FIFOrder MicroArchitecture: Ready-Aware Instruction Scheduling for OoO Processors

Mehdi Alipour[1], Rakesh Kumar[2], Stefanos Kaxiras[1], and David Black-Schaffer[1]

[1]Department of Information Technology, Uppsala University, Sweden
[1]*first.last@it.uu.se*
[2]Department of Computer Science , Norwegian University of Science and Technology, Norway
[2]*first.last@ntnu.no*

***Abstract*—The number of instructions a processor's instruction queue can examine (depth) and the number it can issue together (width) determine its ability to take advantage of the ILP in an application. Unfortunately, increasing either the width or depth of the instruction queue is very costly due to the content-addressable logic needed to wakeup and select instructions out-of-order.**

This work makes the observation that a large number of instructions have both operands ready at dispatch, and therefore do not benefit from out-of-order scheduling. We leverage this to place such ready-at-dispatch instructions in separate, simpler, in-order FIFO queues for scheduling. With such additional queues, we can reduce the size and width of the expensive out-of-order instruction queue, without reducing the processor's overall issue width and depth.

Our design, FIFOrder, is able to steer more than 60% of instructions to the cheaper FIFO queues, providing a 50% energy savings over a traditional out-of-order instruction queue design, while delivering 8% higher performance.

I. Introduction

Out-of-order (OoO) processors identify, select, and execute ready instructions out of program order to exploit instruction level parallelism (ILP). To do so, they employ a number of large, complex, and power-hungry hardware structures such as the instruction queue (IQ), physical register files (PRF), and load/store queues (LSQ). The IQ is responsible for identifying and selecting ready instructions for execution, and is arguably the most complex and power-intensive [1], [2] structure. Its complexity stems mainly from two sources: First, as instructions complete their execution, the IQ needs to *broadcast* their results (destination register or instruction id) to all other waiting instructions. When all the operands for such a waiting instruction become available, it is marked as ready for execution. Second, the IQ needs to *select* instructions for execution from the pool of ready instructions, based on a set of priorities and available functional units.

To support these functionalities, the IQ is implemented as a content addressable memory (CAM). CAMs are particularly expensive to scale up because they require logic for each entry and each port port that detects if the port's data matches that particular entry. For broadcast, the IQ must have a broadcast port for each instruction that can finish in any cycle to mark all relevant instructions as ready. For instruction selection, the CAM must include logic that examines all ready instructions and chooses them by priority, as well as multiple output ports for reading out the selected instructions. The combination of the per-entry and per-port logic required for broadcase and selection, along with the selects and wires needed to output the multiple instructions selected for issue, make the IQ an incredibly complex and energy-hungry circuit that based on figure 2, in average, contributes into 45% of the run-time dynamic energy of the core .

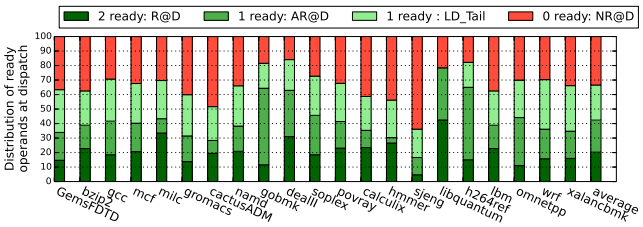

Fig. 1. Classification of instructions at dispatch: both operands ready (ready-at-dispatch, R@D), one operand ready (almost-ready-at-dispatch, AR@D), one operand ready and one from a load (load-tail, LDTail), no operands ready (not-ready-at-dispatch, NR@D). On average 20% of instructions are R@D, and therefore do not benefit from out-of-order scheduling.

Though OoO scheduling enables early execution of ready instructions by bypassing the stalled ones, its complexity leads to significant energy costs. Moreover, the complexity increases superlinearly with the IQ size and issue width, making it challenging to scale up the IQ. Indeed, the delays associated with such complexity force instruction scheduling to be implemented over multiple stages in modern processors.

In this work, we make the critical observation that *not all instructions benefit from the OoO instruction scheduling*. In particular, instructions whose operands are both ready-at-dispatch (*R@D*) can be issued at any time an appropriate functional unit is available. If such instructions can be moved out of the IQ and scheduled using simpler hardware, such as first-in first-out (FIFO) queues, the freed IQ space can be utilized by other instructions that may require OoO scheduling. As instructions can be issued for execution from these simpler FIFO queues in addition to the IQ, the effective issue window size increases, leading to better performance. As the FIFO queues are much simpler and more energy-efficient than the out-of-order IQ, they can deliver additional performance with a very small additional energy cost compared to scaling the IQ [1]. Conversely, for the same performance, the size of the IQ can be reduced due to the instructions placed in the FIFO queue, thereby saving energy.

978-3-9819263-2-3/DATE19/©2019 EDAA

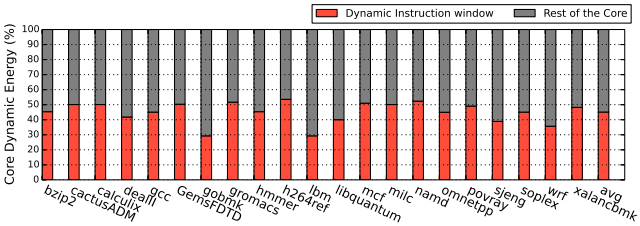

Fig. 2. Distribution of the run-time dynamic energy (both dynamic and static) of a core based on detail parameters of table I and simulated in [3]. Each bar has divided into the run-time energy of dynamic instruction window and the rest of the core energy. Dynamic instruction window includes IQ and its functionality between the dispatch and issue stage such instructions scheduling, wake-up, select, etc. Based on this figure, in a single core processor, more than 40% of the energy is consumed by the dynamic instruction window in average across SPEC2006 benchmarks.

To understand which instructions do not benefit (or only benefit minimally) from out-of-order scheduling, we classify instructions into four categories based on their *operand availability* when they are dispatched to the IQ and the *producer instruction type for non-ready operands*. The first category of instructions are those that have all their operands Ready at Dispatch (*R@D*). These instructions do not need to wait in the IQ to become ready (as they already are) and can be directly sent for execution if functional units are available. In this work we steer *R@D* instructions to simpler, FIFO-based instruction queues. Figure 1 shows that 20% of all dynamic instructions are *R@D* in SPECcpu2006, for a typical 4-wide OoO processor with 128 ROB entries.

The second category of instructions are those that have only one non-ready operand at dispatch, and this operand is produced by a *R@D* instruction. As the producers of these instructions are likely to complete their execution soon (their operands are already ready), these instruction are likely to be ready shortly as well, and we call them Almost Ready at Dispatch (*AR@D*). Also, we discover that if the only non-ready operand of an instruction is generated by another *AR@D* instruction, it does not result in frequent stalls. Therefore, we include these instructions also in the *AR@D* category. Figure 1 shows that 22% of the dynamic instructions are *AR@D*.

The third category includes a dependent instructions with the first instruction receiving at least one of its operands from a load instruction, or Load Tails (*LDTail*). We observe that such chains might reside in the IQ for long intervals, especially if the load misses in the caches. These chains occupy space in the IQ that could be used for other instructions that can potentially be ready sooner. We propose to steer these chains to a simple FIFO to reduce IQ pressure. Once the load receives its data, these dependent instructions can be executed quickly from the FIFO. Our analysis shows about 25% of the dynamic instructions are *LDTail*.

The final category of instructions are Not Ready at Dispatch (*NR@D*). In this category non -f the operands are ready which includes instructions whose operands come from a load dependant instruction, or whose operands come from other *NR@D* instructions. The waiting time for these instructions in the IQ is unpredictable, and they may become ready out-of-order. Therefore, they do benefit from OoO scheduling, and steering them to a FIFO would increase stalls. The remaining 33% of dynamic instructions are *NR@D*.

Based on these observations, we propose a new core design that employs cheap, in-order FIFO queues for instructions that do not need out-of-order scheduling (*R@D*, *AR@D*, and *LDTail*), thus freeing up expensive, out-of-order IQ entries for instructions that benefits the most from them (*NR@D*). Offloading instructions from the IQ to FIFO queues frees space in the IQ for instructions that would otherwise not have been able to be dispatched. This increases the effective issue window (IQ size plus FIFO queue size) and increases performance. As more than 60% of instructions are placed in and issued from the FIFO queues, our design allows us to reduce the issue width of the IQ from 4 to 1. This trade-off, of more, cheaper FIFO queues for a narrower expensive IQ, allows us to reduce energy while improving performance. Our primary contributions include:

- Identifying a easy-to-detect classes of instructions that do not benefit from OoO scheduling (*R@D*: operands are ready at the dispatch stage), or benefit minimally (*AR@D*: operands are almost ready).
- Demonstrating that more than 60% of instructions fall into these classes, and can be effectively offloaded from the expensive IQ to simple FIFO queues, thereby improving energy efficiency.
- Proposing and evaluating a core design that steers these instruction classes to simple FIFO queues and the remaining instructions to a 1-wide out-of-order IQ to provide a 50% energy savings while also delivering 8% performance gain.

II. Background and Motivation

A. The cost of OoO scheduling

OoO cores aim to execute instructions as soon as their source operands become ready, even though older instructions may be stalled [1], [4]. Bypassing stalled instructions to enable early execution of ready instructions improves performance. However, the ability to *identify* instructions as soon as they become ready and *select* among all ready instructions results in an immensely complex and energy-intensive IQ implementation. We describe the two main steps in OoO instruction scheduling below:

Wake-up: Instructions wait in the IQ until their operands become ready and they are selected for execution. The IQ then *wakes-up* the waiting instructions (marks them as ready for execution) as their producers finish execution and generate the required operands. As a waiting instruction can be anywhere in the IQ, the results[1] of executed instructions must be broadcast to all entries in the IQ. Furthermore, as multiple instructions can be executed every cycle, multiple results need to be broadcast simultaneously. For every completed operand broadcast, every instruction in the IQ needs to compare its input operands to see if there is a match, indicating that the operand is ready. This requires multiple comparators per IQ entry and broadcast busses as many operands as can be generated in a cycle. In

[1]Depending on the implementation, the broadcast information can be the result, destination register number, instruction id, or a combination of these.

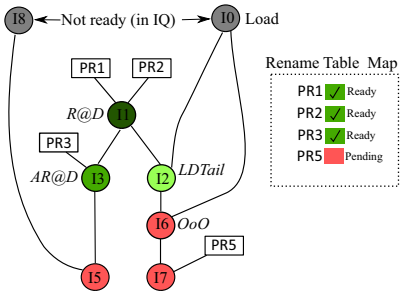

Fig. 3. Instruction dependency graph showing *R@D* (green), *AR@D* (blue), and *LDTail* (red) instructions. The register rename table is shown, indicating that physical registers 1-3 have been written (inputs to the *R@D* and *AR@D* instructions) while physical register 5 (PR5) has not. (I7) has PR5 and (I6) as input operands. Apart from PR5, I6 is pending as well which is why (I7) is not included in the *LDTail*. I6 is a load dependant instruction but it is not a *LDTail* since it has two pending operands.

the case of a match, the operand is marked as available. The instruction itself becomes ready when all operands are ready.

Selection: Instruction issue logic selects instructions for execution from the ready instructions in the IQ by priority. As ready instructions can be anywhere in the IQ, all IQ entries need to be examined in parallel to be able to select among them. Ready instructions are typically prioritized based on their type (memory accesses first to increase MLP) or age (oldest first to avoid chains of stalled instructions). Computing these priorities requires complex comparison trees of instruction opcodes and tags. In addition to the priority logic, the IQ requires as many output ports as the maximum issue width to enable the selected instructions to be read out. As a result of this complexity, the size and energy of the IQ increases super-linearly with the number of entries and issue width [1].

B. Do all instructions need OoO scheduling?

We observe that not all instructions need an out-of-order wakeup and select for early execution. For example, if an instruction is *R@D*, it does not need the result broadcast mechanism of the IQ wakeup to detect operand availability (as they are already available). Such *R@D* instructions can be dispatched to a simpler, and hence, cheaper, FIFO queue to enable them to bypass the stalled instructions in the IQ. As only *R@D* instructions placed in the FIFO queue, there will not be any stalls in this queue. Offloading such *R@D* instructions from the IQ to a simple FIFO queue provides a significant energy saving opportunity as the issue width of the IQ can be reduced because FIFO queue will also supply instructions for execution.

Ready-at-Dispatch (*R@D*, 20%): These instructions have all of their source operands ready when they are dispatched to the IQ. Compiler optimizations that move producer instructions as early as possible, such as load-hoisting or decoupled access-execute [5], are particularly likely to result in *R@D* instructions at runtime. In figure 3, instruction I_1 is a *R@D* instruction since both of its operands are ready when is it is dispatched.

Almost-Ready-at-Dispatch (*AR@D*, 22%): These instructions have one of their operands ready while the non-ready operand comes from either a R@D or AR@D instruction but not from a load. As a result, these instructions are likely to be ready soon, and are also good candidates for scheduling via a FIFO queue. In figure 3, instruction I_3 is *AR@D*. We do not consider instruction I_4 to be *AR@D* as it is likely to take longer to be ready since neither of its operands are ready.

Load-tail (*LDTail*, 25%): Instructions with one ready operand and the other one dependant on a load instructions are classified as load tails (*LDTail*). In figure 3 we see that the load instruction, I_0 has two dependent instructions whose sources are either ready or come from *R@D* (I_2) or *AR@D* (I_6) instructions. Only I_2 is considered a *LDTail* instruction. I_6 is not included as it is likely to take longer since its non-load operand is not yet ready as well. I_7 is not a *LDTail* because it has a source coming from another register that has not yet been written. Load and store instructions should always be dispatched to the IQ to execute them as early as possible to expose both MLP and ILP.

TABLE I
MICROARCHITECTURAL PARAMETERS (BASED ON NEHALEM [6])

Freq, ISA	3.4 GHz, x86-64
L1i/d	32KiB, 8-way, 4clk
L2	256KiB, 8-way, 12clk
L3	1MiB, 8-way, 36clk
DRAM	200clk
Branch Predictor	Two level, front end penalty 10clk
ROB/IQ/RF(Int,FP)/LQ/SQ	128/56/(68,68),48/36
FIFO queues	32 entries, issue up to 3 from head
Technology/VDD/temp	22nm itrs-hp/0.8/360K

TABLE II
FIFO AND IQ CONFIGURATIONS

Design	# FIFOs	IQ Issue Width	RF ports
Baseline	0	4	8
Design #1	1	1	8
Design #2	2	1	8
Design #3	3	1	8
FXA [7]	1	2	10

III. SIMULATION ENVIRONMENT

We use the Multi2sim simulator [8] (x86 target) with SPEC CPU2006 [9], fast-forwarding 1B instruction, cache warming for 250M, and then 1B instructions of detailed simulation. For the energy model we use Cacti and McPAT[10], [3] .

IV. IMPLEMENTATION

To take advantage of the amenability of our identified instruction classes to simpler scheduling, our approach relies on steering appropriate classes to appropriate FIFOs or out-of-order queues, and issuing them from there. This allows us to improve efficiency by reducing the load on the IQ and reducing its required width, as most instructions are steered to the FIFOs and issued from them. To accomplish this, we need to be able to cheaply classify instructions, steer them to the appropriate queue, and identify when *AR@D* and *LDTail* instructions at the head of a queue are ready for execution.

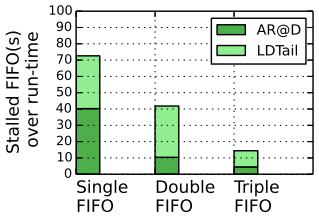

Fig. 4. The distribution of *R@D* FIFO stalls caused by *AR@D* and *LDTail* instructions. Placing the instruction classes in separate queues provides for more out-of-orderness between them, and allows different classes of instructions to bypass each other when they are ready, thereby reducing stalls.

A. Classifying, steering, and waking instructions

The classification of *R@D* instructions simply requires checking if both operands for an instruction are ready at dispatch. We accomplish this by examining the existing Rename Map Table (RMT). Similarly, instructions with one ready operand are either *AR@D* or *LDTail*. To detect *LDTail* instructions, we need an additional bit in the RMT that indicates whether an allocated register is coming from a load instruction. We can then distinguish between *AR@D* and *LDTail* at dispatch by checking that bit for the source registers. The remaining instructions are *NR@D*. Instructions are steered to the appropriate queue based on the FIFO design, as discussed below.

R@D instructions at the head of a FIFO queue require no check before issuing them to a functional unit for execution. However, *AR@D* and *LDTail* instructions may not be ready by the time they reach the head of a FIFO queue. Therefore, we check the ready bit of the pending operation directly in the Rename Map Table before issuing from the FIFO. The FIFOs in our designs examine the three instructions closest to the head of the FIFO and are able to issue up to 3 instructions from the FIFO together. While the FIFOs can each issue up to 3 instructions per cycle, the total issue width of the processor (FIFOs plus IQ) is kept to 4, as in the baseline design.

B. 1st Design: Single FIFO (reduce IQ pressure)

Our first design simply steers all *R@D*, *AR@D*, and *LDTail* instructions to a single FIFO queue (issuing up to 3 instructions per cycle), with the remaining *NR@D* in the OoO IQ (issuing up to 1 instruction per cycle.) The performance and performance per energy results of this implementation are shown in the leftmost bars of figure 6 and figure 7, normalized to the baseline OoO processor (table I). As figure 6 shows, a single 3-wide FIFO with a 1-wide OoO IQ deliers worse performance than a baseline 4-wide OoO IQ.

The fundamental bottleneck of this design is that *AR@D* and *LDTail* instructions are mixed with *R@D* instructions. This causes the FIFO to frequently stall when non-ready *AR@D* or *LDTail* instructions reach the head, which blocks other *R@D* instructions in the FIFO from issuing. The breakdown of stall sources for the single-FIFO design are shown in figure 4. We can see that the FIFO queue was stalled and could not issue instructions (issued zero or less than the bandwidth of three) for 72% of the execution cycles, 40% due to *AR@D* and the remaining 32% were due to *LDTail*. This suggests that keeping the *AR@D* and *LDTail* instructions out of the FIFO that holds *R@D* instructions would reduce *R@D* stalls.

Looking at performance per energy results in figure 7, the single FIFO design outperforms the baseline OoO by 17% due to the cheaper FIFO and 1-wide IQ replacing the much more complex 4-wide IQ. The energy breakdown of the baseline 4-wide IQ vs. our 1-wide IQ and FIFOs is shown in figure 8. We next seek to address the performance loss from a single FIFO queue by adding additional FIFO queues to prevent *R@D* isntrucitons from being stalled by *AR@D* and *LDTail* instrucitons. In essence, by adding more FIFO queues, we hope to increase the out-of-orderness without the need for the full CAM functionality (and cost) of an OoO IQ.

C. 2nd Design: Dual FIFOs (unblocking the R@D FIFO)

To tackle the problem of FIFO stalls, we add a second FIFO for *AR@D* and *LDTail* instructions. This leaves the first FIFO exclusively for *R@D* instructions, which will never stall. As with the previous design, the maximum issue width is 1 for the OoO IQ and 3 across both FIFOs. For selecting between the FIFOs, a higher priority is given to instructions from the *R@D* FIFO.

The performance and performance per energy results of the dual FIFO design are shown in the second set of bars in figure 6 and figure 7. By eliminating the stalls caused by *AR@D* and *LDTail* instructions in *R@D* instruction execution, the dual FIFO design outperforms the single FIFO design overall, and is even better than baseline 4-wide OoO IQ design in a few benchmarks. On average, the dual FIFO design matches the baseline performance, but does so with more energy-efficient scheduling due to its 1-wide OoO IQ and two 3-wide FIFOs. From energy point of view, dual FIFO design outperforms the baseline in terms of performance per energy by over 30% on average.

Despite the second FIFO queue, there are still many FIFO stalls, as seen in the middle bar of figure 4. For this design, the majority of the stalls, 35% of all cycles, are coming from *LDTail* instructions that are blocking the second FIFO. To tackle this problem we separate the *LDTail* instructions from the *AR@D* ones by placing them in a third FIFO queue.

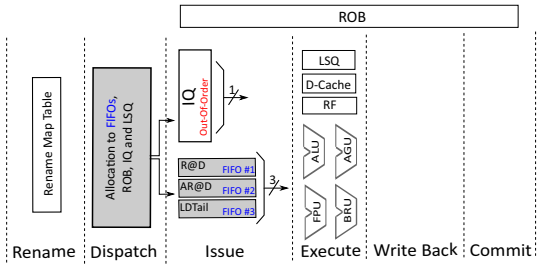

Fig. 5. FIFOrder microarchitecture. Instructions are classified in the rename stage based on the operand ready bits in Rename Map Table. In the dispatch stage, they are steered to the IQ or FIFOs, depending on the instruction classification. The issue stage stores the instructions in either the FIFOs or IQ, and selects ready instructions across the queues for execution.

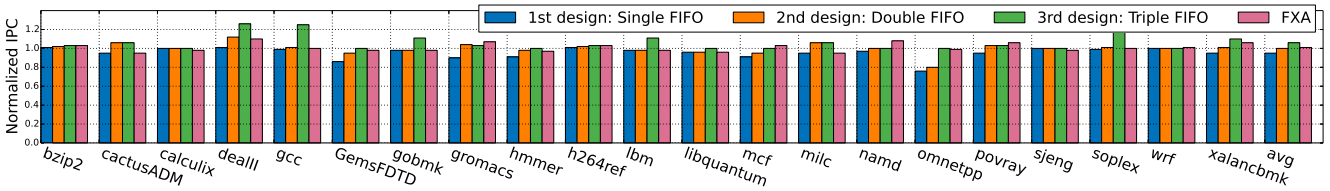

Fig. 6. IPC comparison between three designs of FIFOrder and a related work, FXA [7] normalized to baseline. Baseline: 4-wide OoO. Our designs: 1-wide OoO plus 1, 2, or 3 FIFOs. FXA has a 2-wide OoO (see table I and table II).

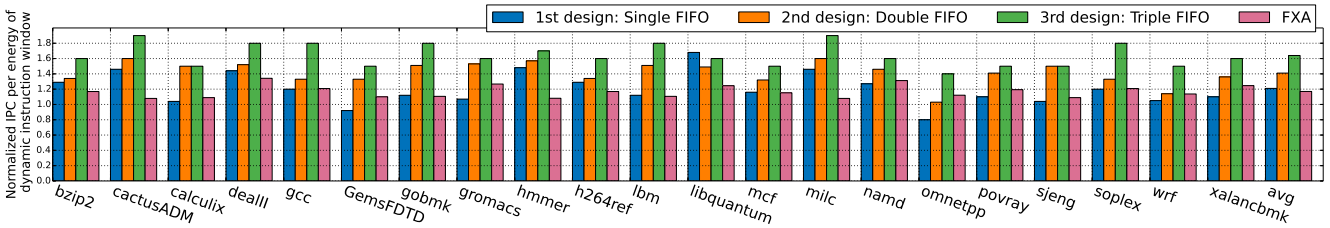

Fig. 7. Normalized performance per energy of the dynamic instruction window. Both baseline and FXA [7] spend more energy on issuing instructions due to their issue-widths of 4 and 2, respectively, compared to 1 in our designs.

D. 3rd Design: Triple FIFO (unblocking AR@D FIFO)

This design provides three separate FIFOs, one for each of the instruction classes to prevent them from stalling each other, along with a 1-wide OoO IQ. For instruction dispatch, the highest priority is given to *R@D* instructions, then *AR@D*, and lowest to *LDTail*. This order attempts to give priority to the instructions that are most likely to be ready soonest. Figure 5 illustrates the final design and pipeline view of the FIFOrder microarchitecture. The shaded and gray parts are the pipeline parts that are affected by the FIFOrder. Functional unites are shared between the IQ and the FIFOs to bring more flexibility at issue time.

The performance and performance per energy result of the three FIFO implementation are shown in the third set of bars if figure 6 and figure 7. As a result of reducing FIFO stalls (figure 4, right bar), this design outperforms the baseline out-of-order by 8% on average and is 55% higher in energy efficiency.

E. Energy Breakdown

For the baseline 4-wide OoO (table II) we consider the size and width of the IQ, and use the number of writes and reads to compute energy with [2]. Since our triple FIFO design delivers the best performance per energy, we pick this design to compare with the baseline in detail. Figure 8 shows the IQ energy breakdown of this design compared to the 4-wide OoO and the FXA [7] designs (both dynamic and static energy). The total energy of the baseline is reduces by 50% in our design, primarily due to an 80% reduction in IQ energy from the smaller IQ width and a 73%-60% reduction in IQ reads/writes, and due to the relatively small energy overheads of FIFO queues.

V. Related Work

A previously proposed technique, FXA [7], also aims to reduce energy consumption of an OoO core by executing some of the instructions in program order. FXA takes a brute force approach of first trying to execute all instructions in program order and then moving only those instructions that could not be thus executed to an OoO queue. Our design splits the instruction stream upfront for in-order and OoO execution. FXA *inserts* a 4-stage in-order pipeline between the dispatch and IQ of an OoO pipeline to filter out instructions that can be executed early and in-order. This requires functional unit replication in the front end. Our design, in contrast, shares the same functional units among in-order and OoO execution. To reduce the area overhead, FXA replicates only integer functional units. As a result, floating point operations are always executed through the OoO IQ. Our design has no such limitation. FXA potentially increases register file reads as operands first need to be read for in-order execution and then again for OoO execution if in-order execution of an instruction fails. A potential solution is to pass the operands from in-order to OoO execution engine, but that requires moving the operands and storing them int the IQ. (see the detail setup of [7] in table I and table II).

We compared our three designs with FXA in terms of performance and performance per energy, as shown by the right most bars of figure 6 and figure 7, respectively. In terms of performance, FXA outperforms the single FIFO design because FXA uses an OoO IQ issue width of 2, or twice that of our design. Our dual FIFO design performs about as well as FXA. Finally, the triple FIFO design that address all stall sources, outperforms FXA both in terms of performance and performance per energy. The energy breakdown of FXA is shown in the middle stacked bars of figure 8. It saves 40% of the energy compare to the baseline. Based on this result, the main reasons our final design consumes less energy compared to FXA is having half of the IQ issue width of FXA and

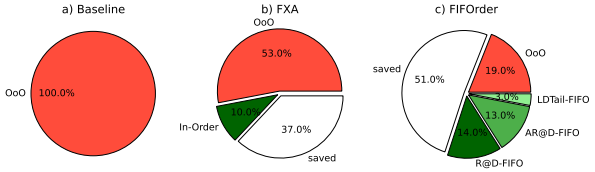

Fig. 8. Energy breakdown of a) the dynamic instruction window for the out-of-order 4-wide baseline, b) FXA [7], and c) the proposed FIFOrder micro-architecture with three 3-wide FIFO queues and a 1-wide IQ across SPEC2006 benchmarks.

offloading the majority of the instructions from the IQ to the FIFOs. Also since FXA passes all of the instructions through the in-order pipeline, before forwarding them to OoO pipeline, regardless of the readiness of their operands, about 7% of its energy is consumed in the in-order part of the pipeline.

One of our contributions is reducing the instruction wake-up and select while keeping the IQ size unchanged. The execution can not get started even if only a single operand of an instruction with multiple input operands is missing. Therefore, including this type of instructions in the pool of instruction to be selected not only does not provide more flexibility for instruction selection but also increases the cost. Ernst et al. [11] proposed a separate wake-up and select policy for instructions with more than one pending-sources. For this class of instructions, they only compare the register renaming tag for the input operand which has the longest slack (last arriving input) to reduce the total number of tag comparison. This problem is addressed by introducing a last tag speculation technique that predicts which input operand of an instruction arrives last, and is used for scheduling execution. However, this approach can identify instructions whose operands are both not ready, which does not simplify later instruction selection as we include them in *NR@D* instructions.

Brown et al. [12] reduced the cost of instruction wake-up and selection by reintroducing pipeline techniques. They pipeline the wakeup and select loops and introduce smaller loops,a critical, single-cycle loop for wakeup; and a non-critical, potentially multi-cycle, loop for select but still all of the instruction pass through wakeup and select stages during instruction scheduling. In comparison, our solution reduces the cost of instruction scheduling for all classes of instructions with all of their operands ready to the ones which have all operands pending. For the ones with two pending operands it is impossible to ignore wakeup and select however, we applied a cheaper IQ (1-wide issue width) for this type of instructions.

Long Term Parking [13] and Load Slice Core [14] classified instruction as urgent and non-urgent, where urgent instructions form a chain of address generating instructions leading up to a load. *LDTail* instructions are conceptually a subset of non-urgent instructions. In [14] they steer the urgent instructions to a secondary FIFO instruction queue to enable MLP-generating urgent instructions to bypass other instructions in an in-order pipeline. In LTP [13], they steer non-urgent instructions to a FIFO (parking) to reduce the IQ pressure. Since non-urgent instructions are woken-up out-of-order, and the FIFO does not have an out-of-order search capability, all instructions are inserted back to the IQ before wakeup. In our design instructions that are placed in the FIFO queues are issued directly from their respective queues, and do not need to pay the latency and energy cost of being re-inserted into the IQ.

VI. Conclusion

Improving out-of-order processor performance has long required increasing the size and width of the instruction queue. However, as this structure must search for independent instructions and read out ready instructions with specific priorities, scaling it up has proven to be extremely energy costly.

To address this problem, we identified classes of instructions that do not benefit from out-of-order scheduling, and proposed an architecture, FIFOrder, that takes advantage of this classification to efficiently execute them. In FIFOrder we use cheap FIFO queues to store and issue instructions that are ready at dispatch or soon to be ready (60% of dynamic instructions), thereby reducing the load on the IQ, and allowing us to reduce its issue width to just 1.

The combination of fewer instructions in the expensive IQ and its reduced width allows us to provide a 50% energy savings while delivering 8% improved performance over a baseline 4-wide OoO processor.

Acknowledgement

This work was supported by the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program.

References

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 206–218, May 1997.

[2] Y. Kora, K. Yamaguchi, and H. Ando, "Mlp-aware dynamic instruction window resizing for adaptively exploiting both ilp and mlp," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, 2013, pp. 37–48.

[3] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 469–480.

[4] M. Alipour, T. E. Carlson, and S. Kaxiras, "Exploring the performance limits of out-of-order commit," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17, 2017, pp. 211–220.

[5] A. Jimborean, K. Koukos, V. Spiliopoulos, D. Black-Schaffer, and S. Kaxiras, "Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14, 2014, pp. 262:262–262:272.

[6] I. Corporation, "Intel® 64 and ia-32 architectures optimization reference manual," http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, Jun. 2016.

[7] R. Shioya, M. Goshima, and H. Ando, "A front-end execution architecture for high energy efficiency," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014, pp. 419–431.

[8] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12, 2012, pp. 335–344.

[9] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[10] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '11, 2011, pp. 694–701.

[11] D. Ernst and T. Austin, "Efficient dynamic scheduling through tag elimination," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02, 2002, pp. 37–46.

[12] M. D. Brown, J. Stark, and Y. N. Patt, "Select-free instruction scheduling logic," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 34, 2001, pp. 204–213.

[13] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud, "Long term parking (ltp): Criticality-aware resource allocation in ooo processors," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48, 2015, pp. 334–346.

[14] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015, pp. 272–284.