Improving Out-of-Order energy efficiency

Reconstructing Out-of-Order Issue Queue - Ipoom Jeong et al - MICRO 2022
A Front-end Execution Architecture for High Energy Efficiency - Ryota Shioya et al - MICRO 2014

Performance at the expense of energy efficiency

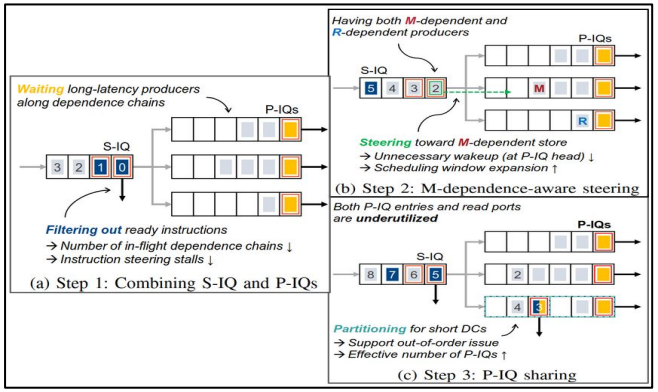
- Out-of-order cores provide high performance at the cost of energy efficiency
- Dynamic scheduling of the OoO IQ is one of the major contributors to this:
  - Data dependencies and availability of ports/FUs for issuing of instructions must be considered
  - These two factors result in complex and costly wakeup and selection circuitry, i.e. high energy consumption
- Solution to high energy consumption: don’t use such complex circuitry (easier said than done)
  - Complexity-effective schemes have been proposed over the years which attempt to mimic the behaviour and characteristics of dynamic scheduling
  - Some of these less costly schemes revolve around the idea of combining multiple energy-efficient in-order IQs, expecting some degree of dynamic scheduling effects between them

Performance at the expense of energy efficiency

- Two to-be-presented approaches to improve energy efficiency:
- Reconstructing Out-of-Order Issue Queue - Ipoom Jeong et al - MICRO 2022
  - The first of the papers to be presented is, in fact, one such scheme dealing in energy-efficient in-order IQs
  - In fact, it combines two previous in-order-IQ-based solutions and merges them
- A Front-end Execution Architecture for High Energy Efficiency - Ryota Shioya et al - MICRO 2014
  - This second paper doesn't deal in energy-efficient in-order IQs
  - Instead, it proposes having an in-order unit/phase of execution (IXU) for ready instructions and an out-of-order execution unit/phase (OXU) to which non-ready instructions are dispatched from the IXU

Zoom out summary

- Both techniques attempt to tackle the issue of bad energy eff. of OoO cores
- Ballerino proposes to do away with the OoO IQ altogether in favor of energy-efficient in-order queues
- FXA suggests inserting an in-order phase before the issue-* stages s.t. this previous in-order phase is so effective that the later OoO phase can be reduced



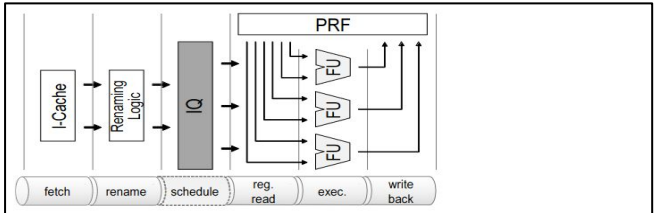


Figure 1: Conventional superscalar architecture.

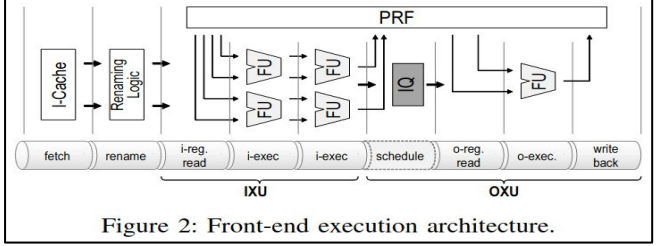


Figure 2: Front-end execution architecture.

First paper: Reconstructing Out-of-Order Issue Queue

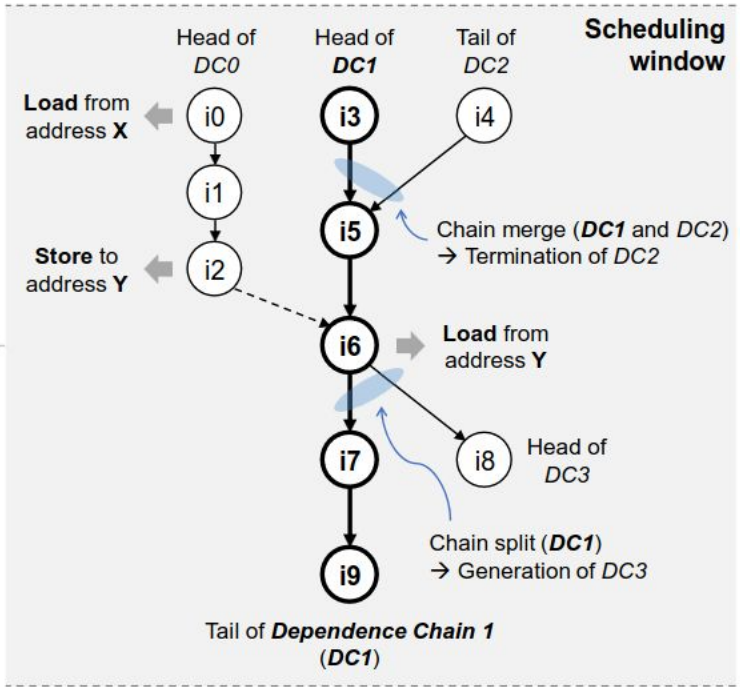
- Novel microarchitecture scheme is proposed, **Ballerino**, from a combination of two previous, compatible microarchitecture designs
- Ballerino is, effectively, a merging of two complexity-effective microarchitecture designs that enables a group of in-order IQs to generate highly optimized issue schedules, i.e. comparable to that of a fully out-of-order IQ while consuming much less energy
- The two conjoined designs are Ballerino = CES + CASINO

Ballerino = **CES** + CASINO

- Complexity-Effective Superscalar (CES) is a *dependence-based* microarchitecture that performs dynamic scheduling by using a group of P-IQs

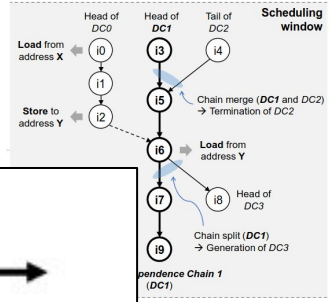
Ballerino = **CES** + CASINO

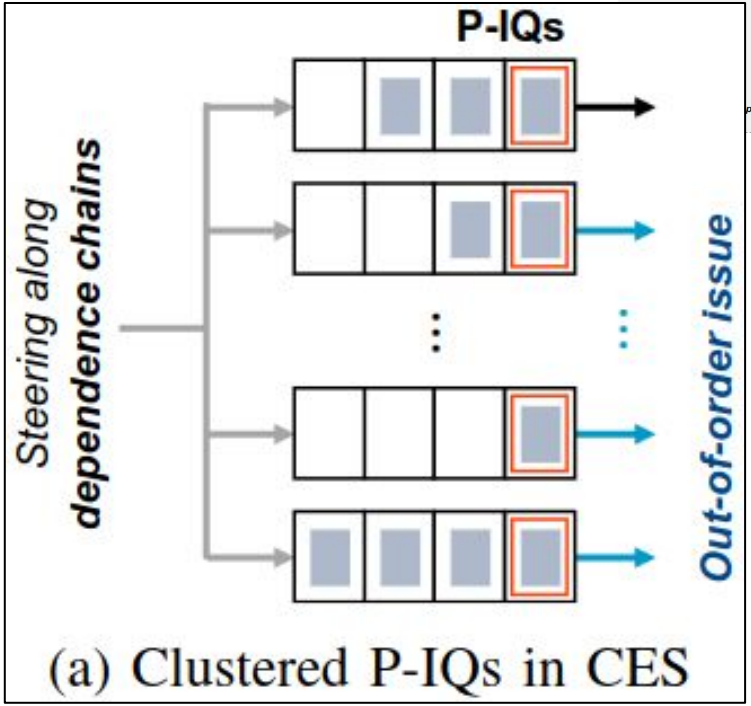
- Complexity-Effective Superscalar (CES) is a *dependence-based* microarchitecture that performs dynamic scheduling by using a group of P-IQs
- In the context of CES, a dependency chain (DC) refers to a sequence of instructions along R-dependencies (solid lines)



Ballerino = **CES** + CASINO

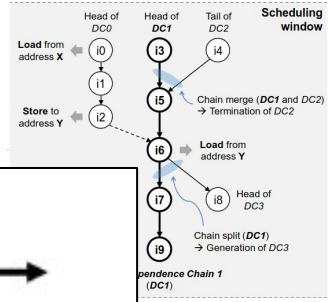
- Complexity-Effective Superscalar (CES) is a *dependence-based* microarchitecture that performs dynamic scheduling by using a group of P-IQs
- In the context of CES, a dependency chain (DC) refers to a sequence of instructions along R-dependencies (solid lines)

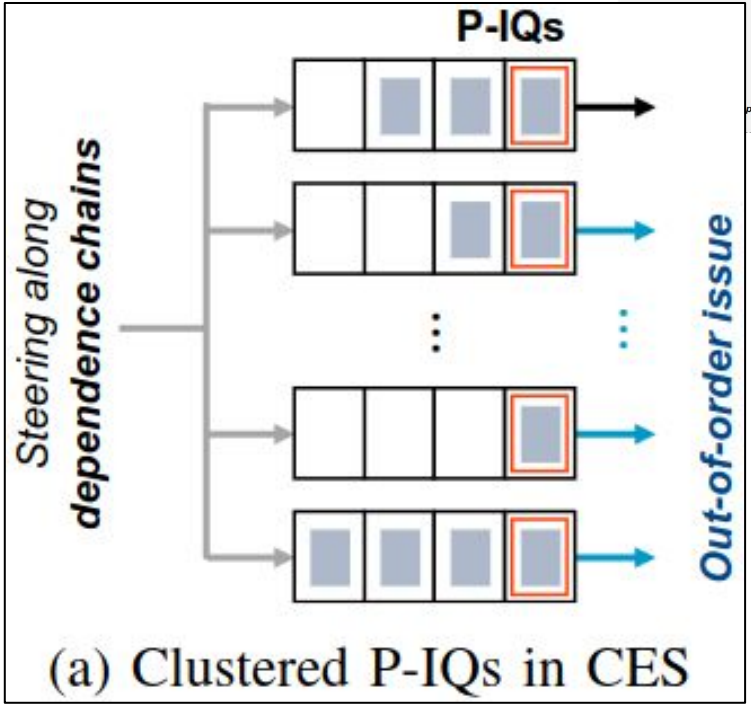




Ballerino = **CES** + CASINO

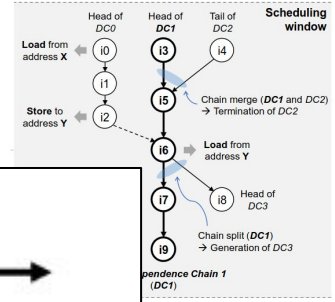
- Dependent instructions (in the same DC) must be executed in sequential order
- Therefore, a significant amount of scheduling energy can be saved by steering each DC into a single P-IQ and examining only its DC's head

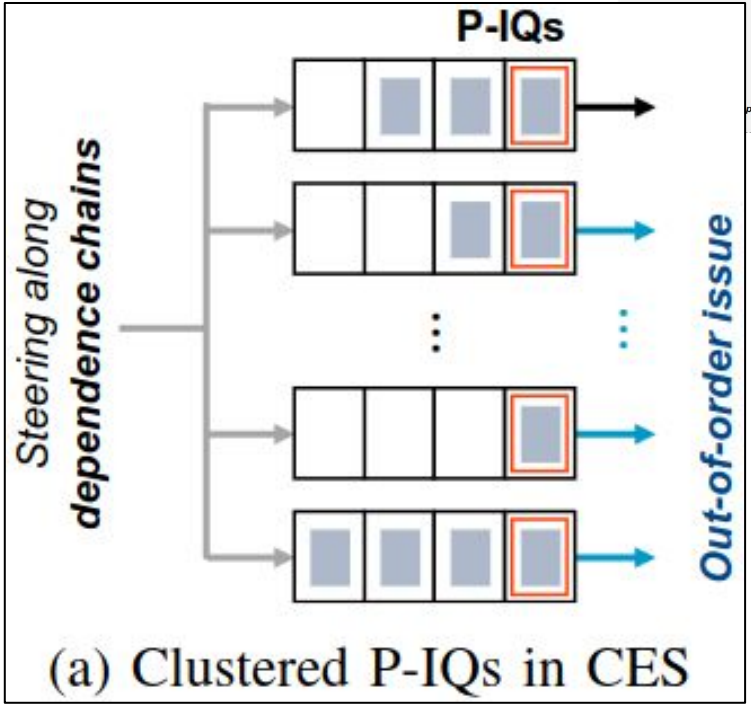




Ballerino = **CES** + CASINO

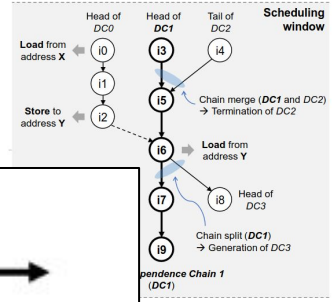
- CES's instruction-steering heuristic puts intrs. into empty P-IQ if:
  - 1) none of its producers are in the P-IQs
  - 2) it becomes a DC head due to chain split
  - 3) there are no free entries in the target P-IQs in which its producers wait for issue
- else: stall

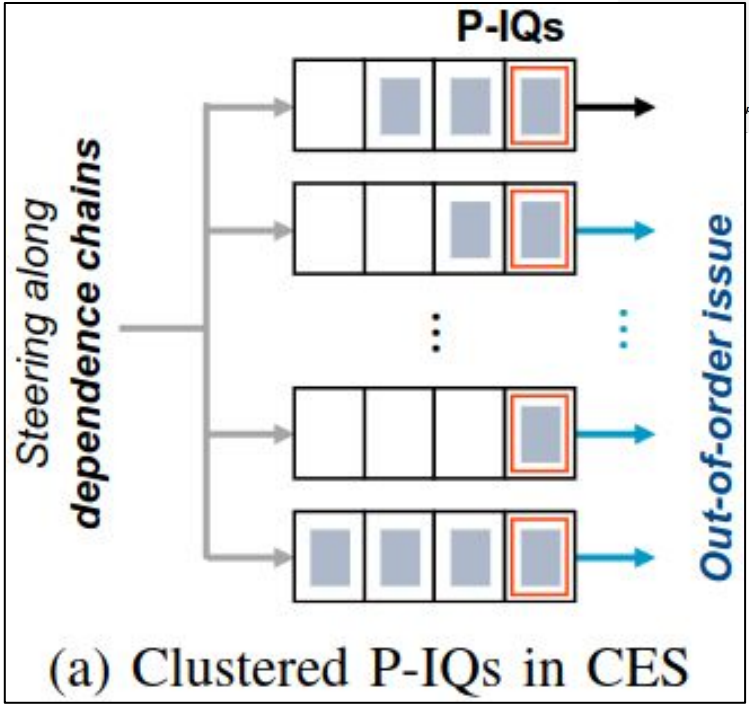




Ballerino = **CES** + CASINO

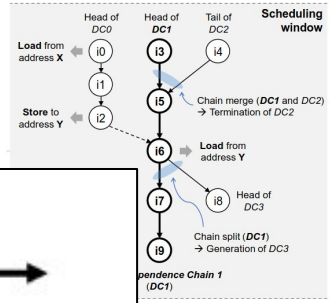
- An instruction could be ready at dispatch (DC heads)…
- Even if it is ready, CES steers them into the P-IQs regardless…
- The major performance bottleneck of CES is a lack of free P-IQs for DC heads (ready-at-dispatch)
- In the Ballerino paper, it is (experimentally) shown that 72% of allocation and 79% stall events are on ready-at-dispatch instructions
- Immediately issuing ready instructions?...

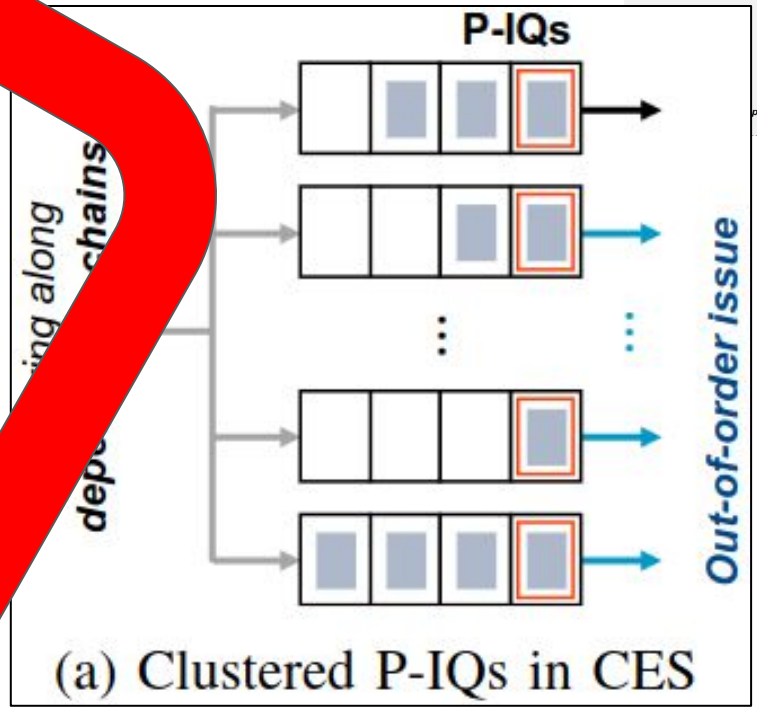




Ballerino = **CES** + CASINO

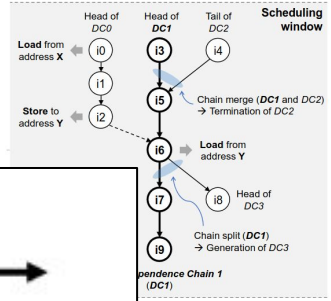
- An instruction could be ready at dispatch…
- Even if it is ready, CES steers them into the P-IQs regardless…
- The major performance bottleneck of CES is a lack of free P-IQs for DC heads (ready-at-dispatch)
- In the Ballerino paper, it is (experimentally) shown that 72% of allocation and 79% stall events are on ready-at-dispatch instructions
- Immediately issuing ready instructions?...

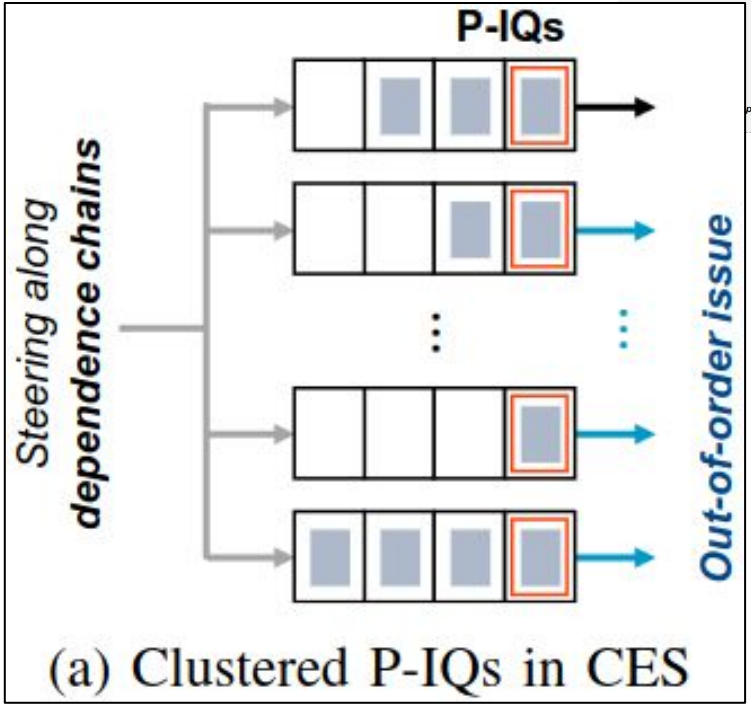




Ballerino = **CES** + CASINO

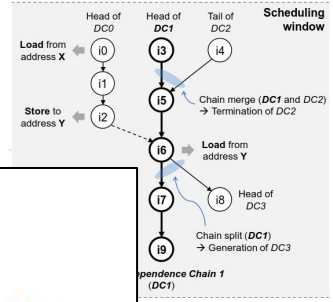
- Adding more P-IQs?
  - According to authors of Ballerino and [10, 26], *"merely increasing the number of the P-IQs is ineffective and impractical considering the growing cost and complexity of circuitry"*.

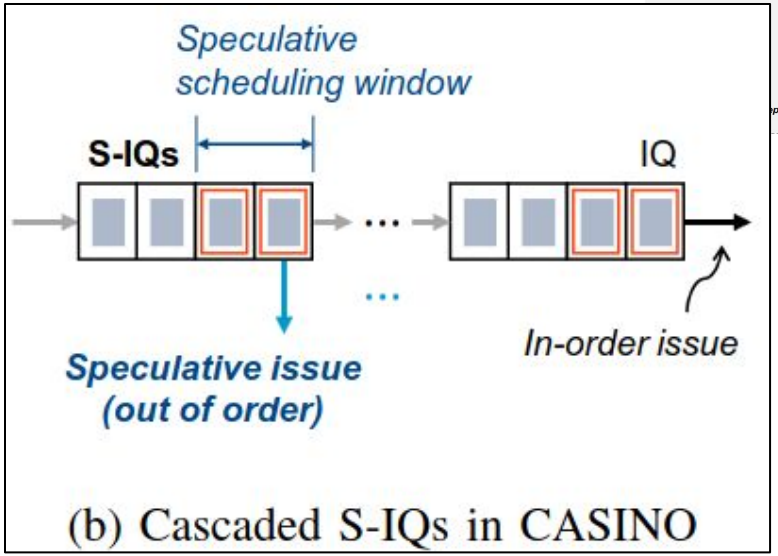




Ballerino = CES + **CASINO**

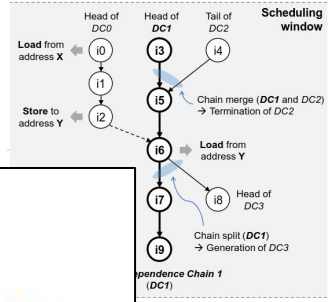
- CASINO captures and issues instructions that become ready shortly after dispatch

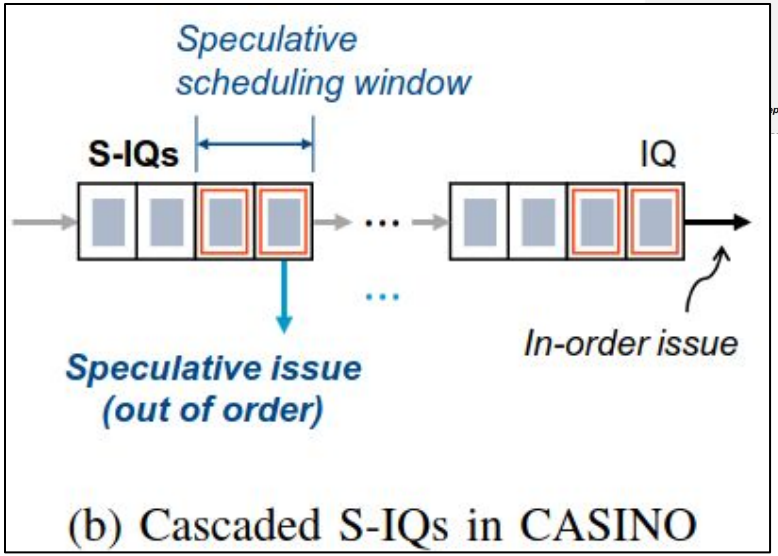




Ballerino = CES + **CASINO**

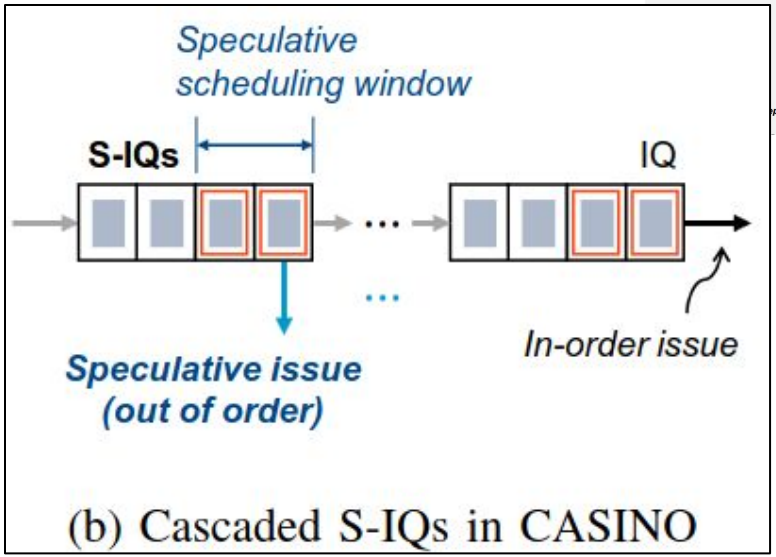
- CASINO captures and issues instructions that become ready shortly after dispatch
- This early-issuing mechanism could be a good solution to tackle CES's allocation/stall events on ready-at-dispatch instructions
- Additionally, the cache-miss intolerance of CASINO could be addressed by CES's P-IQs…

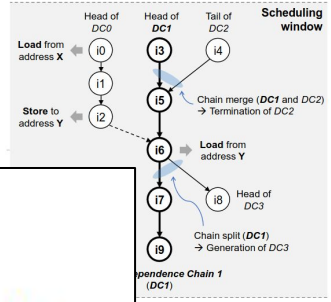




Ballerino = CES + **CASINO**

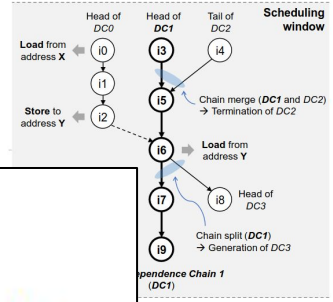
- Additionally, the cache-miss intolerance of CASINO could be addressed by CES's P-IQs…
  - Load-dependent instructions and their subsequent consumers are likely not to be ready at dispatch
  - These instructions dependent on long-latency operations (eventually, probably) enter the last in-order IQ
  - If they finally enter the in-order IQ, they could be mixed with other instructions from different long-latency DCs…

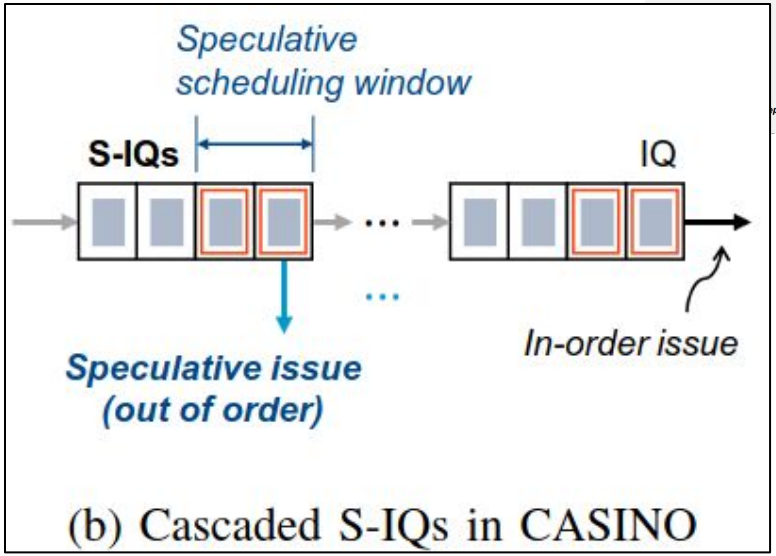




Ballerino = CES + **CASINO**

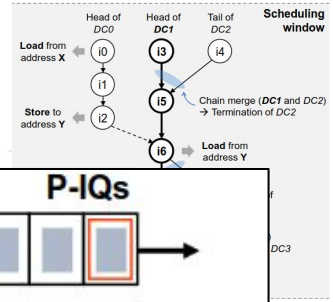
- Additionally, the cache-miss tolerance of CASINO could be addressed by CES's P-IQs…
  - Load-dependent instructions and their subsequent consumers are likely not to be ready at dispatch
  - If these instructions dependent on long-latency operations (eventually, probably) enter the last in-order IQ
  - If they finally enter the in-order IQ, they could be mixed with other instructions from different long-latency DCs…

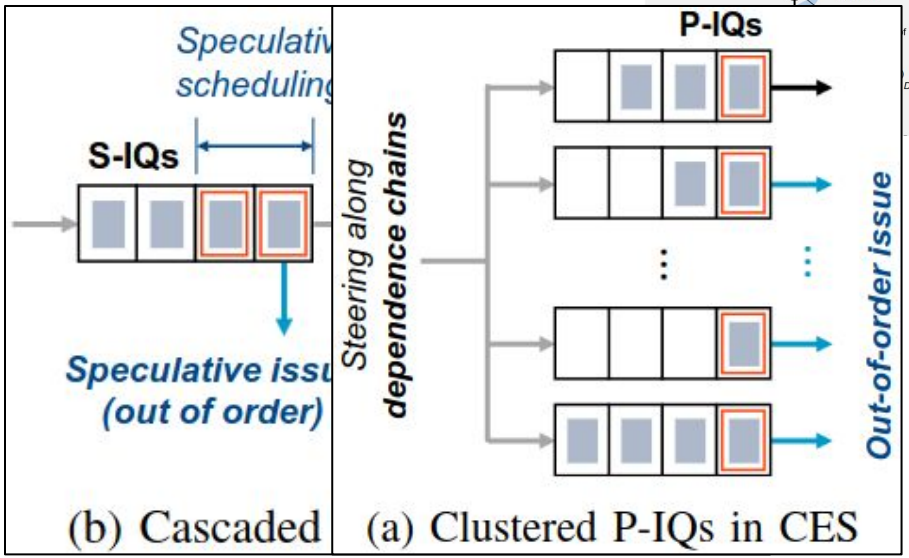


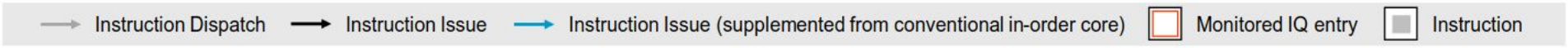


Ballerino = CES + **CASINO**

- Additionally, the cache-miss tolerance of CASINO could be addressed by CES's P-IQs…
  - Load-dependent instructions and their subsequent consumers are likely not to be ready at dispatch
  - These instructions dependent on long-latency operations (eventually, probably) enter the last in-order IQ
  - If they finally enter the in-order IQ, they could be mixed with other instructions from different long-latency DCs…

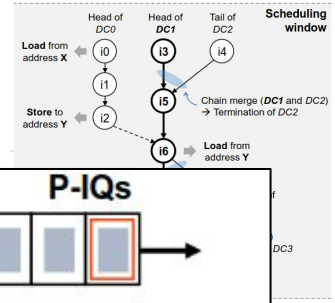


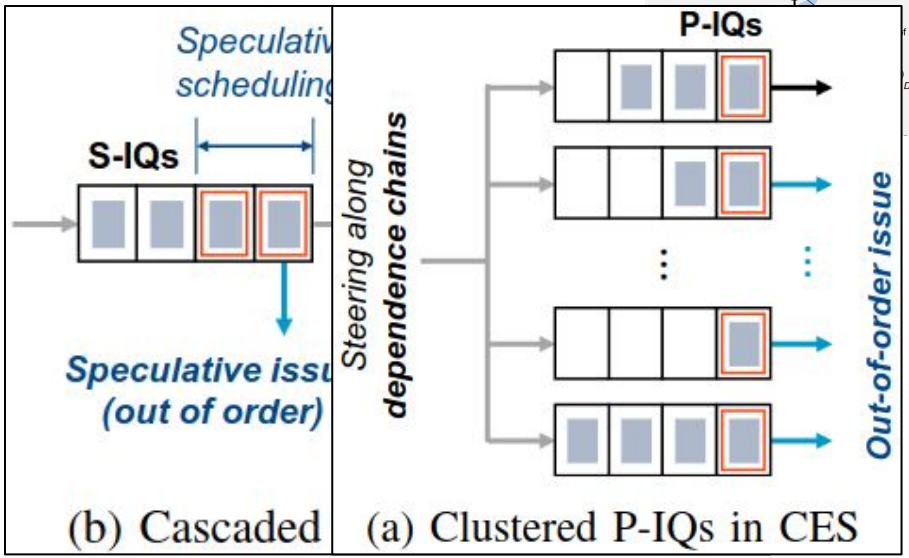


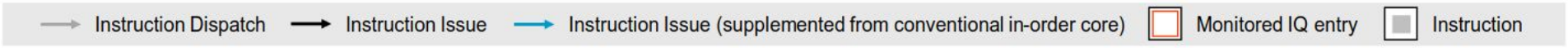


Ballerino = CES + CASINO

- Summing up:

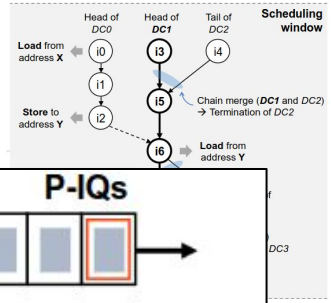


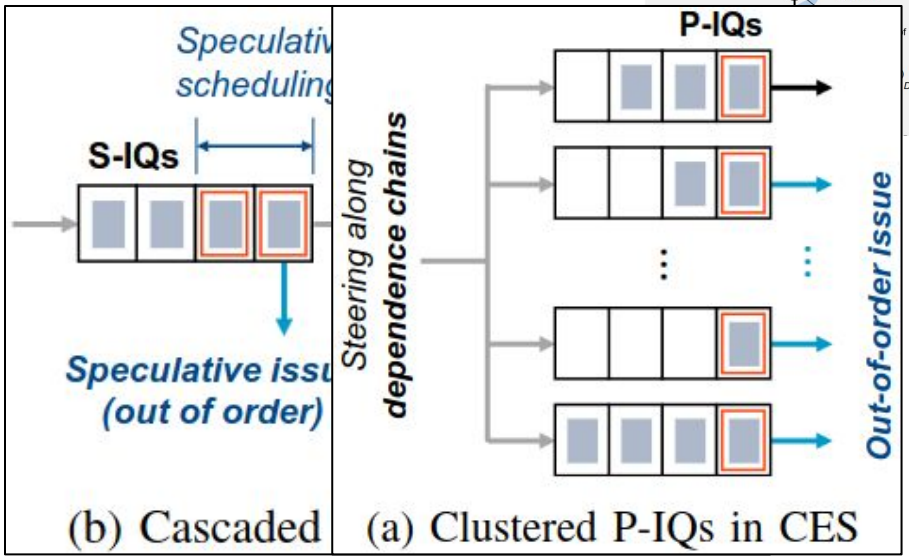




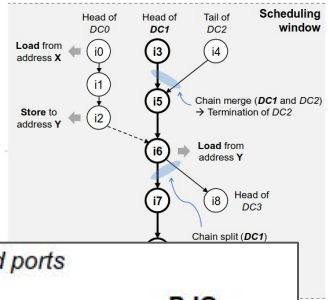
Ballerino = CES + CASINO

- Summing up:
  - The ready-at-dispatch issue functionality of CASINO could significantly mitigate the burden of CES's instruction steering
  - In turn, CES's capability to keep track of multiple DCs can be a good solution to address the cache-miss-intolerance issue in CASINO





Ballerino step-by-step



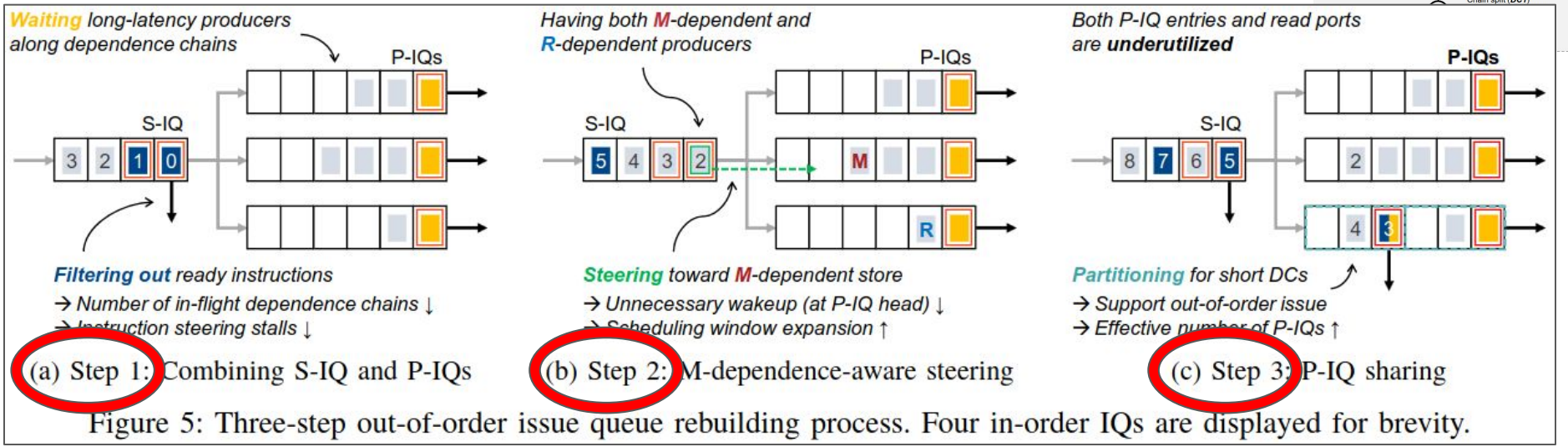
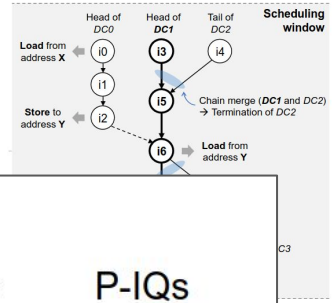
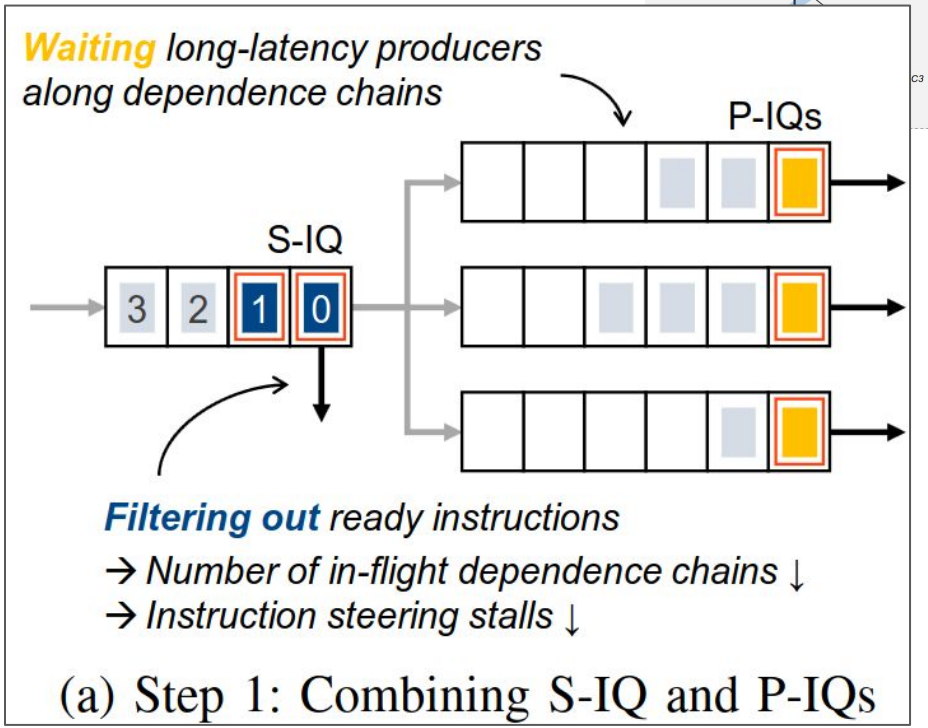


Figure 5: Three-step out-of-order issue queue rebuilding process. Four in-order IQs are displayed for brevity.

Ballerino step-by-step (Step 1)

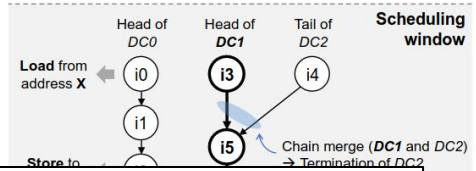


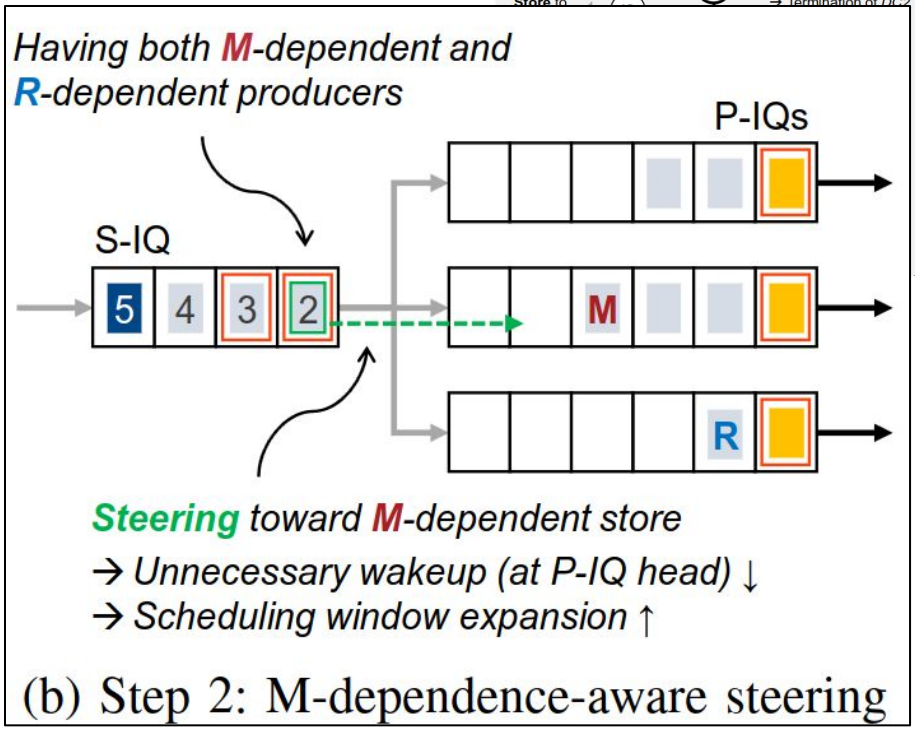
- In front of the clustered P-IQs, a single S-IQ issues ready instructions
- Instructions not captured by S-IQ's scheduling window are not ready; such instructions are passed to the P-IQs and wait for the resolution of their data dependencies
- Since they are steered according to DCs, they can be issued as soon as their producers complete execution
- The steering at the head of the S-IQ stalls if there are no available P-IQs



Ballerino step-by-step (Step 2)

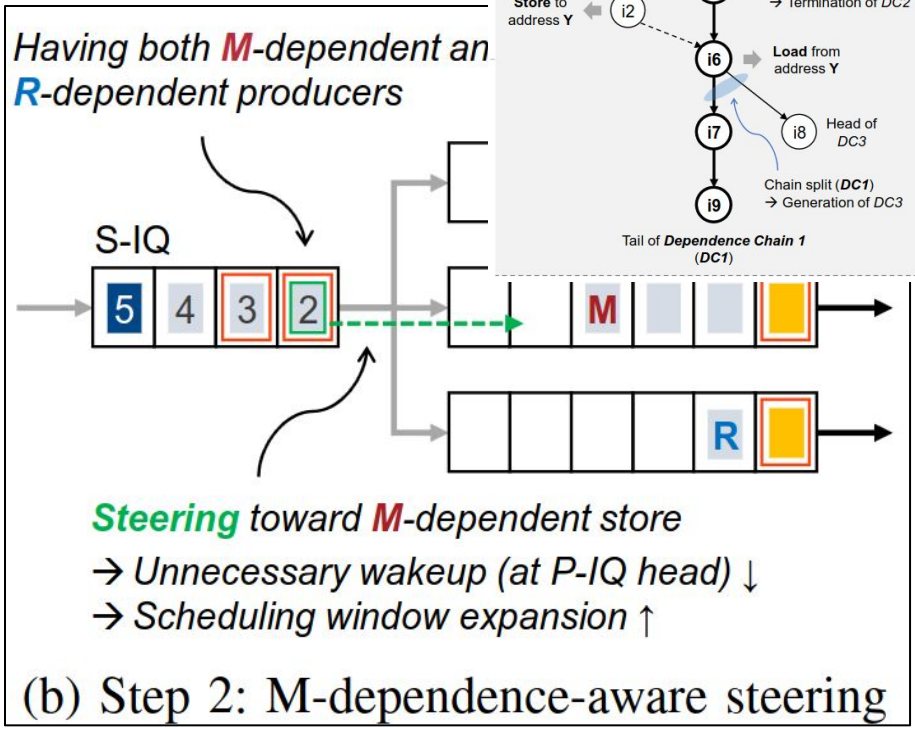
- In CES, instructions are steered based solely on the R-dependence; a producer store and an M-dependent load are always steered to different P-IQ
- The M-dependence between them would block the issue of the P-IQ where the load resides until the producer store is issued (M-related wakeup needed at P-IQ head)
- We can steer this M-dependent load into the P-IQ of its producer store, allowing other P-IQs to be used by other DCs and reducing the occupancy of the P-IQs





Ballerino step-by-step (Step 2)

- In CES, instructions are steered based solely on the R-dependence; a producer store and an M-dependent load are always steered to different P-IQ
- The M-dependence between them would block the issue of the P-IQ where the load resides until the producer store is issued (M-related wakeup needed at P-IQ head)
- We can steer this M-dependent load into the P-IQ of its producer store, allowing other P-IQs to be used by other DCs and reducing the occupancy of the P-IQs



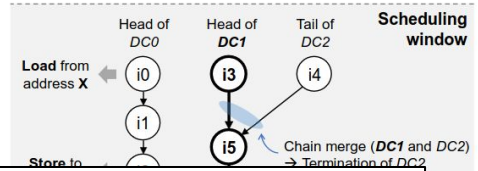
Ballerino step-by-step (Step 2)

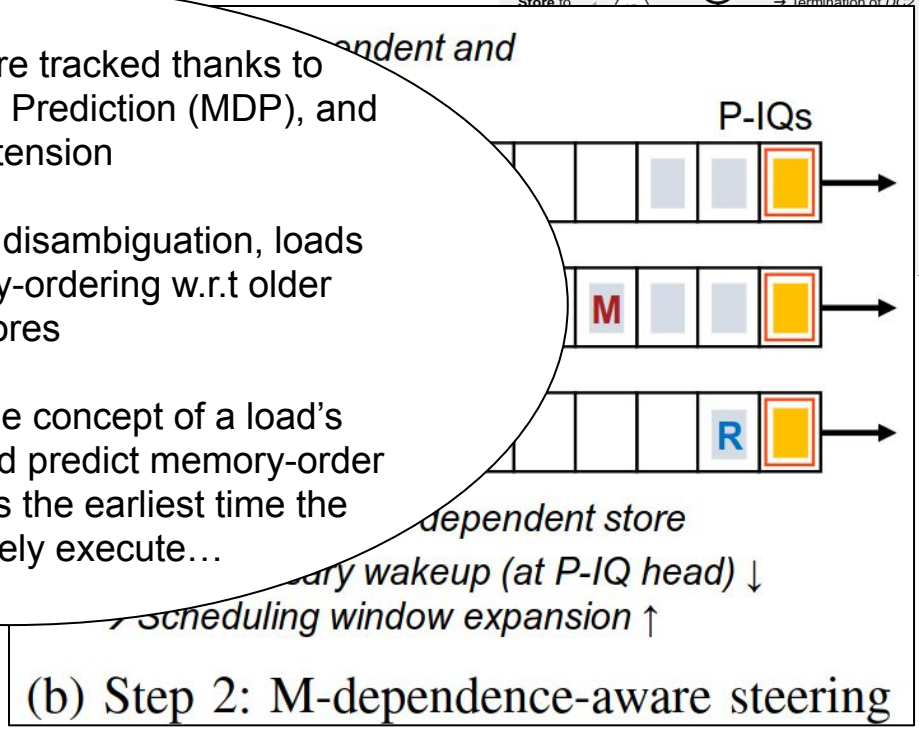
- In CES, instructions are s… solely on the R-dep… store and an M-… always steere…
- The M-depe… block the iss… resides until t… (M-related wak…
- We can steer this M… the P-IQ of its producer st… other P-IQs to be used by other DCs and… reducing the occupancy of the P-IQs

M-dependences are tracked thanks to Memory-Dependence Prediction (MDP), and its extension

Through speculative disambiguation, loads can violate memory-ordering w.r.t older stores

MDP is based on the concept of a load's ***store set*** to detect and predict memory-order violations, as well as the earliest time the load can safely execute…





Ballerino step-by-step (Step 2)

- In CES, instru
  solely on the F
  store and an M
  always steere
- The M-depend
  block the issu
  resides until th
  (M-related wa
- We can steer
  the P-IQ of its
  other P-IQs to
  reducing the c

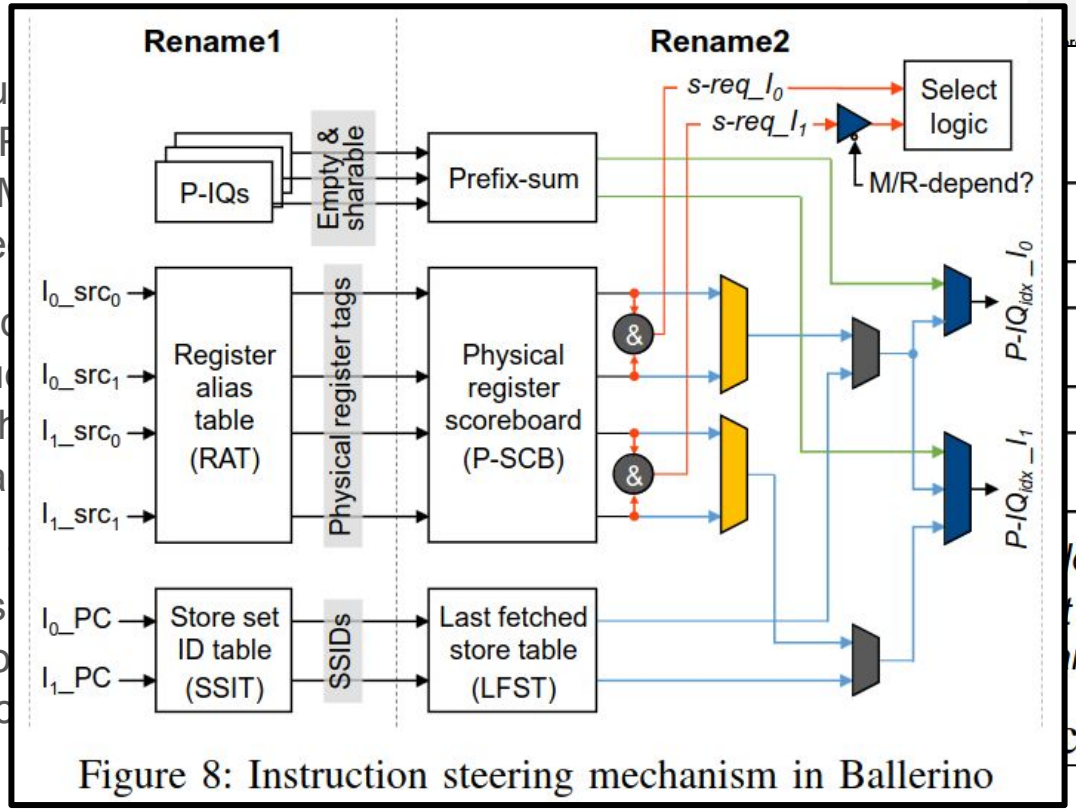
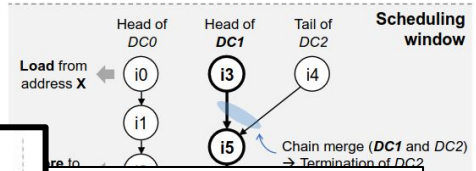
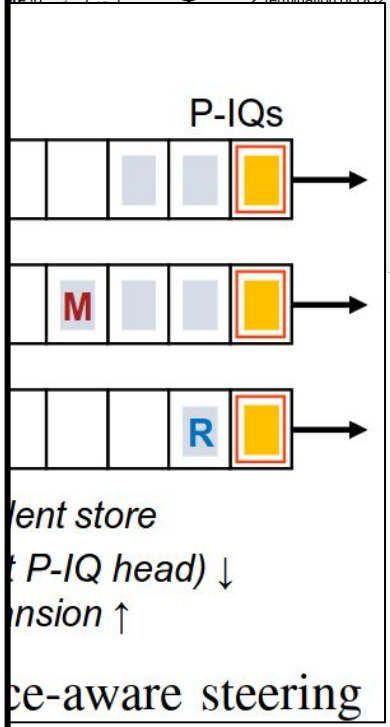


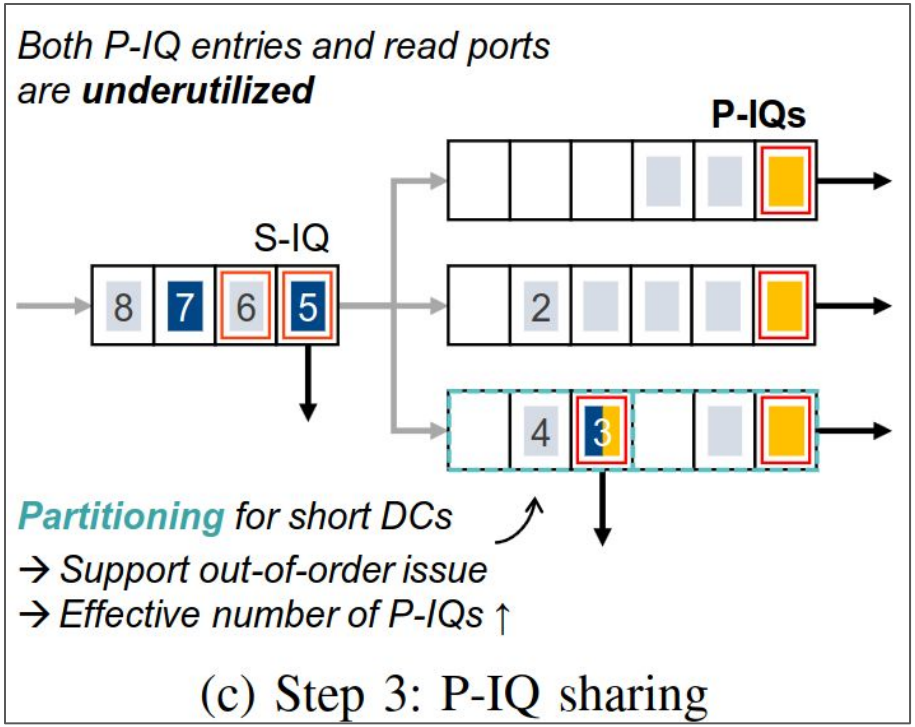
Figure 8: Instruction steering mechanism in Ballerino





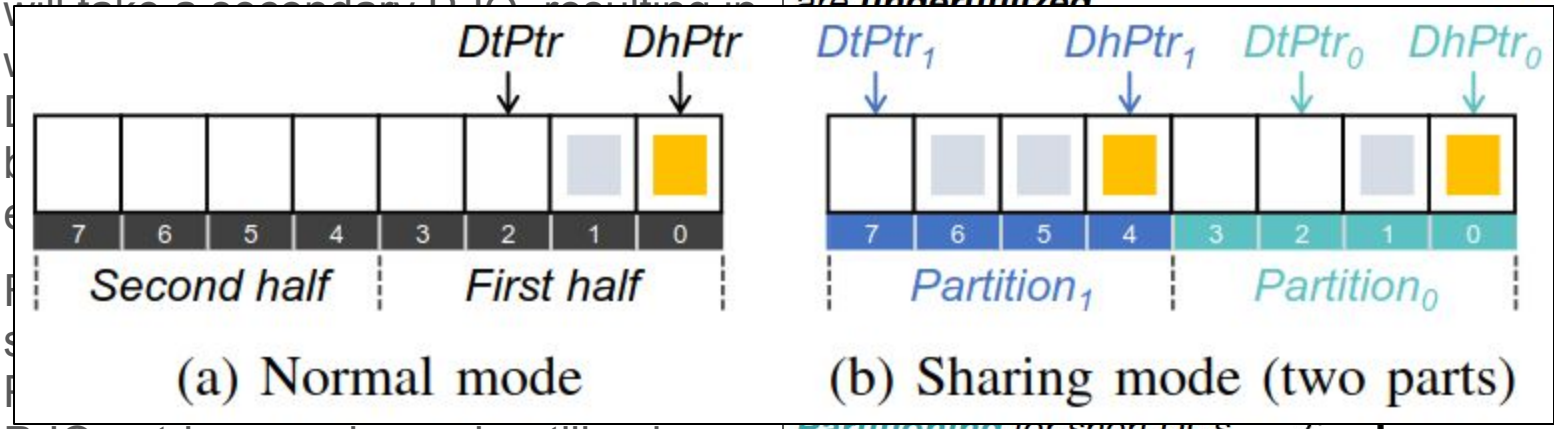
Ballerino step-by-step (Step 3)

- A DC longer than the size of the P-IQ will take a secondary P-IQ, resulting in wakeup operations in the middle of the DC and increasing P-IQ occupancy; but, perhaps, not filling the P-IQ entirely…
- Furthermore, if there are more short-length DCs than the number of P-IQs, stalls take place while some P-IQ entries may be underutilized…
- Solution: multiple short DCs in a single P-IQ (iff…)



Ballerino step-by-step (Step 3)

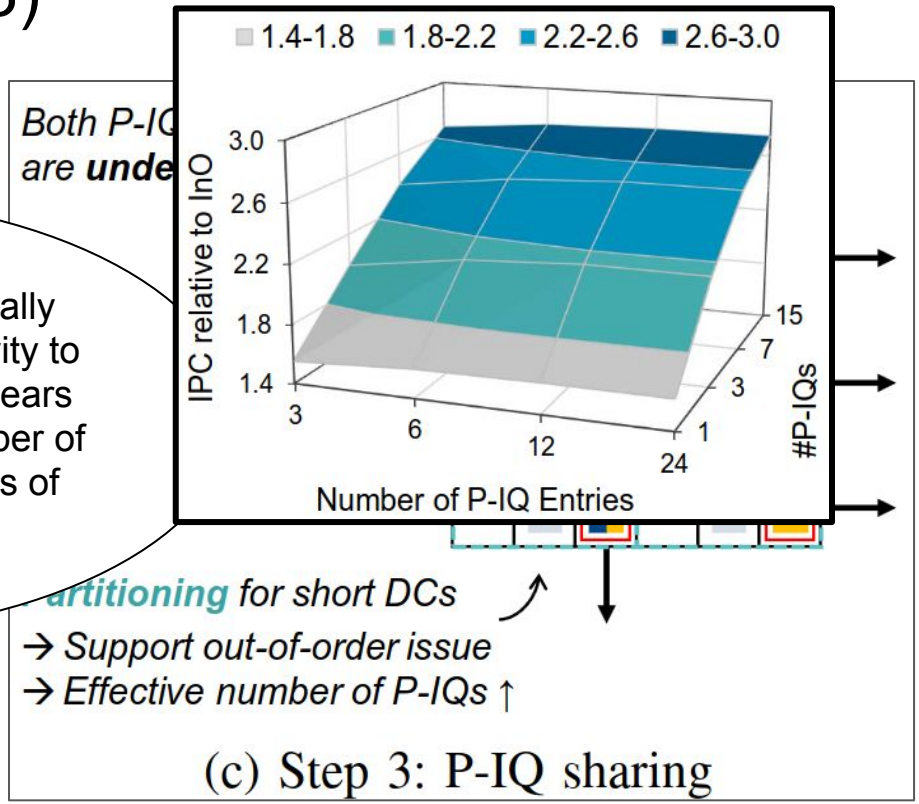
- A DC longer than the size of the P-IQ
- P-IQ entries may be underutilized…
- Solution: multiple short DCs in a single P-IQ (iff…)



Ballerino step-by-step (Step 3)

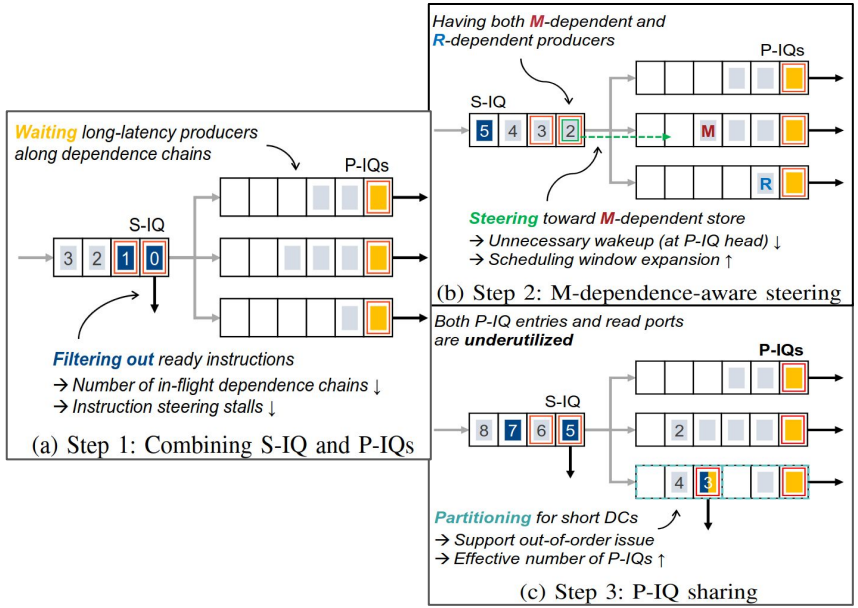
- A DC longer than the size of the P-IQ will take a secondary P-IQ, resulting in wakeup operations i… DC and incr… but, perh… entirely…
- Further… short-leng… P-IQs, stalls … P-IQ entries may be …
- Solution: multiple short DCs in a single P-IQ (iff…)

Ballerino paper experimentally shows relative IPC insensitivity to size of P-IQs, while IPC appears much more sensitive to number of P-IQs, suggesting shortness of DCs…



Ballerino in summary

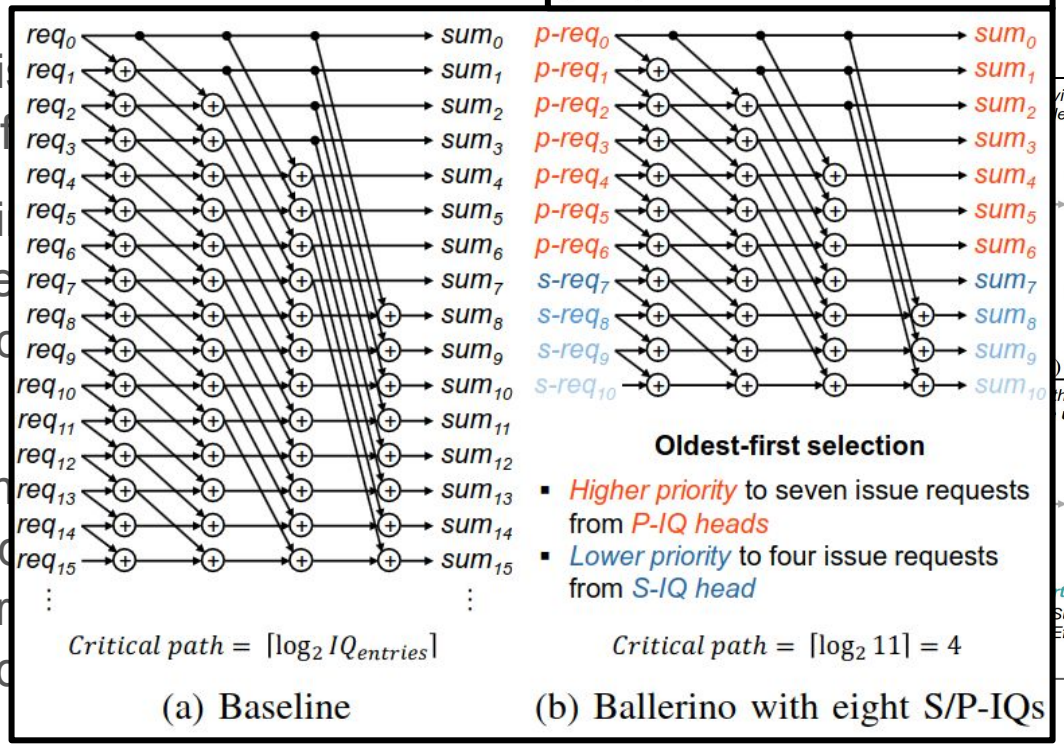
- Ready-at-dispatch instructions are proactively filtered out by the S-IQ
- Non-ready instructions are spread to the clustered P-IQs along their M/R-dependencies
- If execution enters a phase that comprises multiple load-dependent DCs, P-IQs can absorb short DCs by entering sharing mode

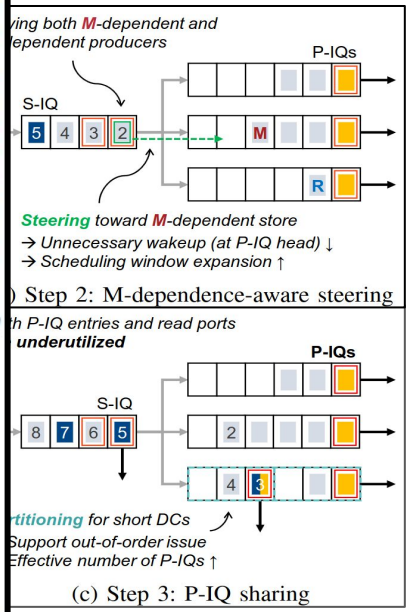


Ballerino in summary

- Ready-at-di
  proactively f
- Non-ready i
  to the cluste
  M/R-depend
- If execution
  comprises n
  load-depend
  absorb shor
  sharing moc

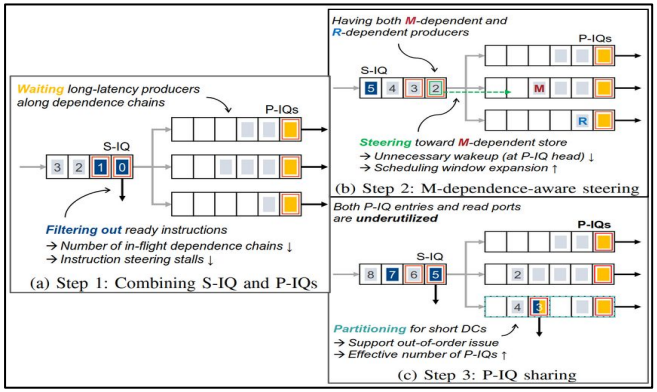
ith req is granted if it is true and *(i − 1)th* cumulative sum is zero

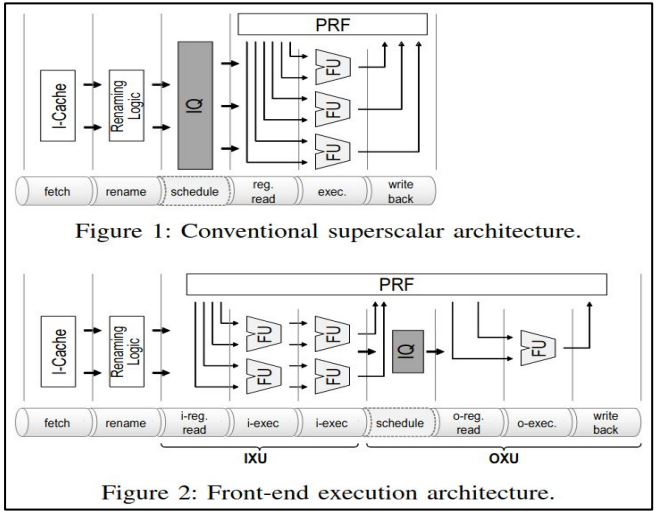




Zoom out summary

- Both techniques attempt to tackle the issue of bad energy eff. of OoO cores
- Ballerino proposes to do away with the OoO IQ altogether in favor of energy-efficient in-order queues
- FXA suggests inserting an in-order phase before the issue-* stages s.t. this previous in-order phase is so effective that the later OoO phase can be reduced





Second paper: A Front-end Execution Architecture (FXA) for High Energy Efficiency

- FXA proposes having two execution units/phases: an in-order execution unit (IXU) and an out-of-order execution unit (OXU)
- The IXU comprises functional units and a bypass network only, while the OXU would consist of the after-dispatch side of a common out-of-order superscalar processor
- Fetched, ready (or bypass-ready) instructions are executed in order in the IXU; fetched, non-ready instructions go through the IXU as NOPs and are dispatched to the OXU for out of order execution

FXA - initial architecture analysis

- IXU's main components are FUs and a bypass network
- It is placed between the rename stage and the IQ
- Registers and the scoreboard are read after the rename stage
- Ready instructions are then executed in-order, such that the IXU functions as a filter for the OXU; anything executed in the IXU is not executed in the OXU

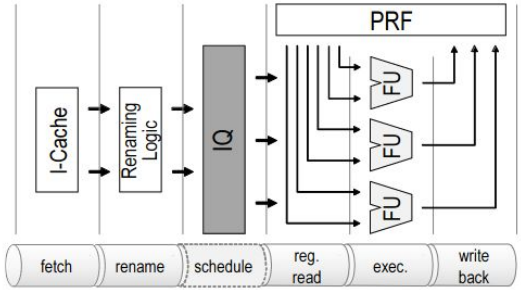


Figure 1: Conventional superscalar architecture.

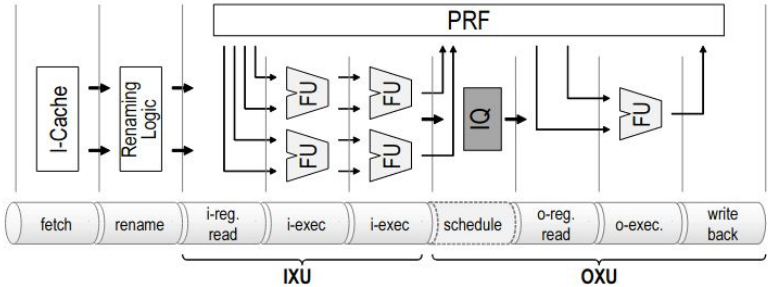


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- IXU's main components are FUs and a bypass network
- It is placed between the rename stage and the IQ
- Registers and the scoreboard are read after the rename stage
- Ready instructions are then executed in-order, such that the IXU functions as a filter for the OXU; anything executed in the IXU is not executed in the OXU

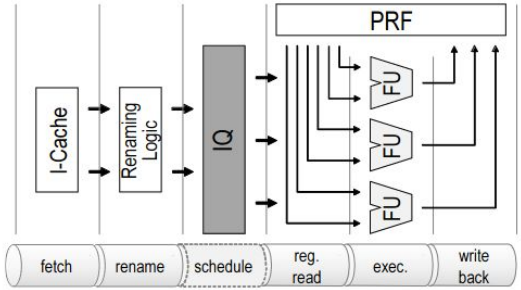


Figure 1: Conventional superscalar architecture.

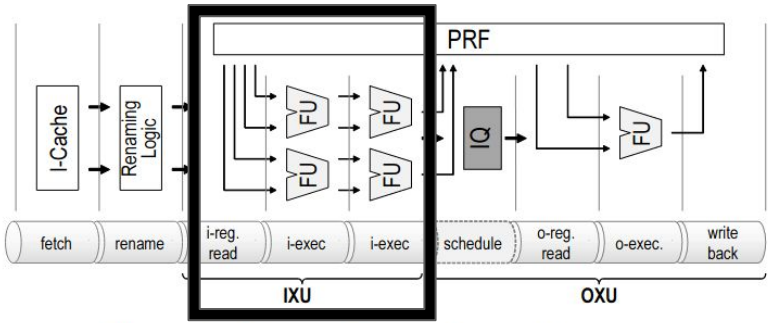


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- An instruction is deemed ready if its source operands are deemed ready in the scoreboard (PRF) or its source operands can be obtained through the bypasses of IXU's FUs
- Non-ready instructions flow through the IXU as NOPs into de IQ, in the interest of simplicity

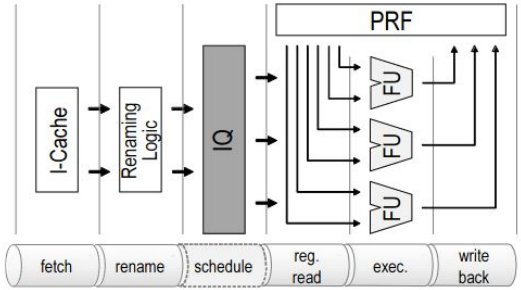


Figure 1: Conventional superscalar architecture.

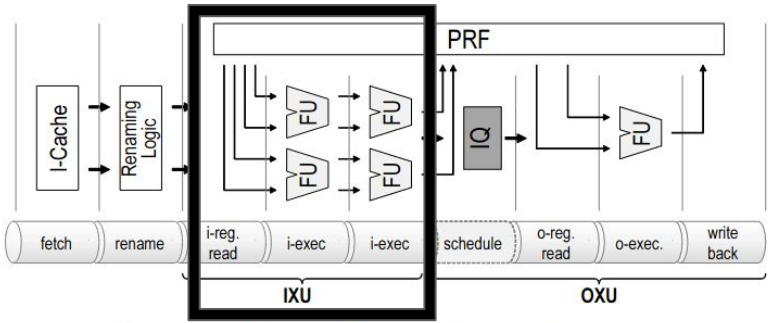
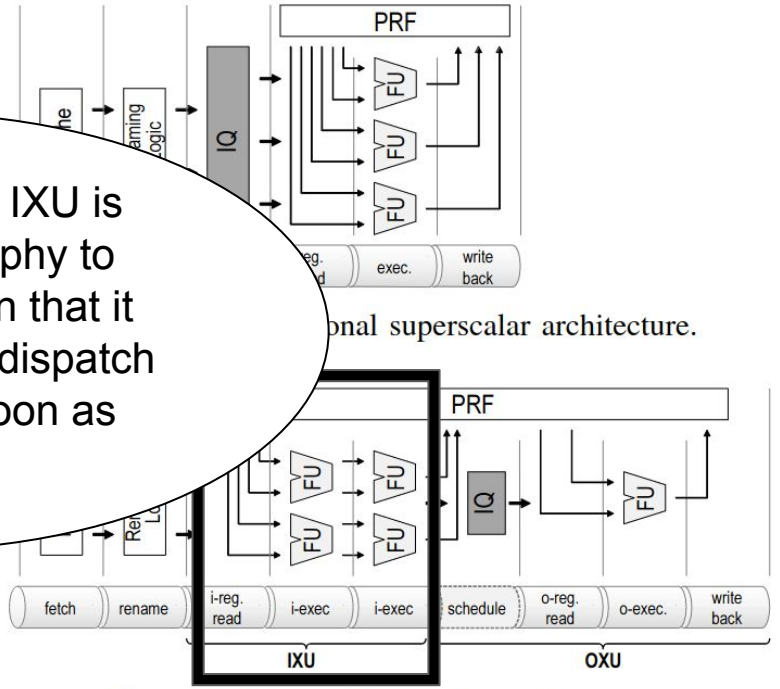


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- An instruction is deemed ready if its source operands are deemed ready in the scoreboa its source operand obtained through IXU's FUs
- Non-ready instructions through the IXU as NOPs into th IQ, in the interest of simplicity

In this sense, the IXU is similar in philosophy to Ballerino's S-IQ, in that it executes ready-at-dispatch instructions as soon as possible…



onal superscalar architecture.

Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- An instruction is deemed ready if its source operands are deemed ready in the scoreboard (PRF) or its source operands can be obtained through the bypasses of IXU's FUs
- Non-ready instructions flow through the IXU as NOPs into de IQ, in the interest of simplicity

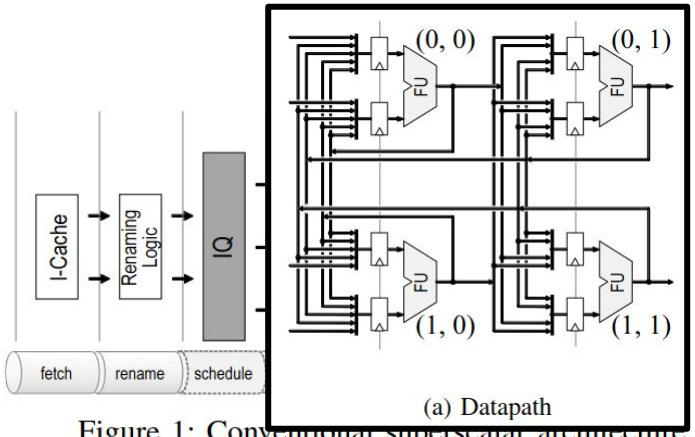


Figure 1: Conventional superscalar architecture.

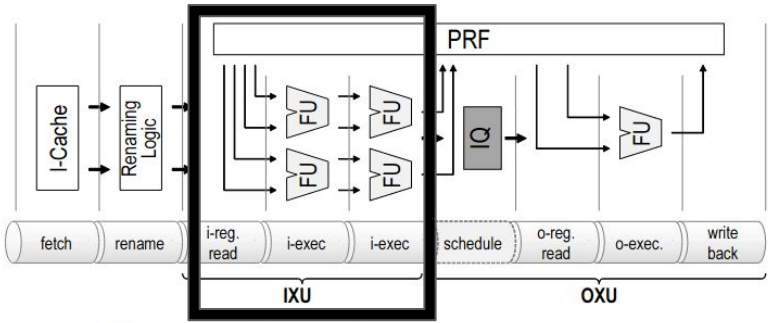


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- IXU's FUs are of the integer type
- Multiple stages of FUs to accommodate for bypasses of dependency chains…

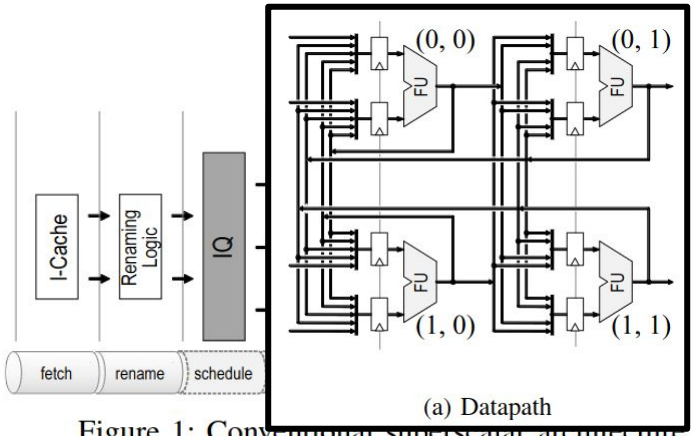


Figure 1: Conventional superscalar architecture.

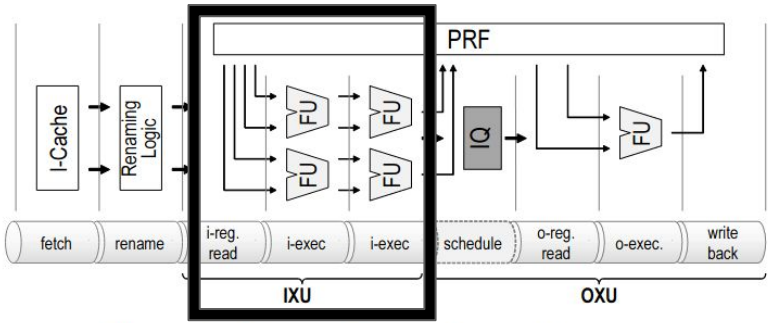
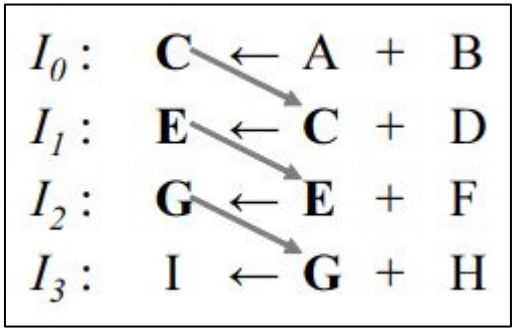


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- IXU's FUs are of the integer type
- Multiple stages of FUs to accommodate for bypasses of dependency chains…



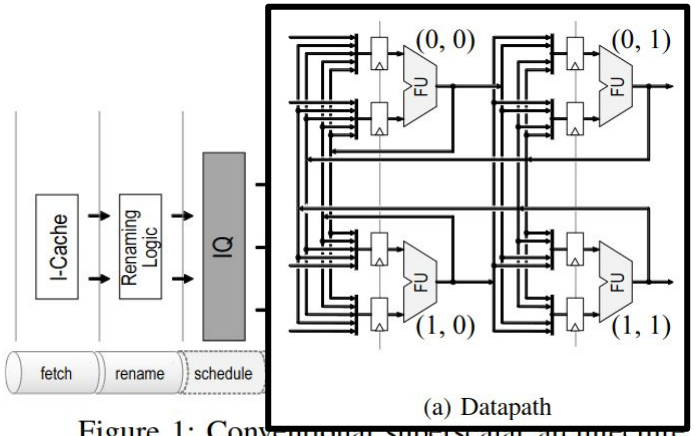


Figure 1: Conventional superscalar architecture.

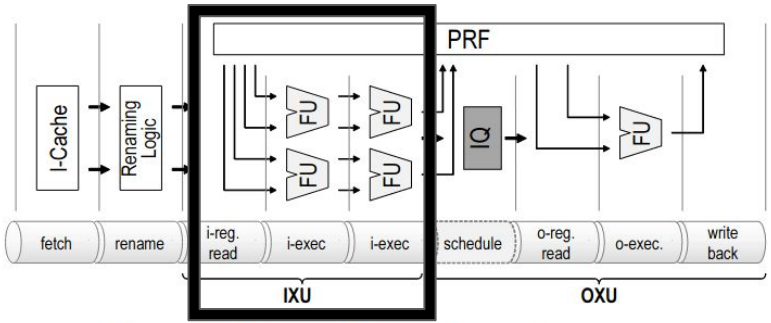
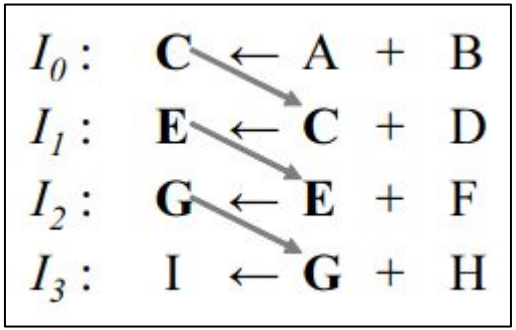


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- IXU's FUs are of the integer type
- Multiple stages of FUs to accommodate for bypasses of dependency chains…



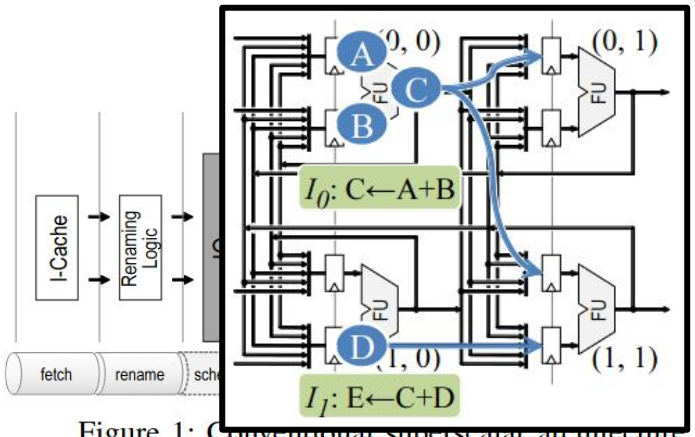


Figure 1: Conventional superscalar architecture.

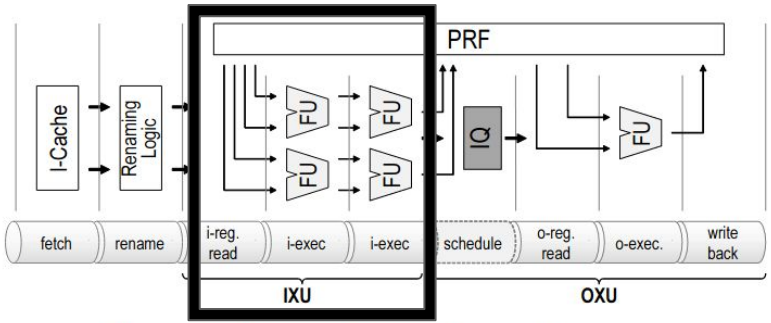
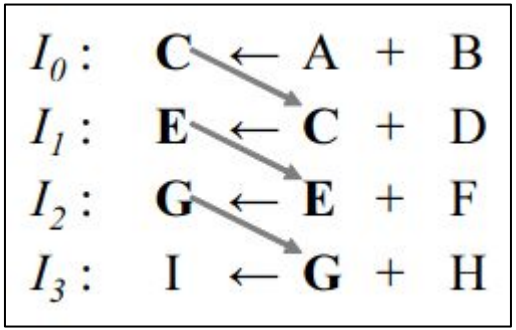


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- IXU's FUs are of the integer type
- Multiple stages of FUs to accommodate for bypasses of dependency chains…



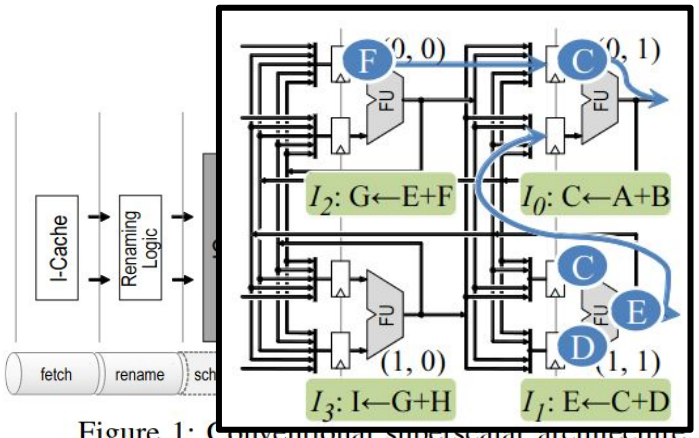


Figure 1: Conventional superscalar architecture.

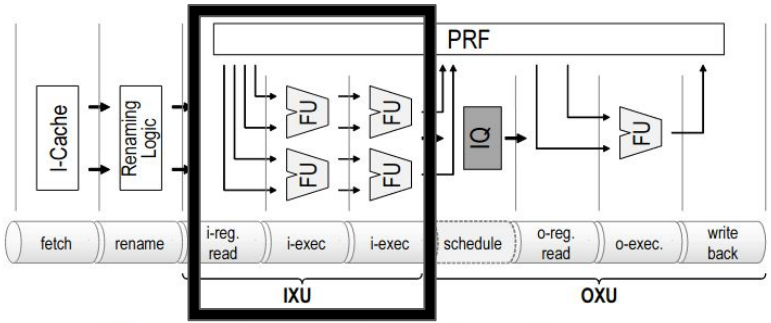
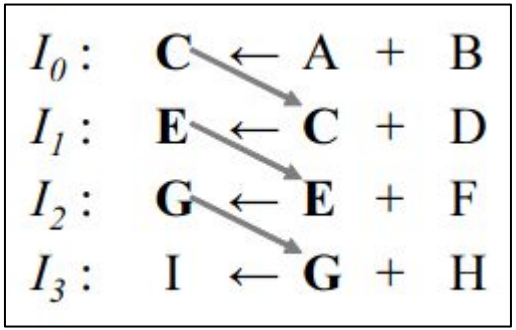


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- IXU's FUs are of the integer type
- Multiple stages of FUs to accommodate for bypasses of dependency chains…



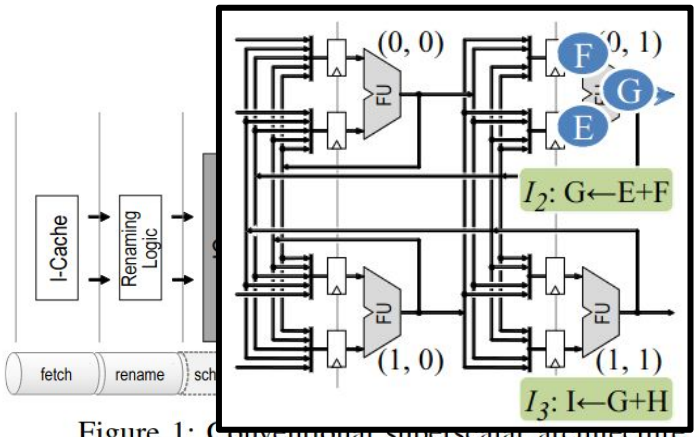


Figure 1: Conventional superscalar architecture.

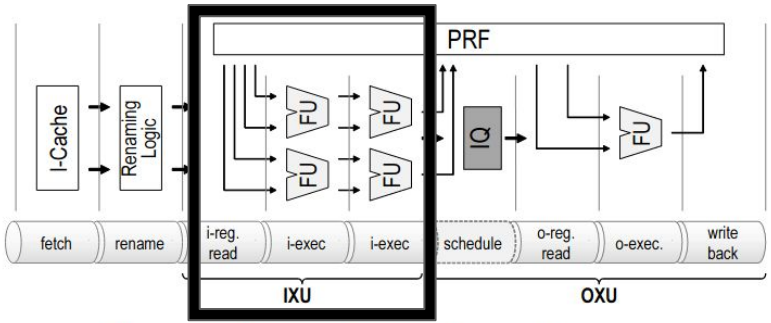
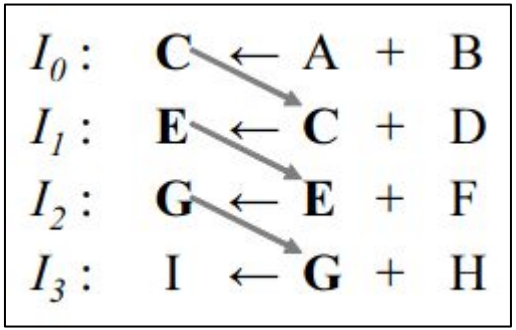


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- I3 cannot be executed with the current configuration of FUs; limit on the length of DC to deem instructions as ready or “bypass-ready”



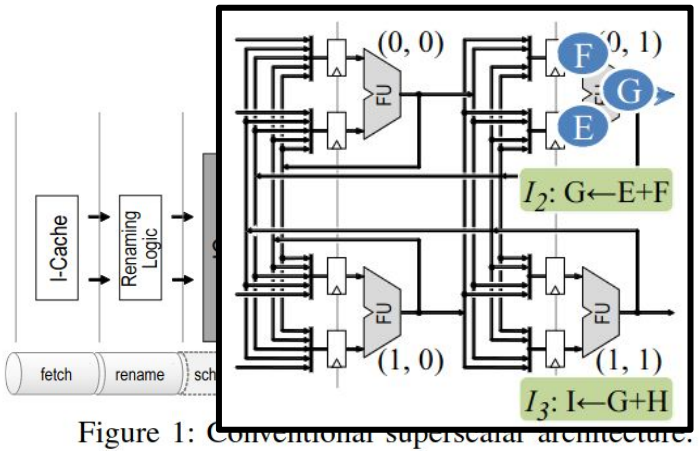


Figure 1: Conventional superscalar architecture.

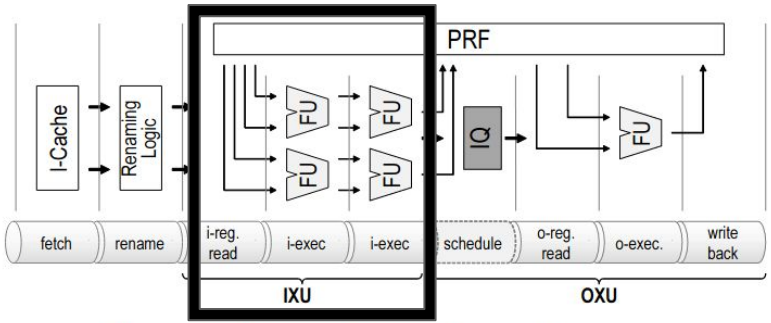


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- Other instructions in the IXU
  - Branches can be executed in the IXU
  - FP operations are executed in the OXU (avoid area overhead and complexity of variable pipeline lengths)
  - Loads/Stores are dispatched to OXU's L/S unit (MDP also for M-dependencies)

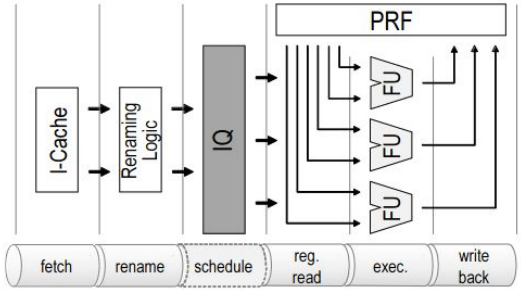


Figure 1: Conventional superscalar architecture.

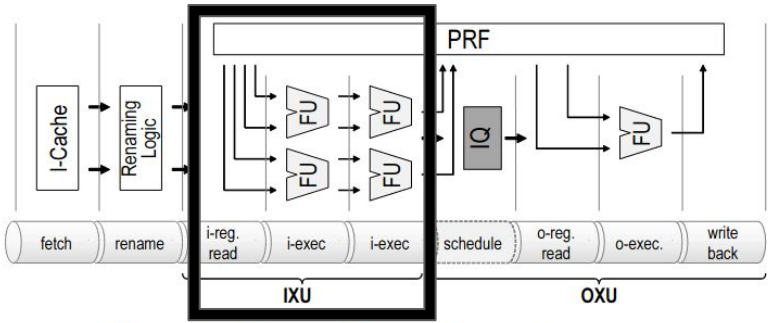


Figure 2: Front-end execution architecture.

FXA - initial architecture analysis

- OXU consists of the after-dispatch side of a common out-of-order superscalar processor
- Non-ready instructions going through the IXU can become ready (wakeup) while in IXU and never receive the necessary wakeup signal, so scoreboard is also read after IXU

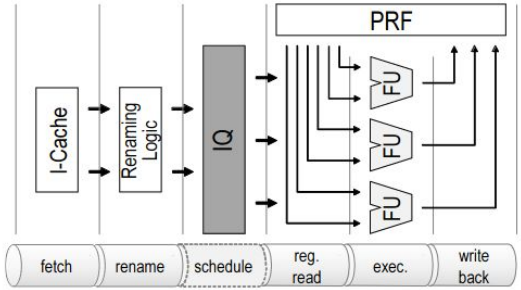


Figure 1: Conventional superscalar architecture.

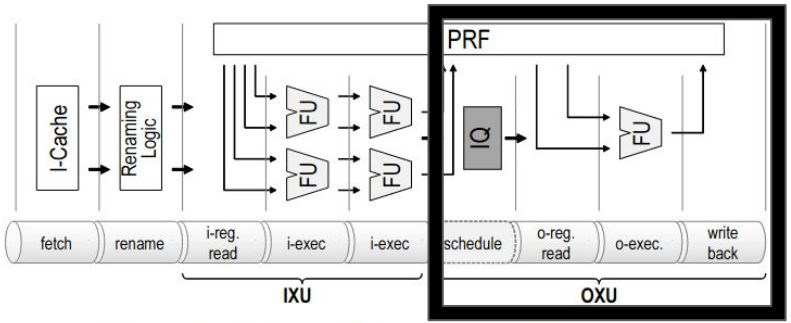


Figure 2: Front-end execution architecture.

FXA - IXU's (downstream) effects

- If the IXU executes many instructions, FXA can improve performance in a manner similar to that used to widening its issue width
  - 4-wide superscalar processor up to four instructions per cycle, FXA up to seven instructions per cycle with 2-wide OXU and an IXU with five FUs 3x1x1
  - IXU is shown to execute many instructions, much in the same way of Ballerino's case with ready-at-dispatch instrs.

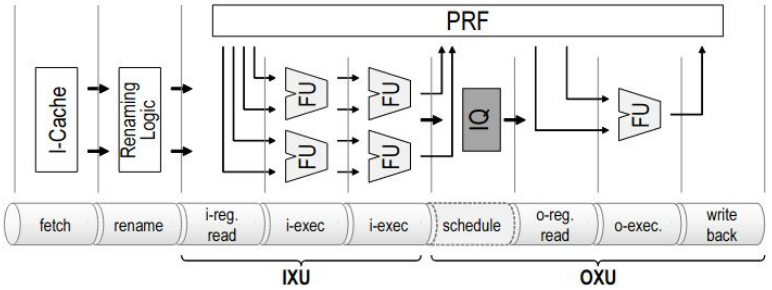


Figure 2: Front-end execution architecture.

FXA - IXU's (downstream) effects

- If the IXU executes many instructions (as it happens to be the case), the OXU can be reduced/shrunken to a point where performance is not lost and more energy efficiency can be gained
  - Reduction in IQ size (FXA's IQ results in 14% energy consumption of original superscalar)
  - Reduction in number of OoO integer FUs (also affecting number of IQ/PRF ports and selection logic reduction)

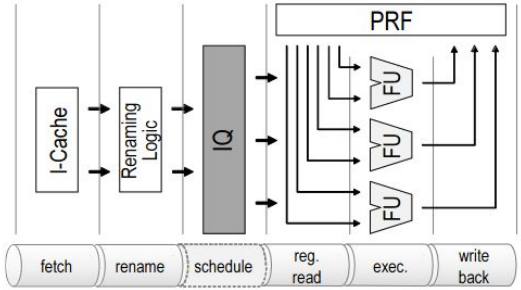


Figure 1: Conventional superscalar architecture.

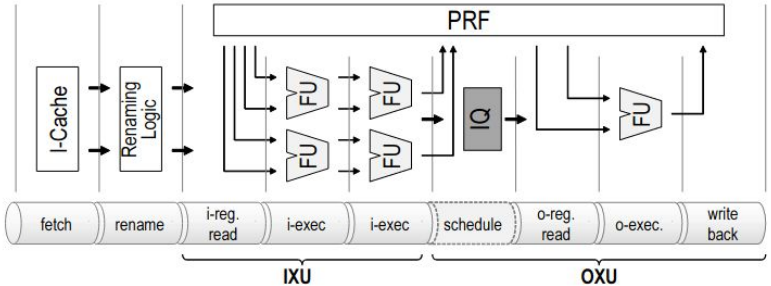


Figure 2: Front-end execution architecture.

FXA - IXU's (downstream) effects

- Branch misprediction penalties can be reduced if detected in the IXU
- If detected in the OXU, though, the misprediction penalty is increased by the number of the stages of the IXU
  - The relatively high number of instructions executed in the IXU (50% on average of SPEC CPU 2006) is claimed to make this unlikely

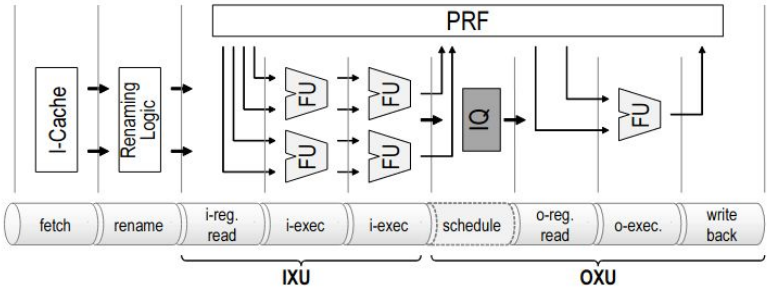


Figure 2: Front-end execution architecture.

FXA - IXU’s (downstream) effects

- Regarding energy efficiency, the dynamic energy consumption of FUs in FXA and conventional superscalar processors is similar
  - The numbers of accesses to the FUs in FXA and conventional superscalar processors are not significantly different, all instructions using FUs are executed once on any FU
  - It should be noted that FUs do not consume much dynamic energy when instructions go through on the FUs in the IXU as NOPs (see pass-through path of FU)

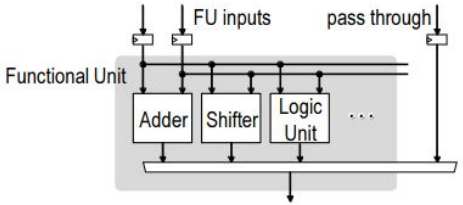


Figure 6: Functional Unit in IXU

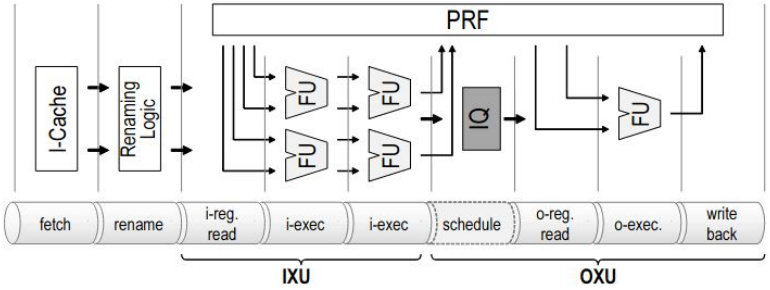


Figure 2: Front-end execution architecture.

FXA - IXU's (downstream) effects

- The static energy consumption is increased by the additional FUs in the IXU, but this increase is relatively small
  - The static energy consumption is proportional to the number of transistors in a circuit
  - However, the low transistors count of integer FUs for the IXU is sufficiently small for this increase in static energy to be relatively small

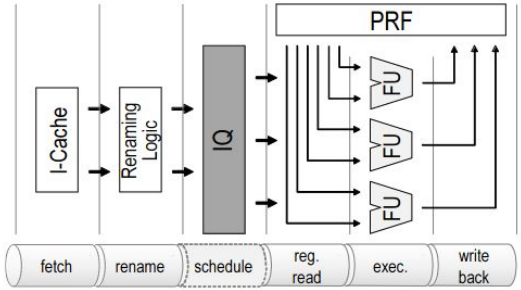


Figure 1: Conventional superscalar architecture.

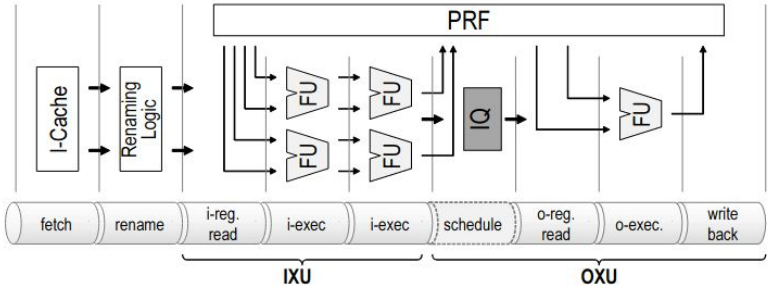


Figure 2: Front-end execution architecture.

FXA - IXU's (downstream) effects

- The energy consumption of each PRF access in FXA and conventional superscalar processors is not significantly different, areas of possible PRFs are almost the same
  - Number of the ports of the PRF required for the IXU is increased, but for the OXU is decreased; number of the ports of the PRF in FXA can be kept from original superscalar processor
  - The ports of the PRF are shared by the IXU and the OXU, and the IXU accesses the shared ports only when the OXU does not access them
  - The number of additional ports as a result of IXU (if not same number of ports) is not so big that this cannot be mitigated through methods of complexity/energy reduction regarding the PRF [5, 19]
  - Number of accesses is also not significantly different

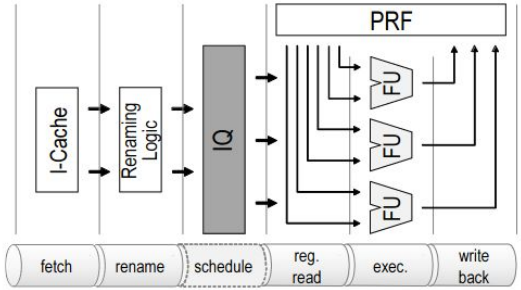


Figure 1: Conventional superscalar architecture.

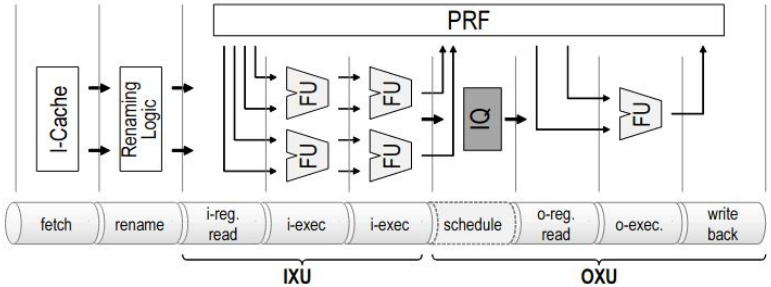
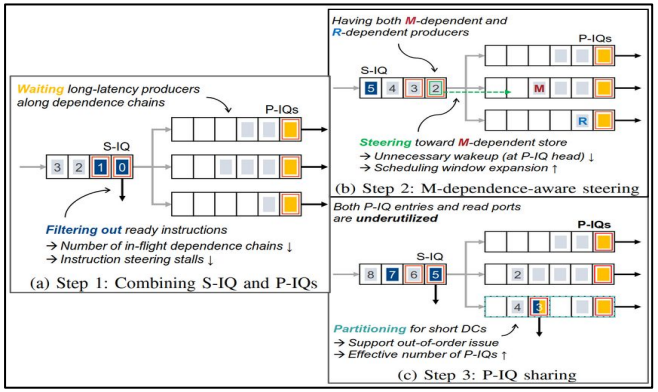


Figure 2: Front-end execution architecture.

Zoom out summary

- Both techniques attempt to tackle the issue of bad energy eff. of OoO cores
- Ballerino proposes to do away with the OoO IQ altogether in favor of energy-efficient in-order queues
- FXA suggests inserting an in-order phase before the issue-* stages s.t. this previous in-order phase is so effective that the later OoO phase can be reduced



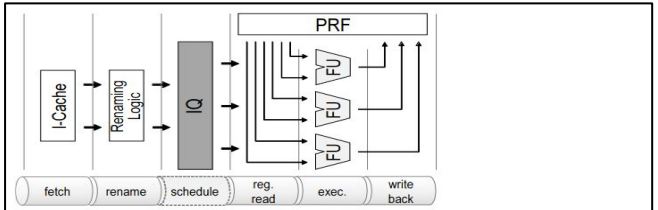


Figure 1: Conventional superscalar architecture.

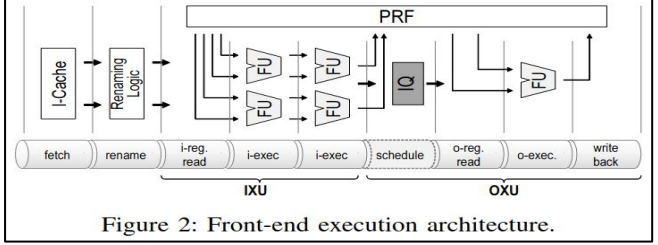
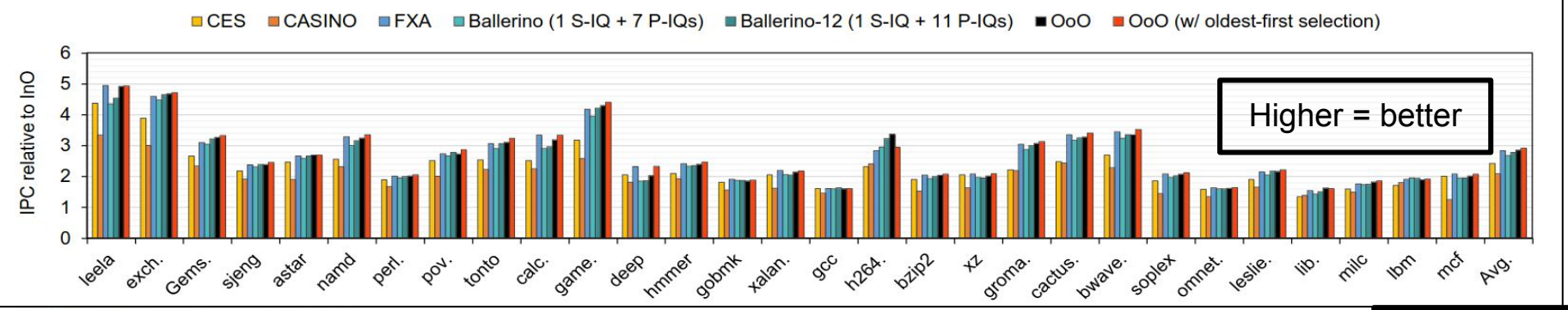
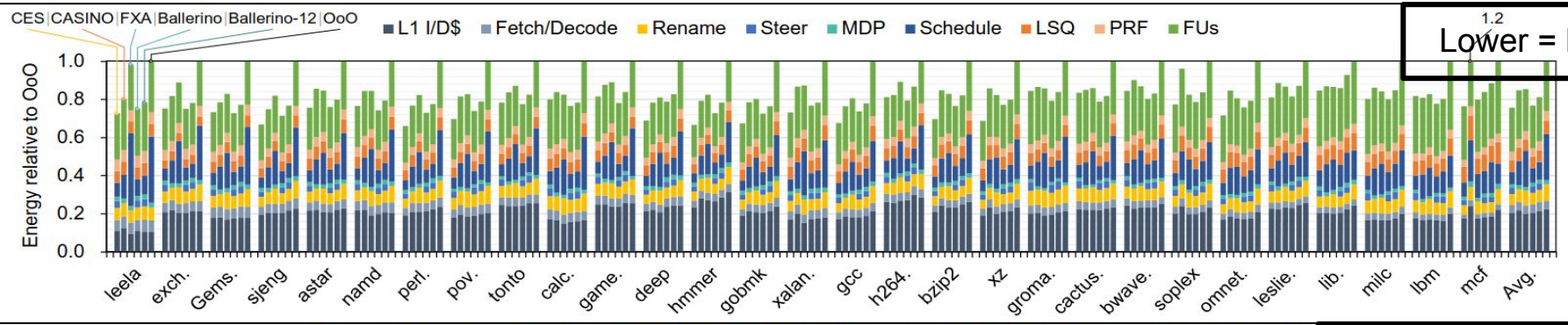
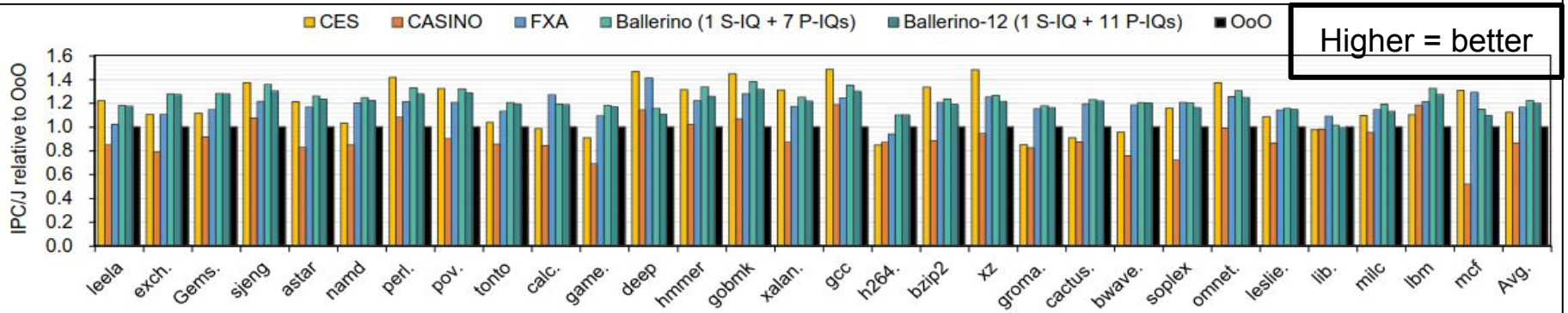


Figure 2: Front-end execution architecture.

Some results from Ballerino’s paper

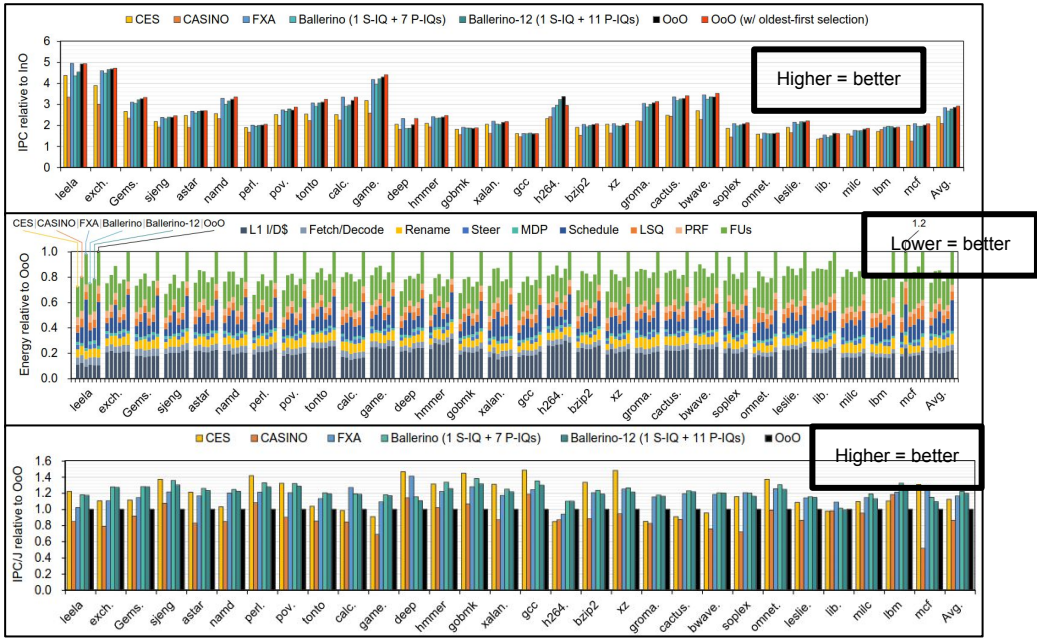

CES
CASINO
FXA
Ballerino (1 S-IQ + 7 P-IQs)
Ballerino-12 (1 S-IQ + 11 P-IQs)
OoO
OoO (w/ oldest-first selection)
IPC relative to InO
Higher = better
leela
exch.
Gems.
sjeng
astar
namd
perl.
pov.
tonto
calc.
game.
deep
hmmer
gobmk
xalan.
gcc
h264.
bzip2
xz
groma.
cactus.
bwave.
soplex
omnet.
leslie.
lib.
milc
lbm
mcf
Avg.


CES
CASINO
FXA
Ballerino
Ballerino-12
OoO
L1 I/D$
Fetch/Decode
Rename
Steer
MDP
Schedule
LSQ
PRF
FUs
Energy relative to OoO
1.2
Lower = better
leela
exch.
Gems.
sjeng
astar
namd
perl.
pov.
tonto
calc.
game.
deep
hmmer
gobmk
xalan.
gcc
h264.
bzip2
xz
groma.
cactus.
bwave.
soplex
omnet.
leslie.
lib.
milc
lbm
mcf
Avg.


CES
CASINO
FXA
Ballerino (1 S-IQ + 7 P-IQs)
Ballerino-12 (1 S-IQ + 11 P-IQs)
OoO
IPC/J relative to OoO
Higher = better
leela
exch.
Gems.
sjeng
astar
namd
perl.
pov.
tonto
calc.
game.
deep
hmmer
gobmk
xalan.
gcc
h264.
bzip2
xz
groma.
cactus.
bwave.
soplex
omnet.
leslie.
lib.
milc
lbm
mcf
Avg.

Some results from Ballerino's paper

- On average, Ballerino and FXA seem to be on par IPC-wise
- Energy-eff-wise, both configurations of Ballerino outperform FXA
- I feel like Ballerino is a bit more complex to implement
- FXA is perhaps too dependent on the presence of ready/bypass-ready integer instructions



Thank you for your attention, questions?

Refresher on dynamic scheduling and out-of-order issuing

- Dynamic scheduling issues instructions “as quickly as possible”, reordering instruction execution to reduce stalls while maintaining data flow dependencies
- It must consider data dependencies (R- and M-type) and availability of ports/FUs for issuing of instructions
  - R(egister)-dependencies correspond to producer-consumer relationship between instructions
  - M(emory)-dependencies correspond to relationship between producer stores and consumer loads
  - Functional units (FUs) are arbitrated through issue ports s.t. an issue port is assigned to an instruction at dispatch according to its opcode

Refresher on dynamic scheduling and out-of-order issuing

1. Port M-1 is assigned to instr. #0 at dispatch according to load balancing and FU needed (opcode)

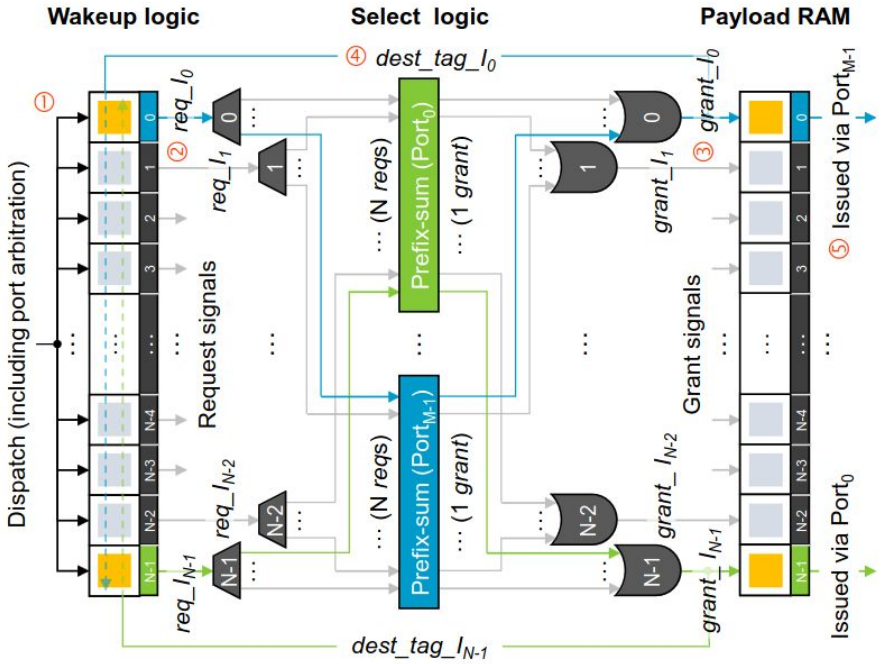


Figure 2: Organization of baseline out-of-order IQ having N entries with M issue ports

Refresher on dynamic scheduling and out-of-order issuing

1. Port M-1 is assigned to instr. #0 at dispatch according to load balancing and FU needed (opcode)

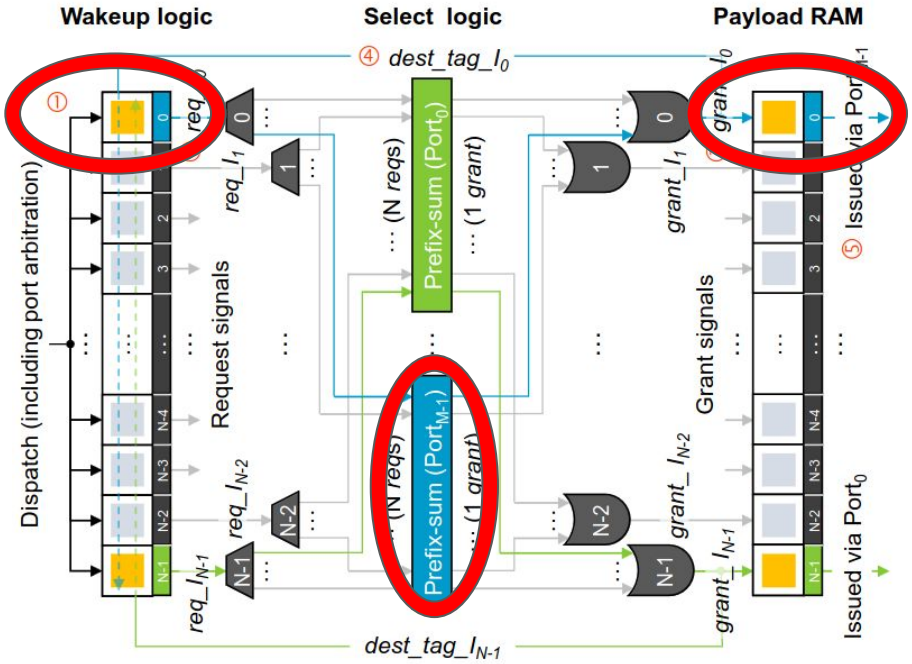


Figure 2: Organization of baseline out-of-order IQ having N entries with M issue ports

Refresher on dynamic scheduling and out-of-order issuing

2. When instr. #0 becomes ready (wakeup), an issue request (*req_I_0*) is sent from the wakeup logic to the select logic (the prefix-sum circuit of Port M-1) that is responsible for selecting an instruction issued in the next cycle

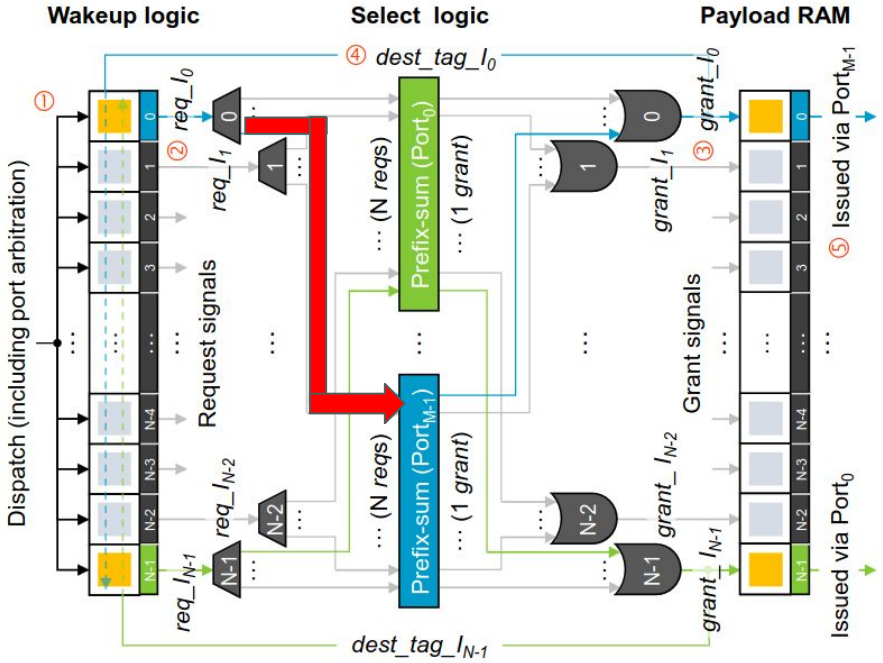


Figure 2: Organization of baseline out-of-order IQ having N entries with M issue ports

Refresher on dynamic scheduling and out-of-order issuing

2. When instr. #0 becomes ready (wakeup), an issue request (*req_I_0*) is sent from the wakeup logic to the select logic (the prefix-sum circuit of Port M-1) that is responsible for selecting an instruction issued in the next cycle

ith req is granted if it is true and *(i − 1)th* cumulative sum is zero

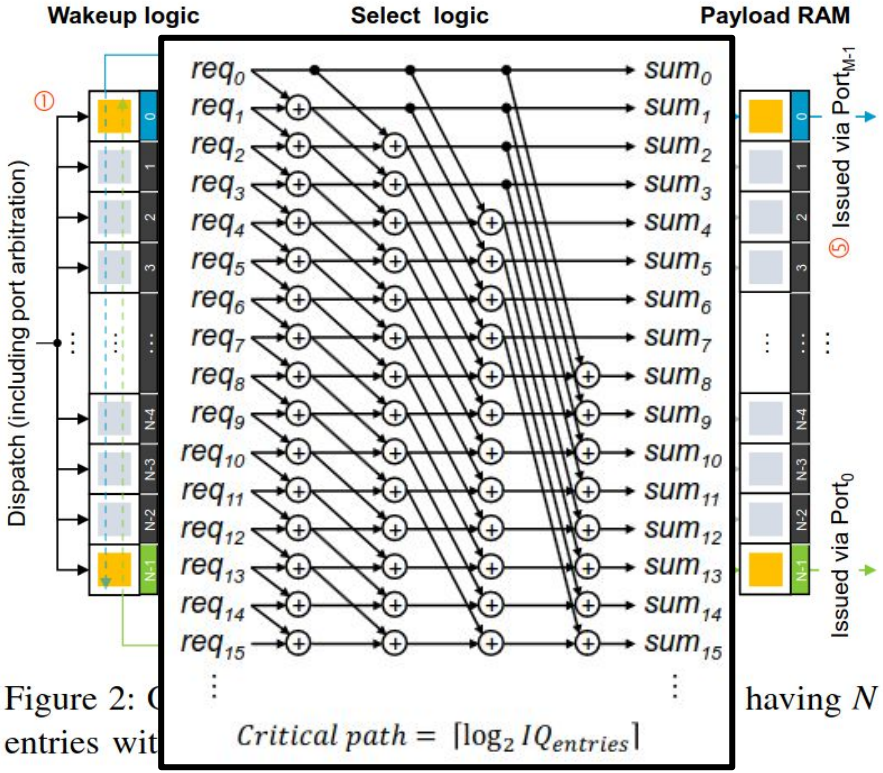


Figure 2: C having N entries wit

Refresher on dynamic scheduling and out-of-order issuing

3. A prefix-sum circuit “grants” one of the input requests, and then sends a grant signal (*grant_I_0*) to the payload RAM

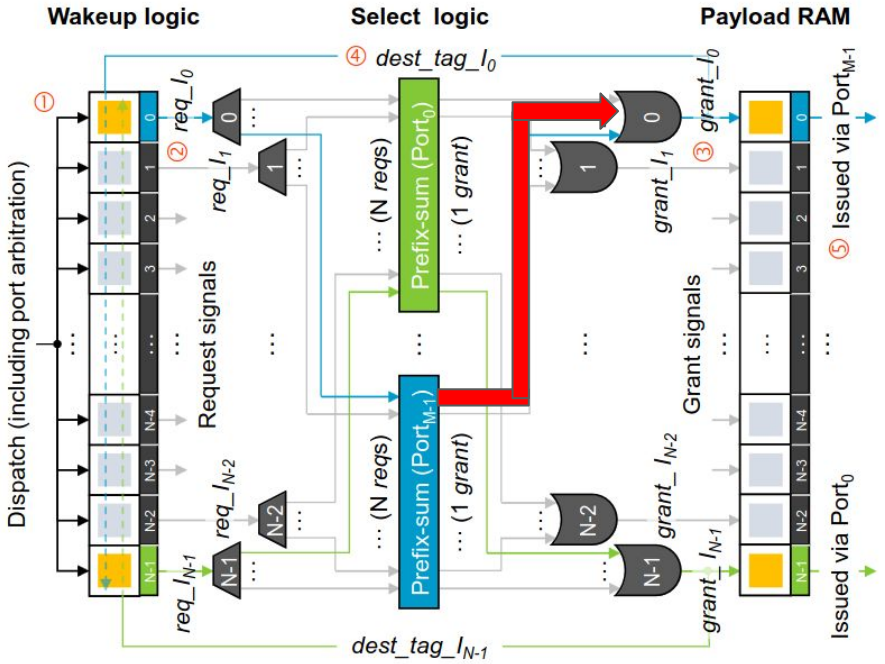


Figure 2: Organization of baseline out-of-order IQ having N entries with M issue ports

Refresher on dynamic scheduling and out-of-order issuing

4. The destination register tag of instr. #0 (*dest_tag_I_0*) is ***broadcast*** to the wakeup logic to update the ready flags of the corresponding source registers

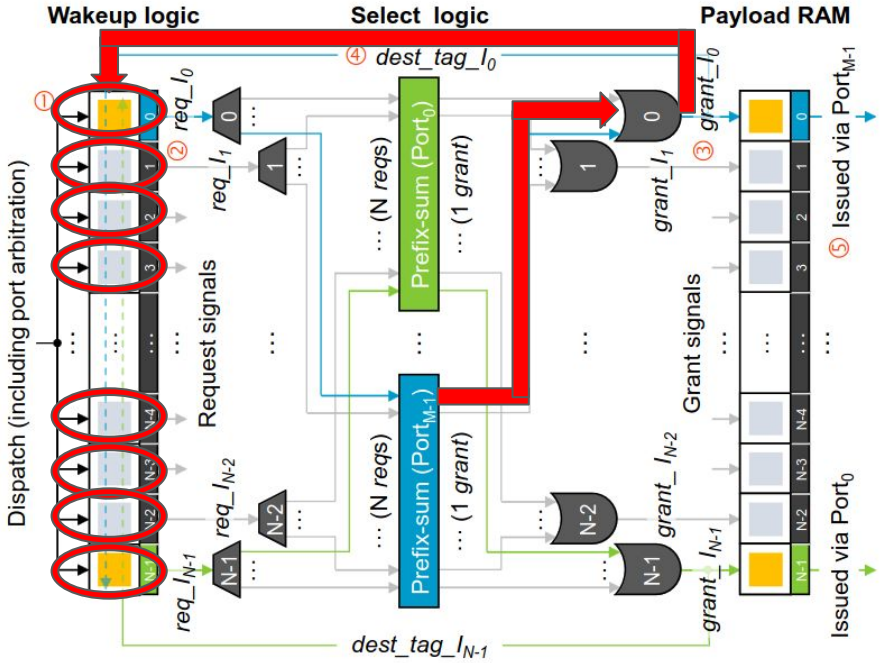


Figure 2: Organization of baseline out-of-order IQ having N entries with M issue ports

Refresher on dynamic scheduling and out-of-order issuing

5. Finally, instr. #0 is issued from the payload RAM to the FU via Port M-1 in the next cycle

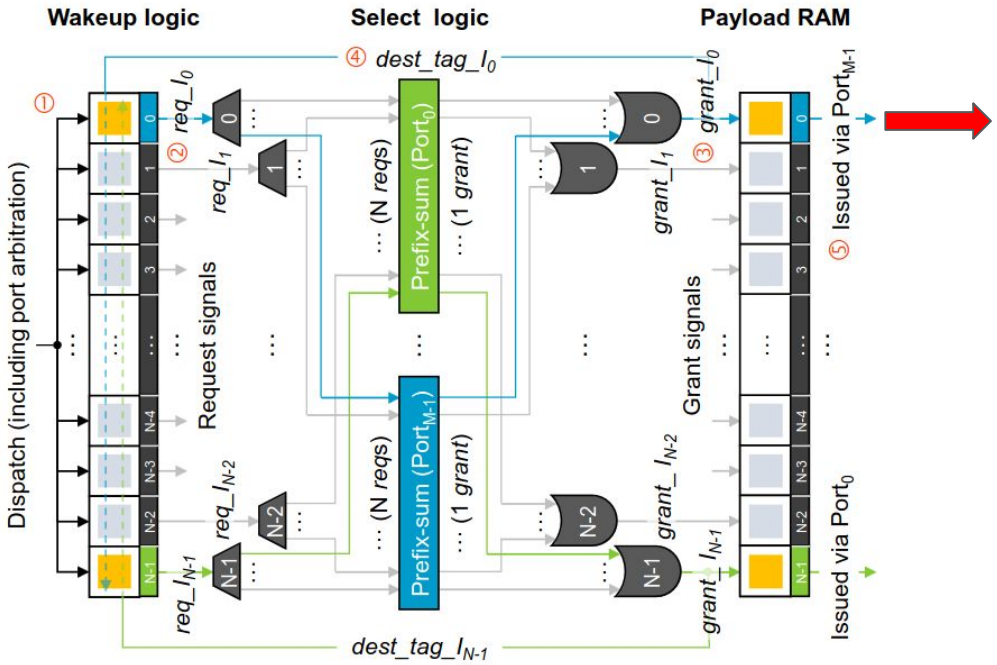


Figure 2: Organization of baseline out-of-order IQ having N entries with M issue ports

Refresher on dynamic scheduling and out-of-order issuing

** M-dependencies (loads depending on older stores) are managed through MDP (Memory-Dependence Prediction): speculative disambiguation and, if load access verification is erroneous, make the load explicitly dependent on the detected store after the fact + prediction…

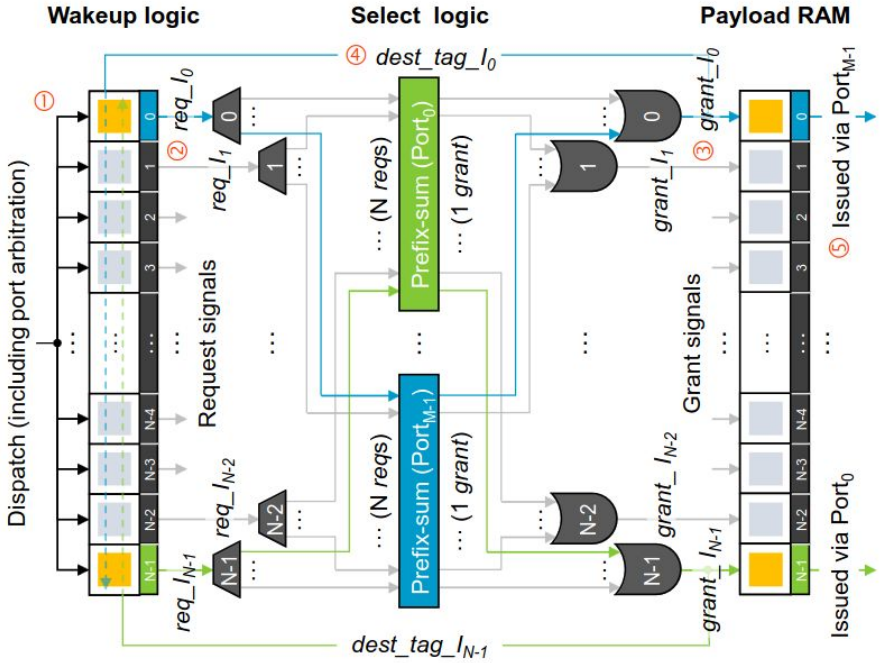
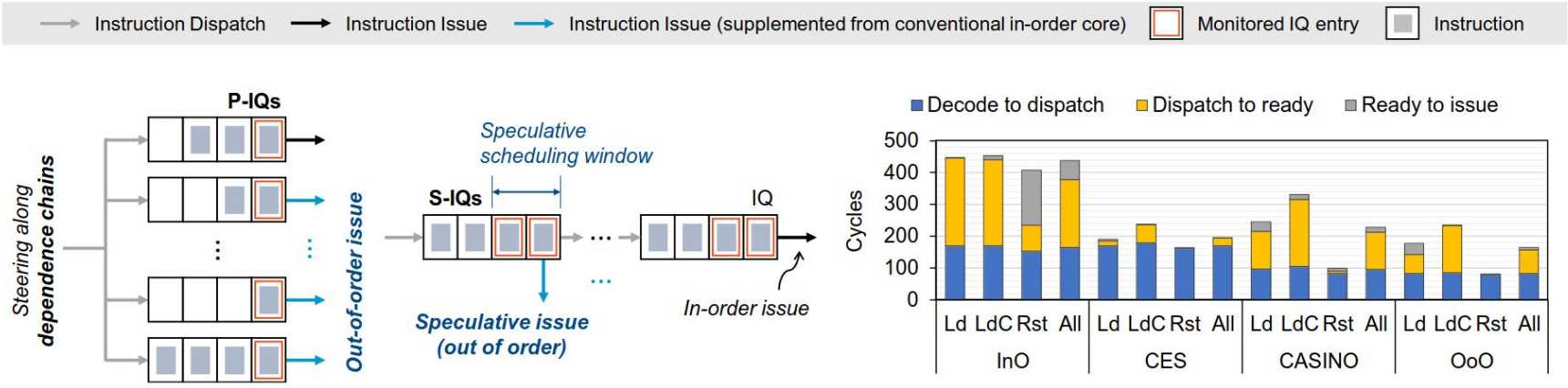


Figure 2: Organization of baseline out-of-order IQ having N entries with M issue ports

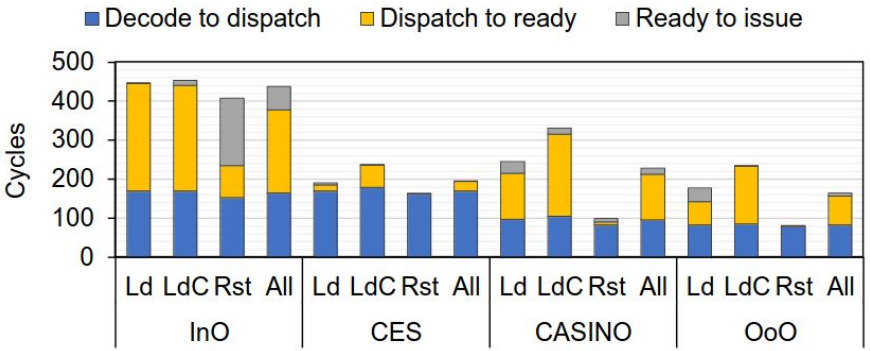
Backup slides



(a) Clustered P-IQs in CES (b) Cascaded S-IQs in CASINO (c) Scheduling performance (avg. decode-to-issue cycles)

Figure 3: Design concepts and scheduling performance of CES [3] and CASINO [2], compared to baseline designs

Backup slides



(c) Scheduling performance (avg. decode-to-issue cycles)

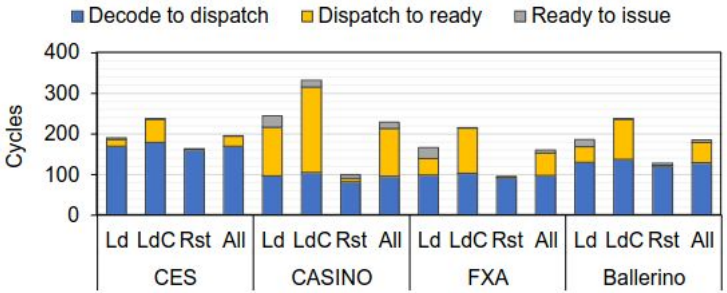


Figure 12: Scheduling performance

Backup slides

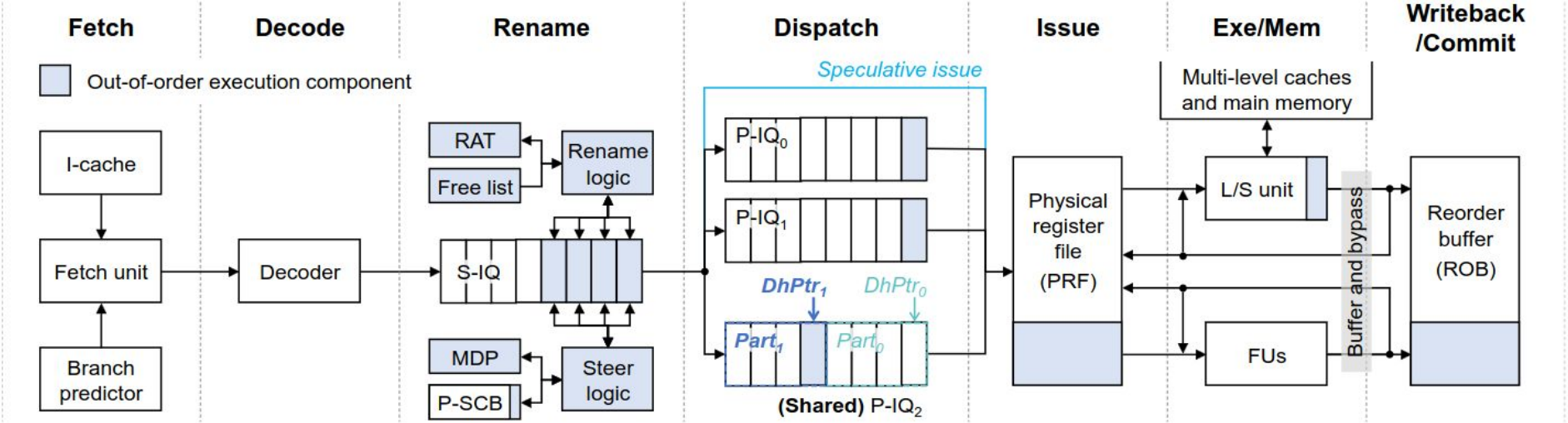


Figure 7: Ballerino core microarchitecture. Components for out-of-order execution are colored in blue. $P\text{-}IQ_2$ is shared by two DCs and second partition ($Part_1$) is activated for issue.

Backup slides

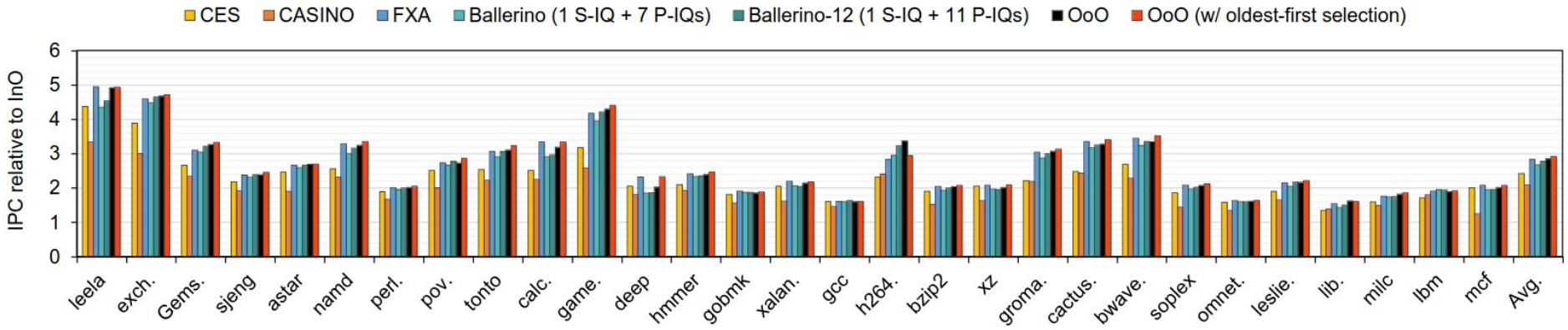


Figure 11: Performance gains of different cores with 8-wide issue capability over in-order core. Ballerino with eight and twelve S/P-IQs respectively achieve 2.7× and 2.8× speedups, which are within 7% and 2% of that of OoO.

Backup slides

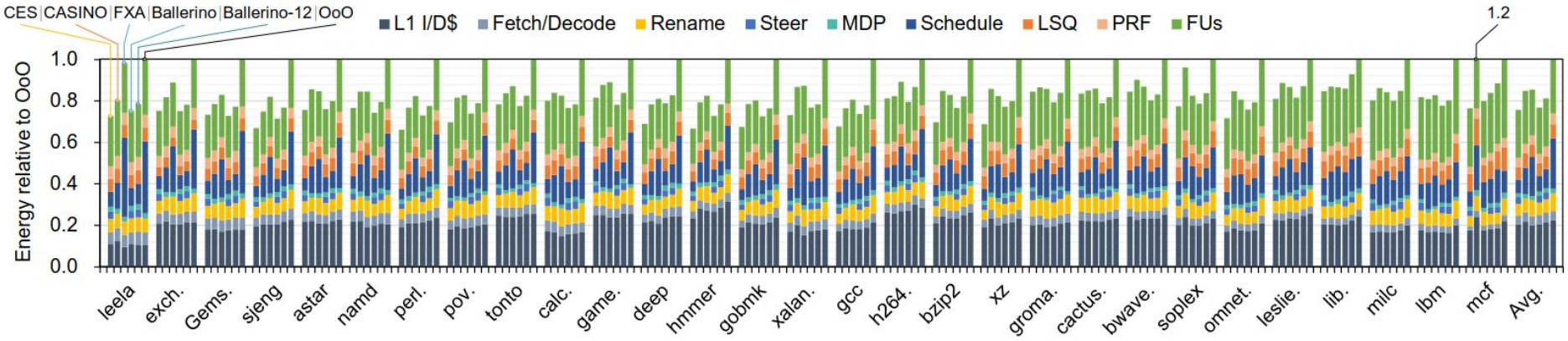


Figure 15: Energy consumption normalized to OoO. From left to right, each stacked bar indicates the core-wide energy consumption of CES, CASINO, FXA, Ballerino, Ballerino-12, and OoO.

Backup slides

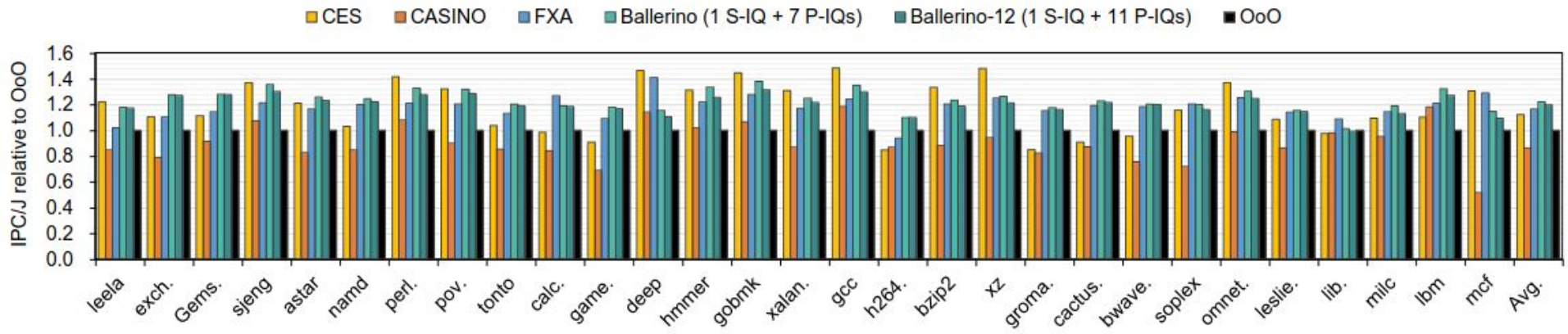


Figure 16: Energy efficiency (performance per energy) normalized to 8-wide out-of-order core