# Delay and Bypass: Ready and Criticality Aware Instruction Scheduling in Out-of-Order Processors

Mehdi Alipour, Stefanos Kaxiras, David Black-Schaffer
Uppsala Univeristy, Sweden
firstname.lastname@it.uu.se

Rakesh Kumar
Norwegian University of Science and Technology, Norway
firstname.latname@ntnu.no

***Abstract*—Flexible instruction scheduling is essential for performance in out-of-order processors. This is typically achieved by using CAM-based Instruction Queues (IQs) that provide complete flexibility in choosing ready instructions for execution, but at the cost of significant scheduling energy.**

**In this work we seek to reduce the instruction scheduling energy by reducing the depth and width of the IQ. We do so by classifying instructions based on their *readiness* and *criticality*, and using this information to *bypass* the IQ for instructions that will not benefit from its expensive scheduling structures and *delay* instructions that will not harm performance. Combined, these approaches allow us to offload a significant portion of the instructions from the IQ to much cheaper FIFO-based scheduling structures without hurting performance. As a result we can reduce the IQ depth and width by half, thereby saving energy.**

**Our design, Delay and Bypass (DNB), is the first design to explicitly address both readiness and criticality to reduce scheduling energy. By handling both classes we are able to achieve 95% of the baseline out-of-order performance while only using 33% of the scheduling energy. This represents a significant improvement over previous designs which addressed only criticality or readiness (91%/89% performance at 74%/53% energy).**

## I. Introduction

Out-of-order processors invest heavily in complex resources for extracting instruction- and memory-level parallelism. Among these resources, the Instruction Queue (IQ) plays a critical role by identifying and issuing ready instructions for execution irrespective of their program order [1], [2], [3], [4]. The amount of parallelism that can be extracted by the IQ is limited by its dimensions: issue *width* limits the number of instructions issued per cycle and *depth* limits the window of schedulable instructions from which it can identify parallelism.

The IQ is also the most complex and power-intensive (18-40% of core power) [4], [5], [6], [7] core component. Its complexity stems from its need to *wake-up* instructions for execution when all their operands becomes available and *select* them for execution based on priority heuristics [8], [9]. For wake-up, the IQ must track operand availability, normally by broadcasting the destination register of completing instructions to all instructions in the IQ to identify which are waiting for it. This requires a complex Content-Addressable-Memory (CAM) structure with comparators for all source operands of each instruction in the IQ for each possible destination operand retiring in a cycle. That is, the number of comparators (and wires) for identifying ready instructions grows super-linearly with both the depth of the IQ and the issue width of the processor. Selecting from the ready instructions requires reduction trees whose complexity grows linearly with the issue width of the IQ and logarithmically with its depth [3], [10], [11]. As a result of this complexity, the power consumption of the IQ grows dramatically with depth and width, as shown in Figure 1. Conversely, the energy of simpler scheduling structures, such as in-order FIFO queues, scales more gracefully [12]. However, this comes at the cost of less flexible scheduling and performance: FIFOs can only consider instructions at the head of their queue.

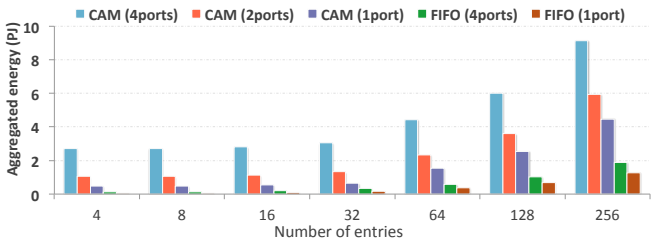


Fig. 1. Energy consumption per access of IQ (CAM-based) and queue (FIFO-based) instruction storage as a function of width (color) and depth (x-axis). IQ energy grows very significantly with both width and depth, while the simpler FIFO implementation is enormously more efficient.

A large body of work has focused on reducing IQ energy consumption. Previous approaches include circuit-level techniques that reduce the cost of wakeup and selection logic [13], [14], [15], dynamically partition the IQ to reduce lookup energy [4], [16], schedule to prioritize performance critical instructions [17] to reduce the IQ size and energy while maintaining the same performance, etc. These approaches reduce the per-instruction IQ energy. However, these approaches *place all instructions in the IQ, regardless of whether or not they benefit from its expensive wake-up and select mechanisms*.

Recent work has taken the different approach of scheduling a subset instructions from simpler, more energy-efficient structures to reduce pressure on the IQ. This allows for the use of smaller and/or narrower IQs without hurting performance. Examples of this approach include filtering instructions that can be executed earlier [18], "parking" instructions that will not be ready for a while [3], and bypassing instructions that do not benefit from out-of-order scheduling [19]. Implicit to the approach of reducing IQ pressure is the need to identify instructions that do not benefit from the expensive scheduling capabilities of the IQ. This can be done by classifying

instructions based on their *criticality* [3] and/or *readiness* [18].

**Ready instructions** have all their operands available. 20% of total instructions (Figure 2) are ready early in the pipeline (e.g., at dispatch), and a significant number (11%) become ready shortly thereafter (Figure 4). These ready instructions *neither need nor benefit from the expensive out-of-order scheduling mechanisms of the IQ*. This presents the opportunity to *bypass* the IQ for ready instructions and execute them directly. While ready instructions typically complete quickly, and therefore do not put as much pressure on the IQ as non-ready ones, bypassing the IQ altogether for them eliminates the installation, wake-up, and select energy needed to issue them from the IQ.

**Critical instructions** hurt performance if their execution is delayed. They often include instructions that lead up to or generate memory-level-parallelism (MLP). Conversely, *non-critical* instructions can be delayed to a significant degree without penalty. Non-critical instructions are typically dependent on long-latency operations and often spend a long time in the IQ before becoming ready, thereby increasing IQ pressure. The long lifetime of non-critical instructions presents the opportunity to *delay* their insertion into the IQ and thereby reduce IQ pressure. Reducing the number of non-critical instructions in the IQ also naturally prioritizes the critical ones.

Prior work has taken advantage of these classes of instructions *separately*: Long Term Parking (LTP) [3] exploited instruction criticality to delay the IQ allocation of non-critical instructions by "parking" them in a cheaper in-order FIFO queue. This reduced capacity pressure on the IQ by keeping instructions that would not be ready for a long time out of the IQ. As a result, LTP was able to reduce the IQ depth. Conversely, the Front-end Execution Architecture (FXA) [18] exploited instruction readiness by executing ready instructions through in-order pipeline stages before placing the remaining instructions in the IQ. This allowed FXA to reduce the IQ width as fewer instructions needed to be sent from the IQ to the functional units.

In this work we demonstrate that there are significant benefits from considering critical and readiness *together*. To do so, we first analyze the distribution of instruction classification across a wide range of benchmarks (Figure 2). We see that if LTP architecture were made ready-aware it could reduce its scheduling energy from 74% of the baseline to 44%. Similarly, a criticality-aware FXA could go from 53% to 46%.

Based on these insights we introduce the *Delay-and-Bypass* (DNB) scheduling architecture that leverages instruction criticality together with instruction readiness. DNB first uses instruction criticality to *delay non-critical instructions* by placing them in a cheap, in-order FIFO scheduling queue, called delay queue (DLQ). Note that during their delay time in the DLQ many non-ready instructions become ready as well. When instructions in the DLQ approach the head of the reorder buffer (ROB) they are taken out of the DLQ and either *bypass*

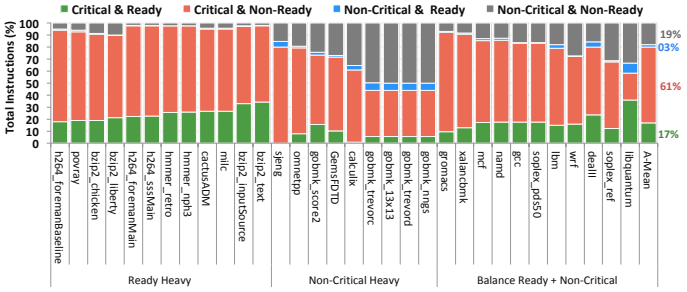


Fig. 2. Readiness and criticality instruction classification in an out-of-order processor.

*the IQ* and are issued directly to the functional units if they are ready, or are inserted into the IQ if not. This reduces IQ occupancy pressure by keeping non-critical instructions out of the IQ for most of their lifetimes and reduces IQ issue width pressure by allowing instructions to issue directly from the cheaper in-order FIFO.

For the remaining critical instructions, DNB *bypasses the IQ for those that are ready* and sends them for immediate execution[1]. Only the Critical & Non-Ready instructions are sent directly into the IQ. This allows the expensive scheduling logic of the IQ to focus on issuing those instructions as soon as they become ready (see Figure 3). DNB thereby ensures that critical instructions execute as quickly and efficiently as possible by either utilizing the expensive IQ scheduling logic for those that are not-ready or by bypassing the IQ for direct execution for those that are ready. This reduces IQ capacity pressure by keeping Critical & Ready instructions out of the IQ and reduces IQ width pressure by allowing them to bypass the IQ altogether.

By addressing both criticality and readiness, DNB is able to reduce both IQ width and depth while retaining performance. Our principal contributions include:

- An instruction taxonomy based on criticality and readiness that provides insights into how to efficiently schedule instructions. In particular, this taxonomy explains why instructions can and should bypass the IQ or be delayed before being inserted into it to save energy without hurting performance.
- An analysis of instruction criticality/readiness showing how previous designs addressing only criticality or readiness have missed significant energy savings opportunities.
- The DNB design that leverages both criticality and readiness of instructions to reduce IQ width and depth, thereby providing a significant power savings at minimal performance degradation.
- Analysis showing that the DNB design is far less sensitive to reductions in the depth and width of the IQ than other designs due to more intelligent instruction steering.

[1]To handle structural hazards and instruction priorities they are actually inserted into a decoupling queue as described in Section IV.

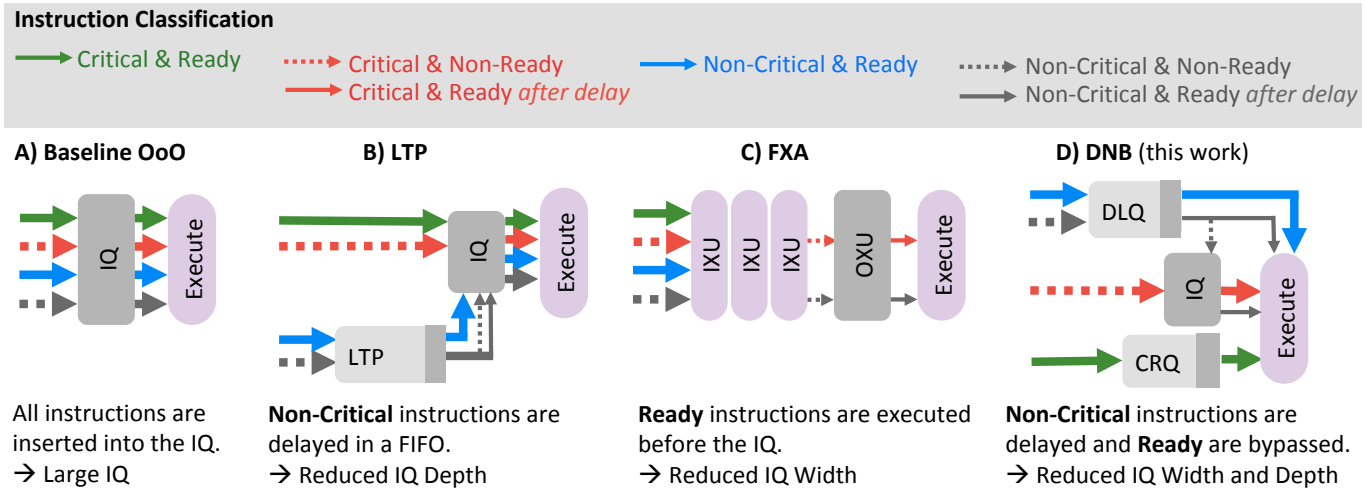


Fig. 3. Instruction placement based on readiness/criticality for the four designs discussed in this work. Ready instructions are shown with solid lines and non-ready in dashed. Non-ready become ready after a delay and are handled differently across the designs. Note that instructions can only be issued from the head of the FIFO queues (dark gray areas in LTP, CR, and DLQ) while any instruction in the IQs/OXU can be issued.

## II. Motivation

### A. Instruction Classification

We classify instructions based on two characteristics: *criticality* and *readiness*. Borrowing the definition of criticality from prior work [3], [20], we define an instruction to be *critical* if it contributes to MLP. In our definition, all memory accesses as well as instructions generating addresses for these accesses are considered critical. We define an instruction to be *ready*[2] if all of its operands are available. Based on these two characteristics an instruction may not require or benefit from IQ allocation, or IQ allocation may be delayed without a performance penalty:

- **Critical & Ready:** As these instructions contribute directly to MLP they are critical for performance and should not be delayed. However, as they are also ready for execution, they do not benefit from the expensive wake-up and select mechanism of the IQ. Therefore, these instructions can bypass the IQ and be issued directly to functional units for execution.
- **Critical & Non-Ready:** Due to their criticality, these instructions should not be delayed. However, as they do not have all their operands available, they cannot be issued for execution immediately. Therefore, they should be allocated IQ entries so that they can take advantage of the IQ's expensive out-of-order wake-up and select mechanism to be selected for execution as soon as they become ready.
- **Non-Critical & Ready:** As these instructions are ready for execution they do not require IQ allocation and can be directly issued for execution. However, since they are non-critical, executing them early does not improve performance. On the contrary, eager execution might hurt performance if they take slots from critical instructions. Therefore, these instructions can both bypass the IQ placement and their execution can be delayed.
- **Non-Critical & Non-Ready:** These instructions are neither performance critical nor they are ready for execution. Placing them in the IQ would occupy entries until their operands become available, potentially at the cost of more critical instructions. On the other hand, delaying their scheduling will not harm performance as they are not performance critical. These instructions should therefore be delayed.

Figure 2 presents the distribution of criticality and readiness across the benchmark applications. (Methodology see Section V and Table I.) The applications are grouped into three categories based on the amount of ready and non-critical instructions, *Ready Heavy*, *Non-Critical Heavy*, and *Balance Ready+Non-Critical*.

As the figure shows through the diversity of instruction classes across the benchmarks, techniques that exploit only readiness or criticality will not achieve savings on all applications. For example, targeting only criticality would miss significant opportunities on applications with more ready instructions as they have very few non-critical instructions. Even a single application can comprise a diverse mix of both ready and non-critical instructions, such as libquantum or dealII, leading to sub-optimal power savings by existing techniques that do not handle both classes.

### B. Exploiting Criticality

Long Term Parking [3] (LTP) is the state-of-the-art mechanism for exploiting instruction criticality to reduce the power consumption of instruction scheduling. LTP classifies instructions in the decode stage based on their contribution to MLP. The critical instructions (the MLP-generating ones) are dispatched to the IQ, whereas the rest of the non-critical

[2]LTP defined "ready" to mean instructions that do not depend on other long-latency instructions. Our definition is more strict in that ready instructions can be immediately executed and are easier to detect in hardware.

instructions are delayed by "parking" them in an energy-efficient FIFO queue. (See Figure 3B.) When the parked instructions get close to the head of ROB, they are moved to the IQ to avoid further delaying their execution, which causes stalls.

As Figure 2 shows, non-critical (Non-Critical & Ready + Non-Critical & Non-Ready) instructions constitute about 22% of the dynamic instruction stream. By delaying their insertion to IQ, LTP can leverage the reduced IQ pressure to reduce the IQ depth by half with minimal performance penalty [3]. However, LTP misses the opportunity offered by ready instructions. These instructions (Critical & Ready + Non-Critical & Ready) account for 20% of the dynamic instruction stream. In addition, the delayed Non-Critical & Non-Ready instructions can become ready by the time LTP decides to pick them for insertion into the IQ if their producers finish execution. Our results in Figure 4 show that more than half of the delayed (parked) instructions become ready for execution while being parked (11% of dynamic instructions). In summary, a total of 32% of instructions are ready for execution when they are inserted to IQ by LTP. These instruction represent a significant power saving potential missed by LTP.

### C. Exploiting Readiness

The Front-end Execution Architecture [18] (FXA) leverages instruction readiness to reduce the energy of instruction scheduling. To avoid IQ insertion for ready instructions, FXA places an in-order execution unit (IXU) between the rename and dispatch stages. All instructions pass through the IXU in-order and it directly executes ready instructions. (See Figure 3C.) In essence, FXA *filters* the instruction stream and only non-ready instructions reach the dispatch stage and are placed in the IQ.

To maximize the number of ready instructions the IXU consists of three pipelined stages of functional units. This gives each instruction three cycles to become ready and execute before being placed in the IQ. Our experiments shows that the total number of ready instructions increases from 18% to 24% due to these additional stages. Note that this delay is different from the delay introduced by LTP, as FXA delays all instructions whereas LTP delays only non-critical ones.

As the IXU filters out 24% of dynamic instructions, FXA leverages the reduced IQ pressure to reduce both IQ width and depth. However, FXA still allocates IQ entries for all the instructions that pass through the IXU unexecuted. Among these are non-critical instructions that could be delayed to further reduce the IQ pressure. Our results in Figure 4 show that very few Non-Critical & Non-Ready instructions become ready during these 3 stages (only 4% of total instructions) and the remaining (15% of total instructions) both pass through the IXU stages and are allocated IQ entries. Delaying the IQ allocation for these 15% of instructions would not only reduce IQ pressure but also give these instructions a further opportunity to become ready. Figure 4 shows that such a delay would significantly increases the number of Non-Critical & Non-Ready instructions that become ready to 11% of total

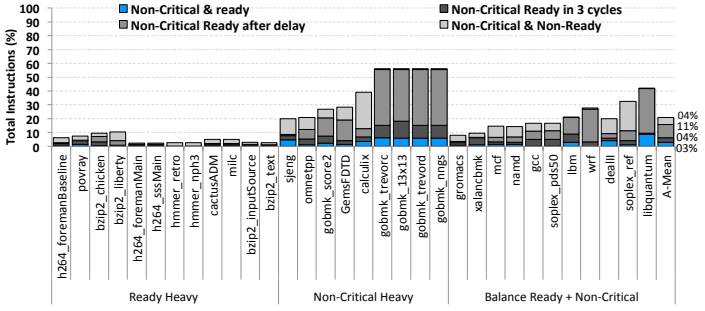


Fig. 4. Readiness of non-Critical instructions, both before issue (blue), after the 3 cycles of the FXA IXU, and after being delayed in the LTP parking. On average, 3% of instructions are Non-Critical & Ready and can bypass. 4% are Non-Critical & Non-Ready that *become* ready within 3 cycles, and 11% are Non-Critical & Non-Ready that *become* ready after being delayed. Combined, 18% of instructions become ready within the time frame of being delayed and could bypass the IQ as a result.

instructions (up from 4% in FXA), thus provide significant opportunity for bypassing the IQ altogether after delay.

## III. Criticality and Readiness

Leveraging both criticality and readiness will increase the number of instructions that can be offloaded from the IQ without hurting performance. This is because only Critical & Non-Ready instructions require the sophisticated scheduling of the IQ, whereas the remaining can either be delayed or bypassed. By taking advantage of both characteristics, only 61% of instructions (Critical & Non-Ready ones) require immediate IQ allocation (Figure 2) compared to 78% (Critical & Ready + Critical & Non-Ready) by only considering criticality and 80% (Critical & Non-Ready + Non-Critical & Non-Ready) by only considering readiness. This drastic increase in offloadable instructions can be leveraged to reduce the IQ depth. In addition, as ready instructions can be issued directly (bypassing the IQ) and in parallel with instructions from IQ, the IQ issue width itself can also be reduced. Reducing both dimensions of IQ provides significant power reduction as shown in Figure 1.

There are two naïve approaches to leveraging both classes: either add bypassing of ready instructions after parking to LTP (LTP+Bypass) or add delaying of non-critical instructions before the IXU to FXA (FXA+Delay). These approaches significantly reduce scheduling energy from 74% to 44% for LTP and from 53% to 46% for FXA, as shown in Figure 5 (performance is discussed in Section VI-A). However, there are significant shortcomings to both:

- *Non-ready instructions in the IXU:* In FXA+Delay [3], the 61% of instructions that are Critical & Non-Ready pass through the IXU while fewer than 1% become ready during its 3 stages. (Overall only 4% of non-ready instructions become ready during those 3 cycles, see Figure 4.) To avoid this extra energy, Critical & Non-Ready instructions should *bypass* the IXU and be placed directly in the IQ. Bypassing the IXU for Critical & Non-Ready instructions

[3]Adding an LTP FIFO to FXA.

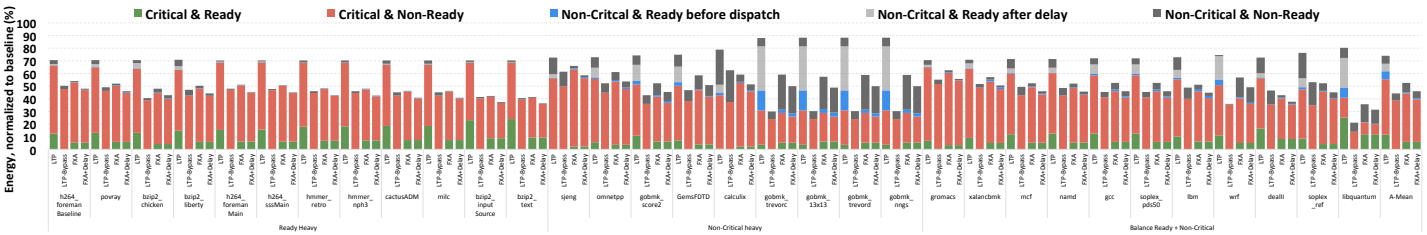


Fig. 5. Instruction scheduling energy reductions for naïve extensions to LTP and FXA.

reduces the percentage of instructions that go through the IXU from 81% in FXA+Delay (all instructions excluding Non-Critical & Non-Ready) to 20%. (in the original FXA 100% go through the IXU).

- *Ready instructions placed in the IQ:* To minimize area overhead, the IXU has no floating point units and limited memory functionality. As a result, all ready floating point instructions must be placed on the IQ before they can be executed, increasing both capacity and port pressure in the IQ. Additionally, the IXU arbitrates with the OXU for shared memory resources and ports, resulting in memory instructions being placed in both and trying to execute in both.
- *Area:* FXA has separate functional units for the in-order IXU and out-of-order OXU which leads to underutilization as many instructions are unable to execute in the IXU.
- *Pipeline depth:* The IXU increases pipeline depth by three cycles, which potentially increases branch misprediction penalty and delays the execution of instructions that require a functional unit only present in OXU.
- *Register file port pressure:* FXA requires that operands be read for instruction execution in the IXU and then again for execution in the OXU if the IXU fails to execute them.
- *Load criticality:* LTP only marks instructions as critical if they contribute to generating an address for a load. This results in some loads being treated as non-critical and delayed. Our simulations show this reduces performance by 7% on average.

To address the aforementioned inefficiencies of the FXA+Delay design, we propose the Delay and Bypass design. DNB shares functional units between the instructions that are identified as ready early in the pipeline and those selected for execution from the IQ. This is achieved by *bypassing* the ready instructions to functional units rather than adding a separate pipeline for executing them before the IQ. DNB further applies bypassing to instructions that are ready after delay, eliminating the need to re-insert them into the IQ. This approach avoids load imbalance, as all functional units are shared, and does not require more RF ports than the original design. By delaying non-critical instructions and bypassing the IQ for ready instructions (both ready at dispatch and ready after delay) the DNB is able to take advantage of both instruction criticality and readiness. In the next section we provide more details of the instructions issue policy and proposed pipeline.

## IV. Delay-and-Bypass (DNB)

DNB *bypasses* the IQ for Critical & Ready instructions (reducing both IQ capacity- and port-pressure, as well as IQ reads/writes), *delays* Non-Critical & Non-Ready and Non-Critical & Ready instructions by placing them in a FIFO (reducing IQ capacity pressure), and *bypasses* the IQ for delayed instructions that are ready when they reach the head of the delay FIFO (reducing IQ port pressure, as well as IQ reads/writes). In DNB only Critical & Non-Ready instructions are placed directly in the expensive IQ CAM, as these are the ones that are critical to execute as soon as they become ready. (See Figure 3D and a comparison across all designs in Figure 6.)

### A. DNB: Details

Figure 7 provides an overview of the DNB architecture: DNB adds a FIFO queue for delaying instructions (the Delay-Queue, DLQ), a FIFO queue for decoupling the back-end execution of Critical & Ready instructions from front-end fetch (the Critical-Ready-Queue, CRQ), and a Critical Instruction Table (CIT) for Iterative Backwards Dependency Analysis (IBDA) [21] to determine instruction criticality. The components common to both LTP and DNB are shown in light gray and additions in white.

**Fetch, Decode and Rename:** The DNB front-end determines instruction readiness and criticality. As in LTP, after instruction fetch the CIT is accessed with the PC to check if the instruction is critical. In rename the operand availability is used to detect if the instruction is ready.

**Detecting Critical Instructions:** As with LTP, DNB uses Iterative Backward Dependency Analysis (IBDA) [21] to iteratively identify chains of instructions that lead to MLP-generating instructions (see Section II-B). In addition, we mark all loads as critical to avoid delaying them and harming performance.

**Detecting Ready Instructions:** DNB uses the register rename stage to identify instructions whose source operands are both available. This incurs no overhead as the register file check is part of renaming.

**Instruction Dispatch:** DNB places Critical & Non-Ready instructions directly into the IQ for execution as soon as they are ready and places and all non-critical instructions into the DLQ to be delayed. Critical & Ready instructions bypass the IQ

| Design | Technique | Awareness | | | Instruction Issue | | | | | | Results | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | C | R | R after delay | R-C | R-NC | NR-C NR after delay | NR-C *R after delay* | NR-NC NR after delay | NR-NC *R after delay* | Performance | Scheduling Energy |
| OoO | | | | | **IQ** | **IQ** | **IQ** | **IQ** | **IQ** | **IQ** | 100% | 100% |
| LTP | Park (delay) | ✓ | | | **IQ** | **Park ➜ IQ** | **IQ** | **IQ** | **Park ➜ IQ** | **Park ➜ IQ** | 91% | 74% |
| FXA | Filter (bypass ) | | ✓ | 3 cycles | Filter | Filter | **Filter ➜ IQ** | Filter (2% in 3c) | **Filter ➜ IQ** | Filter (4% in 3c) | 89% | 53% |
| DNB | Delay & Bypass | ✓ | ✓ | ✓ | Bypass | Delay ➜ Bypass | **IQ** | **IQ** | **Delay ➜ IQ** | Delay ➜ Bypass | **95%** | **34%** |

Fig. 6. Comparison of the four designs. Limitations that increase energy: FXA is only able to handle non-ready instructions that become ready within 3 cycles (6% total) and filters even non-ready instructions, costing energy and adding 3 cycles of delay; LTP inserts all parked instructions into both the LTP parking FIFO and the IQ; DNB inserts inserts Non-Critical & Non-Ready instructions that do not become ready after delay into both the delay queue and the IQ. (R: Ready, C: Critical, NR: Non-Ready, NC: Non-Critical).

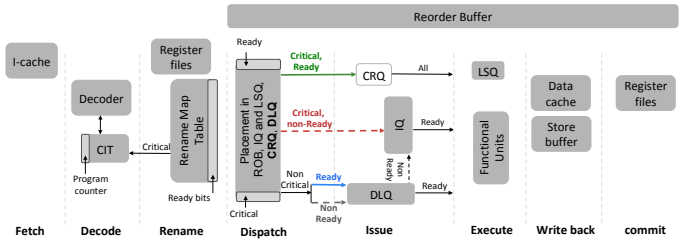


Fig. 7. The DNB microarchitecture. The Critical Instructions Table (CIT) uses IBDA to identify critical instructions while the Rename Map Table is used to identify ready ones. Instructions are placed in the appropriate scheduling structure based on their readiness and criticality.

for immediate execution via the CRQ. The CRQ is needed as ready instructions may not be able to be executed immediately due to structural hazards (lack of available functional units) and/or instruction priorities (newer ready instructions should not be prioritized over older ready instructions). The CRQ decouples the bypassing of ready instructions from the front-end to the back-end. This allows continued fetching even if the ready instruction cannot be immediately executed and is much cheaper than inserting them into the CAM IQ.

**Instruction Issue:** The DNB back-end can select instructions to issue from the IQ, CRQ and/or DLQ. The total issue width remains the same as the Out-of-Order baseline but is distributed across the three sources: up to two instructions from the IQ and a combination of up to two instructions from the CRQ and DLQ, for a total issue width of four. All queues, including the IQ, apply an age-based instruction issue policy. Note that CRQ and DLQ are inherently age-based due to their FIFO nature. Among the three queues, the DLQ has the highest priority as its instructions are generally the oldest and might cause CPU stalls if its instructions reach the head of the ROB and have not issued. The IQ is second priority as it contains memory instructions. CRQ has the lowest priority as all are ready to execute and the and instructions have a very short lifetime in this queue unless there are insufficient functional units available.

**Stalls:** Issuing from the CRQ can only stall when the required functional units are busy, as all instructions in the CRQ are ready. If an instruction at the head of the DLQ is not-ready or cannot be issued, it is placed in the IQ, otherwise it proceeds directly to execute. The DLQ will stall if the IQ is full and the head entry in the DLQ is not ready. If the CRQ or DLQ are full, instructions are placed directly into the IQ. Note that as instructions can be issued from all three sources, the stalling of one is unlikely to halt execution.

### *B. Register File Pressure*

DNB does not increase the number of in-flight instructions (it does not change the ROB size) and therefore does not change the size of the register file. However, as instructions in the DLQ or CRQ that are not at the head cannot be selected for scheduling due to the FIFO nature of the queues, these instructions do not require physical register allocations. This means that virtual register renaming [22], which tracks register allocations without allocating physical space for them, could be used to *reduce* the size of the physical register file without penalty. This approach was successfully used in LTP, where the virtual register renaming allowed them to reduce the size of the physical register file to just cover the number of instructions that were eligible for scheduling in the IQ.

DNB does not increase the number of register file ports that are needed for reading or writing operands as the same number of instructions can issue in each cycle. However, the DLQ needs to check the readiness status of the source operands for the two instructions at its head to determine if they can be executed or should be moved to the IQ. This increases the number of status ports needed. The CRQ requires no such check as all instructions in it were known to have both operands ready when the instruction was dispatched.

### *C. Memory Dependency and Ordering*

In a typical out-of-order processor, load instructions wait for earlier stores to the same address or stores with unknown addresses. If the instruction scheduler executes a load before an older store with the same address, the load instruction will read the wrong data [23]. In such cases the load and all dependant instructions must be re-executed, hurting performance. Memory dependence prediction is used in this

TABLE I
THE HASWELL-LIKE [24] BASELINE MICROARCHITECTURAL PARAMETERS.

| | |
|---|---|
| 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 | 192/60/(130,130),72/42 |
| Prefetcher | enabled |
| Technology/VDD/temp | 22nm itrs-hp/0.8/360K |

TABLE II
SCHEDULING RESOURCE CONFIGURATIONS.

| **Design** | IQ Depth/Width | Other Scheduler | Issue | RF ports |
|---|---|---|---|---|
| Baseline | 64/4 | | 4 | 12 |
| LTP [3] | 32/4 | FIFO 128/4 | 4 | 12 |
| FXA [18] | 32/2 | 3-stage pipeline | 5 | 16 |
| DNB | 32/2 | FIFOs 32/2, 128/2 | 4 | 12 |

case not only to predict if issuing a load is likely to cause a memory-order violation but also to delay the issue of load(s) to avoid memory-order violation. LTP placed dependent loads in the LTP if a store instruction was parked to avoid pipeline squashes due to memory dependency violations. This has the potential of parking load instructions that are critical and reducing the performance. In DNB all loads and stores are placed in the IQ (the same queue), allowing us to use standard memory dependency analysis and recovery techniques.

## V. METHODOLOGY

We use the Multi2sim simulator [25], x86 target, with SPEC CPU2006 [26], fast-forwarding 1B instructions, cache warming for 250M, and then 1B instructions of detailed simulation. For energy modeling we use Cacti and McPAT [27], [28].

We simulate a baseline out-of-order, Haswell-like core [24] (Table I). The baseline IQ has 64 entries and 4r/4w ports. LTP, FXA, and DNB have 32-entry IQs, or half of the baseline's. The DNB RDQ has 32 entries with 2r/4w ports to support inserting 4 Critical & Ready instructions at the same time. The DLQ has 64 entries and 2r/4w. The DNB IQ has 32 entries and 2r/4w ports to support inserting 4 Critical & Non-Ready instructions at the same time. The LTP IQ has 4r/8w ports to cover burst insertion back from the LTP FIFO to the IQ and a 128-entry 4r/4w port LTP, based on the published design. We present energy (dynamic plus static) for the scheduling structures: IQ, FIFOs, and ROB. The FXA IXU is a three stage pipeline with each stage consisting of 4 integer functional units, covering a total of 12 in-flight instructions. All 12 functional units are connected for value read and forwarding. We model this fully-connected forwarding network as a CAM.

## VI. EVALUATION

Our evaluation focuses on the ability of the three designs to reduce instruction scheduling energy while maintaining performance. As such, all results are normalized to the out-of-order baseline with its twice as large (64 vs 32) IQ. In addition, we include a fourth comparison point, *BaselineHalf*, to show the effects of directly reducing the IQ by half without bypass or delay. We expect that BaselineHalf will show the worst performance of the group as it has no ability to offload instructions from the IQ.

The applications are sorted as discussed in Section II-A, and we generally expect applications with more ready instructions to benefit more from FXA and DNB and applications with more non-critical ones to benefit more from LTP and DNB. In addition to energy and performance of the chosen architectures, we analyze sensitivity to reductions in IQ depth and width. We expect DNB to be the least sensitive as it is able to offload (via bypass and delay) a greater range of instructions than the other designs.

### A. Performance

Figure 8 shows that reducing the IQ size of our baseline out-of-order processor from 64 to 32 reduces performance to 84% (BaselineHalf), while the ready/criticality-aware designs fare much better at 91% (LTP) 89% (FXA), and 95% (DNB) of the baseline performance. DNB delivers 4 and 6 percentage points better performance than LTP and FXA by offloading 32% of the instructions, vs. 22% and 24% for LTP and FXA. This is quite close to the maximum potential of offloading 39% of instructions, given that 61% are Critical & Non-Ready and should go directly into the IQ (Section III, Figure 2). Performance analysis by application category largely follows the expected behavior:

- **Ready Heavy:** show better performance with the designs that support *bypassing* of ready instructions (FXA and DNB), and the more ready instructions, the better they do. Despite having somewhat more ready instructions, `hmmer` performs worse than `h264` as it is a more memory-bound application [29], [30] and is therefore more sensitive to reductions in in-flight instructions. Several applications perform better on LTP than FXA, likely due to LTP's wider issue width which combines with FXA's lack of floating point support in the IXU.

  In all cases DNB is able to perform better due to its ability to take advantage of non-critical instructions, with its largest gains over FXA on the applications with the most non-critical instructions (e.g., `bzip2_chicken` and `bzip2_liberty`).
- **Non-Critical Heavy:** show better performance with the designs that support *delaying* non-critical instructions (LTP and DNB), and the more non-critical instructions the better they do. `calculix`, which has essentially no ready instructions, performs remarkably well on FXA as it is rather memory-bound [29], [30] and therefore benefits from the larger number of effective in-flight instructions that FXA provides (12 in the IXU plus the 32 in the IQ).

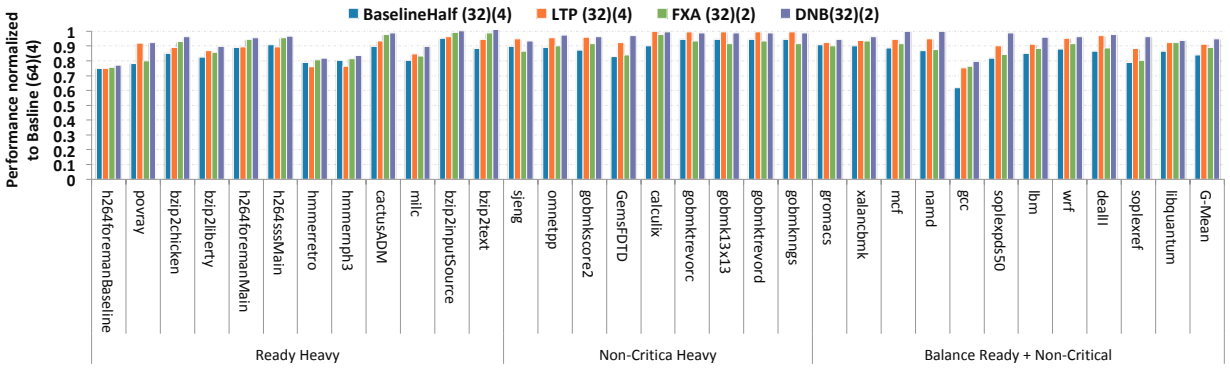


Fig. 8. IPC comparison between all designs normalized to the baseline.

- **Balance Ready + Non-Critical:** demonstrate that DNB's ability to *both bypass and delay* leads to significantly better performance than either alone. LTP performs surprisingly well on several benchmarks due to its wider IQ.

### B. Energy

The energy-saving potential of these designs comes from two sources: reducing the energy cost of each IQ access by making the IQ itself smaller and narrower and avoiding inserting instructions into the IQ via bypass. The three designs have different abilities to achieve these, as outlined in Figure 6:

- LTP only reduces IQ depth by delaying but pays for inserting all instructions into the IQ and must support a wide issue from it.
- FXA reduces IQ depth, width, and accesses by executing ready instructions early, but pays extra for trying to execute those instructions that are not ready early and for the data forwarding paths in the IXU for each instruction.
- DNB reduces IQ depth, width, and accesses, but avoids paying for inserting ready instructions into the IQ, trying to execute non-ready instructions early, and re-inserting instructions that are ready after a delay into the IQ. DNB does pay extra for delaying instructions that do not become ready during the delay, as they have to be inserted into the IQ regardless.

Overall FXA and LTP achieve scheduling energies (IQ and other queues) of 53% and 74% of the baseline, while DNB is significantly more efficient at 34% (see figure 9). In detail:

- **Ready Heavy:** For these applications LTP acts more-or-less as a narrower OoO processor as there are very few non-critical instructions for it to delay. As a result, it saves energy vs. the baseline due to its smaller IQ. FXA is able to significantly reduce the energy for the ready instructions by executing them earlier and for non-critical instructions due to its narrower IQ compared to LTP (4 vs. 2 wide). DNB does even better on Critical & Non-Ready instructions by avoiding passing them through the IXU, as can be seen by its lower energy on the for the applications with fewer ready instructions (left side). FXA's energy on ready instructions is higher than DNB's due the cost of the complex forwarding logic between each functional unit in the IXU.
- **Non-Critical Heavy:** LTP consumes significantly more energy as a large number of non-critical instructions are delayed and each delayed instruction pays both the delayed FIFO energy and the IQ energy. DNB is able to entirely avoid the IQ energy for the non-critical instructions that become ready after delay (Figure 4), which can be seen in the in four last `gobmk` applications, and reduces Critical & Non-Ready energy due to its narrower IQ (4 vs. 2 wide). FXA is able to reduce energy further than LTP on these applications due to its narrower IQ (reduced energy for Critical & Non-Ready and other instructions inserted into the IQ) but pays more energy for those same instructions as they pass through the IXU. Again, DNB spends less energy on ready instructions by avoiding the complex forwarding logic of the IXU. `calculix` is a challenge for DNB as it has very few (5%) Non-Critical & Non-Ready instructions that become ready after being delayed, resulting in essentially all delayed instructions being inserted into the IQ. As a result, DNB is only more efficient than FXA by avoiding putting non-ready instructions through the IXU.
- **Balance Ready + Non-Critical:** These applications show a mixture of the behaviors seen in the other two classes. `Libquantum` shows this behavior well as it has a large number of instructions that can be bypassed (35% Critical & Ready) and delayed (33% Non-Critical & Non-Ready), as well as having roughly 25% of its instructions become ready after being delayed. As a result DNB shows a particularly large energy advantage of 14% percentage points over FXA and 58% percentage points over LTP.

### C. IQ Depth/Width Sensitivity

All three designs are able to offload some instructions from the IQ. However, their ability to maintain performance with fewer IQ entries varies according to the instruction mix in the application, as seen in Section VI-A. Figure 10 explores the overall sensitivity of the designs to the depth of the

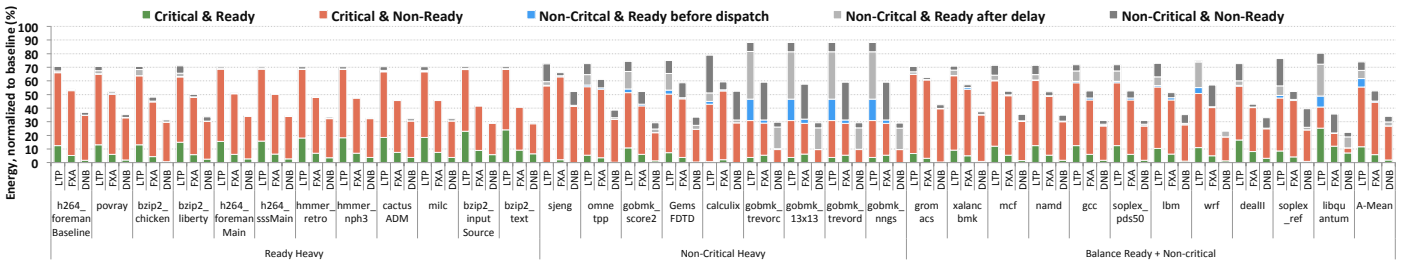


Fig. 9. Instruction scheduling energy reductions. (Not shown: naïve designs achieve 44%/46%, see Figure 5.)

IQ. As expected, the baseline OoO processor loses the most performance as its IQ depth is reduced, as it has no ability to offload instructions to other scheduling structures. DNB is the least sensitive as it has the ability to handle both instruction classes. DNB with a 64-entry IQ is 5% slower than the baseline with a 128-entry IQ, and otherwise delivers performance better than or equal to that of twice the IQ size for the other designs. DNB with a 16-entry IQ even exceeds the performance of both LTP and FXA with their proposed 32-entry IQs. LTP is less sensitive to IQ size than FXA as its offloading of long-lived non-critical instructions is more effective than bypassing of short-lived ready ones. None of LTP, FXA, nor DNB, are able to fully match the baseline OoO with a 128-entry IQ due to reduced IQ width (FXA and DNB) and reduced scheduling flexibility from delaying instructions (LTP and DNB).

The three designs differ significantly in where they can issue instructions from, which directly affect each design's ability to find the best instructions at any given time. LTP can issue 4 only from the IQ, FXA can issue 3 from the IXU and 2 from its IQ, and DNB can issue up to 2 from both the CRQ and the DLQ and 2 from its IQ. This means that LTP cannot issue any parked instructions, FXA can only use its IXU issue width for instructions that are in the IXU, and DNB can only use its CRQ and DLQ issue width for instructions that are at the head of those queues. As a result, the sensitivity to changes in IQ issue width will vary across the designs.

Figure 11 shows that LTP's sensitivity to IQ issue width follows that of the baseline, but with an offset due to the flexibility lost from not being able to issue instructions directly from the LTP FIFO. This is expected, as they both issue all instructions from their IQs and have the same baseline issue width.

FXA is less sensitive as the IXU is able to issue ready instructions regardless of the IQ width, but it suffers from not being able to delay long-lived non-critical instructions to reduce IQ pressure. DNB is able to both take advantage of the ready instructions and avoid the increased IQ pressure from long-lived non-critical instructions, resulting in significantly lower sensitivity to IQ width. At an IQ issue width of 1, DNB is only 17% slower than the 4-wide OoO baseline, while LTP and FXA architectures are 25% and 31% slower, respectively.

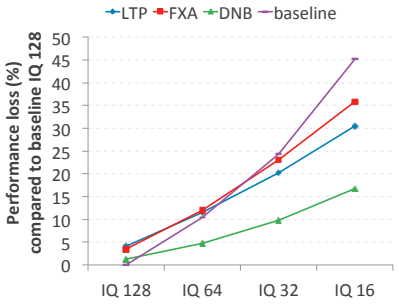


Fig. 10. Sensitivity to IQ depth, normalized to the baseline 128-entry 4-wide OoO. IQ widths and total issue widths are listed in Table II.

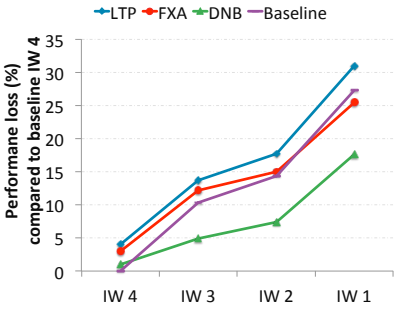


Fig. 11. Sensitivity to IQ issue width with all IQs having depth 64. See table I for more details about each design.

## VII. Related Work

### A. Ready-aware Approaches

**Front-end Execution Architecture:** Shoiya et al. [18] proposed the Front-end Execution Architecture (FXA) which inserts a three-stage in-order pipeline (IXU) with forwarding before IQ allocation (dispatch). The IXU filters out ready instructions by executing them before they reach the IQ. FXA adds dedicated register file ports for the IXU and supports issuing 3 in-order (IXU) plus 2 out-of-order (IQ/OXU) instructions per cycle. FXA suffers from the energy and delay of filtering all instructions through the IXU, even if they are not ready.

**FIFOrder:** The FIFOrder architecture, proposed by Alipour et al. [19], offloads and issues instructions from three FIFO queues covering ready, "almost-ready", and "load tail" instruc-

tions. By separating instructions into these classes they can reduce cross-FIFO stalls due to dependencies on long-latency loads. As a result, the IQ primarily sees memory instructions and they are able to maintain enough out-of-orderness to deliver good performance with an IQ issue width of 1, but are unable to reduce its depth.

### B. Criticality-aware Approaches

**Long Term Parking:** Sembrant et al. [3] proposed Long-Term-Parking (LTP) which delays inserting non-critical instructions into the IQ to reduce the IQ depth. To do so, they detect and cache instruction criticality at the rename stage with a Critical/Urgent Instruction Table (UIT/CIT) that uses iterative backwards dependency analysis [20]. Instructions marked in the CIT are placed directly into the IQ, and the rest are parked into a FIFO queue called Long-Term-Parking (LTP). When the parked instructions reach the head of ROB, they are read out from LTP and inserted into the IQ to be issued. Because instructions in the LTP are not eligible to execute, they do not need physical registers allocated. LTP leveraged this to reduce the size of the physical register file via virtual register renaming. LTP suffers from the need to insert all parked instructions into the IQ before execution, which increases IQ pressure and energy.

**Load Slice Core:** Carlson et al. [20] demonstrated that the Load-Slice-Core (LSC) can improve MLP and the performance of in-order cores. LSC constructs groups, or slices, of MLP-generating instructions that contain the address generating instructions leading up to loads and/or stores. The slices are executed out-of-order with respect to the rest of the instructions, but in-order between slices and between the remaining instructions, by placing them in two separate queues. The MLP-generating slices bypass the rest of the potentially stalled instructions via a bypass queue. Such bypassing enables LSC to extract significant MLP compared to an in-order core. However, the overall performance remains low compared to an out-of-order core, as LSC is unable to extract much ILP.

**Freeway:** Kumar et.al. [31] showed that LSC misses significant MLP opportunities due to inter-slice dependencies which cause the bypass queue to stall frequently. Their proposed design, Freeway, addresses this bottleneck by identifying and moving the dependent slices out of the bypass queue to a new FIFO queue. With dependent slices out of their way, the independent slices execute unobstructed, thus generating higher MLP. Though Freeway boosts MLP, it is still unable to extract ILP. Therefore, the overall performance stays well below that of an out-of-order core.

**Dual Issue Queue:** Moreshet et al. [17] took advantage of IQs that hold instructions after issue for replay to divide the IQ into two portions: the Main IQ (MIQ) for not-yet-issued instructions and the Replay IQ (RIQ) for instructions that have been issued, but have not yet committed, and may need to be replayed. They found that 5-55% of instructions were placed in the RIQ, including load-dependent instructions that may need to be re-issued in case of a cache miss. Both the RIQ and the MIQ are CAM-based structures, but the RIQ is only searched when a load receives its data. This allowed them to use simpler, lower-power circuits for this portion of the IQ.

**Execution Locality:** Pericas et al. [32], [33] take a related approach of categorizing instructions as cheap or expensive depending on how much out-of-orderness they require. Both the Decoupled Kilo-Instruction processor [32] and the Two-Level Load/Store Queue [33] attempt to eliminate or reduce the cost of CAM structures. They observe that programs exhibit "execution locality" which makes it possible for them to provide a large instruction window at a low cost in single- and multi-core processors.

### C. Circuit Approaches

**Half Price:** Kim et al. [14] observed that only a small subset (3-16%) of instructions need to wake-up on both operands simultaneously. To take advantage of this, they proposed the Half Price architecture which provides full-flexibility wake-up and select for instructions awaiting one or zero operands, while instructions waiting for two operands have their operand wake-up serialized, and may be slower. This reduces the wake-up circuitry overhead, but does not reduce the pressure on the IQ.

**Select Free:** Brown et al. [13] reduced the cost of instruction wake-up and select by pipelining the logic. This allowed them to provide a critical circuit for wake-up and a non-critical, potentially multi-cycle, circuit for select. They assume that all woken instructions are selected immediately for execution and thereby avoid the need for selection prioritization.

**Tag Reduction:** Ernst et al. [15] proposed a tag reduction methodology for instructions with multiple operands. They proposed a separate scheduler for instructions with zero to two operands, with reduced tag comparisons for wake-up and select. For instructions with more than two operands, they propose last tag speculation to predict the last arriving tag (latest operand) and execute based on it. This is applicable since earlier operands, even if all are ready but the last one, can not initiate the execution. Based on this observation, they keep a single tag comparator for the last arriving operand and eliminate the rest.

## VIII. CONCLUSION

In this work we have explored how to reduce the energy cost of out-of-order instruction scheduling while maintaining performance. Our approach is to both reduce the width and depth of the IQ and avoid inserting instructions into the IQ that do not benefit from its expensive scheduling. To accomplish this we apply two complementary approaches to scheduling instructions: *delaying*, to reduce IQ pressure and allow only important instructions to access the expensive IQ, and *bypassing*, to reduce IQ pressure and avoid the cost of accessing the IQ altogether.

However, to be able to maintain performance while reducing the IQ width and depth, the right instructions must be delayed and/or bypassed. To achieve this, we classify instructions based on their *criticality* and *readiness*: ready instructions do not need nor benefit from the IQ, and so they should be bypassed, while non-critical instructions are likely to block the IQ for a

long time with little benefit, and so should be delayed to make space for more important ones. We further observe that the instructions that are delayed may become ready during their delay, providing yet another opportunity for bypassing.

While these classes have been used separately to reduce scheduling costs previously, we demonstrate that *combining* them leads to significantly better performance, lower energy, and reduced sensitivity to IQ depth and width. Compared to a standard out-of-order baseline, our criticality- and readiness-aware Delay and Bypass (DNB) design is able to achieve 66% scheduler energy savings while only sacrificing 5% performance, which is significantly better than the best existing criticality-aware design (26% energy savings with 9% performance loss) and the best existing readiness-aware design (47% energy savings with 11% performance loss). Beyond this, by handling both classes of instructions, our DNB design is far less sensitive to both reductions in IQ width and depth, suggesting that it will work well for a wide range of design points.

## REFERENCES


[1] S. Sakai, T. Suenaga, R. Shioya, and H. Ando, "Rearranging random issue queue with high IPC and short delay," in *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*, 2018, pp. 123–131.

[2] H. Ando, "Performance improvement by prioritizing the issue of the instructions in unconfident branch slices," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 82–94.

[3] 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.

[4] 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.

[5] M. K. Gowan, L. L. Biro, and D. B. Jackson, "Power considerations in the design of the alpha 21264 microprocessor," in *Proceedings of the 35th Annual Design Automation Conference*, ser. DAC '98, 1998, pp. 726–731.

[6] K.-S. Hsiao and C.-H. Chen, "An efficient wakeup design for energy reduction in high-performance superscalar processors," in *Proceedings of the 2Nd Conference on Computing Frontiers*, ser. CF '05, 2005, pp. 353–360.

[7] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: Speculation control for energy reduction," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA '98, 1998, pp. 132–141.

[8] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S.-i. Mori, "A high-speed dynamic instruction scheduling scheme for superscalar processors," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 34, 2001, pp. 225–236.

[9] D. S. Henry, B. C. Kuszmaul, G. H. Loh, and R. Sami, "Circuits for wide-window superscalar processors," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, May 2000.

[10] H. Wong, V. Betz, and J. Rose, "High-performance instruction scheduling circuits for superscalar out-of-order soft processors," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 1, pp. 1:1–1:22, Jan. 2018.

[11] H. wong, V. Betz, and J. Rose, "Microarchitecture and circuits for a 200 mhz out-of-order soft processor memory system," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 1, pp. 7:1–7:22, Dec. 2016.

[13] 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.

[12] 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.

[14] I. Kim and M. H. Lipasti, "Half-price architecture," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA '03, 2003, pp. 28–38.

[15] 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.

[16] R. Canal and A. González, "A low-complexity issue logic," in *Proceedings of the 14th International Conference on Supercomputing*, ser. ICS '00, 2000, pp. 327–335.

[17] T. Moreshet and R. I. Bahar, "Power-aware issue queue design for speculative instructions," in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC '03, 2003, pp. 634–637.

[18] 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.

[19] M. Alipour, R. Kumar, S. Kaxiras, and D. Black-Shaffer, "Fiforder microarchitecture: Ready-aware instruction scheduling for ooo processors," in *Proceedings of the 25th International Symposium on Design, Automation and Test in Europe*, ser. DATE19, 2019, pp. 710–715.

[20] 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.

[21] C. B. Zilles and G. S. Sohi, "Understanding the backward slices of performance degrading instructions," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00, 2000, pp. 172–181.

[22] T. Monreal, A. González, M. Valero, J. González, and V. Viñals, "Delaying physical register allocation through virtual-physical registers," in *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 32, 1999, pp. 186–192.

[23] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA '98, 1998, pp. 142–153.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] M. Alipour, T. E. Carlson, D. Black-Schaffer, and S. Kaxiras, "Maximizing limited resources: a limit-based study and taxonomy of out-of-order commit," *Signal Processing Systems*, vol. 91, no. 3-4, pp. 379–397, 2019.

[30] A. Jaleel, "Memory characterization of workloads using instrumentation driven simulation," http://www.glue.umd.edu/ajaleel/workload, 2010.

[31] R. Kumar, M. Alipour, and D. Black-Schaffer, "Freeway: Maximizing MLP for slice-out-of-order execution," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, 2019, pp. 558–569.

[32] M. Pericas, A. Cristal, R. Gonzalez, D. A. Jimenez, and M. Valero, "A decoupled kilo-instruction processor," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, ser. HPCA '06, 2006, pp. 52–62.

[33] M. Pericàs, A. Cristal, F. J. Cazorla, R. González, A. Veidenbaum, D. A. Jiménez, and M. Valero, "A two-level load/store queue based on execution locality," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, 2008, pp. 25–36.