

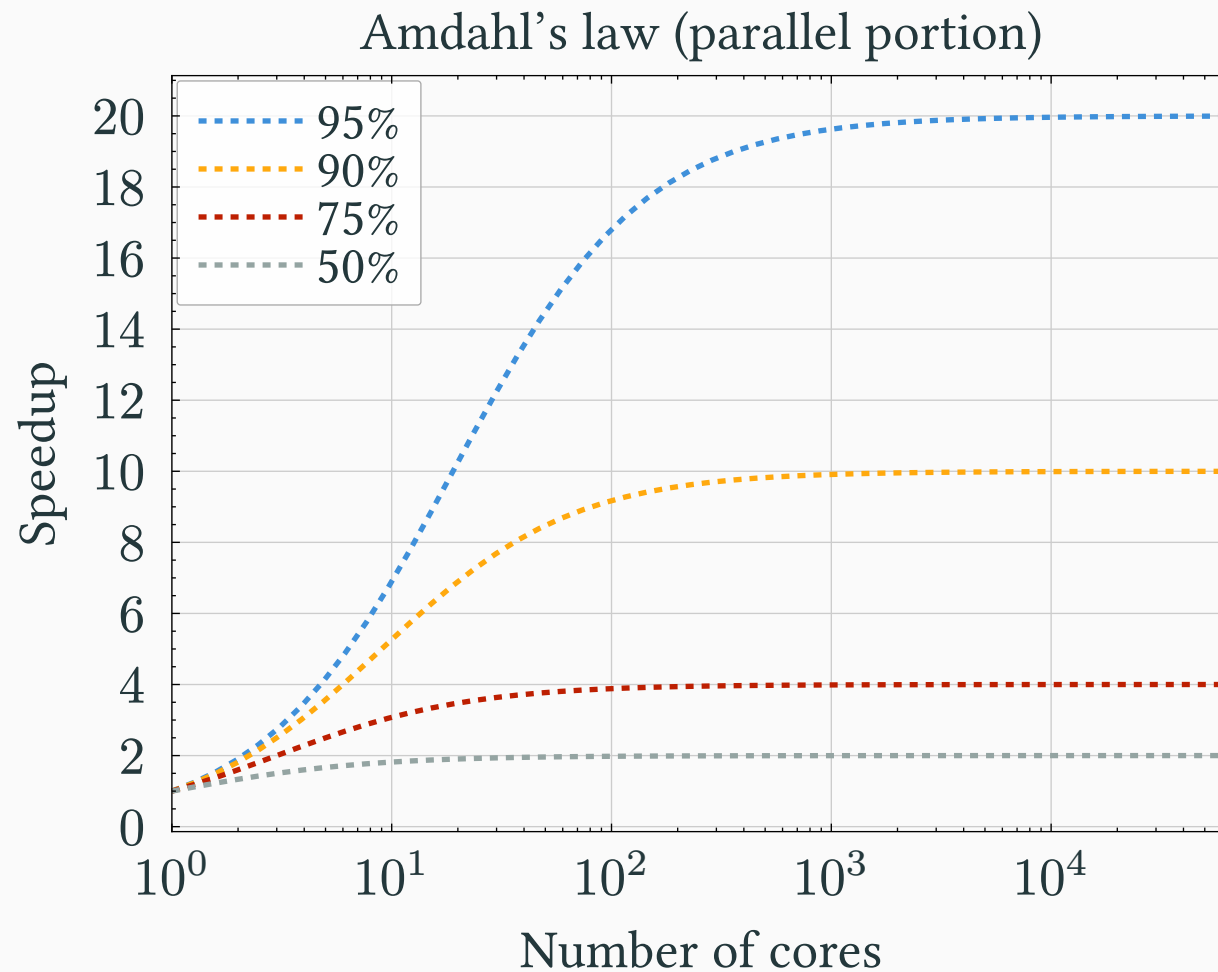
Designing high-performance in-order processors to close the gap with out-of-order

Internship defense

Joachim Le Fournis (supervised by Simon Rokicki)

2026-08-25

TARAN@INRIA Rennes

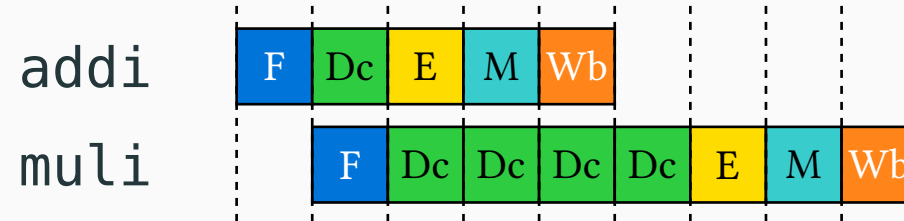


First issue : hazards

```
1 addi a1, zero, 5  
2 muli a1, a1, a1
```

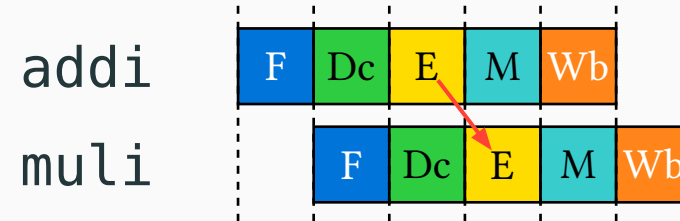
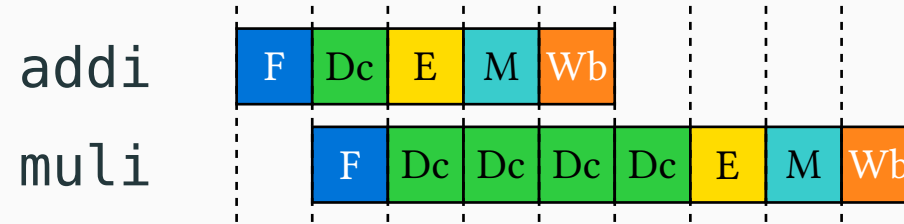
First issue : hazards

```
1 addi a1, zero, 5
2 muli a1, a1, a1
```



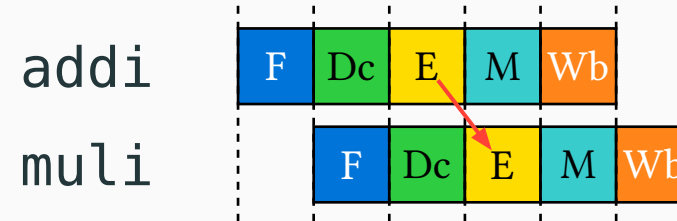
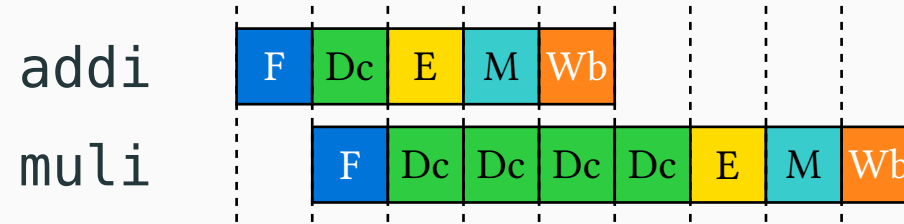
First issue : hazards

```
1 addi a1, zero, 5
2 muli a1, a1, a1
```



First issue : hazards

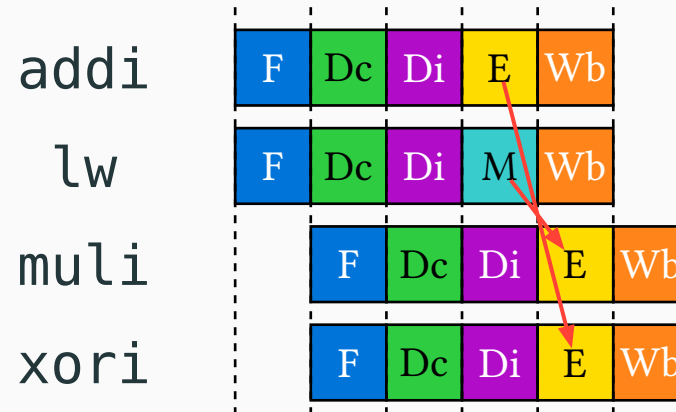
```
1 addi a1, zero, 5
2 muli a1, a1, a1
```



Data hazards (RaW, WaW)

First issue : hazards

Functional hazards

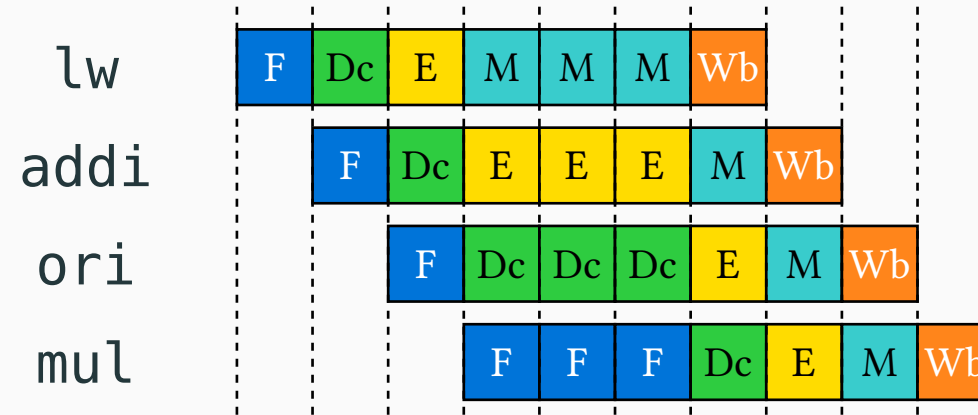


First issue : hazards

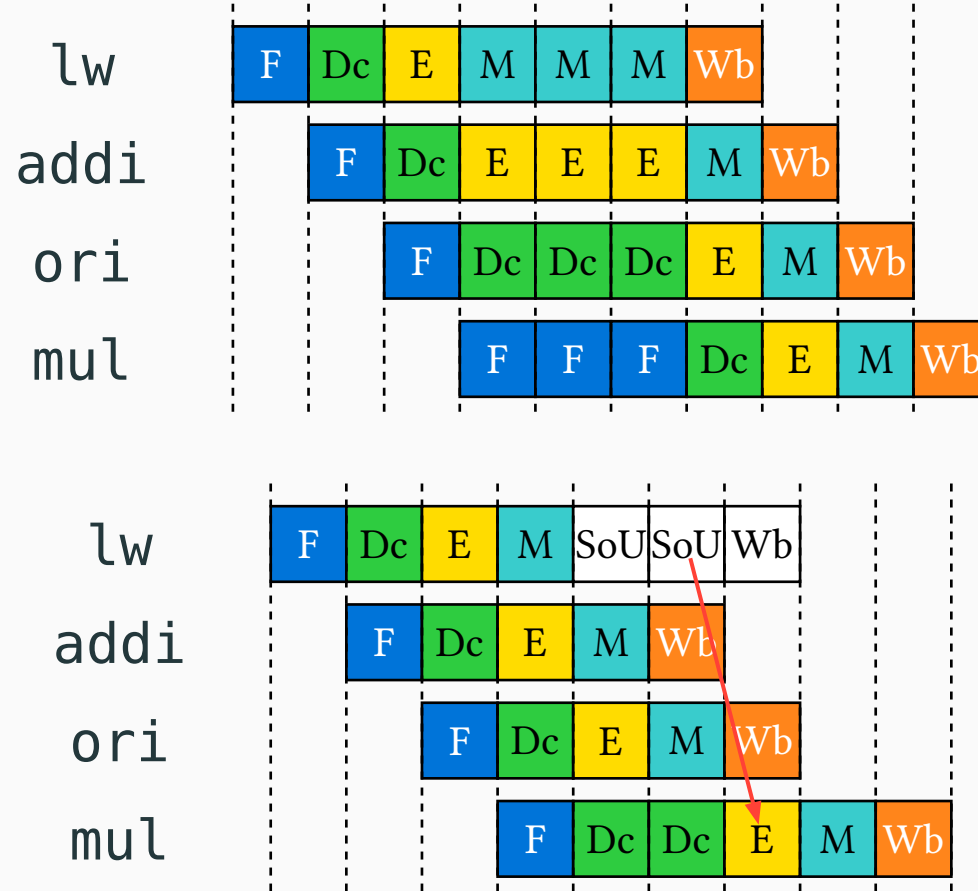
Functional hazards



Second issue : variable latencies



Second issue : variable latencies



$$\text{ILP} = \text{Instruction-level parallelism} = \frac{\text{\#executed instructions}}{\text{\# cycles}}$$

$$\text{ILP} = \text{Instruction-level parallelism} = \frac{\# \text{executed instructions}}{\# \text{ cycles}}$$

Dynamic

- Directly on processor (OoO=Out-of-Order)
- Working with algorithms (Tomasulo) on instruction window

$$\text{ILP} = \text{Instruction-level parallelism} = \frac{\# \text{executed instructions}}{\# \text{ cycles}}$$

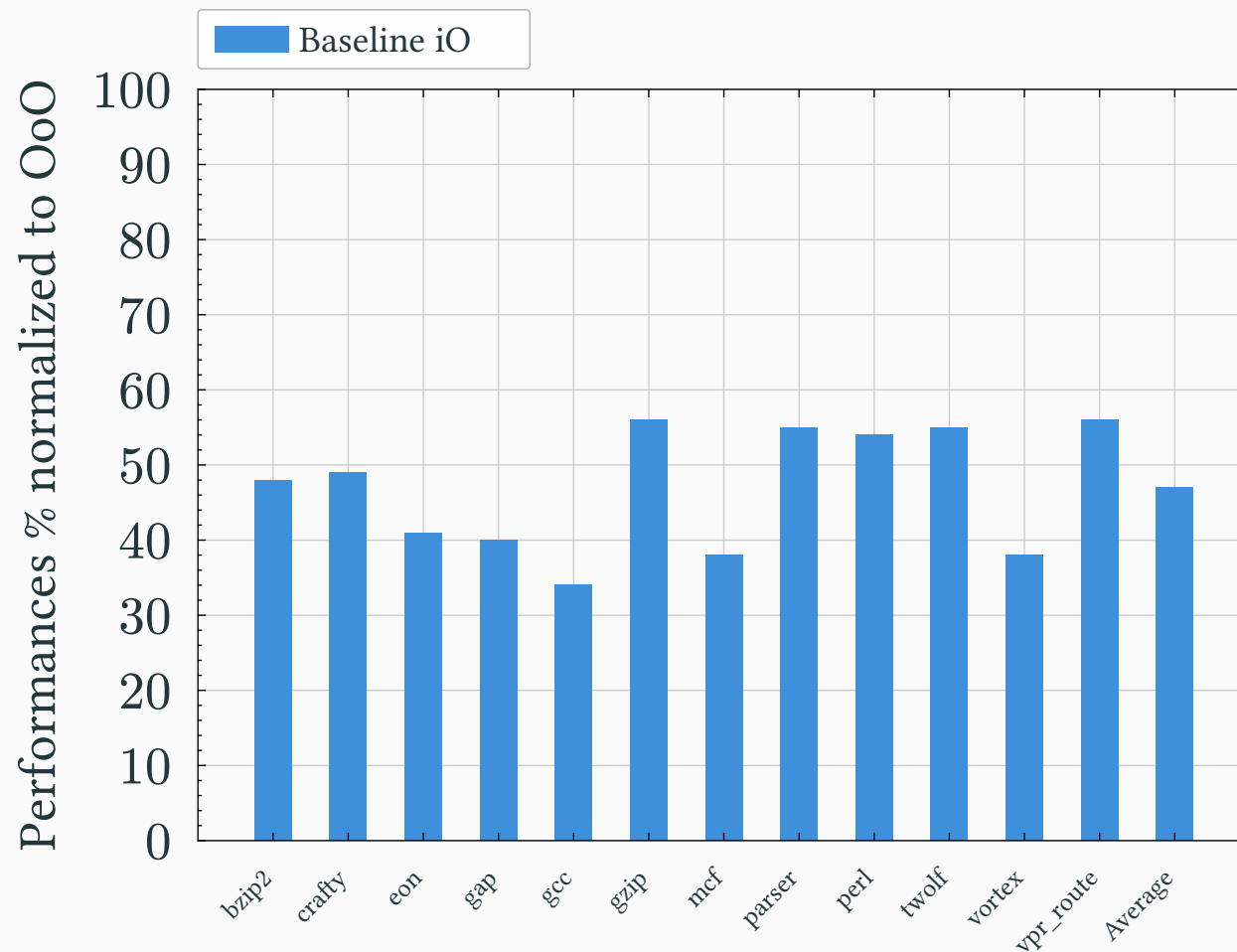
Dynamic

- Directly on processor (OoO=Out-of-Order)
- Working with algorithms (Tomasulo) on instruction window

Static

- During compile-time
- With different solutions (cyclic/acyclic analysis)

Performance disparity



Internship goals

- Understand the origin of the gap between static and dynamic scheduling
- Implement a RISC-V processor (rv32i)
- in-Order, Superscalar
- Which will *in the future* implement the ideas explained here

Dynamism

vs

Speculation

⇒ Change of schedule during execution

- React to hardware change (branch prediction, cache misses, ...)
- To produce better suited schedules

⇒ Execute instruction that we don't know if we are allowed to / if we should

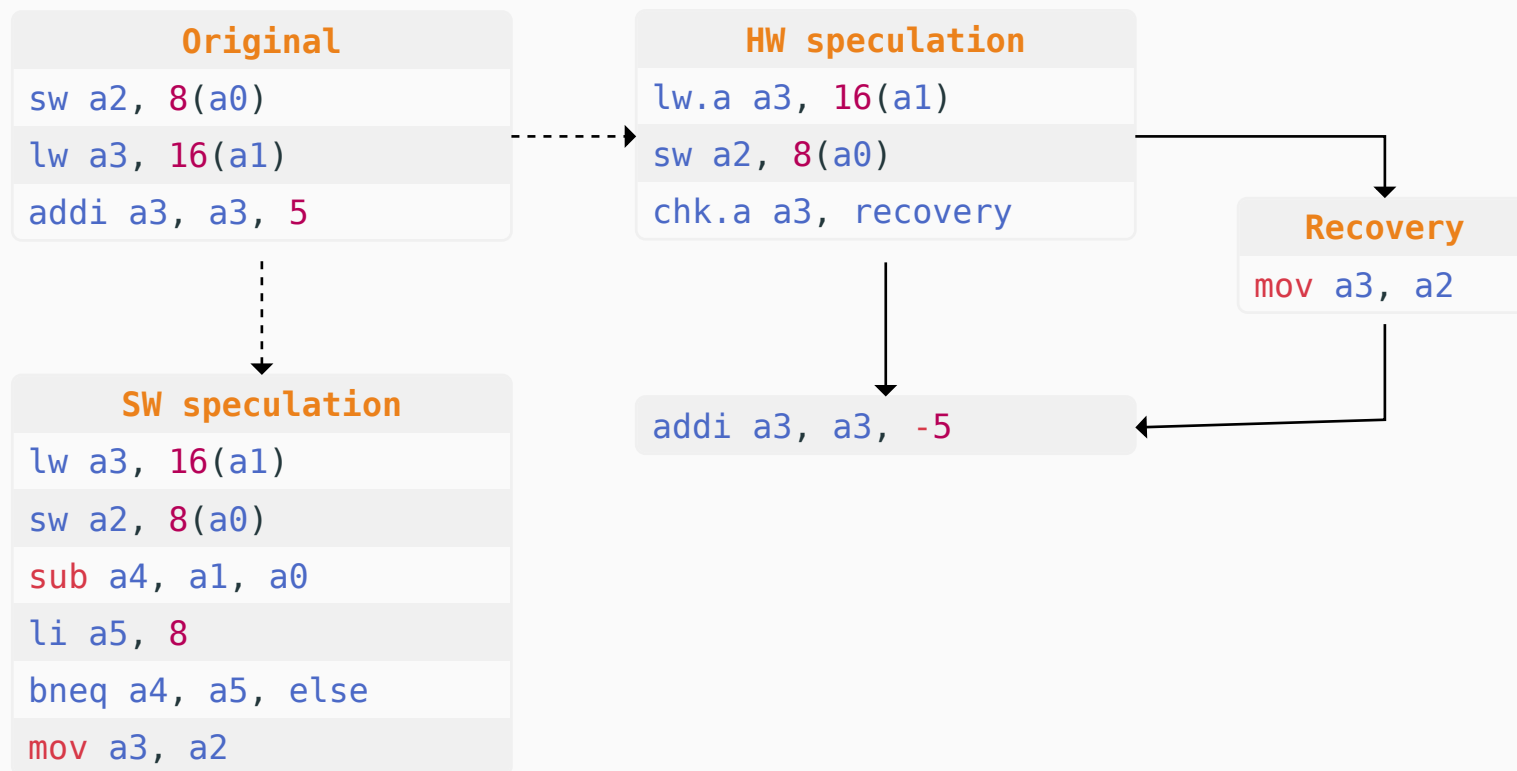
- Speculative execution across branches
- Speculative execution with memory accesses

Impossibilities to move memory access

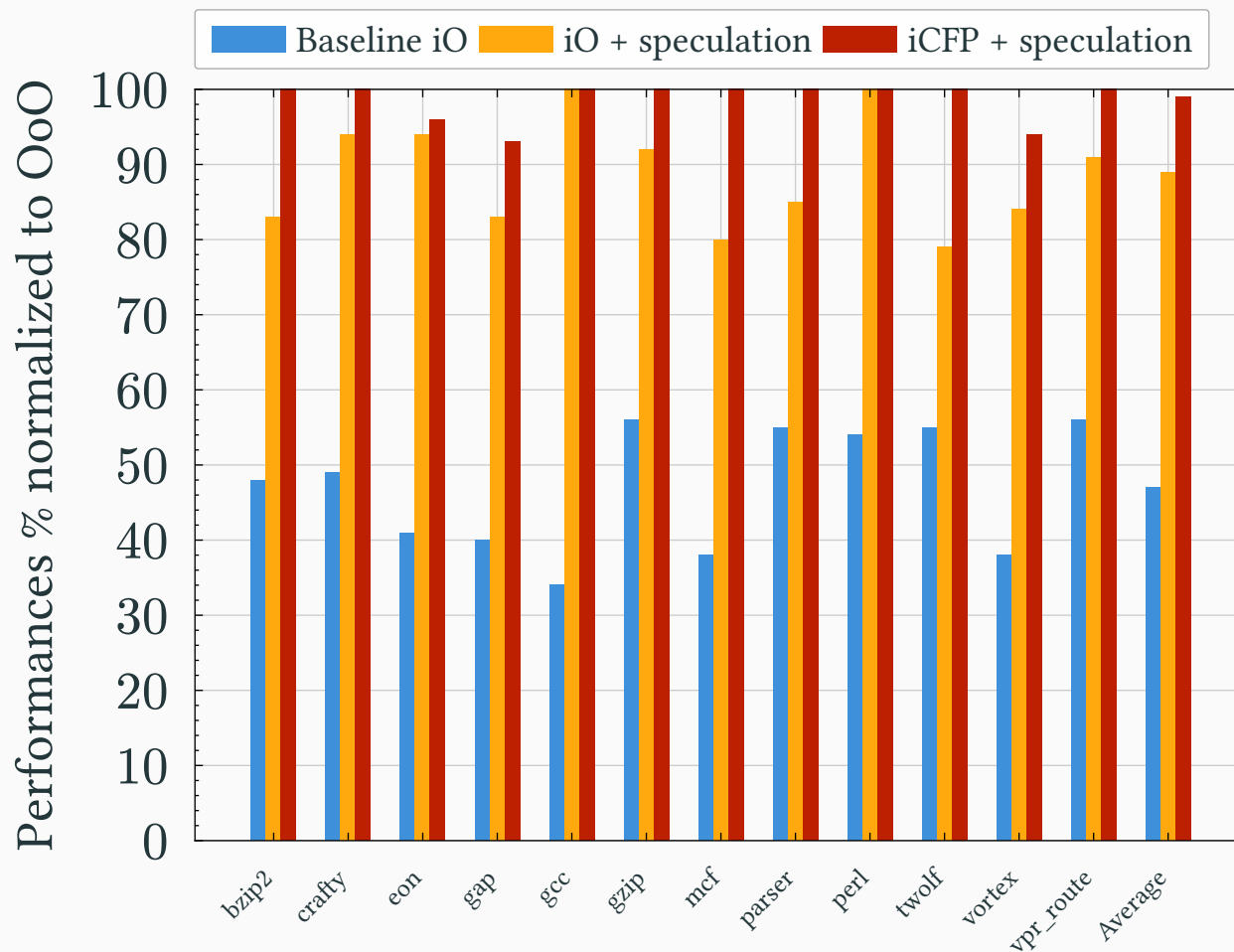
```
1 beqz a1, else
2 lw a2, 0(a1)
3 addi a2, a2, 2
```

```
1 sw a2, 8(a0)
2 lw a3, 16(a1)
3 addi a3, a3, 5
```

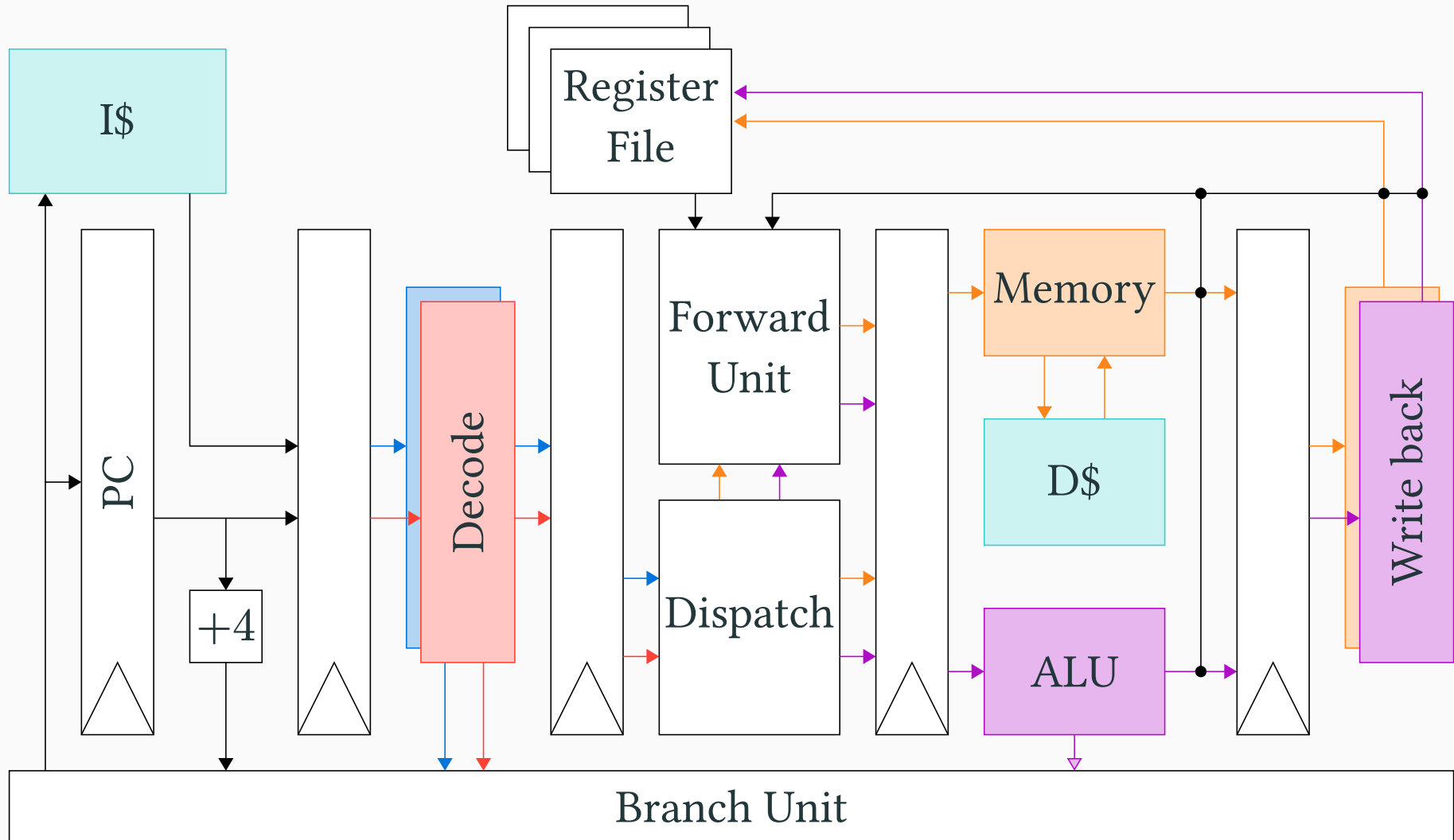
Solutions to this issue



Experimental results



Structure of the processor



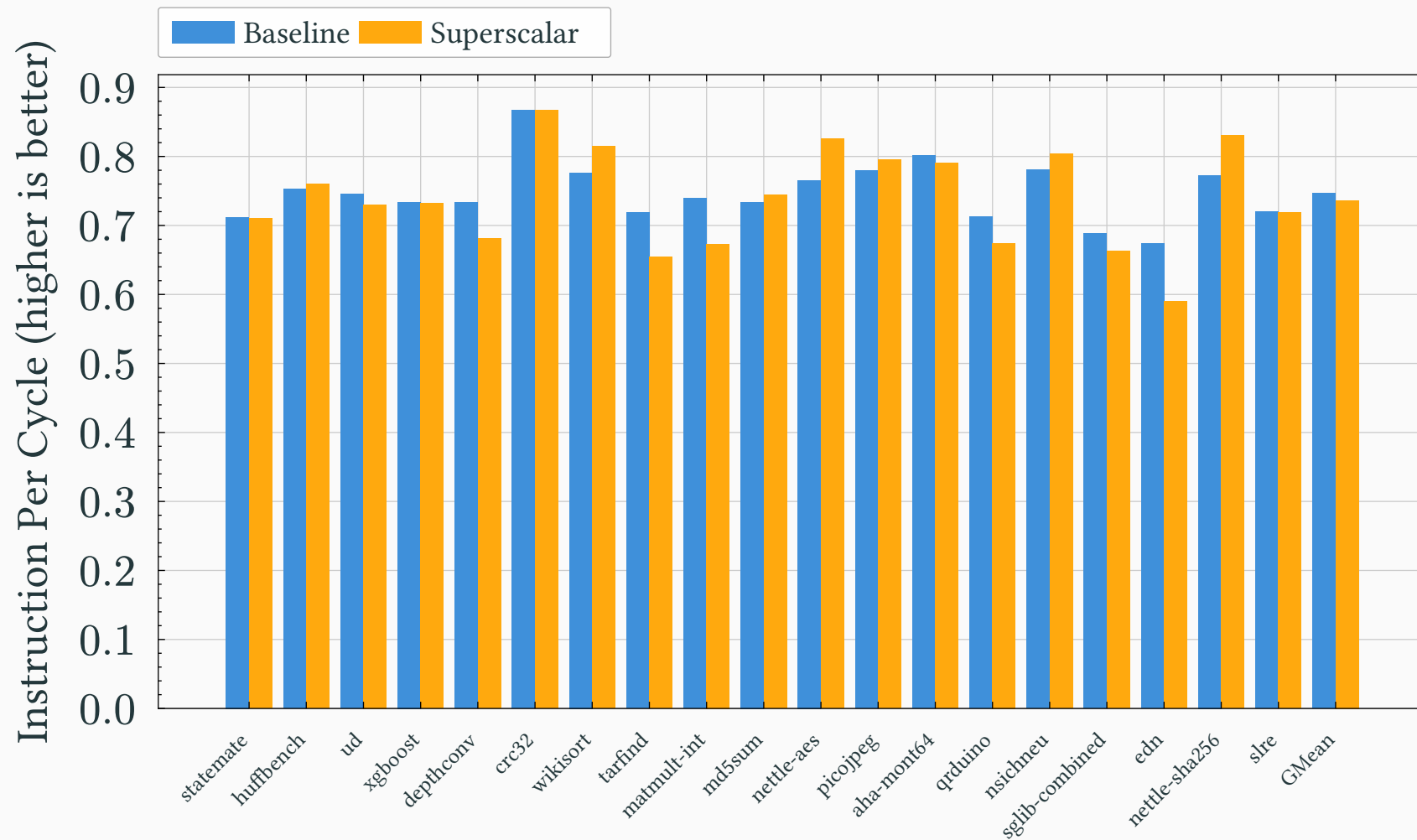
High Level Synthesis

```
void doCycle(struct Core& core) {  
    struct FtoDC ftodc_temp[2];  
    bool stalls[7];  
    fetch(core.ftodc[0], &ftodc_temp[0]);  
    fetch(core.ftodc[1], ftodc_temp[1]);  
    ...  
    if (!stalls[STALL_FETCH]) {  
        core.ftodc[0] = ftodc_temp[0];  
        core.ftodc[1] = ftodc_temp[1];  
    }  
    ...  
}
```

Kanata traces

755: s68748 (t0: r755): 0x10C8C beq a5, a4, 0x2e	F	1	Dc	1	Di	Ex	WB												
756: s68752 (t0: r756): 0x10C90 addi s0, s0, 320	F	1	Dc	1	Di	1	Ex	WB											
757: s68756 (t0: r757): 0x10C94 addi s1, zero, 0			F	1	Dc	1	Di	Ex											
758: s68760 (t0: r758): 0x10C98 lw a5, 0(s0)			F	1	Dc	1	Di	M	WB										
759: s68764 (t0: r759): 0x10C9C addi s1, s1, 1					F	1	Dc	Di	Ex	WB									
760: s68768 (t0: r760): 0x10CA0 addi s0, s0, 4					F	1	Dc	Di	1	Ex	WB								
761: s68772 (t0: r761): 0x10CA4 jalr ra, a5, 0x0							F	Dc	1	Di	Ex	WB							
762: s68776 (t0: r762): 0x10CA8 bltu s1, s2, 0xffff2							F	Dc	1	Di	1	Ex	WB						
763: s68780 (t0: r763): 0x10CAC lw ra, 12(sp)								F	1	Dc	1								
764: s68784 (t0: r764): 0x10CB0 lw s0, 8(sp)								F	1	Dc	1								
765: s68788 (t0: r765): 0x10CB4 lw s1, 4(sp)										F	1	Dc							
766: s68792 (t0: r766): 0x10CB8 lw s2, 0(sp)										F	1	Dc							
767: s65728 (t0: r767): 0x100C0 addi a5, zero, 0													F	Dc					
768: s65732 (t0: r768): 0x100C4 beq a5, zero, 0x0													F	Dc					

Performances



Contribution and future work

- Implemented a superscalar RISC-V in comet (in the future : additional ALU, use in OoO Comet)
- Produced some experiment contributing the McFarlin's article (bias/predictability, delay slot, ...)

Contribution and future work

- Implemented a superscalar RISC-V in comet (in the future : additional ALU, use in OoO Comet)
- Produced some experiment contributing the McFarlin's article (bias/predictability, delay slot, ...)
- Learned a lot for future work (in M1) :
 - Producing multiple schedules, add in the ISA a way to pick among them at runtime (cache miss, branch prediction)
 - **Specifying in ISA confidence (and latency) of branch and value prediction**

Thank you for your attention

Any question ?

Appendix

Amdahl's Law

$$S_{\text{latency}} = \frac{1}{1 - p + \frac{p}{s}} \quad \text{where } S_{\text{latency}} \approx \text{task speedup}$$

s = number threads

$p \approx$ parallel portion

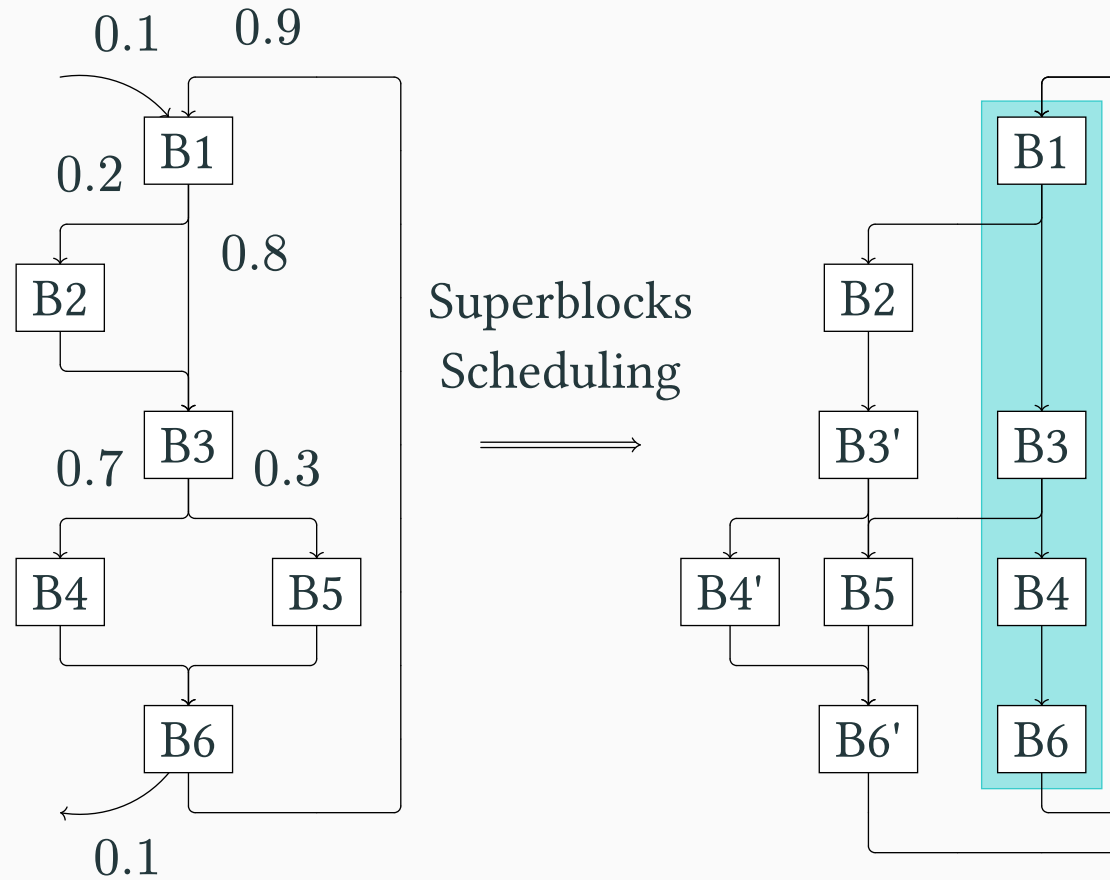
- Not that many application can have a high p (at best, media treatment can go up to 0.95)
- A theoretical and very simple model
- Some other model don't have such pessimistic results (Gustafson's Law)

Cyclic static scheduling

```
while(i != n) {  
    x = a[i];  
    y = b[i];  
    z = x+y;  
    c[i] = z;  
    i++;  
}
```

```
x = a[i];  
y = b[i];  
i++;  
while(i != n) {  
    z = x+y;  
    x = a[i];  
    y = b[i];  
    c[i-1] = z;  
    i++;  
}  
z = x+y;  
c[i-1] = z;
```

Acyclic static scheduling

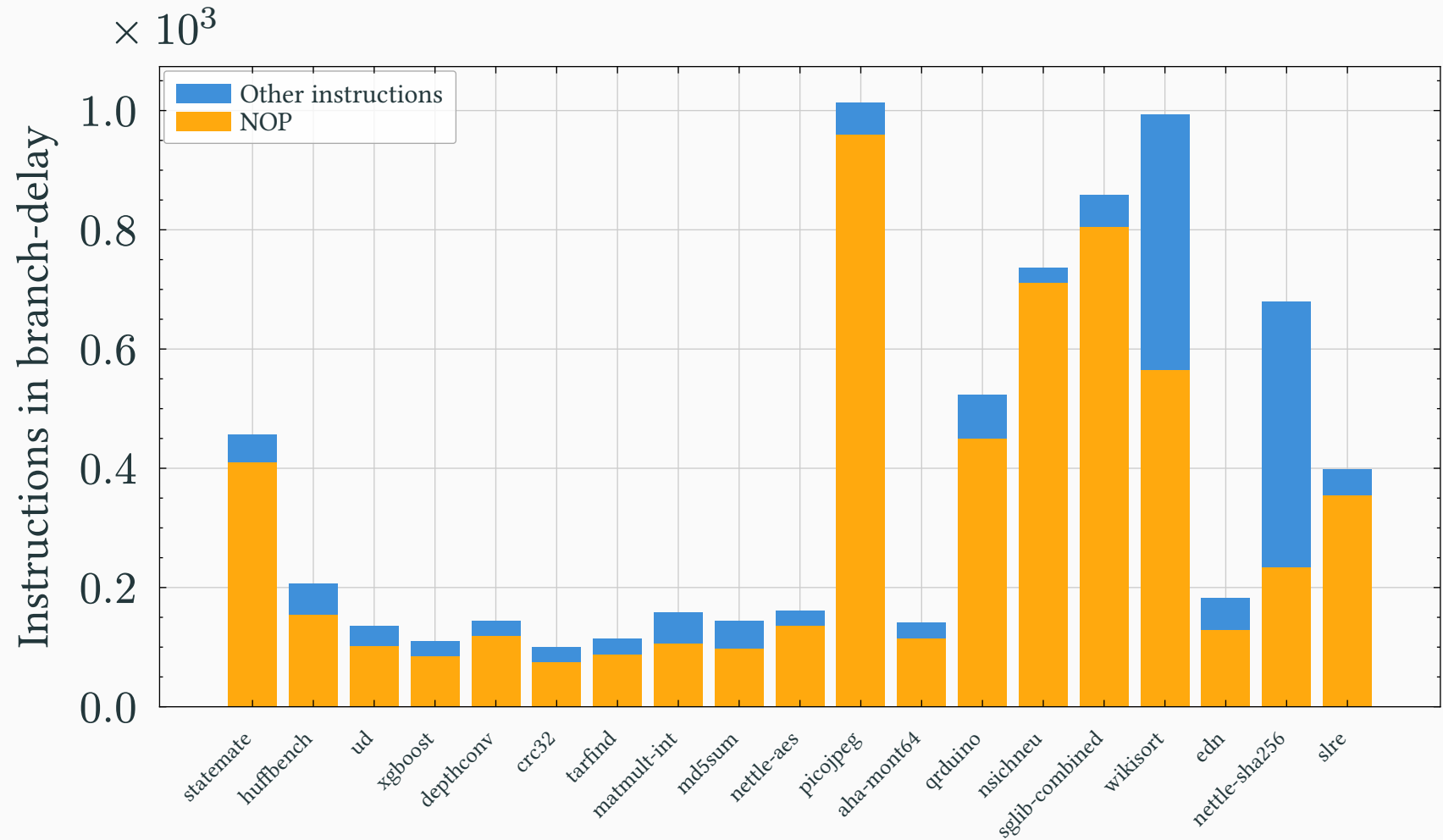


- L1 : 1-3 cycles (≈ 1 ns)
- L2 : 4-40 cycles (≈ 10 ns)
- L3 : 40-100 cycles (≈ 50 ns)
- RAM : 200 cycles (≈ 100 ns)

Early branch computation

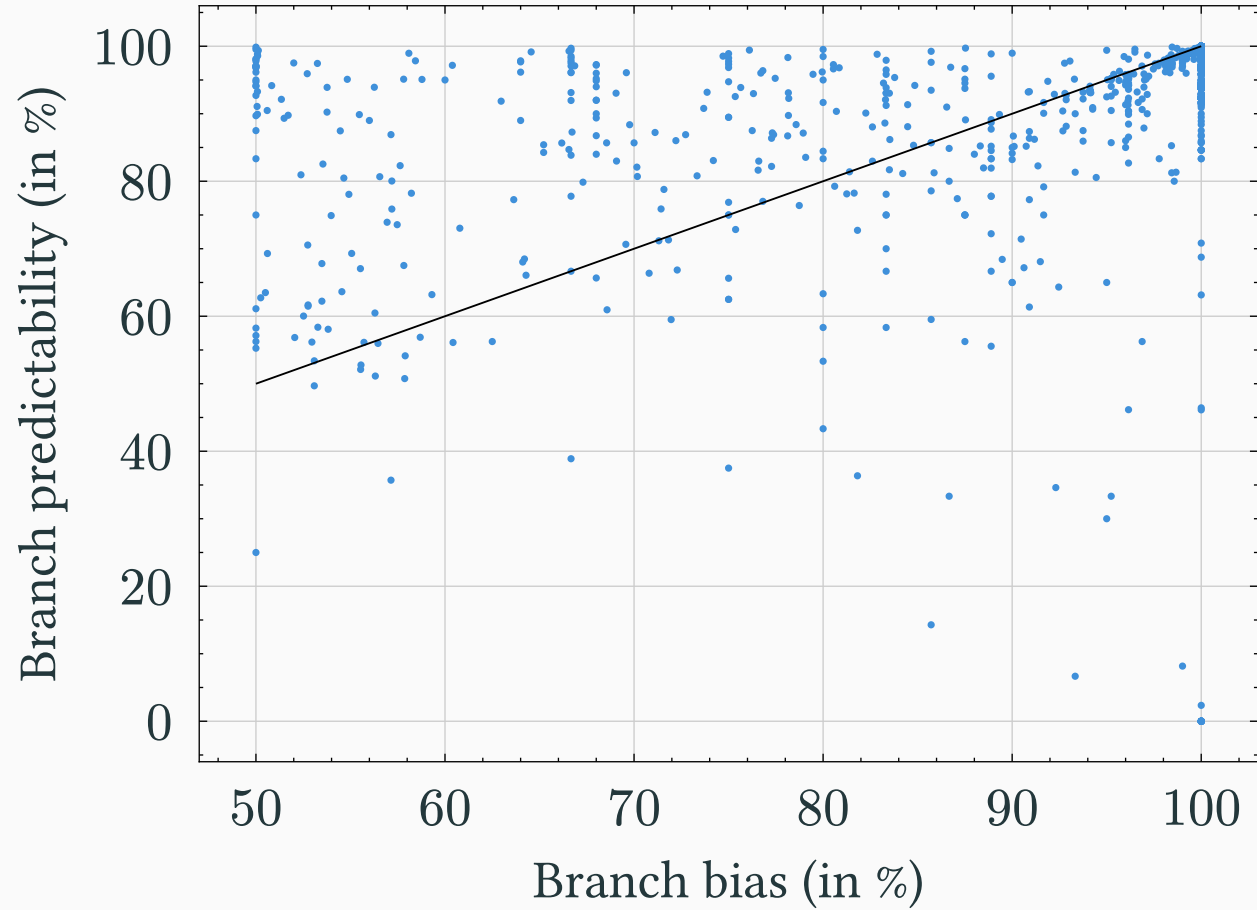
```
1  x++;  
2  if (y != 0)  
3      z++;
```

Early branch computation



1. We want to reorder instruction, but we have branching in the middle of them
2. We're generous : we suppose we have statistics about the execution
3. Static scheduling $\Rightarrow$ only one schedule for each part of the code

Bias vs predictability

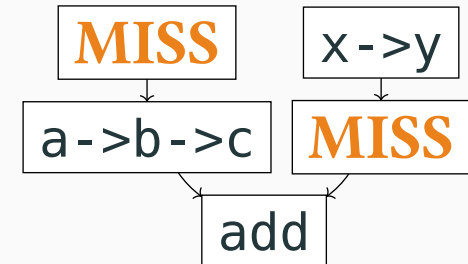
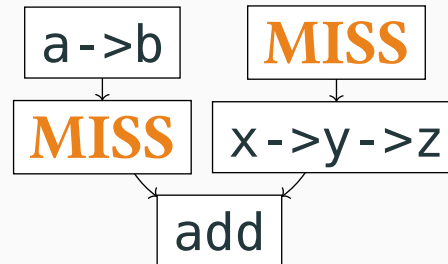
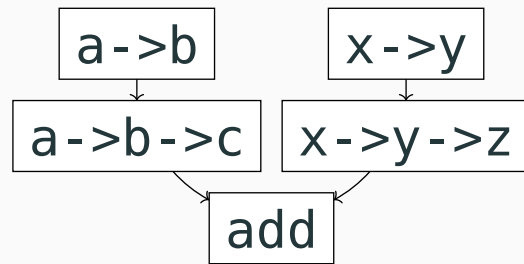


Speculation isn't dynamism

```
1 int u = a->b->c + x->y->z;
```

Speculation isn't dynamism

```
1  int u = a->b->c + x->y->z;
```



McFarlin experimentation

1. Execute benchmark on OoO simulator
2. Extract code snippets out of the trace
3. Filter them and keep the most frequent one
4. Use these snippet as a database of “static schedules”
5. Modify the frontend of the simulator to use this database
6. Execute again the benchmarks on the simulator

McFarlin experimentation

1. Execute benchmark on OoO simulator
2. Extract code snippets out of the trace
3. Filter them and keep the most frequent one
4. Use these snippet as a database of “static schedules”
5. Modify the frontend of the simulator to use this database
6. Execute again the benchmarks on the simulator

We removed all dynamism 😊 !

AKA we added speculation to in-Order

Both instruction dispatched

