Both CPUs have a tournament branch predictor
- Both support a sustained fetch of 2 instructions per cycle
- The out-of-order CPU has 512 general purpose physical registers (256 for integers and 256 for floating-point)
- The out-of-order CPU has another 290 physical registers
- The out-of-order processor features a Reorder Buffer (ROB) with 192 entries

[1]In the ./python_scripts directory: caches.py, m5out/config.json, m5out/config.ini

[2]In the in the opt/gem5/cpu directory, for each CPU Architecture a cpu.cc and a BaseCPU.py can be found

B. Assembler Features

A feature to note of the gcc assembler (GNU Assembler), is that the `main` routine is called from a startup/setup routine. As such, an empty `main`() program has a $2\times$ speedup due to the reordering of the instructions called before this routine (which should itself result in a $1\times$ speedup).

We will build upon these features to justify later solutions and provide commentary.

II. Proposed Solutions

A. BenchMax

For this benchmark, our goal is to maximize the difference in performance between the in-order and the out-of-order processor. To achieve that, we need to develop code that is easily exploitable through dynamic instruction scheduling and instruction level parallelism (ILP). Therefore, we want to force several heavy instructions, such as `fld` and `fdiv` (if possible, with dependencies between them), followed by multiple, independent instructions, so that the out-of-order processor can mask the execution time of the heavy parts with the execution of the independent parts. This solution will be referred to in the future as "simple code".

Such a simple solution like that should provide some speedup, but it doesn't do much to exploit the resources available.

For example, array accesses can vary in latency, depending on whether it is a cache hit or miss. As such, we can slow these accesses by forcing cache misses, which in turn lets the out-of-order processor execute more independent instructions during those access times. To do this, we can traverse the array in multiples of the cache lines, or use matrices and traverse them in column-major order. Since we are using an array of doubles for this example and cache lines have a size of 64B, we can increment the index, `i`, in multiples of **8**, per iteration.

However, now thinking of the longer access times, there is one special strategy we can use. Looking back on the previous example, we defined `c` as **register** so that it could be accessed without the need for loads. On another hand, if we do not set it as **register**, then any declaration such as `c = 5`, becomes a set of instructions: `fld`, `lui`, `fsd`.

So, any large enough amount of `double` declarations can easily clog the in-order processor, by marking all respective FU entries of the scoreboard. Since the out-of-order processor has many reservation stations and load buffers, it is significantly harder to clog it using this strategy. As such, the out-of-order processor will perform many times better than the in-order processor. The pseudo-code for the proposed solution is presented in Algorithm 1, shown below.

Algorithm 1 *BenchMax* Solution

```
a: N element array of doubles
b, k: double
M: Constant
for $0 \le i < N$ do
    for $0 \le j < N$ do
// column-major order: cache misses
        b ← a[j][i]
        b ← b / 1.00001 // fp division
/* Some independent operations */
    end for
    for $0 \le j < M \times N$ do
        // Declarations
        k ← 1
        k ← 2
        k ← 3
        . . . // about 1300 lines
    end for
end for
```

In the example above, M×N declarations are executed in each iteration i, where M and N are previously defined constants. Therefore, a total of M×N^2 declarations are executed. Of course, it is still important to be moderately reasonable in the amount of explicit k declarations performed.

B. *BenchMin*

Unlike the previous benchmark, our goal for *BenchMin* is for its performance to be as similar as possible on both processors. Recalling from the Introduction we have to consider that the assembly instructions that are run before calling the main routine, are also reordered by the out-of-order processor.

Therefore, in order to minimize the performance differences, the developed code must run worse on the out-of-order processor than on the in-order processor. There are a few things we can exploit, in order to achieve that. Particularly, one of the weak points of the out-of-order processor lies in the branch resolution, or rather, the fact that any branch *miss* may result in more lost instructions than on the in-order processor. So, if we can get the branch predictor to miss frequently, we are bound to get a very low speedup, maybe even a slowdown. Since both processors use tournament branch predictors, the solution won't be as simple as creating a predictable pattern to exploit the predictors, something that could be done if they were simple 1-bit or 2-bit Branch Predict Buffers.

As such, we propose the use of randomness for the branch conditions, that is, using the result of a rand() function to determine the branches. One simple example of this is a coin-flip program. In fact, the performance of such a simple program worsens as we increase the number of iterations, N, performed, and can even lead to a slowdown. So, we can fine-tune that number, in-order to solve the problem. However, as we are not exploiting much of the available resources, N needs to be quite large.

We can improve on this, by having the branch condition be true close to 50% of the time, maximizing the branch predictor's miss rate. We can also place many if statements in series have them lead to gotos, to force many different jumps and guarantee a miss on the first iteration. Finally, since we also want to guarantee that each branch miss results in a harsh loss of instructions, we can develop conditions that take longer to compute. Thus, we arrive at the inequation $x^2 < N \implies x < N/x$. For a random value of x (of type double), this condition does not result in a hit rate too far off from 50%, but most importantly, it results in a division by another double. As such, whenever that condition results in a miss, many instructions are lost.

Algorithm 2 *BenchMin* Solution

```
x, y, z, ... : double
for $0 \le i < N$ do
    x ← rand()%N
    y ← rand()%N
    z ← rand()%N
    . . . /* more random variables */
    if $x < N/x$ then
        goto case_a
    else if $y < N/y$ then
        goto case_b
    else . . . /* more branches */
    end if
    /* label declarations*/
    /* with more gotos*/
end for
```

Using this strategy, very few iterations of the code will be needed for it to result in a slowdown. So it will require a much smaller N to solve the problem, taking less time to execute on either processor.

C. *BenchX*

1) Vanilla: For the vanilla approach we design a simple algorithm, discarding architectural and Hardware knowledge, so that we can produce a simplistic baseline from where we can build upon. After investigating some common benchmarking tests, we opted for **Large Matrix Multiplication**.

Algorithm 3 Simple Matrix Multiplication Algorithm

```
A, B, C: int N×N matrices
for $0 \le i < N$ do
    for $0 \le j < N$ do
        for $0 \le k < N$ do
            C[i][j] += (A[i][k]×B[k][j])
        end for
    end for
end for
```

2) Non-specific Optimizations: Before targeting any specific architecture, we can use our Computer Engineering perspective to optimize the overall performance on both processors:

- **Adding the register keyword**: This will hint the compiler to store this values inside processors registers so that we can minimize the need for memory accesses. We choose to apply it to the loop iterators (`i`, `j`, `k`), and even some matrix values that remain unchanged in some loops (`temp`).
- **Using loop unrolling**: It also reduces the amount of instructions and loop overhead since we have less jumps overall. From the previous remarks we know that each cache line is 64B long. Since the matrices are of the `int` type (4 bytes) our unroll strategy will operate, in the same iteration, over 16 matrix values.
- **Switching the loop nesting order**: By switching the `j` and `k`, we access matrix `B` in line-major order (remark the cache spatial locality), minimizing the amount of cold misses.
- **Partition the matrix loading**: Since the inner `for` loop operates over the same line for all matrices, we can store the pointers to the current lines for each matrix (`int* registers`), instead of loading each value one by one in line-column coordinates. As such, loading each subsequent value only takes 1 cache access, instead of 2.
- **Full loop unrolling**: We can extend the previous loop unrolling approach, unrolling the inner `for` loop completely.

3) In-Order Optimization: The above mentioned techniques provide the gross of the available optimizations. However, there is one last change we can make: **instruction reordering**. To minimize data-dependencies, we can reorder the instructions in various places. One example of this is having an *interleaved* initialization (alternating between values of matrices `A` and `B` instead of initializing `A` completely and then `B`).

4) Out-of-Order and Blind Optimizations: We did not find any optimizations exclusive to the out-of-order processor. In fact, most non-specific optimizations were done targeting the in-order processor, knowing that they likely also benefit the other one.

As such, and taking into account the fact that the proposed optimizations for the in-order processor do not hinder the out-of-order processor at all, our blind variant will be equal to the in-order variant, as it achieves maximized performances on both processors.

Algorithm 4 *benchX Blind* (simplified)

A, B, C: `int N×N` matrices; aa, bb, cc: `int*`; temp: `int`

```
1: for $0 \leq i < N$ do
2:     aa ← A[i], cc ← C[i]
3:     for $0 \leq k < N$ do
4:         bb ← B[k], temp ← aa[k]
5:         cc[0] += temp×bb[0], cc[1] += temp×bb[1], . . .
6:         cc[N-1] += temp×bb[N-1]
7:     end for
8: end for
```

III. Experimental Results

A. BenchMax

Since our goal was to maximize the difference in performance, we are more concerned in obtaining the highest possible speedup rather than the lowest possible execution times for either of the processors. However, aside from the calculation of the speedup, the execution times are also useful to show us how computationally heavy the programs become, for the speedups we obtain.

We tested the execution times for different solutions and varied the number of iterations, `N`, and the number of declarations (by varying `M`), with respect to the obtained speedup.

Table I shows us the the execution times (in-order and out-of-order respectively) and the respective speedup, for different algorithms.

TABLE I
Execution Times of Different Solutions

Algorithm	In-Order [ms]	Out-of-Order [ms]	Speedup
Simple Code	1.65	0.635	2.596
Solution ($M = 6$)	16.0	1.98	8.075
Solution ($M = 150$)	260.7	29.6	8.827

The results above, for our solution, apply for `N` = 20. As we can see, one of the best results is with `M` = 150, with a speedup of about 8.83×. We can also see that the speedup seems to increase with `M`, since, for `M` = 6 the performance difference is smaller. In fact, we can see, on figure 1, that increasing `M` does result in an increase in speedup, but that this increase stagnates for much higher values of `M`. We also plot, on figure 1, the speedup with respect to `N`, and the results are similar. It is interesting to note that the speedup stagnates, but does not decrease, which indicates that the reorder buffer does not get full. The stagnation of the speedup can thus be explained by a limitation in the pipeline stages, which appears to be **9**.

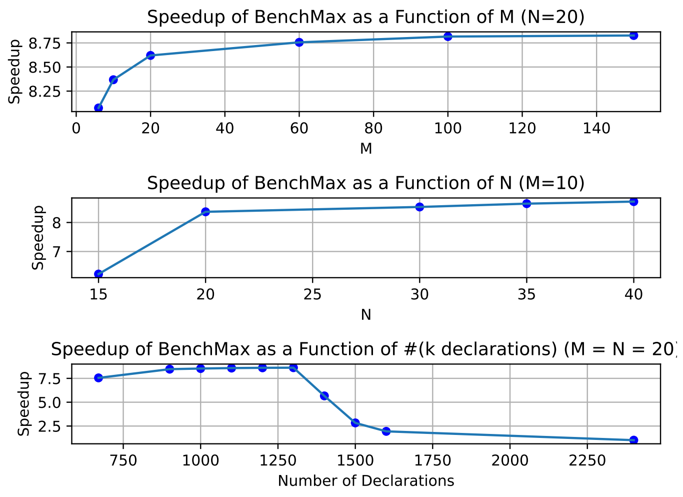

Fig. 1. Our solution's performance with respect to different parameters.

However, perhaps even more interesting, is the fact that, by increasing the number of explicit `k` declarations (from

1300), the speedup drops immensely. We believe it might have to do with the L1I cache. As mentioned in previous sections, the L1I's size is 16kB. If we check our solution's assembly code, we see that its `main` routine occupies **3974** instructions, each 4B long (**15896B** total) which can fit in the L1I. Therefore, most accesses will correspond to a hit during the code's execution. However, by increasing the number of explicit declarations, we easily overflow the cache's size, which means that there will be cache misses on each iteration (the longer the code, the higher the number of misses per iteration). Since L1I misses flush the pipeline and block it due to increased latency, getting them with such frequency can severely impact both processors' performance. This is coherent with our experiments, as even somewhat small increases in explicit declarations result in the decrease of the speedup. In fact, if the declarations occupy twice the L1I's size, then the entire cache will be overwritten twice per iteration, maximizing the miss rate.

B. BenchMin

Unlike the previous benchmark, our goal here is to obtain a speedup as close to $1\times$ as possible. Again, we tested the execution times for different solutions and for a varying number of iterations, `N`.

Table II shows us the the execution times (in-order and out-of-order respectively) and the respective speedup, for different algorithms.

TABLE II
EXECUTION TIMES OF DIFFERENT SOLUTIONS

Algorithm	Performances [ms] / [ms]	Speedup
Coin-Flip ($N = 40923$)	4.375 / 4.373	1.000457
benchMax (2600+ lines)	106.26 / 105.85	1.003883
Solution ($N = 381$)	1.091 / 1.091	1.000000
Solution ($N = 10000$)	86.15 / 107.1	0.804261

As shown above, both algorithms lead to very similar performances by both processors, indicating that they do in fact run worse on the out-of-order processor. However, it takes a much higher `N` for the coin-flip to reach that point, where as for our proposed solution, it only takes 381 iterations. For a higher value of `N`, the execution time is longer for the out-of-order processor, resulting in a *slowdown*, rather than a speedup (as seen for `N` = 10000). As explained in the previous section, this happens because the number of branch misses increases with the number of iterations (for both processors) and each branch miss produces a big loss of instructions, which is predominant for the out-of-order processor (PC is *only* updated upon committing the branch instruction). Of course, we are also taking into consideration the stalls produced on the branches due to the computationally heavy branch conditions. Although not shown in the pseudo-code, the sections marked by the labels also have heavy random branch conditions before jumping back to the `for` loop, which also contributes.

Finally, it is also interesting to note that, by adding too many instructions to the *benchMax* solution, its performance becomes comparable to *benchMin* solutions. Therefore, abusing L1I cache misses is an effective strategy for this. Of course, the number of lines chosen is enough to fill the L1I approximately twice, to maximize its miss rate.

C. BenchX

TABLE III
EXECUTION TIMES OF DIFFERENT SOLUTIONS

Algorithm	In-Order [ms]	Out-of-Order [ms]
Vanilla	4.025	1.020
Non-specific	1.719	0.407
Full loop-unrolling	1.243	0.326
benchX In-Order	1.236	0.326
benchX Out-of-Order	1.243	0.326
benchX Blind	1.236	0.326

The results for *benchX* show a notable speedup for the in-order and out-of-order ($3.26\times$ and $3.13\times$ respectively), when compared to *Vanilla*. Unrolling the entire inner loop resulted in significantly better performance than anticipated, showing amazing results on both processors, due to the reduced number of branches, and consequently, fewer instructions. Moreover, since the fetch stage can fetch multiple instructions per clock cycle, reducing branches can help mitigate cases where more instructions could be fetched in the same cycle, but were discarded due to a branch prediction that happens in the first cycle of the fetch. The *in-order* variant's performance only improves the respective processor's performance by 7μs. That can be attributed to the fact that the earlier nonspecific optimizations already minimized dependencies between instructions to a large extent. Consequently, the impact of further instruction reordering on performance was minimal. Of course, the out-of-order processor does not benefit at all from this, hence the exact same performance on all variations of *benchX*.

Although our architectures had powerful features such has vector and matrix-registers as well as support for *SIMD*, it is difficult to hint to the compiler their use with the O0 flag. However, this is an interesting topic to explore in future works.

Finally, the blind optimization approach was the same as the in-order variant, since it allowed for the best performance of each processor, as indicated in the previous section.

IV. CONCLUSION

In this lab assignment, we explored the performance differences between in-order and out-of-order processor architectures using the gem5 simulator. Through the development of various benchmarks, we aimed to maximize or minimize the performance differences between the two architectures, as well as optimize a matrix multiplication algorithm for both processors.

Overall, our experiments highlight the importance of understanding architectural features and leveraging optimization techniques to maximize performance on different processor architectures. Through careful analysis and experimentation, we can explore software implementations to take advantage of the strengths of specific processor designs, enhancing overall system performance.

REFERENCES

[1] *Gem5 Documentation*. Jason Lowe-Power. [Online]. Available: https://www.gem5.org/documentation/

V. Biography Section

João Barreiros C. Rodrigues is an Electrical and Computer Eng. (Computing Systems branch) MSc. student at Instituto Superior Tecnico - Universidade de Lisboa and an intern at INESC-ID under the orientation of Prof. Horacio Neto. He has been a Teaching Assistant in the same institution since November 2023, teaching the undergrad courses of Digital Systems, Programming and Computer Architectures. His main interests are Digital Systems Design, Reconfigurable Computing, Advanced Computer Architectures, Embedded and IoT Systems.

Martim Rendeiro Bento is an Electrical and Computer Eng. (Electronic Circuits and Systems branch) MSc. student at Instituto Superior Tecnico - Universidade de Lisboa. He has been awarded a merit scholarship (LEEC 2021/2022) in the same institution. He has been working as a teaching assistant since the second semester of the year 2022/23, on Computer Architectures, Digital Systems, Algorithms and Data Structures and Computer Networks and the Internet. His main interests are Digital Systems Design, Reconfigurable Computing, Advanced Computer Architectures and Integrated Systems Design (Systems on a Chip).

Rodrigo Manuel Ferreira Simões Coimbra is an Electrical and Computer Eng. (Control, Robotics and AI branch) MSc. student at Instituto Superior Tecnico - Universidade de Lisboa. He has been awarded a merit diploma (LEEC 2020-2023) in the same institution. His main interests are Artificial intelligence, Advanced Programming and Advanced Computer Architectures.