

Changing instruction set of an in-order processor to reduce the performance disparity with out-of-order processors

Joachim Le Fournis
*Intern at TARAN, INRIA
Ecole Normale Supérieure
Rennes, France*
joachim.le-fournis@ens-rennes.fr

Simon Rokicki
*Internship supervisor
TARAN, IRISA
Rennes, France*
simon.rokicki@irisa.fr

Abstract—During this internship, we focused on the origins of disparities in performance between in-Order and Out-of-Order processors, how to categorize them (speculation or dynamism), and whether it would be feasible to modify the instruction set to improve the in-Order performances. In addition to this analysis, we began developing high-performance in-Order processor using High-Level-Synthesis in C++.

Index Terms—Hardware/Software codesign, In-Order, Speculation, static scheduling, High Level Synthesis, Superscalar, VLIW

I. INTRODUCTION

Nowadays, while parallel computing tends to become more and more popular, it may look like the origin of performance of program is located within the number of core able to execute it. However, it has been predicted with Amdahl's law that the real limit of performance will be at the single-core performance. This leads to searching for a balance of the trade-off between single-core performance and number of cores.

Historically, two main solution exist for both of these factors : to enable high single core performances, we need Out-of-Order (OoO) processors, which can execute instructions in the order they prefer, as long as the semantic of the program is preserved. However, OoO cores are way more complicated, which will cause higher electric consumption and increased die size than in-Order (iO) processors.

Because of the conflict between them, there has been quite a lot of back and forth between these technologies (for instance with predictions, with some of them being designed for OoO and then ported on iO cores, like value prediction [1]), causing both of them to improve over time. This caused OoO to be used by most general-purpose processors, and iO to be limited to core with specific usages, such as embedded systems or Graphical Processing Units. Some other proposition were made lately to create a new competitor between both of them, so-called slice Out-of-Order [2], but they are still not as famous as OoO or iO, on which we will focus here.

In this article, we will focus on the origins of performance differences between the iO and OoO processors, to find out how we can modify the instruction set architecture (ISA) of iO in order to retrieve most of the performances

introduced by OoO that can't yet be expressed within the current ISA.

II. SCHEDULING

To have efficient processors, we need to avoid the most we can hazards :

- Data hazards : from one instruction to another, data is produced and/or consumed (like Read-After-Write, Write-After-Write, ...)
- Control hazards : the first instruction arriving is a conditional branching, which doesn't allow you to know if you can execute the next ones or not until you have executed the branching
- Structural hazards : on superscalar processors (which can execute multiple instructions at once), not all execution units can do the same operations (some are specific for memory access, other for arithmetic computation, other for floating point operators). If multiple instructions are decoded at the same time, which requires the same execution units, and that the hardware hasn't got enough for all of them, some instructions will be stalled

Moreover, we also want to hide latencies : some instruction (like load) can take multiple cycles to execute (like discussed in Section III.C), and we would like to add distance as long as possible between loading data and their usage to be able to avoid the most we can the stall that would appear if the load is very long.

Modifying the code and reordering instructions in order to use the most we can the hardware at is full potential is called scheduling. We differentiate mostly two types of scheduling : the static scheduling, which is generated within the compiler, and the dynamic scheduling which is produced by the processor directly.

A. Static scheduling

Static schedulers usually split their job in two different categories, which will use different ideas : cyclic scheduling, which exclusively operates on the loop in the code, and acyclic scheduling, which tries to increase ILP on any part of the code.

There's mostly two thing than a cyclic scheduler can change within a loop :

- Loop unrolling : instead of testing the exit condition at every cycle of the body of the loop executed, it instead creates copies of the loop's body, and checks the condition way less frequently.
- Software pipelining : when a loop is unrolled, there may be an instruction causing a latency because of the result of it being used immediately. One way of optimizing that would be to move this instruction within the previous loop's cycle's body, to increase distance between the instruction and its usage.

Acyclic scheduling operates on the rest of the code, by trying to create regions of code which are very likely to be executed together, and then hoist some instruction within it to create distance between possibly conflicting instructions, and distributing instruction so that it avoids structural hazards.

To find these region of code, static schedulers usually starts by finding the tiniest region possible : basic blocks, which consists of instructions which will all or none be executed. When these basic blocks are found, it can generates more complex blocks, such as superblocks or hyperblocks [3] by using traces (statistics on the executions, such as the number of time a basic block is executed after another).

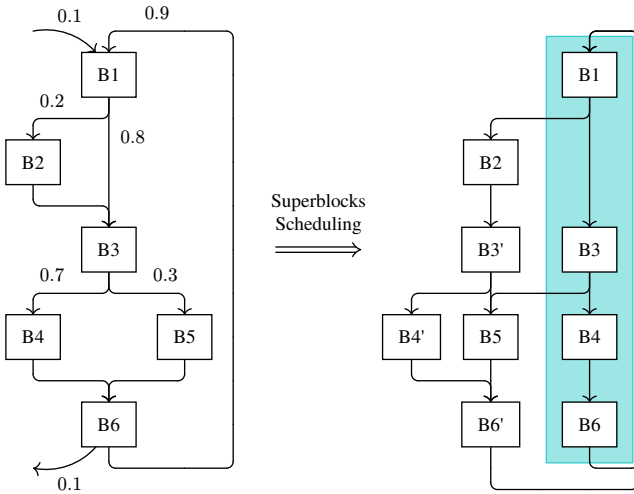


Fig. 1. Construction of a superblock (highlighted) : starting with basic blocks (indicated with B1,...) and a trace (written on edges), find the most common path, and arrange the code to have a single entry (in B1 here) and multiple outputs (in B3 and B6) by duplicating tail code (indicated with ')

When the compiler can create static schedules with very important ILP, it is possible to go further that simple superscalar processors, and directly mark within the ISA the ILP of the instructions. Instead of simply putting instructions one next to the other, without any guarantees between them, it is possible to bundle a few of them together, while indicating there won't be any data or control hazard. Such systems are so-called Very Long Instruction Word (VLIW).

This concept was extended in the Intel Itanium series and its IA-64 ISA by adding to each bundle a template number, making the functional units explicit, removing the complex

hardware checking for functional hazard from the hardware. This was called Explicitly Parallel Instruction Computing (EPIC) [4].

IA-64 did include a lot of other ideas (register windows and stacking, predication, hints in instructions, ...), but among them, we will focus a little further in Section III.C on its advances/speculative data load instructions.

B. Dynamic scheduling

Meanwhile, OoO processors do all this task of scheduling instructions to avoid latencies and increase ILP at runtime directly within the core itself. To do so, the processor builds up an instruction window in its frontend, using complex speculation (mostly branch predictors). With these instruction already fetched, it can use some hardware implemented algorithm (such as Tomasulo's one [5]) to find a valid scheduling of these instruction. To keep the execution correct, instruction needs to be committed in the right order, which is possible thanks to a Re Order Buffer (ROB).

III. TWO ORIGINS FOR THE PERFORMANCES OF OoO

[6] tries to split into two categories the origin of performance gain of OoO against iO processors :

- Dynamism : instead of having a fixed schedule for a set of instruction for the whole execution, the dynamic scheduler can react to specific events happening during the execution (giving the "dynamic" in its name), and produce a better suited scheduling to hide the consequences of these events
- Speculation : because the OoO processors are way less restricted on scheduling, they can speculatively execute instruction that would require specific instruction to appear statically

In the following paragraphs, we will see situations where OoO processors are gaining performances that statically scheduled processor with RISC-V ISA can't match, tell if they are caused by increase speculation capacity or dynamism, and try to discuss about proposed ideas to close the gap between OoO and iO in these situations.

A. Early branch computation

In some situations, a branch condition doesn't depend on its previous instruction, like this :

```
1  x++;
2  if (y != 0)
3  z++;
```

A dynamic scheduler could notice that here, checking if $y \neq 0$ or not could be computed earlier than the branching itself.

With the next the Program Counter (PC) already known while decoding the conditional branch, we don't need to use complex branch predictors to avoid stalling the pipeline, and because it is correct, we don't need any rollback system.

This is challenging for static scheduler, because most of the code region computation algorithms require branching at the end of the block, like explained in Section II.A. To lower this effect, a few solution have been proposed [7] :

- A new “prepare-to-branch” instruction could be added to the ISA, that would tell the CPU that on the next conditional branching, the condition is already evaluated. Thus, generating the right PC after that is really easy and doesn’t require any branching speculation, while not causing any stalls in the pipeline.
- Another proposition that has been implemented within a few different architectures (such as MIPS) is the branch delay slot : instead of introducing another instruction before the effective branch, it is proposed to execute every time the instruction (or a few ones) right after the branch.

However, like it is foreseen in [6], static scheduler usually aren’t that much limited by the lack of the solution mentioned above, and tends not to use this much these solutions. To show that, we did compile `embench-iot` for `mipsel` (little-endian MIPS) with `gcc` and did static measurement, shown in Fig. 2.

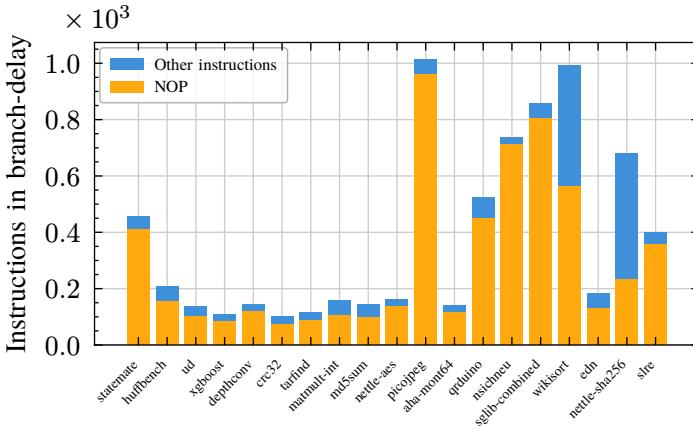


Fig. 2. Usage of the branch-delay slot (instruction right after branches) when compiling `embench-iot` on MIPSel : in yellow when it is filled with NOP (situation we want to avoid) and in blue when it is filled with other instruction

Most of the time, as shown here, the static scheduler can’t take any advantage of the branch delay slot introduced, and has to fill the program with NOP in order to keep it correctly executed (in average accross all `embench` benchmark, 78.43% of instructions with branch delay slots are followed by NOP).

B. Bias vs predictability of scheduling

As we explained earlier in Section II.A, bigger regions that simple blocks are usually constructed with statistics of execution (traces) that for instance show how many times one side or the other of a branching is taken, and if one side is taken way more often that the other one (wich we reference by the term bias), the block may be extended with it. Thus, static scheduling requires an important bias on the program branches in order to produce a efficient scheduling.

On the other side, dynamic schedules are produced thanks to branch predictors located on the chip, which are able to change their prediction during the execution reacting

to events, which is a direct form of previously introduced dynamism.

In short, we could say that performance for static schedulers is proportional to bias of branch, while dynamic schedulers’ performance are proportional to predictability of a branch. We did execute the `embench-iot` benchmarks, and extracted the following results :

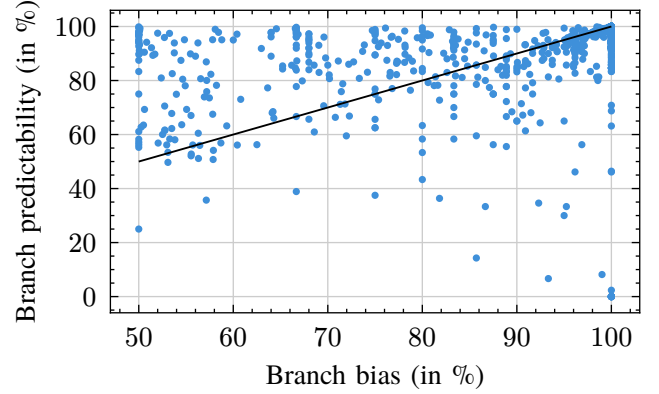


Fig. 3. Branch predictability (% of right prediction) vs bias (% of $\max(\frac{\text{taken}}{\text{total}}, 1 - \frac{\text{taken}}{\text{total}})$) for `embench-iot` with at least 10 occurrences

These benchmarks were executed on the basic RISC-V o3 CPU of `gem5` (a 8192kB 2-bits tournament branch predictor with 2048kB of 2-bits local predictor with 2048kB of local history table and a 8192kB 2-bits global predictor).

Because of the important difference between bias and predictability, OoO has a great advantage compared to iO. One way static schedulers could try to close the gap with its counterpart would be by producing multiple schedules and allowing the processor to switch between them by adding instruction within the ISA to let it decide the most adapted one based on its branch predictor. This solution hasn’t been that much studied in the literature probably because it doesn’t look very fancy due to the increased binary size, which lowers code density, reducing performances.

C. Studying memory accesses

a) Lack of speculation for in-Order processors:

As we mentioned in Section II.A, when code regions are generated, we would want to hoist loads the furthest away possible from the usage, in order to hide latencies of stall-on-use or instruction re-executed by iCFP. However, in many situation, there are limits of hoisting instruction for static scheduling, that OoO doesn’t suffer thanks to speculation. We will study two brief example of these :

1 <code>beqz a1, else</code>	1 <code>sw a2, 8(a0)</code>
2 <code>lw a2, 0(a1)</code>	2 <code>lw a3, 16(a1)</code>
3 <code>addi a2, a2, 2</code>	3 <code>addi a3, a3, 5</code>

Listing 1. Limits of instruction hoisting without speculation

On the left program of Listing 1, we would want to hoist the load on `line 2` away from its usage on `line 3`. However, `a1` may be storing a null pointer, which we wouldn’t want

to load on **line 2**, and to avoid this, we have a condition on **line 1**. If the load of **line 2** was hoisted above the condition on **line 1**, we may be generating unnecessary exception, that we want to avoid.

On the right program of Listing 1, we would similarly want to hoist the load on **line 2** away from its usage on **line 3**. However, the compiler sometimes doesn't know statically if load address in `a3` overlaps with the store address in `a0`, and hoisting the load above **line 1** might be invalid because we could read the value that would be overwritten by store **line 1**.

To address these issue that the OoO doesn't suffer from thanks to its speculation capacities, we could either do it with software (to fix the second example) at the cost of a very high overhead because of the comparison required (see Fig. 4), or we need to introduce two new kind of speculation instruction in the ISA [3]:

- Control speculation consists of executing an instruction even before knowing if its execution is required or not. To implement this on iO processors, we need hardware to defer the possible exception raised while executing this instruction : a Not-a-Thing (NaT) bit is stored with each register, that is set or cleared by a hoisted control-speculative-load instruction (`ld.s` on itanium [4]).
- Data speculation consist of executing an instruction even before knowing if it can be executed correctly. To usual solution to implement it on iO processors is to add a new data-speculative-load (`ld.a` on itanium [4]), which will issue a load immediately but will add this load in a special table called Advanced Load Address Table (ALAT), implemented similarly to a cache (associative structure, separating tag and index, ...). The following stores instructions will simply remove the entries in the ALAT if they overlap with the store executed.
- On itanium, both of these speculation can be executed on the same time with `ld.as`.

When these speculative load instructions are used, if we are lucky, they won't cause any issue. However, in other situation, we need to recover the invalid load we executed. We do so by adding a check instruction (`chk.s`, `chk.a`, `chk.as` on itanium [4]) before the usage of the loaded value, with the register loaded and an offset to a recover snippet (which will load from memory the data without any speculation this time, or will copy from the origin register of the store that happened before). This snippet will be executed only when the NaT bit is set, or if the loaded register doesn't appear anymore in the ALAT.

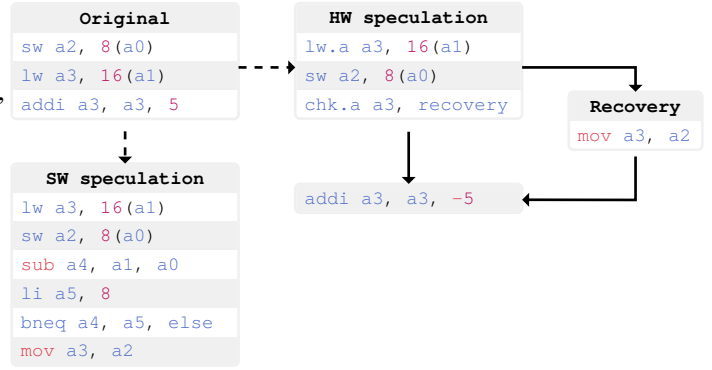


Fig. 4. Software data speculation vs hardware data speculation

Not that many experiment have been made on these speculative-load, but according to [8], these instruction should be beneficial : in this article, the compiler was pushed to the limits by aggressively using the speculative loads each time instead of loads, and it still increased the performances.

b) Speculation isn't dynamism:

As it is highlighted in [6], even with these previous improvements, the static scheduler still produces instructions in a certain order that won't change during execution. One great example of the lack of dynamism is this simple C code : $a \rightarrow b \rightarrow c + x \rightarrow y \rightarrow z$

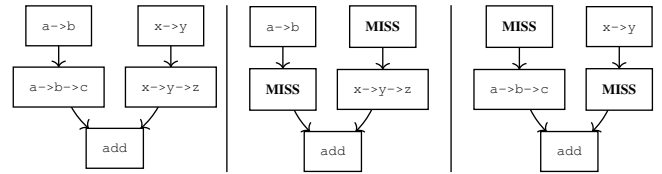


Fig. 5. Data dependencies of $a \rightarrow b \rightarrow c + x \rightarrow y \rightarrow z$ in three scenarios : without any misses, and with two cache misses in different location

In Fig. 5, three scenarios are shown, depicting possibles cache misses. Depending on the order the loads are scheduled, it is possible or not to hide one of the cache miss during the other one. However, for a given schedule, it is not possible to hide for both scenarios the miss within the other one. Meanwhile, an OoO processor can produce multiple schedules during the execution, and if both situation happens during the same execution, it can produce for each one a different scheduling, which will increase performance compared to a static scheduler.

There would be multiple way to get back the performance loss here, which would consist of adding a little bit of dynamism into the in-Order processors :

- This could be explicit within the ISA, by introducing new instruction similarly to Section III.B, allowing the processor to use different scheduled code depending of the hit/miss. However, this would, like in Section III.B, reduce the code density and introduce complexity on static schedulers
- Or implicit : instead of classical stall-on-use processors (which don't stall the moment the cache miss appears, but only when the loaded data is consumed),

there are a few solution to continue execution even after this consumption of the loaded data. Among them, the one we will focus here is the in-order Continual Flow Pipeline (iCFP [9]). The main idea is that instead of only keeping the destination register of the load as invalid, we keep executing further after the consumption of the register, marking the destination register of this instruction as invalid too, and adding it to a queue of wrongly executed instruction. As soon as the load is resolved, we execute again the instructions located within the queue, and remove every mark of invalid registers. Other implicit solution are described in [9] with lower hardware costs but won't be described here.

- Another implicit solution, that has gained popularity in the last few years would be to use value prediction within iO processors[1]. However, we didn't really studied this solution during this internship.

IV. EVALUATING SPECULATION AND DYNAMISM

After seeing the previous two sections, one could ask himself two main questions :

- How hard would it be for a static scheduler to get to the scheduling performances of a dynamic one without specific modification on the processor ?
- If most of the speculation can be achieved on iO processors too, in what proportion does it help to gain back the lost performances compared to OoO, or in the complement way, how much does OoO gain from dynamism ?

To answer these two question, [6] did extract dynamic schedules of an OoO processor, and did process it in two different ways that answers respectively these questions :

A. Finding distance between base program and schedule

Like previously explained in Section II.B, the frontends of OoO are quite complicated because they tend to use a very large instructions window. However, we could ask ourselves compared to the static scheduling techniques showed previously, how far away does the dynamic scheduler searches for instruction to produce a good scheduling. [6] did answer this by firstly computing the average region size (number of basic blocks) computed by different static scheduling : superblocks did manage to get about 3.2 basic blocks, and the best solutions (Atom frames adaptative) can get up to 6.3 basic blocks in average. Compared to that, it was computed on the OoO core for the dynamic scheduler the number of basic blocks present within RoB, and found this :

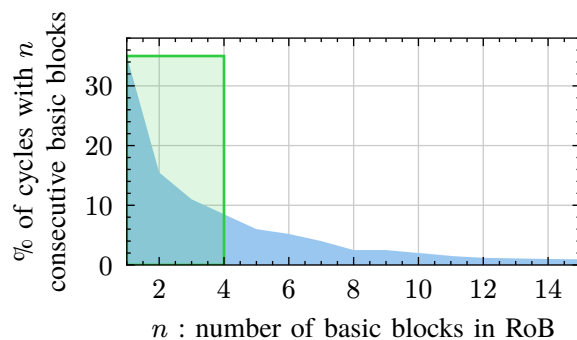


Fig. 6. Capacity of overlapping code within dynamic scheduler : 70% of OoO cycles contain 4 or less consecutive basic blocks

Even if Fig. 6 contains a long tail, it is shown that even the simplest static scheduling techniques such as superblocks can produce code region as big as the dynamic schedulers produces in 70% of the cycles.

B. Extracting static schedules out of the dynamic schedules

To answer the second question, the authors of [6] did develop a technique in order to extract from the OoO static schedules that will be executed on an OO, removing dynamism from the OoO while keeping speculation.

To explain a little bit further, the strategy was to extract snippet of code (ending with backward branches and function returns), note the order in which instructions are executed in this snippet, and which snippet is executed after it. After some filters, the most common schedule is selected to become the static schedule. It is then appended to a database, with the sequence of PC of the snippet as key.

Then, they execute the same benchmark by using the same simulator as previously, but which frontend is changed so that it doesn't generate a dynamic schedule anymore but load instructions for the database. This allows the database to act as a sort of static schedule, but without emitting a single binary and allowing the speculation to still happen.

At the end, this somehow allows to exactly know the repartition between speculation and dynamism within the OoO processor simulated. When comparing the executions of the simple iO, this OoO without dynamism (and only speculation), and the baseline OoO, here are the performances :

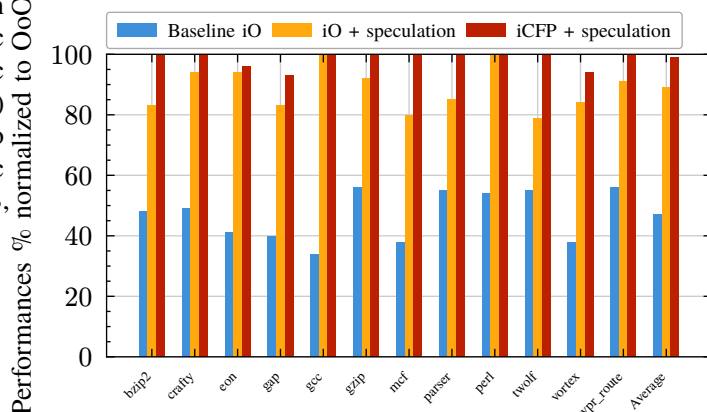


Fig. 7. Baseline iO vs iO with speculation normalized to OoO performances
Figure extracted from [6]

Fig. 7 allows us to see that in average, for a similarly provisioned processor, an iO can only achieve 47% of the performances of an OoO. However, when letting an iO express all its speculation (or differently said, when removing dynamism of an OoO), this percentage grows to 89%.

This number get even up to 99% in average when adding just the right amount of dynamism within the iO processor : as it is explained in [6], almost all performance that still lacks to this boosted iO is the capacity to handle cache miss and branch miss-prediction. Thus, having a iCFP processor with these speculation allows it to get to this performance.

V. IMPLEMENTING AN iO SUPERSCALAR PROCESSOR

A. Processor synthetised

Before experimenting with the ISA changes proposed above, we needed a baseline rv32i (32-bits RISC-V with 32 registers, without any extensions) processor with already good enough performance, which could be easily modified in the future. We decided to start with an iO 2-wide superscalar processor (fetching/decoding two instructions together, and executing both as long as they are no hazards between them). Moreover, we want to be able to simulate it or synthesis it with possibly cache in front of it.

For these reasons, we started from the Comet processor [10], which is a tiny open-source in-order processor. We modified it to get the above requirements, and it is open-source too on the same repository. Fig. 8 is a simple diagram of its pipeline :

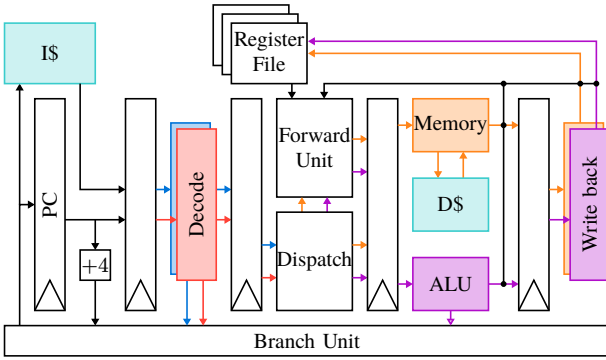


Fig. 8. Superscalar in-order processor (Comet)

Even if the processor itself isn't that complex compared to modern other processors, it took quite a lot of time to get to this point, because of a lot of diverse difficulties. Among them, the two most important ones are management of forwarding and branch unit, which are hard because of the number of edge cases to think of (for instance the first instruction is a branch and the second one is ALU, or it is in the reverse order, so that it is totally differently forwarded and the stalls aren't the same at all).

To help us designing the processor, the first step was adding to the simulator Kanata CPU traces, that can be open within Konata pipeline viewer :

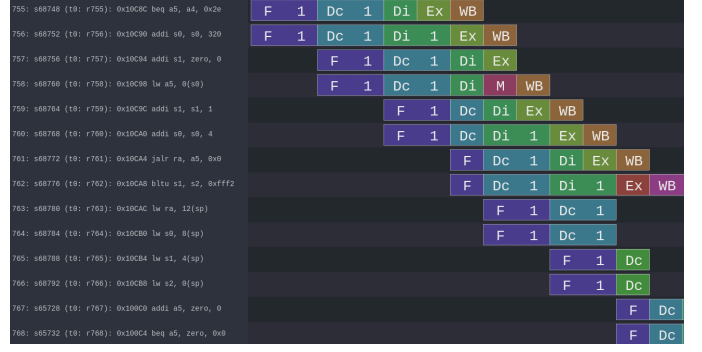


Fig. 9. Konata showing a program displaying different situation : both instructions not dispatched in the same cycle, both instruction in the same cycles and branching. The Ex and WB in a different color show that the instruction are located within these stages, but they are invalidated (We=false)

B. High-Level Synthesis

This processor is written in C++ and is synthetised with Catapult 2023.1 High-Level Synthesis tool (HLS). Thanks to Catapult, it is possible to synthetise Comet for a large variety of comet. Currently, it synthetise on both Xilinx Artix 7 FPGA and 28nm ASIC.

```
1 void doCycle(struct Core& core) {
2     struct FtoDC ftdc_temp[2];
3     bool stalls[7];
4     fetch(core.ftodc[0], &ftdc_temp[0]);
5     fetch(core.ftodc[1], ftdc_temp[1]);
6     ...
7     if (!stalls[STALL_FETCH]) {
8         core.ftodc[0] = ftdc_temp[0];
9         core.ftodc[1] = ftdc_temp[1];
10    }
11    ...
12 }
```

The choice of using HLS was done in-order to simplify hardware/software codesign : in order to implement the proposed instructions mentioned earlier, it is possible to write easy C++ code within our `core.c` file that would implement them, but also use the modified `core.c` to generate a simulator. The simulator is simply forwarding some syscalls to the host OS, while executing exactly the written pipelined processor. This allows an efficient simulator (see [10]) while still being cycle accurate.

C. Results

We used the simulator that we can generate with the same C++ code[10] and ran the embench-iot benchmark suite on it. It was compiled for RISC-V using `riscv32-none-elf` tool-chain version 15.2.

These benchmarks tend to be quite small, which doesn't allow to have quite precise benchmarks, but are open-source, easy to compile for RISC-V and MIPS and quick enough to be run on software on gem5 for the previous experiments, which lead us to this choice of benchmarks.

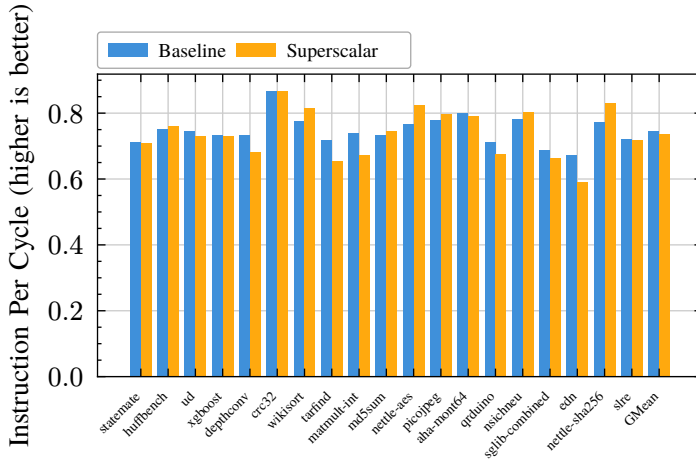
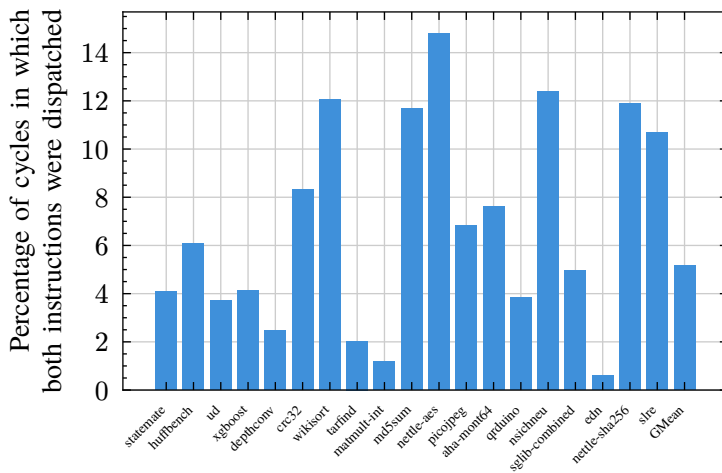


Fig. 10. Execution performances (IPC) on embench-iot between baseline and superscalar comet

As you can see, the Instruction-per-Cycle is very similar (sometimes higher than the baseline processor we started with, but in average slightly lower) to what we had before, and we didn't really change the performance of the processor. To explain this, we also did extract the percentage of cycles in which both instructions can be dispatched.



As we can see, in average we have about 5% of the cycles in which we could use both the execution ALU stage and the memory one. Moreover the conditional branchings cost more (because the condition is found one cycle later, or more (if the instruction is the second one in dispatch) because of the dispatch cycle added before the execution stage). This deeper pipeline also affects forward which can now require one (or more) cycle more than before.

We thus have a slightly increase of performance when both instruction can be dispatched in the same cycle (memory+execution), which is counterbalanced with a decrease of performance because of the more complex pipeline. This is perfectly seen in both `nettle-aes` and `edn` benchmarks, which respectively have the highest and lowest percentage of both instruction dispatched, which directly corresponds to a important increase and decrease of IPC on Fig. 10.

VI. FUTURE WORK

Even if this internship is now finished, there is still a lot of possible ways to continue on this topic that we would like to explore in the future. Among them, the one we highlight these ones :

- Reproducing the experiment of [6] and try to implement different solution he highlighted in his article that weren't experimented yet (such as producing multiple static scheduling and switching between them during the execution, as we explained earlier)
- Another solution to hide a lot of latencies in in-Order processor is value prediction[1], and we would like to experiment on how we can use multiple branch/value predictors which different level of confidence and latencies of result. We would interface it with software hinting (difficulty to predict the right value inlined within the code)

REFERENCES

- [1] P. Ravenel, "Amélioration de la performance des processeurs à exécution dans l'ordre sous contrainte de complexité matérielle," Doctoral dissertation, 2025.
- [2] K. Lakshminarasimhan, A. Naithani, J. Feliu, and L. Eeckhout, "The Forward Slice Core Microarchitecture," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, in PACT '20. Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 361–372. doi: 10.1145/3410463.3414629.
- [3] J. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. 2005, p. .
- [4] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, no. 5, pp. 24–43, 2000, doi: 10.1109/40.877948.
- [5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.
- [6] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: is it speculation or dynamism?," *SIGPLAN Not.*, vol. 48, no. 4, pp. 241–252, Mar. 2013, doi: 10.1145/2499368.2451143.
- [7] J. A. DeRosa and H. M. Levy, "An evaluation of branch architectures," *Proceedings of the 14th annual international symposium on Computer architecture - ISCA '87*, pp. 10–16, 1987, doi: 10.1145/30350.30352.
- [8] J. Baiocchi, R. Villamarin, and M. Mock, "An Empirical Study of Data Speculation Use on the Intel Itanium 2 Processor," in *2012 16th Workshop on Interaction between Compilers and Computer Architectures (INTERACT)*, Los Alamitos, CA, USA: IEEE Computer Society, Feb. 2005, pp. 22–33. doi: 10.1109/INTERACT.2005.2.
- [9] A. Hilton, S. Nagarakatte, and A. Roth, "iCFP: Tolerating all-level cache misses in in-order processors," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 431–442. doi: 10.1109/HPCA.2009.4798281.
- [10] S. Rokicki, D. Pala, J. Paturel, and O. Sentieys, "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications," in *ICCAD 2019 - 38th IEEE/ACM International Conference on Computer-Aided Design*, Westminster, CO, United States: IEEE, Nov. 2019, pp. 1–8. [Online]. Available: <https://hal.science/hal-02303453>