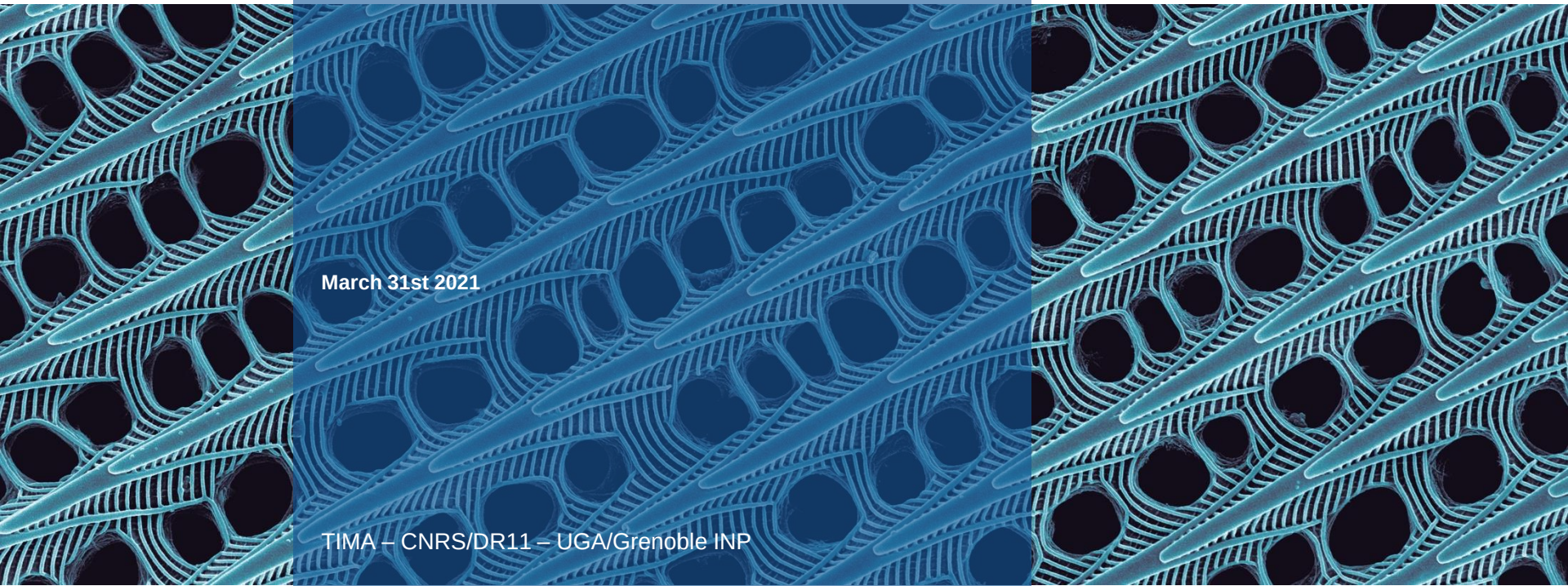
RISC-V for High Performance General Purpose Processors

Arthur Perais (arthur.perais@univ-grenoble-alpes.fr)



March 31st 2021

TIMA – CNRS/DR11 – UGA/Grenoble INP

This Talk – Touching on a Couple Issues

- **Approaching RISC-V from the statement « ISA has no impact on performance »**

- **One example of a microarchitectural performance feature enabled by RISC-V**

Disclaimer : This is my opinion from my experience so far

Context : Single-Thread is Here to Stay

- **Parallel workloads still need sequential performance**

$$Speedup = \frac{1}{(1 - P) + \frac{P}{\#cores}}$$

Amdahl's Law

P : Parallel fraction of the program

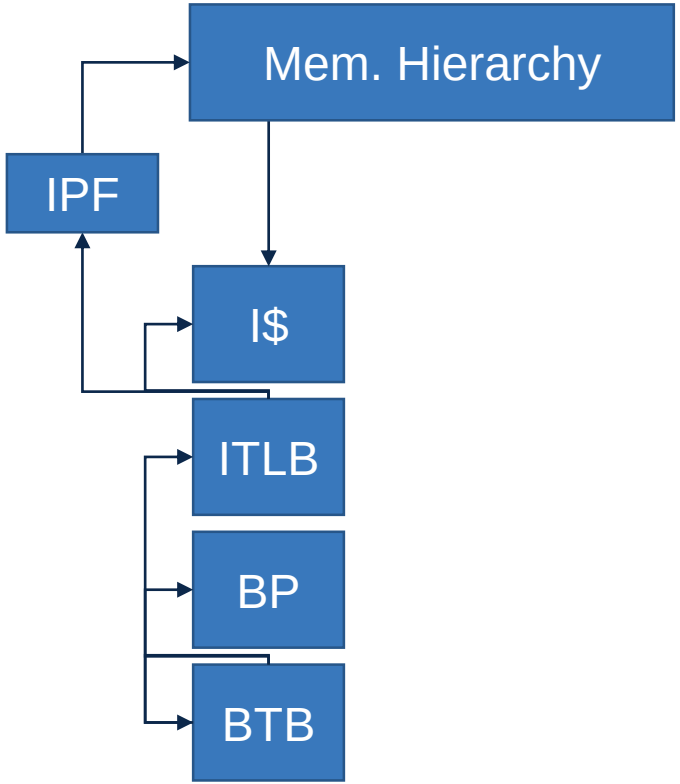
- **Many workloads without (much) DLP/TLP**
  - Sequential performance paramount

ISA is Irrelevant in High Performance Designs

Fact ?

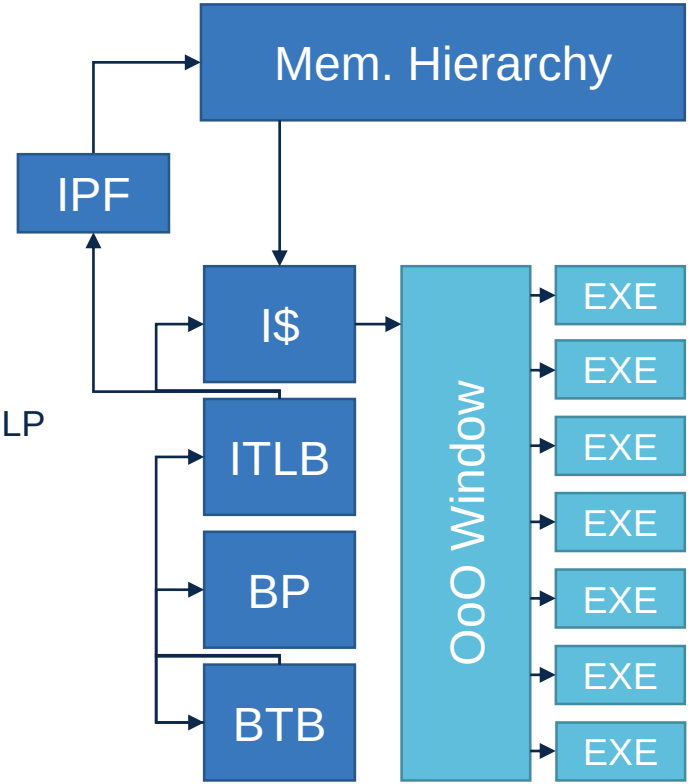
What do we Need for High Sequential Performance ?

- **Fast instruction delivery**
  - Branch Predictors (cond., ind.), BTB
  - I-Caches/TLBs/Prefetchers



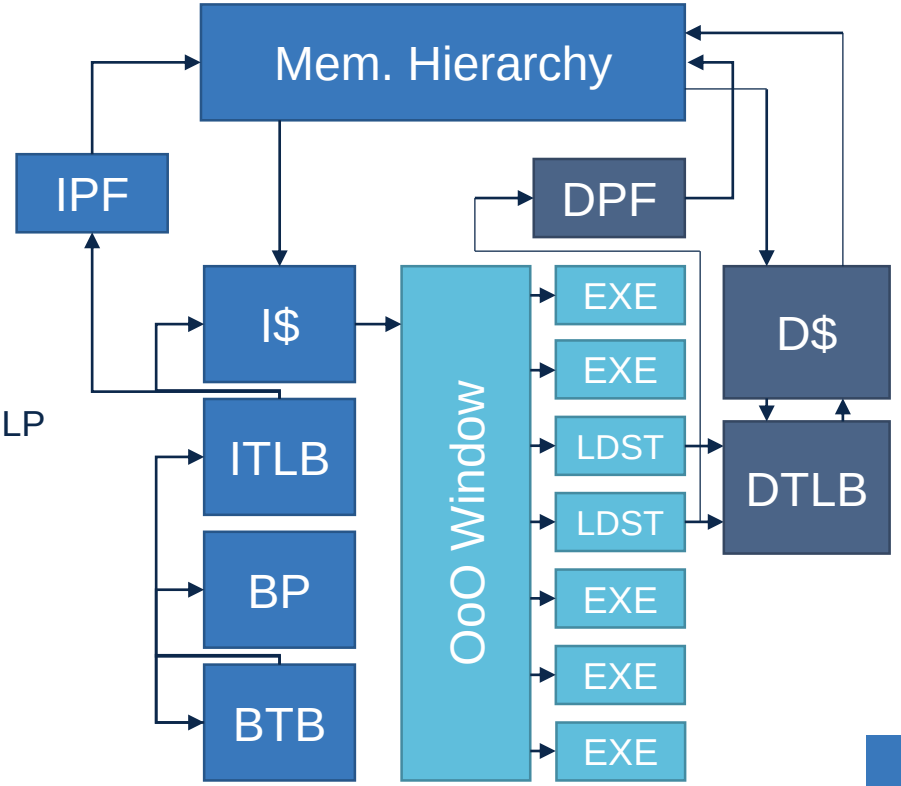
What do we Need for High Sequential Performance ?

- **Fast instruction delivery**
  - Branch Predictors (cond., ind.), BTB
  - I-Caches/TLBs/Prefetchers
- **Minimize structural stalls**
  - Large instruction window to extract ILP/MLP
  - Large buffers
  - Wide execution



What do we Need for High Sequential Performance ?

- **Fast instruction delivery**
  - Branch Predictors (cond., ind.), BTB
  - I-Caches/TLBs/Prefetchers
- **Minimize structural stalls**
  - Large instruction window to extract ILP/MLP
  - Large buffers
  - Wide execution
- **Fast data delivery**
  - D-Caches/TLBs/Prefetchers



Is ISA itself going to impact the level of performance we can reach ?

- **ISA « Tax » to enable high performance (not exhaustive)**
  - x86 : Legacy
    - Variable-length encoding. Tax = Uop Cache
    - Several others (push/pop dependency, partial reg. write, …)

Is ISA itself going to impact the level of performance we can reach ?

- **ISA « Tax » to enable high performance (not exhaustive)**
  - x86 : Legacy
    - Variable-length encoding. Tax = Uop Cache
    - Several others (push/pop dependency, partial reg. write, …)
  - ARMv8 : No obvious quirk that I have dealt with so far
    - I am sure we can find some

Is ISA itself going to impact the level of performance we can reach ?

- **ISA « Tax » to enable high performance (not exhaustive)**
  - x86 : Legacy
    - Variable-length encoding. Tax = Uop Cache
    - Several others (push/pop dependency, partial reg. write, …)
  - ARMv8 : No obvious quirk that I have dealt with so far
    - I am sure we can find some
  - **RISC-V : ?**

Executing a lot of Instructions

- **ISA-independent problem : Instructions consume resources**
  - Physical register
  - Instruction Queue (IQ) entry
  - Load/Store Queue (LSQ) Entry
  - Reorder Buffer (ROB) Entry

Executing a lot of Instructions

- **ISA-independent problem : Instructions consume resources**
  - Physical register
  - Instruction Queue (IQ) entry
  - Load/Store Queue (LSQ) Entry
  - Reorder Buffer (ROB) Entry

- **ISA-dependent problem : Work per instruction**

Executing a lot of Instructions

- **ISA-independent problem : Instructions consume resources**
  - Physical register
  - Instruction Queue (IQ) entry
  - Load/Store Queue (LSQ) Entry
  - Reorder Buffer (ROB) Entry

- **ISA-dependent problem : Work per instruction**

C code

```
uint64_t a = *c
uint64_t b = *(c + 1)
c = c + 2
```

Executing a lot of Instructions

- **ISA-independent problem : Instructions consume resources**
  - Physical register
  - Instruction Queue (IQ) entry
  - Load/Store Queue (LSQ) Entry
  - Reorder Buffer (ROB) Entry

- **ISA-dependent problem : Work per instruction**

C code	ARMv8
uint64_t a = *c uint64_t b = *(c + 1) c = c + 2	ldp x1, x2, [x3], #16

Executing a lot of Instructions

- **ISA-independent problem : Instructions consume resources**
  - Physical register
  - Instruction Queue (IQ) entry
  - Load/Store Queue (LSQ) Entry
  - Reorder Buffer (ROB) Entry

- **ISA-dependent problem : Work per instruction**

C code	ARMv8	RV64G

uint64_t a = *c
uint64_t b = *(c + 1)
c = c + 2
	ldp x1, x2, [x3], #16	LD x1, 0(x3)
		LD x2, 8(x3)
		ADDI x3, x3, 16

Executing a lot of Instructions

Do I use 3x pipeline resources to do the same amount of work in RISC-V ?

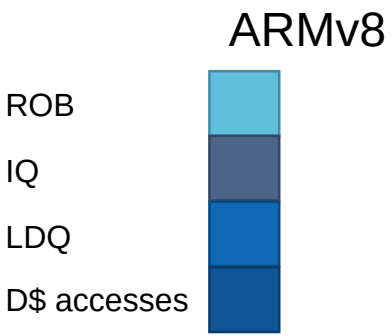
- **High-performance context : Probably ?**
  - ldp x1, x2, [x3], #16 = 1 internal operation (**uop**)
    - 1 ROB, 1 Scheduler, 1 LDQ, 1 D-Cache access

Executing a lot of Instructions

Do I use 3x pipeline resources to do the same amount of work in RISC-V ?

- **High-performance context : Probably ?**
  - ldp x1, x2, [x3], #16 = 1 internal operation (**uop**)
    - 1 ROB, 1 Scheduler, 1 LDQ, 1 D-Cache access
  - RISC-V = 3 uop = 3 ROB/Scheduler, 2 LDQ, 2 D-Cache access. **Same work done.**

```
uint64_t a = *c
uint64_t b = *(c + 1)
c = c + 2
```

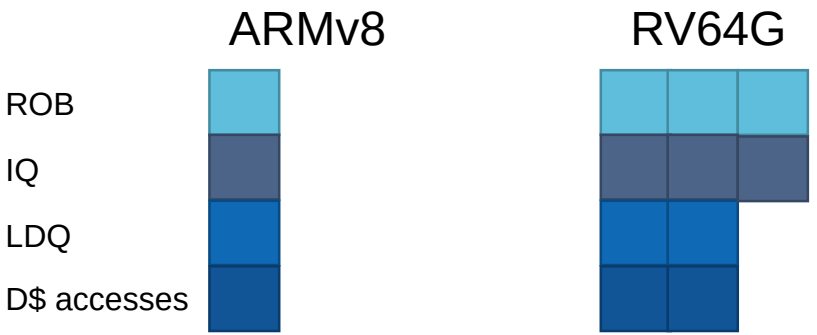


Executing a lot of Instructions

Do I use 3x pipeline resources to do the same amount of work in RISC-V ?

- **High-performance context : Probably ?**
  - ldp x1, x2, [x3], #16 = 1 internal operation (**uop**)
    - 1 ROB, 1 Scheduler, 1 LDQ, 1 D-Cache access
  - RISC-V = 3 uop = 3 ROB/Scheduler, 2 LDQ, 2 D-Cache access. **Same work done.**
  - Other example : ldr x1, [x2 + x3] pretty useful, not supported natively in RISC-V

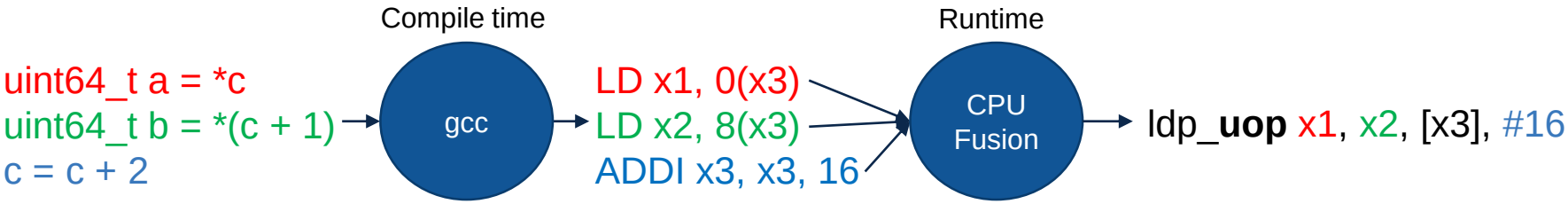
```
uint64_t a = *c
uint64_t b = *(c + 1)
c = c + 2
```



Executing a lot of Instructions

Do I use 3x pipeline resources to do the same amount of work in RISC-V ?

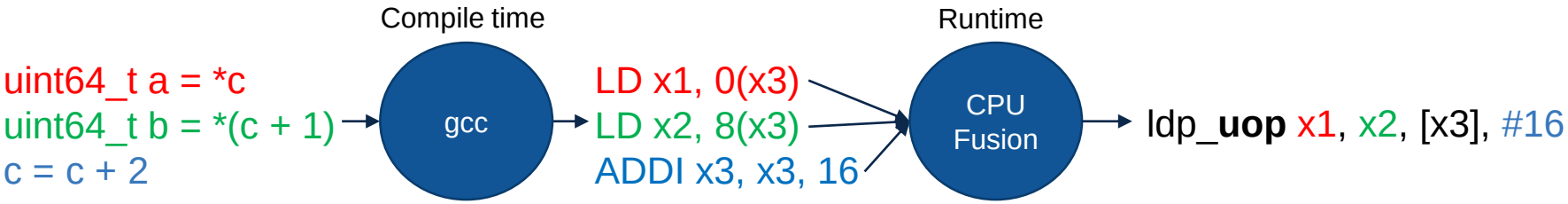
- **Instruction fusion is envisioned to save the day**



Executing a lot of Instructions

Do I use 3x pipeline resources to do the same amount of work in RISC-V ?

- **Instruction fusion is envisioned to save the day**



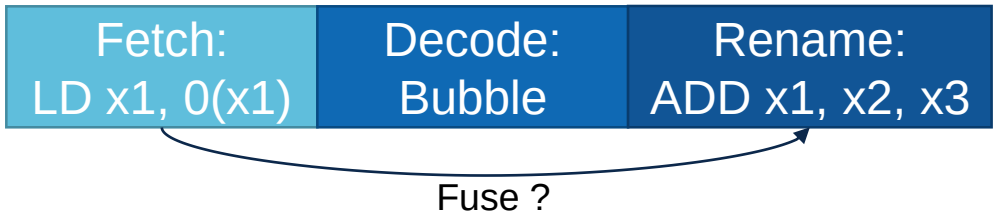
RISC-V high performance « tax » : Fusion

Instruction Fusion - Considerations

- **Impact on microarchitecture :**

ADD x1, x2, x3
LD x1, 0(x1)

  - What if instructions do not enter the pipeline together ? Fusing accross **cycles** ?

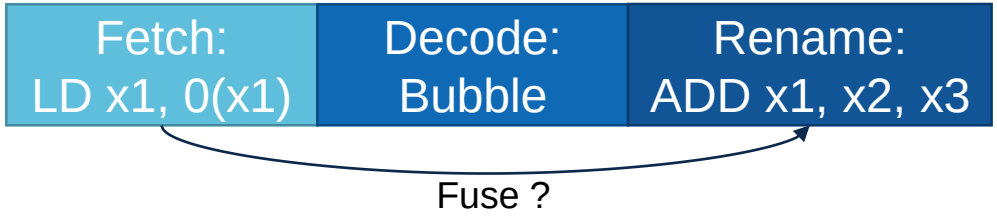


Instruction Fusion - Considerations

- **Impact on microarchitecture :**

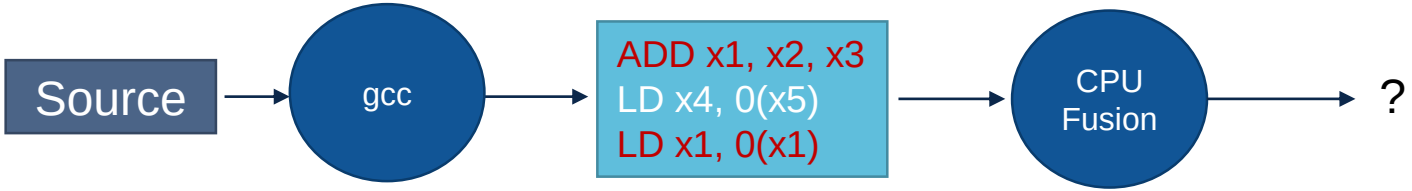
ADD x1, x2, x3
LD x1, 0(x1)

  - What if instructions do not enter the pipeline together ? Fusing accross **cycles** ?
    - Potentially hard (instructions cycles away may be far away physically)
    - If cannot fuse accross cycles**, missed opportunity**
      - ARMv8/x86 have load base reg + offset reg as part of the ISA : **0 missed opportunities**



Instruction Fusion - Considerations

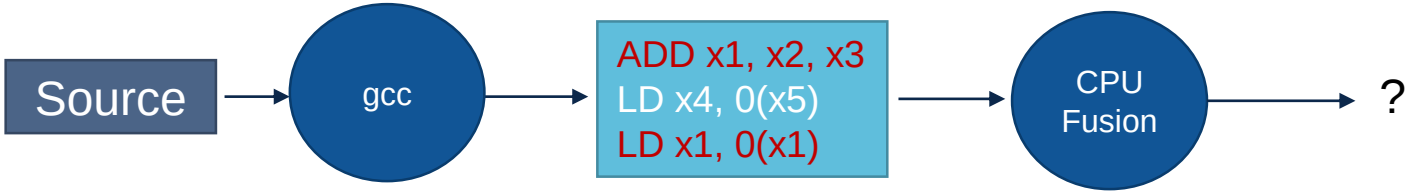
- **Compiler intervention :**



  - Can HW fuse « over » instructions ?

Instruction Fusion - Considerations

- **Compiler intervention :**



  - Can HW fuse « over » instructions ?
    - Yes : Complexity to « unfuse » if LD x4, 0(x5) causes an exception
    - No : **Missed opportunity** if compiler is not aware of available fusion idioms
      - Again, ARMv8/x86 have load base reg + offset reg as part of the ISA : **0 missed opportunities**

Instruction Fusion - Considerations

- **« Those issues are ISA independent »**
  - x86/ARMv8 : cmp + branch fusion, important, but just **one** idiom
    - Remove it, lose 0-5% perf (ballpark)

Instruction Fusion - Considerations

- **« Those issues are ISA independent »**
  - x86/ARMv8 : cmp + branch fusion, important, but just **one** idiom
    - Remove it, lose 0-5% perf (ballpark)
  - RISC-V :
    - ADD + LD = Load [base reg + offset reg]
    - LD + LD = Load pair
    - ADDI + LD (+ LD) = Pre-indexed load (pair)
    - LD + (LD +) ADDI = Post-indexed load (pair)
    - Shift + ALU = Shifted ALU
    - Etc.
  - **Many idioms needed to equalize amount of work per HW resource vs. x86/ARMv8**
    - Likely a pillar of high-performance RISC-V

ISA Irrelevant ?

- **All ISAs have a « tax » or « entry ticket » for high performance**
  - To deal with the ISA aspects that are not adapted to high performance

- **No evidence that ISA makes or breaks performance *so far***
  - Intel/Amd demonstrate that x86 can lead in sequential performance
  - Apple M1 demonstrates that ARMv8 can reach x86-level of sequential performance
  - Assuming no blocking issue with scaling instruction fusion, high performance RISC-V seems possible

Does RISC-V Enable Specific Performance Features ?

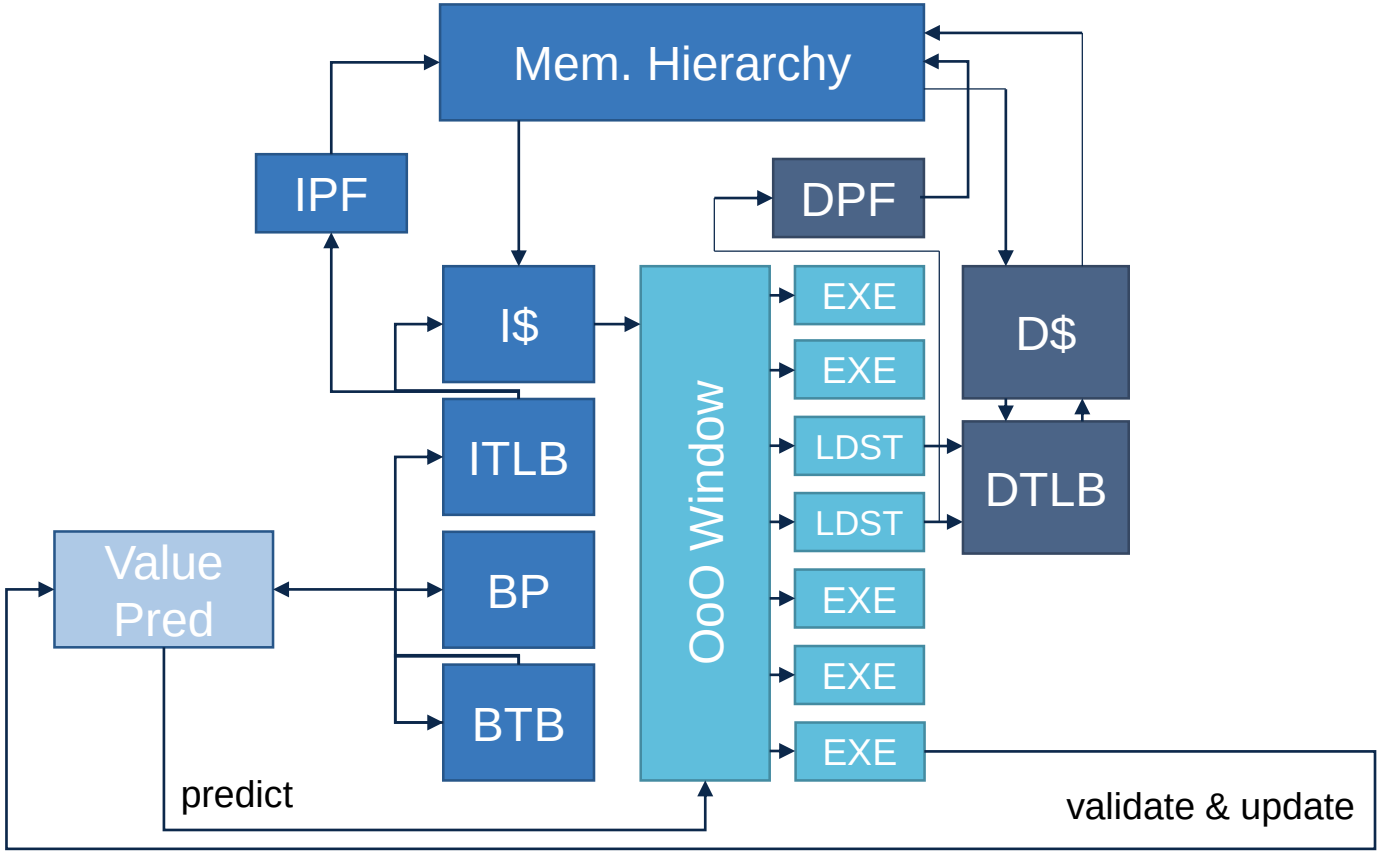
One example feature

One Example

- **Read-After-Write dependencies**
  - Encode program semantics
  - True dependencies
  - Limit ILP, hence the work we can do each cycle

- **Value Prediction**
  - Predict instructions' results (like branch direction)

Value Prediction



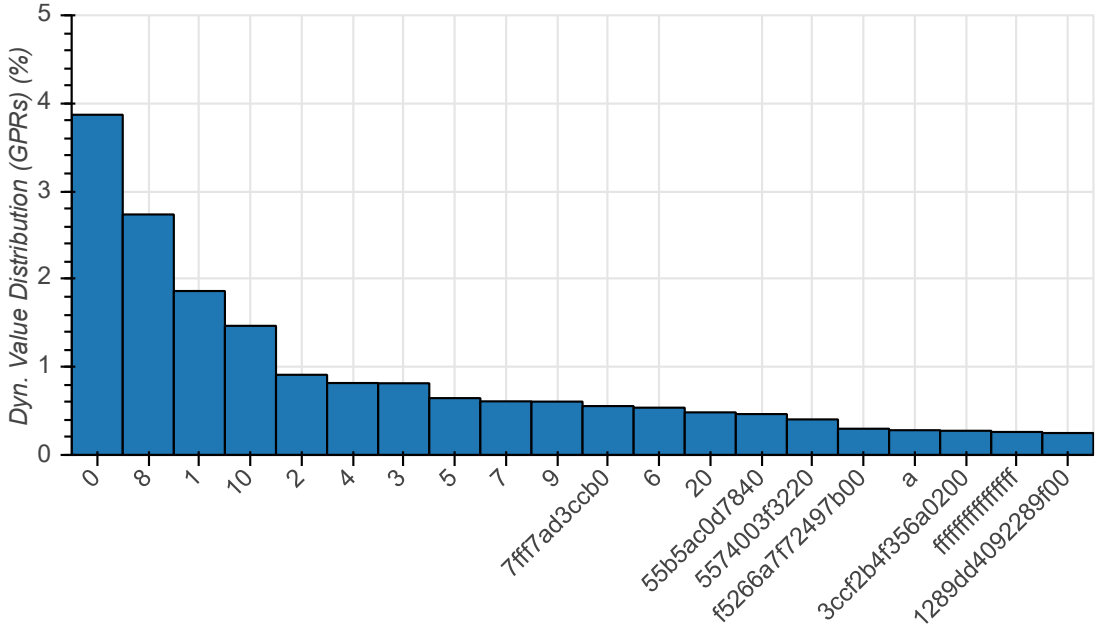
Value Prediction

- **Value Prediction in General :**
  - Predictor is big (predicting 64-bit values)
  - Additional datapath to send multiple 64-bit values from the predictor to the backend each cycle

  ➢ Improves performance but adds complexity and costs area/power

Value Prediction

- **Variation : Limited Value Prediction**
  - Distribution of 64-bit results in SPEC2k17 (x86, train inputs)
  - 0x0 and 0x1 = close to 6% of the dynamic results



Perais, « A Case for Speculative Strength Reduction », IEEE CAL, 2021

Value Prediction

- **Variation : Limited Value Prediction**
  - Just predict zeroes/ones
    - Smaller predictor
    - Can write predictions to PRF implicitly through renaming and hardwired zero/one physical registers

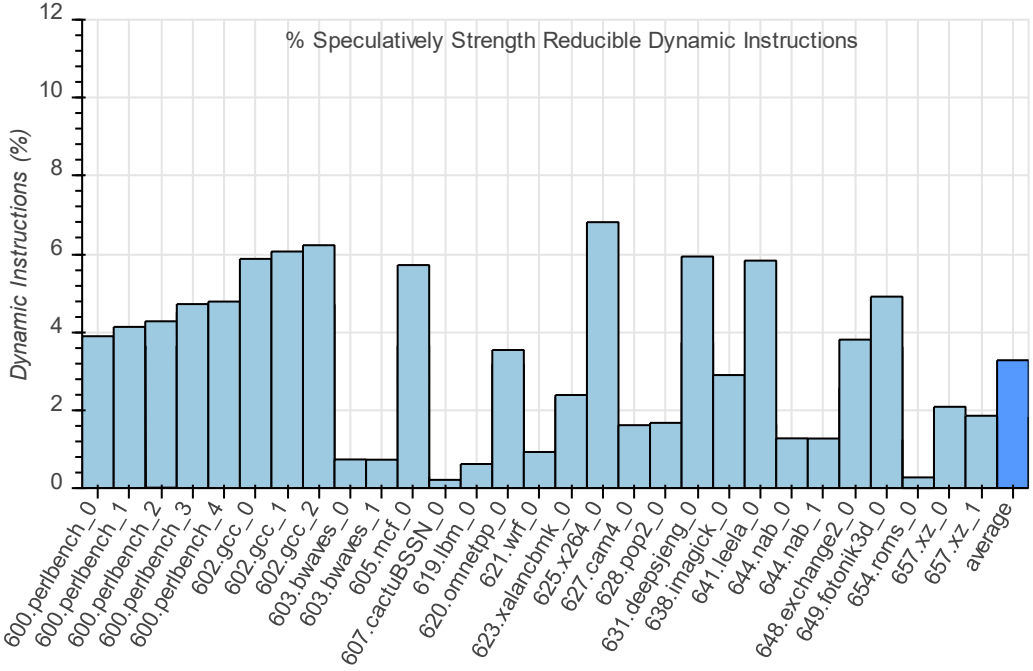
Speculative Strength Reduction (SSR)

- **Variation : Limited Value Prediction**
  - Just predict zeroes/ones
    - Smaller predictor
    - Can write predictiones to PRF implicitly through renaming and hardwired zero/one physical registers

- **Build on top : Speculative Strength Reduction (SSR)**
  - Execute instructions at rename, enabled by predicting 0x0/0x1
  - **add r0, r1, r2** reduces to **mov r0, r1** if r2 predicted to be 0 (**move eliminated**)
    - Several others
  - Saves on **latency** (0-cycle) and **backend resources** (scheduler, functional unit)

Speculative Strength Reduction (SSR)

- **Percentage of dynamic instructions that can be speculatively strength reduced**
  - State of the art value predictor, x86, SPEC2k17speed *train*



Perais, « A Case for Speculative Strength Reduction », IEEE CAL, 2021

Impact of ISA on SSR

- **Strength Reduction and x86**
  - Most x86 instructions have side effects (writing the ccflags)
    - Instruction reduced to move still need to go compute the flags in the backend
    - SSR helps latency, but does not save as much as it could on pipeline resources

Impact of ISA on SSR

- **Strength Reduction and x86**
  - Most x86 instructions have side effects (writing the ccflags)
    - Instruction reduced to move still need to go compute the flags in the backend
    - SSR helps latency, but does not save as much as it could on pipeline resources

- **Strength Reduction and ARMv8**
  - Only few instructions write the ccflags and cannot disappear at Rename.
    - However : cmp + branch fusion can deal with those

Impact of ISA on SSR

- **Strength Reduction and x86**
  - Most x86 instructions have side effects (writing the ccflags)
    - Instruction reduced to move still need to go compute the flags in the backend
    - SSR helps latency, but does not save as much as it could on pipeline resources
- **Strength Reduction and ARMv8**
  - Only few instructions write the ccflags and cannot disappear at Rename.
    - However : cmp + branch fusion can deal with those
- **Strength Reduction and RISC-V**
  - No ccflags = no side effects
    - Reduced instructions can always disappear at Rename

Not a huge advantage vs ARMv8, but a significant one vs x86

Conclusion

- **High-performance RISC-V**
  - **If you pay the fusion tax**
  - SSR : Dataflow-based optimization facilitated by ISA regularity

  - Statement about performance, not other features (virtualization, security, etc.)

That’s all folks !

SLS : http://tima.univ-grenoble-alpes.fr/tima/fr/sls/slsoverview.html

Perso : http://aperais.fr

www.cnrs.fr

Backup : RISC-V Fusion Idioms (source : wikichips)

Pattern	Result

// rd = array[offset]
add rd, rs1, rs2
ld rd, 0(rd)
	Fused into an indexed load

// &(array[offset]) slli rd, rs1, {1,2,3} add rd, rd, rs2	Fused into a load effective address

// rd = array[offset]
slli rd, rs1, {1,2,3}
add rd, rd, rs2
ld rd, 0(rd)
	Three-instruction fused into a load effective address

// rd = rs1 & 0xffffffff slli rd, rs1, 32 srli rd, rd, 32	Clear upper word

// rd = imm[31:0]
lui rd, imm[31:12]
addi rd, rd, imm[11:0]
	Load upper immediate

// rd = *(imm[31:0])
lui rd, imm[31:12]
ld rd, imm[11:0](rd)
	Load upper immediate

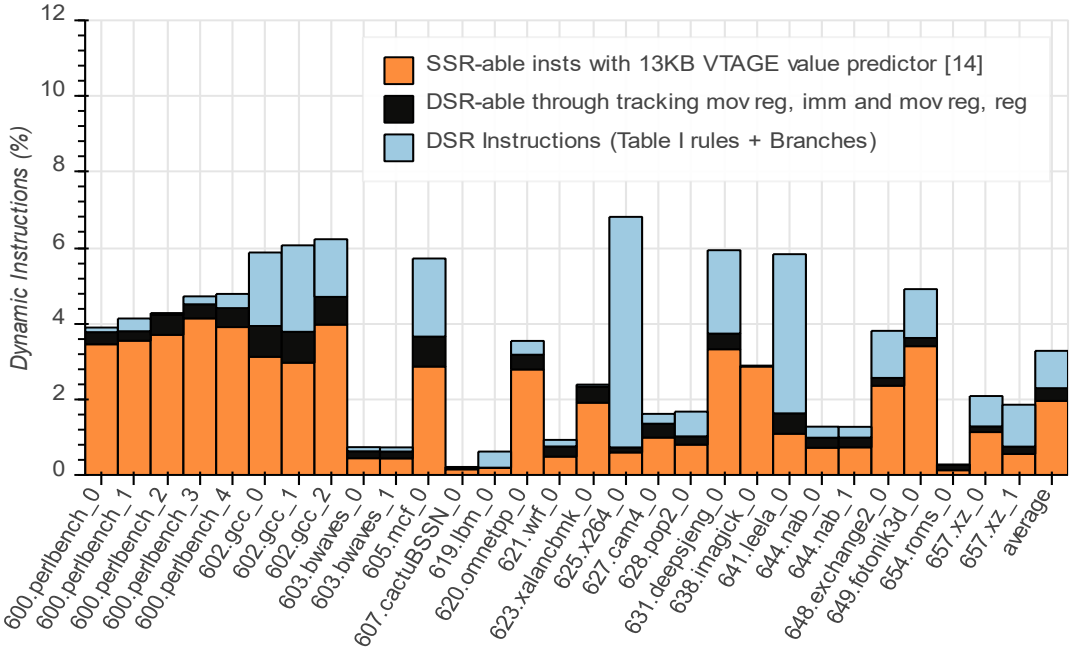
// l[dw] rd, symbol[31:0] auipc rd, symbol[31:12] l[dw] rd, symbol[11:0](rd)	Load global immediate

Pattern	Result
// far jump (1 MB) (AUIPC+JALR) auipc t, imm20 jalr ra, imm12(t)	Fused far jump and link with calculated target address
addiw rd, rs1, imm12 slli rd, rs1, 32 SRLI rd, rs1, 32	Fused into a single 32-bit zero extending add operation
mulh[[S]U] rdh, rs1, rs2 mul rdl, rs1, rs2	Fused into a wide multiply
div[U] rdq, rs1, rs2 rem[U] rdr, rs1, rs2	Fused into a wide divide

// ldpair rd1,rd2, [imm(rs1)]
ld rd1, imm(rs1)
ld rd2, imm+8(rs1)
	Fused into a load-pair

// ldia rd, imm(rs1)
ld rd, imm(rs1)
add rs1, rs1, 8
	Fused into a post-indexed load

Backup : CAL Figure



Perais, « A Case for Speculative Strength Reduction », IEEE CAL, 2021