Reconstructing Out-of-Order Issue Queue

Ipoom Jeong*, Jiwon Lee*, Myung Kuk Yoon†, and Won Woo Ro*

*School of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea
†Department of Computer Science and Engineering, Ewha Womans University, Seoul, Korea
Email: {ipoom.jeong, jiwon.lee, wro}@yonsei.ac.kr, myungkuk.yoon@ewha.ac.kr

***Abstract*—Out-of-order cores provide high performance at the cost of energy efficiency. Dynamic scheduling is one of the major contributors to this: generating highly optimized issue schedules considering both data dependences and underlying execution resources, but relying heavily on complex wakeup and select operations of an out-of-order issue queue (IQ). For decades, researchers have proposed several complexity-effective dynamic scheduling schemes by leveraging the energy efficiency of an in-order IQ. However, they are either costly or not capable of delivering sufficient performance to substitute for a conventional wide-issue out-of-order IQ.**

In this work, we revisit two previous designs: one classical dependence-based design and the other state-of-the-art readiness-based design. We observe that they are complementary to each other, and thus their synergistic integration has the potential to be a good alternative to an out-of-order IQ. We first combine these two designs, and further analyze the main architectural bottlenecks that incur the underutilization of aggregate issue capability, thereby limiting the exploitation of instruction-level and memory-level parallelisms: 1) memory dependences not exposed by the register-based dependence analysis and 2) wide and shallow nature of dynamic dependence chains due to the long-latency memory accesses. To this end, we propose Ballerino, a novel microarchitecture that performs balanced and cache-miss-tolerable dynamic scheduling via a complementary combination of cascaded and clustered in-order IQs. Ballerino is built upon three key functionalities: 1) speculatively filtering out ready-at-dispatch instructions, 2) eliminating wasteful wakeup operations via a simple steering technique leveraging the awareness of memory dependences, and 3) reacting to program phase changes by allowing different load-dependent chains to share a single IQ while guaranteeing their out-of-order issue. The net effect is minimal scheduling energy consumption per instruction while providing comparable scheduling performance to a fully out-of-order IQ. In our analysis, Ballerino achieves comparable performance to an 8-wide out-of-order core by using twelve in-order IQs, improving core-wide energy efficiency by 20%.

***Keywords*-Dynamic Scheduling, Data Dependence, Steering**

I. Introduction

Out-of-order execution is a fundamental microarchitectural technique to achieve high single-thread performance in modern microprocessors. The essence of out-of-order execution is to parallelize the execution of instructions (i.e., instruction-level parallelism (ILP)), as well as the accesses to the memory hierarchy (i.e., memory-level parallelism (MLP)). Dynamic scheduling plays a central role in exploiting such parallelisms by deriving data dependence chains from the dynamic instruction stream and issuing ready-to-execute instructions regardless of their relative order. It is well known that dynamic scheduling generates issue schedules highly optimized for the underlying pipeline and memory hierarchy. Therefore, dynamic scheduling is suitable for wide-issue designs, providing much higher (around 2–3x) performance than in-order scheduling that has limited issue capability [1], [2].

Albeit superior performance, it has been claimed that out-of-order cores are not appropriate for energy-constrained systems due to the complexity of dynamic scheduling; wakeup and select operations cause significant complexity and energy overhead, which is further exacerbated as the scheduling window grows [3], [4], [5], [6], [7]. A promising solution is to replace an out-of-order issue queue (IQ) with multiple energy-efficient in-order IQs, expecting some degree of dynamic scheduling effects between them [2], [8], [9], [10]. However, such limited forms of dynamic scheduling are not very well suited for wide-issue superscalars because most of them focus solely on one aspect of dynamic scheduling, such as MLP, resulting in substantially low performance. We should consider various factors determining the issue order altogether to achieve comparable performance to an out-of-order IQ without the aid of power-hungry wakeup and select operations.

To address this issue, we revisit two existing microarchitectures – one *dependence-based* design and the other *readiness-based* design – and observe that they are complementary to each other, which implies that their combination is a good starting point for reconstructing out-of-order IQ. The former one, complexity-effective superscalar (CES) [3], paved the way for concurrently tracking multiple data dependence chains by steering them to parallel in-order IQs (P-IQs). However, CES requires too many P-IQs to achieve near-out-of-order performance because every ready-at-dispatch instruction should allocate a separate P-IQ. The latter one, CASINO [2], proposed a simple and effective filtering mechanism that immediately issues ready-at-dispatch instructions using a speculative in-order IQ (S-IQ). One serious drawback of this microarchitecture is that it may experience a significant performance degradation under cache misses since non-ready instructions eventually enter the single in-order IQ where they are issued in original program order. To summarize, the speculative issue functionality of CASINO could be a good solution to reduce the steering stalls caused by ready-at-dispatch instructions in CES. Conversely, CES's ability to keep track of multiple data dependences can be applied to CASINO, specifically for the non-ready (and thus not filtered out) instructions, to support out-of-order issue when they become ready.

Based on this observation, we propose Ballerino, a novel microarchitecture carrying out *BAL*anced and cache-miss-to*LER*able dynamic scheduling via cascaded and clustered *IN-O*rder IQs. Ballerino is developed in three steps. As a first step, we put together the aforementioned two scheduler designs – the S-IQ ahead of the multiple P-IQs – and analyze the architectural bottlenecks preventing the further exploitation of ILP and MLP. We identify two major bottlenecks: 1) memory dependences not considered in the existing steering mechanism reduce the effective issue bandwidth and 2) load-dependent instructions and their consumers stay inside the scheduling window for a long time, and thus still causing a lack of P-IQs while taking up only a small portion of scheduling window entries.

To address the first issue, we incorporate the awareness of memory dependences into instruction steering in Step 2. We extend memory dependence prediction (MDP) [11], [12] to maintain the steering information of producer stores. The following consumer loads are steered based on this information, overriding the existing steering mechanism. By supplementing this feature, the scheduling resources can be utilized more efficiently while expanding the scheduling window as quickly as possible. For the second issue, we introduce the concept of P-IQ sharing that facilitates a single P-IQ to concurrently handle multiple dependence chains from long-latency loads. If a new P-IQ is required but there are no empty P-IQs, a steer logic selects one of the P-IQs and activates sharing mode. In this operation mode, a P-IQ is equally partitioned and each partition acts as a distinct FIFO queue to accommodate instructions from different dependence chains while providing the opportunities for out-of-order issue. By leveraging the behavior of such dependence chains, we also propose a novel implementation that minimizes design cost and complexity. Together, the two architectural techniques augment the effective number of P-IQs and maximize their utilization.

The contributions of this paper are as follows:

- We conduct an in-depth analysis on a wide range of microarchitectures pursuing complexity-effective dynamic scheduling. We observe that although they could be promising alternatives for energy-efficient dynamic scheduling, none of them succeed in achieving sufficient performance to substitute for high-end out-of-order processors.
- We leverage the observations from two different designs – specifically, their potentially synergistic effects – to reconstruct the instruction scheduler with the aim of achieving comparable performance to the 8-wide, fully out-of-order IQ by only using in-order IQs. Two microarchitectural techniques are also proposed to mitigate the bottlenecks that arise when two designs are joined together. The proposed techniques are adaptive to different phases of execution, thereby maximizing the utilization of scheduling resources.
- We present a Ballerino core microarchitecture with its implementation details. The proposed design strikes a good balance between exploiting ILP and MLP while providing high energy efficiency. Our evaluation on a detailed cycle-level simulator demonstrates that twelve in-order IQs with some extended functionalities achieve around 98% of the performance of the out-of-order IQ. As a result, Ballerino shows 20% better energy efficiency than the baseline out-of-order core.

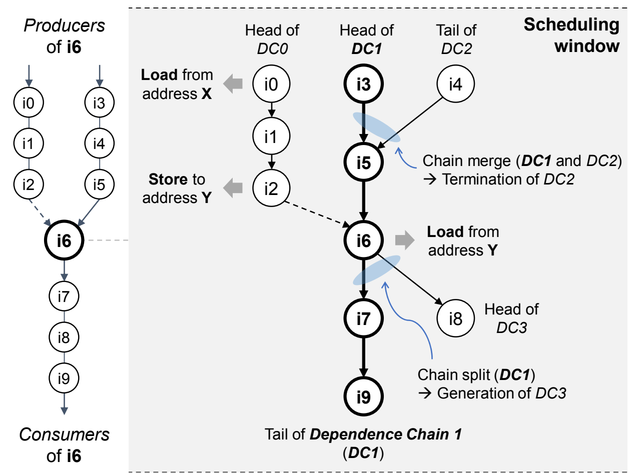


Figure 1: Definition of dependence chain. Solid and dashed arrows indicate register dependences and memory dependences, respectively.

II. Background and Motivation

In this section, we first give an overview of dynamic scheduling that lays the foundation for out-of-order execution. We then provide an in-depth analysis on two core microarchitectures pursuing energy-efficient dynamic scheduling: CES [3] and CASINO [2]. To this end, we present our key observation that the two scheduling techniques are complementary to each other, and thus their synergistic combination could be a promising alternative to a wide-issue out-of-order IQ for higher energy efficiency.

A. Dynamic Scheduling

A key design philosophy behind dynamic scheduling is to extract ILP and MLP from an instruction window by issuing instructions as quickly as possible. Data dependence is one of the factors that determine such parallelisms, since instructions can be executed only when all of their source operands are ready. There are two types of data dependences: *register dependence (R-dependence)* and *memory dependence (M-dependence)*. The R-dependence is the producer-consumer relationship via a register tag and identified at the decode of each instruction. On the other hand, the M-dependence occurs through a memory address, and so can be captured after the address calculations of both the producer store and consumer load are completed.

Figure 1 illustrates an example data dependence graph of ten dynamic instructions (*i0* – *i9*). We use the term *dependence chain (DC)* to refer to a sequence of instructions along the R-dependences (solid arrows) [3]. Within a DC, each instruction is allowed to have up to one producer and one consumer. If two destination registers are read by only one consumer, it terminates one of DCs (*chain merge* at *i5*). On the other hand, if a destination register is read by more than one consumer, new DCs are generated at that point (*chain split* at *i6*). Within a scheduling window, dependence *head* and *tail* refer to the oldest and youngest instruction of a DC, respectively. At any given moment, only dependence heads are eligible for issue. In the figure, a dashed arrow denotes the M-dependence between an older store (*i2*) and a younger load (*i6*) targeting the same address *Y*.

In modern out-of-order cores, the IQ keeps track of the R-dependences and issues ready-to-execute instructions on a cycle-by-cycle basis. As dynamic instructions are scheduled out of program order honoring the R-dependences, both ILP and MLP can be exploited by fully utilizing the available execution resources. Figure 2 shows our baseline scheduler, a unified IQ that consists of the CAM-based wakeup logic without compaction (i.e., random queue), the select logic implemented by prefix-sum circuits, and the payload RAM [4], [13], [14]. Other implementations – such as the matrix-based wakeup logic [15], [16], [17] and select logic that consists of a tree of arbiters [3], [18] – have also been published.

To support arbitration for the heterogeneous functional units (FUs), the IQ issues instructions through issue *ports*, each of which has the dedicated FUs (see Table I). The number of issue ports is equal to the issue width, and each port issues up to one instruction in a cycle. At dispatch, an issue port is assigned to each instruction, according to the opcode of an instruction and the FUs dedicated to each port. If multiple FUs of the same type exist across different ports, simply the one assigned to the least number of in-flight (dispatched but not issued) instructions is selected for load balancing. Detailed operations of instruction select in each issue port is further discussed in Section IV-E.

In Figure 2, $Port_{M-1}$ is assigned to I_0 at dispatch ①. When I_0 becomes ready, an issue request (req_I_0) is sent from the wakeup logic to the select logic (the *prefix-sum* circuit of $Port_{M-1}$) that is responsible for selecting an instruction issued in the next cycle ②. A prefix-sum circuit

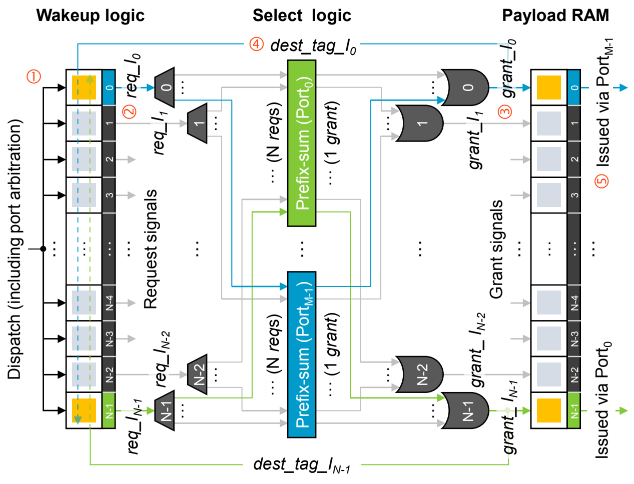
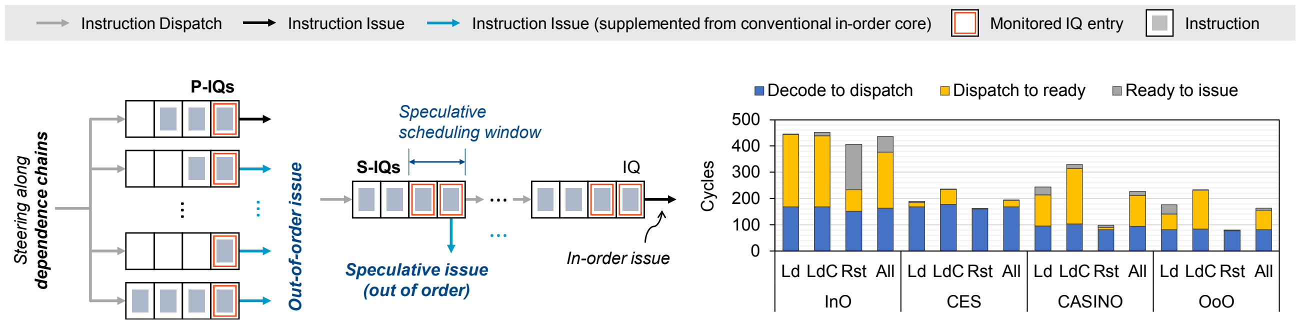


Figure 2: Organization of baseline out-of-order IQ having *N* entries with *M* issue ports

grants one of the input requests, and then sends a grant signal ($grant\_I_0$) to the payload RAM ③. At the same time, the destination register tag of I_0 ($dest\_tag\_I_0$) is broadcast to the wakeup logic to update the ready flags of the corresponding source registers ④. Finally, I_0 is issued from the payload RAM to the FU via $Port_{M-1}$ in the next cycle ⑤. The wakeup–select scheduling loop determines the critical path of the IQ and cannot be pipelined to support the back-to-back issue of dependent instructions [3], [18].

Note that, besides the R-dependences, two other factors should also be considered for better scheduling performance when selecting instructions to issue: 1) the M-dependences between instructions and 2) the ages (i.e., relative order) of them. First, a load must fetch the value from its *producer store* (i.e., the youngest older store targeting the same address), if any. If a load is issued earlier than its producer store, it would read a wrong value either from a store queue (SQ) or memory hierarchies. When this *memory order violation* is detected, the core's microarchitectural state must be recovered and the following instructions need to be re-executed. Our baseline machine performs MDP to detect the M-dependences between stores and loads [11], [12]. The key insight is that an M-dependent store-load pair is repeatedly encountered during execution, and such a load has a high probability of being ready earlier than its producer store again in the future. Therefore, once an order violation is detected, future order violations of the same pair can be avoided by delaying the issue of the M-dependent load until its (possible) producer store is issued.

Second, it is well known that the age of an instruction is heavily correlated with its criticality, and thus assigning higher issue priority to older instructions usually provides better performance [4], [14], [19], [20]. Typically, out-of-order cores have employed a compaction circuit [21] or age



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

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

matrices [16], [22], [23] to provide the capability of giving higher issue priority to the oldest instruction (i.e., oldest-first selection), at the cost of extra hardware complexity and lengthened critical path. These mechanisms are essential for wide-issue cores with large instruction windows, since memory order violations and issue port conflicts are frequently observed as more in-flight instructions are examined for out-of-order issue. The impacts of MDP and oldest-first selection are discussed in Sections III-B and VI-A, respectively.

B. Energy-Efficient Dynamic Scheduling

1) Complexity-Effective Superscalar (CES): CES [3] is a *dependence-based* microarchitecture that performs dynamic scheduling by using a group of P-IQs, as depicted in Figure 3a. The key insight of CES is that dependent instructions – instructions in the same DC – must be executed in a sequential order. Therefore, a significant amount of scheduling energy can be saved by steering each DC into a single P-IQ and examining only its dependence head (i.e., the instruction at the head of the P-IQ). Instructions in a P-IQ are issued in order but independently from the others steered to different P-IQs. Ideally, this scheduling scheme could offer comparable performance to a fully out-of-order IQ, provided that it has a number of P-IQs sufficient to accommodate all the in-flight DCs. A heuristic proposed in [3] steers an instruction to a new (empty) P-IQ in three cases: 1) none of its producers are in the P-IQs, since it is ready or some of its producers are under execution; 2) it becomes a dependence head due to chain split (e.g., *i8* in Figure 1); and 3) there are no free entries in the target P-IQs in which its producers wait for issue.

2) CASINO: CASINO [2] is built on a stall-on-use in-order core by leveraging the observation that an S-IQ(s) ahead of a conventional in-order IQ can *speculatively capture* a large amount of ready-to-execute instructions (Figure 3b). Each cycle, CASINO examines a predefined number of instructions at the head of each S-IQ (i.e., speculative scheduling window). If some ready instructions are detected, they are issued immediately while the preceding non-ready instructions are passed to the next IQ. In this manner, an S-IQ could issue dependent instructions along a DC on a cycle-by-cycle basis. If no ready instructions are detected, an S-IQ moves the speculative scheduling window toward younger instructions by passing a fixed number of instructions to the next IQ. Instructions inserted into the last IQ are issued in program order. Together, they effectively perform a restricted form of dynamic scheduling by using the S-IQ(s) as a *filter* for a conventional in-order IQ.

C. Scheduling Performance Analysis

The above two designs successfully reduce the complexity of dynamic scheduling, while achieving a certain level of performance by applying different insights into the microarchitectures: *R-dependence* and *readiness*. The R-dependence is a static feature that is determined by the relative order of instructions and their source and destination operands. On the other hand, the readiness is a dynamic feature that varies with microarchitectural factors such as execution resources and out-of-order issue capability. We observe that these two features can be synergistically integrated to compensate for the inherent drawbacks of the two microarchitectures.

Figure 3c shows the breakdown of the average decode-to-issue cycles of dynamic instructions on different microarchitectures: in-order core (InO), CES, CASINO, and out-of-order core (OoO). Instructions are subdivided into three types: load (*Ld*), load consumer (*LdC*), and the rest (*Rst*). The dependences to loads are determined at dispatch by checking whether an instruction is directly and/or indirectly dependent on any older loads that are not completed yet. The readiness of source operands are examined after the source registers are renamed. Note that we mark a load as ready only when both M/R-dependences are resolved. In this experiment, we allow up to 160 micro-operations (μops) to be in-flight between decode and issue (the sum of the allocation queue and out-of-order IQ of Skylake [24]). Our experimental methodology and microarchitectural parameters are presented in Section V.

From the results of CES, we observe that *the parallel ar-*

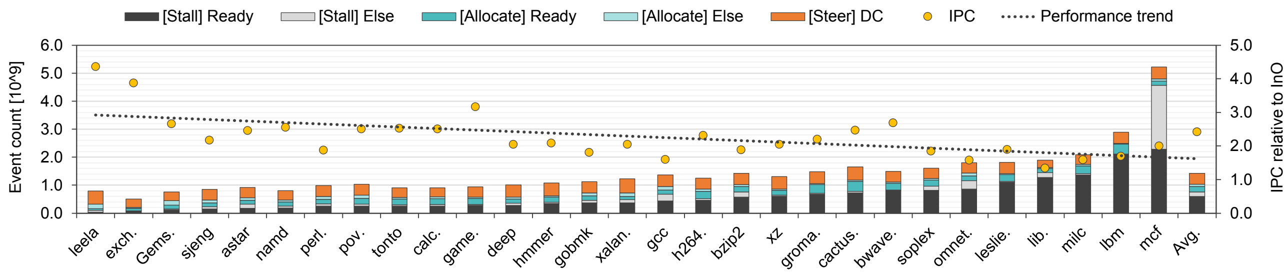


Figure 4: Breakdown of instruction steering results in CES with eight P-IQs [3]. From left to right, speedup over in-order core degrades due to increased steering stalls caused by ready-at-dispatch instructions.

rangement of P-IQs incurs significant delays from decode to dispatch. To further investigate this behavior, we measure the instruction steering statistics of CES at dispatch (Figure 4). According to our evaluation, 27% of the cases successfully steer to the P-IQs along DCs (*[Steer] DC*); the remainder either allocate new P-IQs, or stall if there are no free P-IQs. Note that most of *Allocate* and *Stall* events are caused by *Ready* instructions; they account for 72% and 79% of P-IQ allocations and steering stalls, respectively. It reveals that the major performance bottleneck of CES is a lack of free P-IQs caused by a large amount of ready-at-dispatch instructions [1], [2], [25]. This is an inherent limitation of CES, because its performance gains are degraded as the number of steering stalls increases, as shown in Figure 4 (applications are sorted from left to right according to *[Stall] Ready*). OoO has some ready-to-issue delay of *Ld*, because it frequently reaches the maximum limit of parallel memory accesses (i.e., MLP).

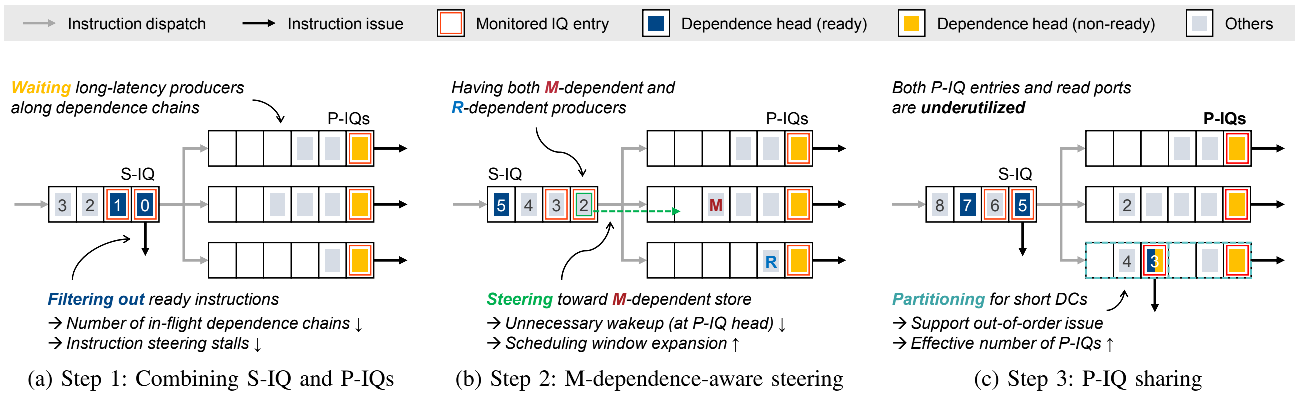
Nevertheless, dependence-aware scheduling in CES is still an attractive approach. *Once instructions are dispatched and become ready, almost all of them are issued immediately regardless of their order* (gray bars in Figure 3c). However, counter-intuitively, this mechanism restricts the ILP extraction when the number of in-flight DCs exceeds that of the P-IQs. In this situation, instruction dispatch is stalled and (possibly ready) instructions must wait for the release of the P-IQs allocated to different DCs. Such missed opportunities for issue become a critical performance limiter as more of stalled instructions are ready or dependent on prior ready instructions waiting for dispatch. One brute-force approach is pushing more P-IQs into the pipeline. However, merely increasing the number of the P-IQs is ineffective and impractical considering the growing cost and complexity of circuitry [10], [26].

CASINO exhibits remarkably different behavior. First, *the S-IQ effectively captures and issues instructions that become ready shortly after dispatch.* In Figure 3c, such instructions are classified as *Rst*, and show very small delays not only from dispatch to ready, but also from ready to issue. This implies that speculative issue functionality of CASINO would be a good solution to tackle the dispatch stall issue in CES by filtering out ready-at-dispatch instructions and their consumers before they are steered to the P-IQs. Second, even though CASINO suffers less from dispatch stalls due to the sequential arrangement of the IQs, this characteristic makes CASINO *not inherently cache-miss-tolerant.* An instruction that directly depends on a load becomes ready after the memory reference is resolved. Therefore, load-dependent instructions and their subsequent consumers (*LdC*) are likely not to be ready at dispatch. Passing through the S-IQ, they finally enter the last IQ and mixed with other instructions from different DCs. If the last IQ already gets stalled by the older DCs depending on cache-missing load(s), these instructions cannot be issued even though they become ready [9], [10].

The sequential nature of CASINO makes it vulnerable to cache misses, and thus not suitable for wider issue superscalar processors. The main reason for this is that instructions dependent on long-latency operations eventually enter the last in-order IQ, and cannot be issued out of order according to the readiness of their source operands. A dependence-based approach in CES could address this issue by buffering such instructions in the separate P-IQs, and keeping track of individual dependence heads. When one of the long-latency operations completes, instructions that belong to the corresponding DC can be issued immediately while bypassing the other instructions in different P-IQs.

III. Rebuilding Out-of-Order IQ

The speculative issue functionality of CASINO could significantly mitigate the burden on the instruction steering of CES. Conversely, the CES's capability to keep track of multiple outstanding DCs can be a good solution to address the cache miss tolerance issue in CASINO. Based on our analysis, we first combine the two IQ designs to completely eliminate the steering stalls caused by ready-at-dispatch instructions. Then, we further explore the architectural bottlenecks causing the underutilization of the aggregate issue capabilities. To this end, we propose two simple but effective steering schemes to realize the potential


Instruction dispatch
Instruction issue
Monitored IQ entry
Dependence head (ready)
Dependence head (non-ready)
Others
Waiting long-latency producers along dependence chains
P-IQs
S-IQ
Filtering out ready instructions
→ Number of in-flight dependence chains ↓
→ Instruction steering stalls ↓
Having both M-dependent and R-dependent producers
Steering toward M-dependent store
→ Unnecessary wakeup (at P-IQ head) ↓
→ Scheduling window expansion ↑
Both P-IQ entries and read ports are underutilized
Partitioning for short DCs
→ Support out-of-order issue
→ Effective number of P-IQs ↑

(a) Step 1: Combining S-IQ and P-IQs (b) Step 2: M-dependence-aware steering (c) Step 3: P-IQ sharing

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

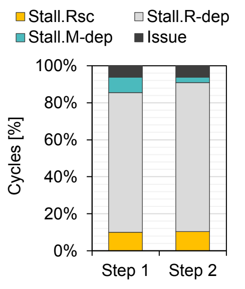
performance benefits. Throughout this section, we assume a 8-wide issue machine using eight in-order IQs following the convention in [3]. Section VI-E3 discusses the performance impact of different IQ configurations.

A. Combining Register Dependence and Readiness

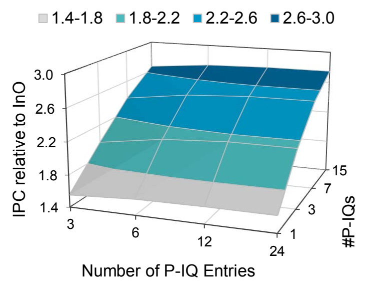
As a first step, we combine the two IQ designs, as depicted in Figure 5a. To simplify the figures, we assume that the S-IQ examines two instructions in each cycle. In front of the clustered P-IQs, a single S-IQ sequentially issues not only ready instructions but also their consumers. The effectiveness of the S-IQ is maximized when execution enters a phase that encounters multiple DCs not dependent on long-latency operations. In this situation, the S-IQ effectively filters out dynamic instructions along multiple DCs, which otherwise would require multiple P-IQ allocations. In Figure 5a, *i0* and *i1* at the head of the S-IQ are ready (i.e., belong to different DCs), and thus issued immediately. Without the S-IQ, they should allocate two separate P-IQs. Instructions not captured by the S-IQ are not ready and (possibly) dependent on long-latency operations. Such instructions are passed to the P-IQs and wait for the resolution of their data dependences. Since they are steered according to DCs, they can be issued as soon as their producers complete execution. The steering at the head of the S-IQ stalls if there are no appropriate P-IQs. More details are explained in Section IV-C.

B. M-Dependence-Aware Steering

As discussed earlier, MDP is essential for dynamic scheduling to reduce performance-critical memory order violations. According to our evaluation, MDP reduces memory order violations by 96%, resulting in an average speedup of 1.5× in the baseline. However, we observe that MDP could undermine the out-of-order issue capability of the clustered P-IQs, which has not been explored in previous studies. In CES, instructions are steered based solely on the R-dependences. Therefore, a producer store and M-dependent load are always steered to different P-IQs since



(a) Issue statistics



(b) IPC sensitivity to P-IQ in Step 2

Figure 6: Analysis on architectural bottlenecks

the dependence between them is not exposed through a register tag. Furthermore, as the younger load is likely to be ready (i.e., reaching the head of the P-IQ) earlier than its producer store, the M-dependence between them would block the issue of the load's P-IQ until the producer store is issued. The left stacked bar in Figure 6a shows the breakdown of issue cycles at the heads of the P-IQs and reveals that 9% of issue stalls are caused by M-dependent loads waiting for the issue of the producer stores.

At this point, two P-IQs are occupied by instructions along two DCs that are connected through an M-dependence; the former one ending in a producer store (e.g., *i2* in Figure 1) and the latter one starting from an M-dependent load (e.g., *i6* in Figure 1). Thus, we can create new opportunities to exploit further parallelisms by steering them into a single P-IQ and allowing the other P-IQ to be used by the following non-ready instructions. This is achieved by a simple modification to the existing steering mechanism: steering an M-dependent load along the M-dependence rather than the R-dependence (Figure 5b). Steering these two DCs into a single P-IQ does not impose a negative impact on scheduling performance for two reasons. First, it puts an M-dependent load right after its producer store, and so the M-dependent load reaches the head of the P-IQ exactly when it is eligible

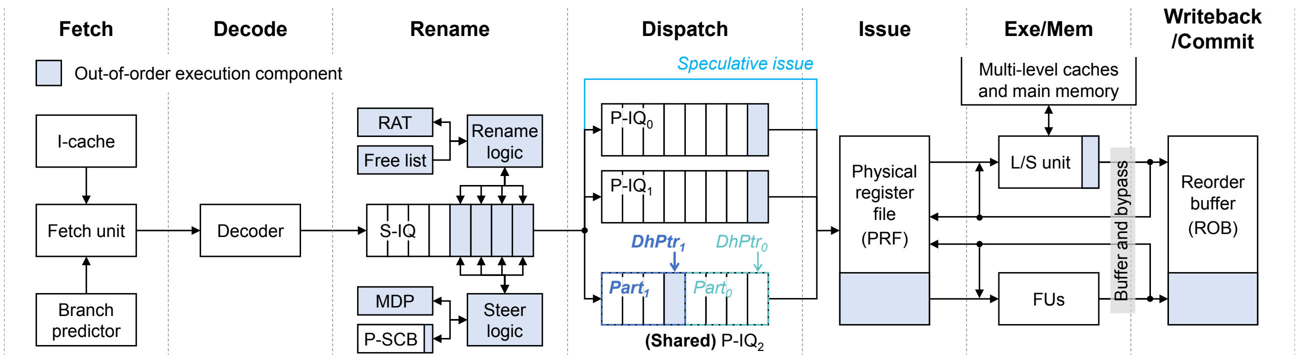


Figure 7: Ballerino core microarchitecture. Components for out-of-order execution are colored in blue. $P\text{-}IQ_2$ is shared by two DCs and second partition ($Part_1$) is activated for issue.

for issue. Second, a store always ends a DC because it does not have R-dependent consumers. Therefore, steering an M-dependent load (and its consumers) to the producer store's P-IQ does not incur the intermingling of instructions from different DCs behind the store.

C. *Tolerating Cache Misses via P-IQ Sharing*

Besides the capability of issuing ready instructions as soon as possible, another important feature of out-of-order execution is to tolerate cache misses. State-of-the-art high-end processors are equipped with multi-level cache hierarchies having various access latencies. If the memory access of a load misses in the L1 cache, it will take additional tens to hundreds of cycles to complete. In this case, instructions that read the loaded value (and their subsequent consumers) must wait for the completion of the memory access. The situation becomes more complicated when multiple parallel memory accesses hit in the different levels of cache hierarchies. Even though such memory accesses usually take a long time (see dispatch to ready cycles of *LdC* in Figure 3c), their consumers should be issued as soon as possible because they may have blocked the head of the instruction window for a long time, thereby limiting the opportunities for further ILP extraction.

One promising approach is the dependence-based scheduling of CES that keeps track of individual dependence heads waiting for loaded values, because it is complexity-effective as well as not hurting performance at least within its own scheduling window. However, various shapes and numbers of in-flight DCs would be the sources of inefficiency [2], [27]. A DC longer than the size of the P-IQ will take the secondary P-IQ (Section II-B1), resulting in unnecessary wakeup operations in the middle of the DC. On the other hand, if there are short-length DCs more than the number of the P-IQs, parallelisms are limited while some P-IQ entries are underutilized.

Rather than adding more P-IQs having large number of entries, we propose a new IQ design capable of accommodating multiple DCs in parallel, while providing them with opportunities for out-of-order issue. The proposed design is based on two key observations. First, most of the time dynamic instructions are derived from a bunch of short-length DCs, and therefore a large portion of P-IQ entries are underutilized; Figure 6b shows that the performance of the Step 2 design is very sensitive to the number of P-IQs, but not that much to the size of them. Second, with the existing steering mechanism, the actual issue operations are carried out only for a small fraction of the time, resulting in the underutilization of read ports; the right stacked bar in Figure 6a shows that a P-IQ issues instructions only for, on average, 6% of the time, mainly due to the stalls caused by dependences to long-latency loads.

Our analysis paves the way for a novel P-IQ sharing technique: *allowing a single P-IQ to be shared by multiple DCs while only one DC is selected for issue in a given cycle*. As such, the proposed technique maximizes the utilization of both the entries and read port of each P-IQ. In the best-case scenario, it provides scheduling performance comparable to two separate, ordinary P-IQs. In Figure 5c, *i3* and it's consumer *i4* are steered to the lowest P-IQ by sharing it with two older instructions, which otherwise would block the head of the S-IQ until they become ready or one of the P-IQs becomes free; in this case, the S-IQ loses opportunities for speculative issue of younger ready instructions, *i5*. When *i3* becomes ready, it can be issued bypassing the older non-ready instructions from a different DC, using its own hardware pointer. See Section IV-D for more details.

D. *Putting It Together*

The proposed instruction scheduler is adaptive to the various execution phases of each application. Ready-at-dispatch instructions are proactively filtered out by the S-IQ. The other instructions are spread to the clustered P-IQs along their M/R-dependences, which removes most of wasteful wakeup operations and leads to minimal scheduling energy consumption per instruction. If execution enters a phase that comprises multiple load-dependent DCs, the P-IQs absorb such DCs with minimal scheduling performance penalty. In the next section, we describe the implementation of the proposed microarchitecture in great detail.

IV. Ballerino Microarchitecture

A. *Overview*

The microarchitecture of Ballerino is illustrated in Figure 7. Hardware components responsible for out-of-order execution are colored in blue (i.e., area overhead over an in-order core). The S-IQ and P-IQs are the variants of a conventional in-order IQ with the speculative issue and sharing functionalities, respectively. The other pipeline structures are similar to those of the baseline out-of-order core [28], [29].

B. *Register Renaming*

Instructions are fetched, decoded, and then inserted into the S-IQ. Meanwhile, an issue port is assigned to each instruction by considering both its opcode and load balancing, similar to the baseline (Section II-A). The source and destination operands of instructions are renamed using a register alias table (RAT) and physical register free list. We assume a two-stage, pipelined register renaming [30], [31]. In the first stage (*Rename1*), RAT lookups are performed to get the current architectural-to-physical mappings of source and destination operands; destination mappings are written to a recovery log to restore the RAT when mis-speculations or exceptions are detected. Then, the R-dependences between intra-group instructions are analyzed to honor true data dependences. In *Rename2*, new mappings of destination operands (from the free list) are written to the RAT and source operand mappings are fixed by reflecting the result of an R-dependence analysis from *Rename1*.

C. *Speculative Issue and Steering*

Instruction steering is conducted in parallel to register renaming, as shown in Figure 8. For brevity, we present the steering of two instructions (with two source and one destination operand) and omit some control and datapaths. Following the RAT lookups, the steer logic examines a number of instructions (equal to the rename width) within the speculative scheduling window by referring to the corresponding entries of a physical register scoreboard (P-SCB) in *Rename2*. A P-SCB entry holds not only the readiness of each physical register but also the steering information of its producers that are not issued yet. Using this information, ready instructions send issue requests to the select logic (red arrows), while the others are steered to the P-IQs honoring the M/R-dependences (light blue arrows). If an instruction has multiple source operands whose producers are in the P-IQs, one is selected by their relative order (yellow muxes) [3]. If both M/R-dependences are detected,

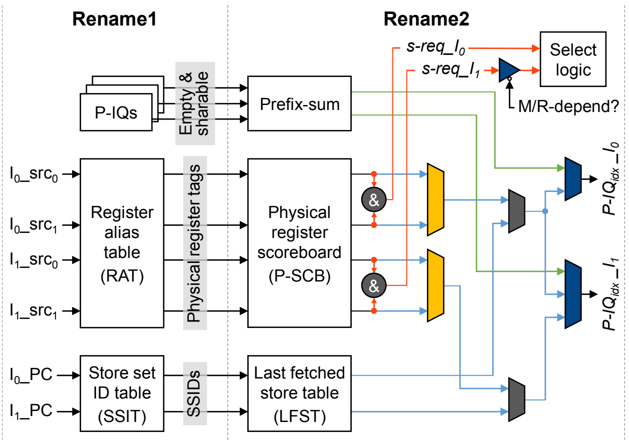


Figure 8: Instruction steering mechanism in Ballerino

the M-dependence has priority over the R-dependence (gray muxes).

An instruction is steered to a new P-IQ (either empty or sharable) as a new dependence head (green arrows) in the following three cases: 1) none of its producers are in the P-IQs, 2) the target P-IQ is full, and 3) it is ready but its issue request is not granted by the select logic due to the issue port contention. The third case may steer a ready instruction to the P-IQ, but it does not impact scheduling performance because such an instruction is again examined for issue in the next cycle at the head of the P-IQ. Intra-group M/R-dependences are used to determine the speculative issue (an enable logic) and the target P-IQ (a lower blue mux) of a younger instruction. Such M- and R-dependences are detected via SSID comparison (explained later in this section) and R-dependence analysis from *Rename1*, respectively. If any of the P-IQs cannot accommodate an incoming instruction, steering gets stalled.

To support speculative issue and steering, Ballerino maintains the states of individual physical registers: *readiness* and *producer location*. Such information is stored in each P-SCB entry. The *readiness* is presented as a 1-bit *Ready* flag. The *producer location* is encoded as a combination of an IQ_{index}, the index of the P-IQ where its producer currently resides (if any), and a 1-bit *Reserved* flag indicating whether any of its consumers has been steered to the same P-IQ. Note that producers only at the tails of the P-IQs are considered to examine the M/R-dependences; otherwise, scheduling performance would degrade due to the multiple DCs intermingled inside a P-IQ (Section II-B1). When an instruction I_p allocates a new P-IQ, the P-IQ index is written to the IQ_{index} field of its destination register R_p, and the *Reserved* flag is set to zero. Later, the consumer of R_p is steered to the same P-IQ by referring to these two fields, and setting the *Reserved* flag indicating that the producer of R_p (i.e., I_p) is not located at the tail of the P-IQ any more;

this prevents another consumer of R_p from being steered to the same P-IQ, as the chain split occurs. When I_p completes execution, the IQ_{index} and *Reserved* fields of R_p are cleared and the *Ready* flag is set.

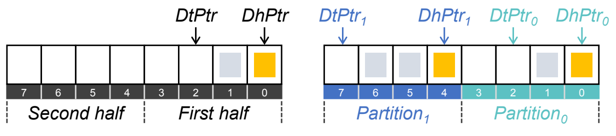
Ballerino also employs MDP to keep track of the M-dependences. As described in the original paper [11], a load has a *store set*, a set of stores on which the load has ever depended. When a memory order violation occurs, the identifier of the store set (*SSID*) is recorded in the store set ID table (SSIT) entries corresponding to the producer store and the M-dependent load. In later iterations, the producer store uses the *SSID* as an index to update the corresponding last fetched store table (LFST) entry that holds the hardware pointer of the most recently fetched (and in-flight) store from the active store set. This pointer consecutively imposes the M-dependences between the producer store and the following load and/or stores in the same store set, and serializes such memory operations to prevent potential memory order violations. If a fetched load has a valid *SSID* in the SSIT, it accesses the LFST to get the pointer of the most recently fetched store belonging to its store set; a fetched store performs the same operation except that it finally updates the LFST entry with the pointer of its own. The LFST entry is released when the store performing the most recent update to it is issued.

To support M-dependence-aware steering, we extend an LFST entry with the ability to track the *producer location*, i.e., the IQ_{index} and *Reserved* flag. Every time a store belonging to a store set is steered, it updates these two fields (instead of those in the P-SCB) as well as the hardware pointer field. Later, when the following load (or store) in the same store set is steered, this information is used to select the target P-IQ, overriding its R-dependences.

D. P-IQ Sharing

The P-IQ operates in two modes: *normal* mode and *sharing* mode. In normal mode (Figure 9a), it behaves identically to a traditional circular FIFO queue and holds instructions belong to a single DC. In sharing mode (Figure 9b), the P-IQ is divided into equal-sized partitions (two in this example) and they operate as distinct FIFO queues. To support sharing functionality, each P-IQ has additional head and tail pointers for a newly added partition ($DhPtr_1$ and $DtPtr_1$). As such, instructions from multiple DCs are accommodated in a single P-IQ. Initially, an empty P-IQ is allocated to each dependence head and operates in normal mode. When an instruction at the head of the S-IQ needs to allocate a new P-IQ but there are no empty P-IQs, the steer logic selects one of the eligible P-IQs and activates sharing mode.

To reduce implementation complexity, we impose three constraints on sharing mode. First, a P-IQ can have up to two partitions. Even though it is feasible to populate three or more partitions, it incurs additional hardware overhead (described below) as well as increasing the complexity



(a) Normal mode (b) Sharing mode (two parts)

Figure 9: Two operation modes of P-IQ

of the steer logic. Second, a P-IQ is eligible for sharing only when its head and tail pointers point to physically the same half of the queue (either first or second half in Figure 9a). Otherwise, two logical partitions might be mapped to three physical partitions (e.g., one to [2:5] and the other to [0:1] and [6:7]), which makes pointer manipulation further complicated. Note that this constraint also prevents a P-IQ from being shared if more than half of its entries are already occupied by instructions from a single DC. Third, as discussed in Section III-C, each P-IQ in sharing mode examines only one dependence head in a cycle (i.e., only one head pointer is activated). This eliminates the need for additional ports to support the out-of-order issue of instructions from different DCs. We further investigate the performance impact of the latter two constraints in Section VI-C.

In Figure 9a, the P-IQ can be selected for sharing as both of its head and tail pointers are within the first half; every time an instruction is enqueued or dequeued, each P-IQ in normal mode examines the locations of active head and tail pointers and asserts a *shareable* signal to the steer logic if they are in the same half. The steer logic refers to this signal to activate sharing mode if necessary. In Figure 9b, sharing mode is activated and $Partition_1$ serves three instructions from the second DC. At the end of each cycle, the P-IQ selects the next head pointer as follows. If the current head pointer issues an instruction, it is used again in the next cycle to enable back-to-back issue of single-cycle instructions. If not, the producer of the current dependence head would probably be a long-latency load. In this case, the P-IQ activates the other head pointer to provide opportunities for out-of-order issue to the other DC.

E. Select Instructions for Issue

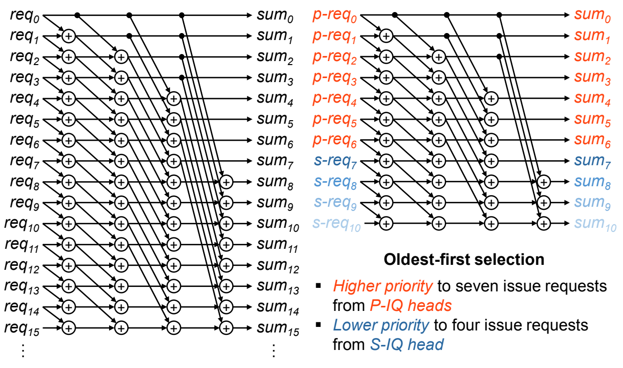
In this work, we assume the baseline select logic that comprises the prefix-sum circuits [14], as shown in Figure 10a. A prefix-sum circuit has N inputs (reqs) and N outputs (sums), where N is equal to the number of IQ entries. Each req is a 1-bit issue request from the corresponding entry in the wakeup logic, and each sum is the cumulative sum of issue requests from the first wakeup logic entry. As discussed in Section II-A, each prefix-sum circuit is tied to a specific issue port where up to one instruction is allowed to issue in a cycle. Therefore, i^{th} req is granted if it is true and $(i-1)^{th}$ sum is zero. The number of adders on the critical path is equal to $\lceil \log_2 N \rceil$.

Ballerino also uses the prefix-sum circuits as a select logic (Figure 10b), but there are two major differences. First, each prefix-sum circuit has a smaller number of inputs than its counterpart in the baseline, equivalent to the sum of the number of P-IQs (p-reqs from the P-IQ heads) and the rename width (s-reqs from the S-IQ head). As the P-IQs and S-IQ are implemented using FIFO queues where issue candidates are always positioned at the heads of the queues, the select logic does not necessarily specify the physical locations of granted instructions. Instead, it is sufficient to notify the grant(s) to each FIFO, allowing the issue of an instruction at the head (P-IQ) or instructions indexed by the relative locations from the head (S-IQ). After receiving grant signals, individual FIFOs issue instructions referring to their own head pointers. This significantly reduces the complexity of the select logic (i.e., the number of inputs and adders as well as the length of the critical path), which translates into lower power consumption.

The second difference is that the proposed select logic partially benefits from *oldest-first selection* without additional hardware overheads. Ballerino facilitates the oldest-first selection based on two key insights: 1) the relative order of instructions is already encoded in their locations (either in the P-IQs or S-IQ), and 2) the prefix-sum circuit always gives highest priority to the uppermost input. Therefore, by connecting issue requests from the P-IQs to the upper part of each prefix-sum circuit, older instructions in the P-IQs automatically have higher priority than younger ones in the S-IQ without any extra logic for tracking the ages of instructions. Note that instructions at the heads of the P-IQs are selected randomly because instruction steering is conducted without considering the relative order of instructions.

F. Execute, Writeback, and Commit

The rest of the pipeline operates very similarly to a conventional out-of-order core. Up to eight instructions are issued in each cycle, and completed instructions are committed one-by-one at the head of a reorder buffer (ROB). On a mis-speculation, instructions following the wrong path and/or consuming the wrong values are flushed from the pipeline. Such instructions are dequeued from the ROB tail one-by-one, and restore the RAT using the corresponding recovery logs. Each flushed instruction clears the P-SCB entry of its destination operand, and also clears the LFST entry if it is a store that conducts the last update to that entry. Although the *Reserved* field of each P-SCB and LFST entry is updated by its consumer, a mis-speculated consumer does not restore that field for two reasons. First, to correctly restore the *Reserved* field, each and every steered instruction must store the location of the P-SCB or LFST entry that it updates, which incurs additional hardware cost and complexity. Second, although skipping the restoration may prevent newly fetched instructions from being steered



req_0 … req_{15}
sum_0 … sum_{15}
p-req_0 … p-req_6
s-req_7 … s-req_{10}
sum_0 … sum_{10}
Oldest-first selection
Higher priority to seven issue requests from P-IQ heads
Lower priority to four issue requests from S-IQ head
$Critical\ path = \lceil \log_2 IQ_{entries} \rceil$
$Critical\ path = \lceil \log_2 11 \rceil = 4$

(a) Baseline (b) Ballerino with eight S/P-IQs

Figure 10: Prefix-sum circuit of select logic

following their DCs, it does not affect the correctness of execution. Furthermore, such entries might also be cleared if their non-speculative producers complete execution during recovery.

G. Discussions on Ballerino

1) Operating Frequency: Our primary goal is to judiciously design an energy-efficient IQ as an alternative to conventional IQs in high-end processors that consume significant energy (albeit superior performance). As the actual performance is highly correlated with the clock cycle time as well as IPC, the proposal should not increase the delay of the critical path of the core, i.e., the wakeup–select loop. Two sub-paths are changed in Ballerino: 1) accessing the P-SCB registers (Figure 8) instead of traversing the entire wakeup logic with destination tags (④ in Figure 2), and 2) selecting issue candidates among instructions at the FIFO heads (Figure 10b) rather than all in-flight ones (Figure 10a). These changes do not lengthen the critical path, so we conclude that Ballerino is able to operate at the same frequency as the baseline. We also evaluate the energy efficiency of Ballerino operating at the lower frequency (and voltage) levels in Section VI-E2.

2) Unified vs. Distributed IQ: A unified IQ has been adopted by Intel, RISC-V, and IBM POWER processors [17], [24], [32], [33]. On the other hand, AMD Zen and ARM Cortex-A processors have employed a distributed IQ where each FU has a dedicated IQ [34], [35], [36]. The former provides better capacity efficiency by allowing the IQ entries to be shared by any instructions regardless of their opcodes. However, the number of inputs to each prefix-sum circuit should be equivalent to the number of the IQ entries, which increases the complexity of the select logic. The latter facilitates the simplified select logic by reducing the number of inputs to each prefix-sum circuit, but the IQ entries cannot be fully utilized because each IQ entry

Table I: Core and Memory System Configurations

Component	InO	OoO, CES, CASINO, FXA, Ballerino
Core	**8(/4/2)-wide superscalar** @3.4(/2.5/2.0) GHz 4(/4/2)-wide decode & dispatch, 8(/4/2)-wide issue & commit	
Branch predictor	TAGE: 17-bit GHR with one bimodal and four tagged predictors (overall 32 KiB) 512 sets, 4-way set associative BTB	
MDP [11]	-	1024-entry SSIT, 7-bit SSID
Recovery penalty	8 cycles	11 cycles
Reorder logic	64(/32/16)-entry scoreboard (SCB)	224(/128/48)-entry reorder buffer (ROB)
Load queue	-	72(/48/24) entries
Store queue	16(/8/4) entries	56(/32/16) entries
Physical registers	-	180(/128/32) int 168(/96/32) fp
Functional units	4 int ALUs (P0, P1, P5, P6), 1 int DIV (P0), 1 int MUL (P1), 2 fp ADDs (P0, P1), 1 fp DIV (P0), 2 fp MULs (P0, P1) 4 AGUs (P2, P3, P4, P7), 2 branches (P0, P6)	
L1 I/D	32 KiB, 8-way, 4-cycle latency, 8 MSHRs, stride-based prefetcher	
L2 cache	256 KiB, 8-way, 12-cycle latency, 32 MSHRs	
L3 cache	1 MiB, 4-way, 42-cycle latency, 64 MSHRs	
Main memory	4 GiB, DDR4 DRAM, 2400 MT/s, 1 channel, 1 rank	

is reserved for specific instruction types. Ballerino takes advantages of both the unified and distributed designs: all the IQ entries are shared by the FUs via port arbitration at dispatch (Section IV-B), while each prefix-sum circuit has a small number of inputs by monitoring only the heads of IQs (Section IV-E). Therefore, the IQ entries can be fully utilized while the select logic is implemented with low complexity similar to the one in a distributed IQ.

3) Hardware Overhead: The overheads of implementing Ballerino (with eight S/P-IQs) over CES (with eight P-IQs) are as follows: 1) the S-IQ and P-SCB have three and six more read ports than their counterparts, respectively, 2) each P-IQ has one more pair of head and tail pointers, 3) LFST is extended to maintain steering information (64 bytes), 4) the steer logic has four more muxs to support M-dependence-aware steering, and 5) each prefix-sum circuit has three more inputs. Note that, prefix-sum circuits are also required in CES for arbitration to the heterogeneous FUs.

V. Experimental Methodology

In our experiments, we use an execution-driven, cycle-based x86 processor simulator [37]. The memory system is modeled by an integrated DDR4 DRAM simulator [38]. We run the applications from both SPEC CPU2006 and CPU2017 benchmark suites [39], [40]. For each application, we use the *reference* input set and run the most representative region of 300 million instructions chosen using the SimPoint methodology [41], after a warm-up phase of 300 million instructions. Energy consumption is estimated via modified version of McPAT [42], [43] at the 22 nm process technology. We also incorporate the modeling of the newly added structures (e.g., MDP) and control logic (e.g., steer logic) on top of an MR2 model introduced in [43].

The microarchitectural parameters and scheduling window configurations of evaluated designs are listed in Table I and Table II, respectively. We evaluate a wide range of microarchitectures that can be broadly categorized into three groups:

- Baseline: An in-order core (InO) and an out-of-order core (OoO).
- Prior work: CES [3], CASINO [2], and a front-end execution architecture (FXA) [1].
- This work: Ballerino (w/ eight S/P-IQs) and its variants.

Table II: Scheduling Window Configurations

Microarchitecture	Configuration: 8(/4/2)-wide superscalar
InO	96(/64/32)-entry **in-order IQ** [8(/4/2)r4(/4/2)w]
OoO	96(/64/32)-entry **out-of-order IQ** [8(/4/2)r4(/4/2)w]
CES [3]	8(/4/2)× 12(/16/16)-entry **P-IQ** [1r4(/4/2)w]
CASINO [2]	8(/6/4)-entry **S-IQ$_0$** [4(/3/2)r4(/3/2)w], 40(/52/-)-entry **S-IQ$_1$** [4(/3/-)r4(/3/-)w], 40(/-/-)-entry **S-IQ$_2$** [4(/-/-)r4(/-/-)w], 8(/6/28)-entry **in-order IQ** [4(/3/2)r4(/3/2)w]
FXA [1]	3-stage **IXU** [4(/4/2)r4(/4/2)w], 48(/32/16)-entry **out-of-order IQ** [4(/4/2)r4(/4/2)w]
Ballerino	8(/8/4)-entry **S-IQ** [4(/4/2)r4(/4/2)w], 7(/3/1)× 12(/16/16)-entry **P-IQ** [1r4(/4/2)w]

The baseline 8-wide out-of-order core is modeled after Skylake microarchitecture [24]. For a fair comparison, we use the same pipeline configuration across all of the evaluated microarchitectures. In addition, we set the overall number of IQ entries to the same, except for FXA and Ballerino. In FXA, we set the IQ size to the half of that of the baseline. In Ballerino, the size of the S-IQ does not need to be larger than the 2× dispatch width. In CASINO, we find the optimal combination of the S-IQ(s) and in-order IQ in size that achieves the best performance using the same number of entries as the baseline. For 4-wide and 2-wide out-of-order cores, we set the sizes of the ROB and IQ following the most widely adopted values in previous studies. Then, we perform sensitivity experiments to find the optimal parameter values of the other structures allowing less than 10% performance impact. The IQs in CASINO and FXA are configured according to the conventions in the original papers. The number of IQs in CES and Ballerino is scaled according to the issue width, while the total number of IQ entries is set to the same as the baseline.

VI. Results and Analysis

A. Ballerino Performance

Figure 11 compares the performance of Ballerino (normalized to InO) to a wide range of microarchitectures sharing the same purpose: *support dynamic scheduling with high energy efficiency.* Besides CES and CASINO, we also evaluate FXA [1] to further explore the efficacy of our proposed techniques. FXA is a readiness-based microarchitecture having an in-order execution unit (IXU) that consists of the functional units and a bypass network, in front of a common out-of-order back-end. The IXU comprises multiple pipeline stages, executing ready-at-dispatch instructions

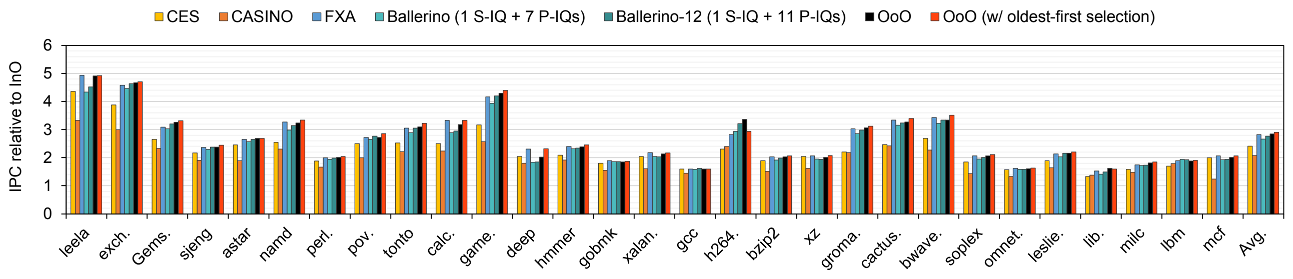


Figure 11: Performance gains of different cores with 8-wide issue capability over in-order core. Ballerino with eight and twelve S/P-IQs respectively achieve 2.7× and 2.8× speedups, which are within 7% and 2% of that of OoO.

as well as their consumers whose data dependences are resolved inside the IXU. Instructions not executed by the IXU are dispatched to the back-end and scheduled out of order by the conventional out-of-order IQ.

CES, CASINO, FXA, and Ballerino (w/ eight S/P-IQs) achieve speedups of 2.4×, 2.1×, 2.8×, and 2.7× over InO, respectively. CASINO shows a relatively low speedup since the sequential arrangement of the IQs is not well suited for a wide-issue design (further explored in Section VI-E1). Ballerino achieves significant performance improvements over both CES (11%) and CASINO (29%) by effectively filtering out ready instructions from the speculative scheduling window of the S-IQ as well as tracking the data dependences of the other instructions in the clustered P-IQs. A variant of Ballerino with four more P-IQs (Ballerino-12) delivers comparable performance to FXA. Even though the back-end of FXA provides fully out-of-order issue capability, its IQ size is set to one-half of the baseline to reduce energy consumption. Therefore, the effective size of its scheduling window is limited even considering the capacity of the IXU. In some applications, Ballerino outperforms FXA by taking advantage of the larger scheduling window that is adaptively partitioned and allocated to dynamic instructions from different DCs. Using twelve in-order IQs, Ballerino-12 achieves performance within 2% of that of OoO.

The right-most bas in each application shows the performance impact of the capability to track the criticality of individual instructions (i.e., ages) and prioritize the oldest one in each port. It can be implemented with either the compaction circuit [21] or age matrices [16]. Our simulation reveals that supplementing this feature improves the performance of OoO by 2%, assuming that it does not lengthen the clock cycle time. This result infers that, as already discussed in [4], the age-based select policy would provide no benefit in some implementations if the reduced execution cycles and the increased cycle time are not properly balanced. On the other hand, Ballerino enables the age-based select mechanism (except for among the instructions at the P-IQ heads), without increasing the critical path of the wakeup–select scheduling loop (i.e., clock cycle time).

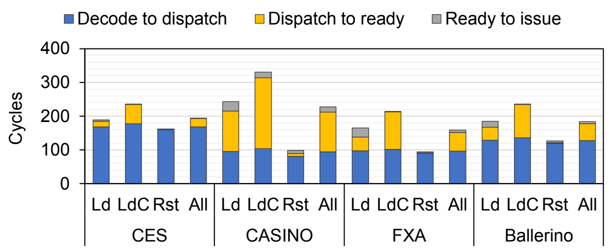


Figure 12: Scheduling performance

B. Scheduling Performance

According to Figure 12, Ballerino shows lower decode-to-issue delay compared to CES and CASINO. More specifically, Ballerino has slightly larger decode-to-dispatch delay than CASINO, which is much smaller than CES. In addition, the ready-to-issue delay of *LdC* in Ballerino is almost zero, similar to CES. As discussed in Section II-C, the ready-to-issue delay of *Ld* indicates that Ballerino reaches the maximum limit of capable MLP exploitation. Note that a load-independent instruction (*Rst*) in Ballerino experiences some delay from ready to issue, because sometimes they get stalled in the middle of the S-IQ due to the steering stall at its head. CES experiences the same situation in the allocation queue, but such instructions are not counted as ready in CES because they are not renamed yet.

C. Impact of Proposed Techniques

Figure 13 and Figure 14 demonstrate the performance impact of the proposed techniques and the breakdown of instructions issued from different IQs on the variants of Ballerino, respectively. CES has eight P-IQs that keep track of up to eight in-flight DCs concurrently. Applying M-dependence-aware (MDA) steering to CES improves performance by 4 percentage points. By substituting one P-IQ with an S-IQ (Step 1), steering stalls caused by ready-at-dispatch instructions are completely eliminated. In addition, the S-IQ can issue consecutive instructions from different DCs in a cycle. As a result, the S-IQ speculatively issues

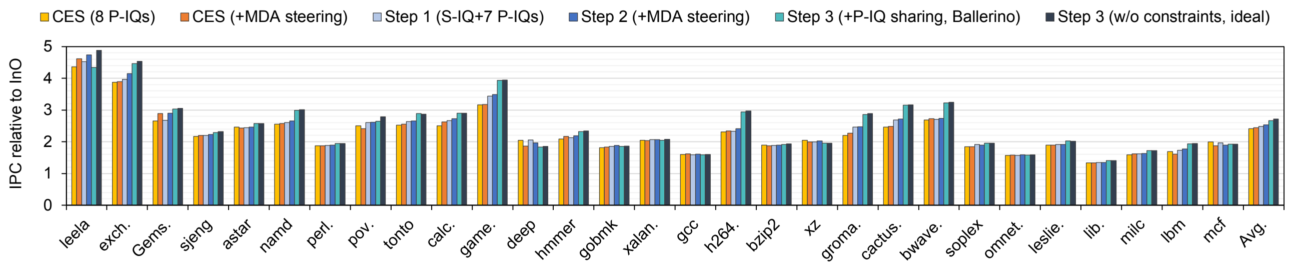


Figure 13: Performance gain over in-order core. Proposed microarchitectural techniques are applied step by step.

41% of dynamic instructions and improves performance by 7 percentage points. Nevertheless, Step 1 still suffers from a lack of free P-IQs, because 1) individual DCs starting from M-dependent loads unnecessarily allocate the P-IQs and 2) most of the P-IQs are usually occupied by the load-dependent DCs waiting for the resolution of memory accesses. We address the first issue via MDA steering in Step 2, which leads to a performance gain of 5 percentage points. Note that applying MDA steering does not change the distribution of instructions that much. This is because each MDA steering event increases the effective number of P-IQs by one only from the steering of a consumer load to the issue of the producer store. Yet, it accelerates the expansion of the scheduling window by allowing younger non-ready instructions to be steered during this period, which otherwise would block the speculative issue at the S-IQ head.

Step 3 increases the effective number of the P-IQs by allowing up to two DCs to share a single P-IQ. It improves the utilization of the entries and read ports of the P-IQs, which translates into a performance improvement of 13 percentage points over Step 2. In Step 3, a cluster of P-IQs issues 6 percentage points more instructions than Step 2, which facilitates the S-IQ to find ready instructions more aggressively. Step 3 exhibits some performance degradation in *leela* due to the increased memory order violations caused by its enlarged scheduling window. Step 3 achieves an additional performance gain of 5 percentage points assuming no constraints discussed in Section IV-D (*ideal*); P-IQ sharing is enabled regardless of the pointer locations and both partitions are able to issue ready instructions in any cycle. In other words, our final design implementation provides comparable performance to an ideal design, while leveraging the P-IQs having the same numbers of entries and ports as those of CES.

D. Energy Consumption and Efficiency

Figure 15 shows the breakdown of core-wide energy consumption for CES, CASINO, FXA, Ballerino, and Ballerino-12 normalized to that of OoO. We classify the core components into nine categories. Among them, *Schedule* includes the ROB and various types of the IQs. Note that CES

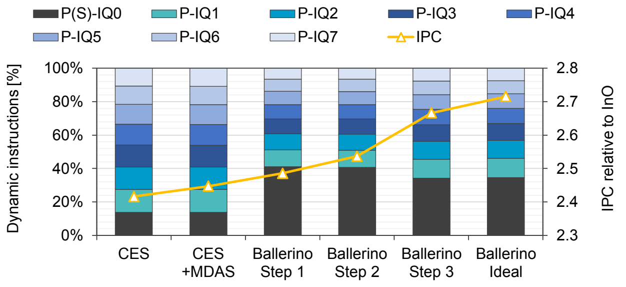


Figure 14: Distribution of instructions in Ballerino variants

and Ballerino variants consume the additional energy for *Steer* that performs P-SCB lookups, intra-group dependence check, etc. By alleviating power-hungry wakeup and select operations, CES, CASINO, FXA, Ballerino, and Ballerino-12 reduce the overall (*Schedule*) energy consumption of OoO by 24% (74%), 15% (48%), 14% (22%), 23% (68%), and 19% (62%), respectively. Even though FXA issues about half of dynamic instructions in the IXU, the out-of-order IQ at the back-end consumes significant scheduling energy, which leads to relatively high energy consumption. CASINO also dissipates higher scheduling energy than CES and Ballerino variants, because the sequential arrangement of the S-IQs fundamentally necessitates a number of read ports per S-IQ – equal to the dispatch width – to prevent the front-end from being stalled. Furthermore, non-ready instructions within the speculative scheduling window of an S-IQ are passed to the next IQ, which requires an additional copy operation. CES and Ballerino provide similar energy savings.

Figure 16 presents the energy efficiency of each microarchitecture in terms of *performance per energy*, the inverse of the energy-delay product (EDP), relative to that of OoO. As shown in the figure, Ballerino (Ballerino-12) achieves energy efficiency that is 9% (7%), 42% (39%), 5% (3%), and 22% (20%) higher than CES, CASINO, FXA, and OoO, respectively. The reason is that Ballerino yields performance close to OoO (highest) while consuming energy close to CES (lowest) via the synergistic combination of principal scheduling factors: *Readiness*, *M/R-dependences*, and *oldest-*

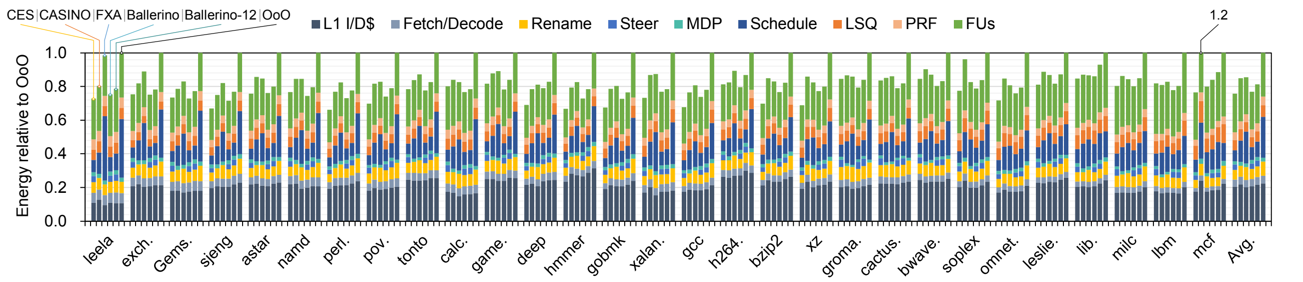


Figure 15: Energy consumption normalized to OoO. From left to right, each stacked bar indicates the core-wide energy consumption of CES, CASINO, FXA, Ballerino, Ballerino-12, and OoO.

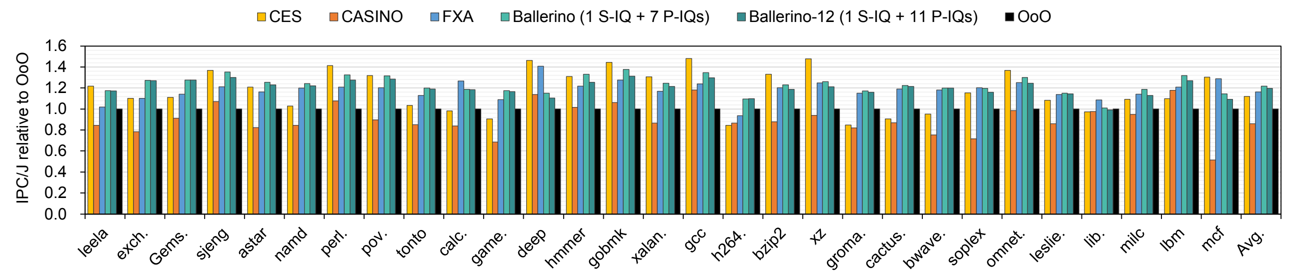


Figure 16: Energy efficiency (performance per energy) normalized to 8-wide out-of-order core

first selection. Note that CASINO shows notably poor energy efficiency in *mcf* mainly due to its inherent vulnerability to variable load latencies (Section II-C).

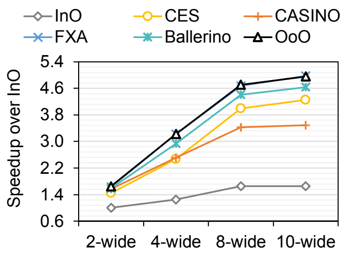
E. Sensitivity Analysis

1) Issue Width: Figure 17a shows the speedup (in terms of execution time) of each microarchitecture over 2-wide InO with respect to the issue width (see Table I and Table II). The performance of each microarchitecture increases linearly with the issue width and operating frequency. CASINO offers great speedup over InO with a 2-wide issue width, but exhibits relatively poor scalability than others. This is because it is primarily designed as an alternative to a 2-wide out-of-order core by focusing solely on one aspect of dynamic scheduling (i.e., readiness). Sequentially adding more S-IQs does not yield a dramatic speedup because 1) each S-IQ cannot precisely handle the back-to-back issue of instructions from multiple DCs and 2) instructions not speculatively issued by the S-IQs are eventually inserted into the last IQ where they are issued in original program order.

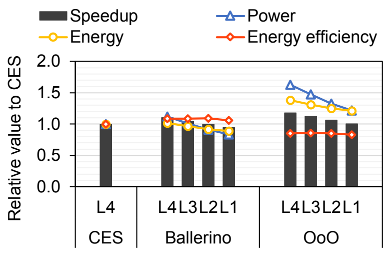
On the other hand, CES and Ballerino show good scalability since they provide functionality to keep track of the individual DCs and issue ready instructions out of program order. For all the configurations, Ballerino achieves higher performance than CES since 1) Ballerino proactively filters out ready-at-dispatch instructions using the S-IQ and 2) MDA steering and P-IQ sharing increase the effective number of the P-IQs. As a synergistic effect, Ballerino's scheduling window grows faster than that of CES, resulting in more aggressive extraction of ILP and MLP. Across all configurations, FXA achieves similar performance improvements to those of OoO by filtering out about half of dynamic instructions in the IXU and performing fully out-of-order scheduling for the rest at the back-end. Figure 17a also shows the speedups on state-of-the-art 10-wide issue designs running at 3.4 GHz [44]. Beyond the 8-wide issue width, InO and CASINO show negligible speedups since their maximum achievable ILP is not higher than 8. On the other hand, the other microarchitectures provide similar performance gains ranging from 5% to 6%.

2) Frequency and Voltage: Figure 17b shows the speedup, power, energy, and energy efficiency of Ballerino and OoO (normalized to CES) with respect to different frequency and voltage levels [45]. L4, L3, L2, and L1 denote the frequency and voltage levels of [3.4 GHz, 1.04 V], [3.2 GHz, 1.01 V], [3.0 GHz, 0.98 V], and [2.8 GHz, 0.96 V], respectively. As the level goes down, both static and dynamic power consumption decrease. If Ballerino is implemented as an efficiency core operating within the same power budget as CES, it would run at L3, exhibiting 5% higher performance and 9% higher energy efficiency. Assuming the same performance as CES, Ballerino and OoO would run at L2 and L1, providing 9% higher and 27% lower energy efficiency, respectively. Ballerino running at L4 shows 27% higher energy efficiency than OoO running at L3 while providing comparable performance.

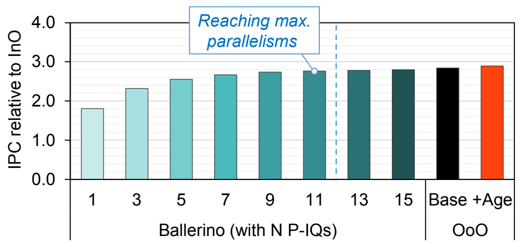
3) Configuration of P-IQs: Figure 17c demonstrates the effect of a varying number of P-IQs on the performance of



(a) Issue width



(b) Frequency and voltage



(c) Configuration of P-IQs

Figure 17: Sensitivity analysis on various hardware configurations

Ballerino. As the number of P-IQs increases, Ballerino is able to keep track of more in-flight DCs in parallel, thus ultimately resulting in better exploitation of ILP and MLP. With up to eleven P-IQs, adding a P-IQ to the back-end directly translates into performance improvement. More than eleven P-IQs, however, the performance impact becomes smaller since only a handful of further parallelisms can be exploited. Ballerino with eleven P-IQs (Ballerino-12) offers comparable performance to OoO, with additional overhead of eight more P-SCB read ports, a 1-bit lengthened P-SCB/LFST entry, and four more inputs to each prefix-sum circuit, over the one with seven P-IQs. Note that even with overall twelve S/P-IQs, the critical path of a prefix-sum circuit remains as 4 ($\lceil \log_2 15 \rceil$), and the number of inputs is slightly more than that of a distributed IQ assuming that the IQ entries are equally partitioned across issue ports.

VII. Related Work

Throughout the paper, a large body of prior work has been discussed. This section presents some other related work.

To reduce the power consumption of dynamic scheduling, hybrid scheduling schemes have been proposed. These schemes handle instructions that will not benefit from the power-hungry wakeup and select logic in a special way, to reduce the size and/or issue width of the out-of-order IQ. Such *non-critical* instructions are initially steered to the simple in-order IQ. Later, they are either pushed back to the out-of-order IQ to be scheduled with minimal wakeup and select operations [46], [47], or directly issued from the in-order IQ for energy efficiency [25], [48]. Delay and Bypass (DNB) [25] classifies dynamic instructions based on both criticality and readiness. Then, it inserts only the critical and non-ready instructions to the out-of-order IQ, while steering the others to the energy-efficient in-order IQs.

Another approach is to promote some particular instructions using the separate in-order execution engines or scheduling them via the dedicated in-order IQs. Two pass pipelining [49] uses two in-order back-end pipelines to cope with variable load latencies. Ready-at-dispatch instructions are executed in the *advance* pipeline, while the others are deferred to the *backup* pipeline. Recently, slice-out-of-order cores have been explored as a new class of cores aiming at complexity-effective dynamic scheduling on a stall-on-use in-order core [8], [9], [10]. These cores primarily focus on exploiting MLP by extracting (backward [8], [9] or forward [10]) load slices from the original instruction stream and promoting them using the separate in-order IQ(s). These restricted forms of out-of-order execution prevent a conventional in-order pipeline from stalling due to the long-latency memory accesses (i.e., cache misses) by triggering such memory accesses earlier or putting aside the instructions dependent on such memory accesses.

Replay-based scheduling schemes leverage the repetitive nature of instruction issue schedules [50], [51], [52], [53], [54]. Such scheduling schemes memoize the issue order of dynamic instructions on the out-of-order pipeline, and replay the same schedules in the energy-efficient in-order pipeline in future iterations. Ideally, they could achieve near-out-of-order performance, but must be equipped with the out-of-order IQ as well as handling mechanisms for unexpected microarchitectural events such as mis-speculations during replay phases.

Some recent work leverages both criticality and repetitiveness of dynamic instructions [4], [55]. Ando proposed to prioritize the issue of instructions in unconfident branch slices to reduce the mis-speculation penalty [4]. Some of the out-of-order IQ entries are reserved for such sliced instructions, and they are prioritized over the other instructions when they are granted by the select logic. This effectively expedites the resolution of (possibly) mispredicted branches and subsequent recovery process. Diavastos and Carlson proposed an efficient dynamic scheduling built upon a precise load delay tracking scheme [55]. They adopted systolic priority queues (SPQs) [56], and arranged them in parallel to facilitate out-of-order scheduling with low complexity. Dispatched instructions are steered to one of the SPQs by considering opcodes, data dependences, and load balancing. Each SPQ reorders instructions according to the predicted issue time, and only an instruction at the head is eligible for issue. Ballerino conducts out-of-order scheduling by dynamically examining the readiness, M/R-dependences, and ages of instructions by only using in-order

IQs. Therefore, neither the complex out-of-order IQs nor issue time predictions are required.

VIII. Conclusion

In this work, we propose a novel microarchitecture named Ballerino, built upon three key principles that drive dynamic scheduling: *readiness*, *M/R-dependences*, and *oldest-first selection*. Starting from a combination of two compatible microarchitecture designs, two simple and effective architectural techniques are supplemented to enable a group of the in-order IQs to generate highly optimized issue schedules comparable to that of the fully out-of-order IQ while consuming much less energy. Ballerino first filters out the ready-at-dispatch instructions and their consumers using the S-IQ. Then, the other instructions are partitioned into multiple DCs honoring both M/R-dependences, and scheduled by the individual P-IQs. With a simple modification, we facilitate P-IQ sharing that allows multiple short-length DCs to share a single P-IQ, being provided with the opportunities for out-of-order issue. The proposed design partially benefits from the oldest-first selection by leveraging the relative order of instructions encoded in their locations in the scheduler. As a synergistic effect, Ballerino with twelve S/P-IQs achieves performance comparable to the 8-wide out-of-order core while consuming 19% less energy, which leads to a core-wide energy efficiency improvement of 20%.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments that helped to improve the quality of the paper. This work was supported in part by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-00853, Developing Software Platform for Programming of PIM), in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1G1A1092196), and in part by Samsung Electronics Company, Ltd., Hwaseong, Korea. W. W. Ro is the corresponding author.

References

[1] 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*, 2014, pp. 419–431.

[2] I. Jeong, S. Park, C. Lee, and W. W. Ro, "Casino core microarchitecture: Generating out-of-order schedules using cascaded in-order scheduling windows," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 383–396.

[3] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997, pp. 206–218.

[4] H. Ando, "Performance improvement by prioritizing the issue of the instructions in unconfident branch slices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 82–94.

[5] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, 2003, p. 93.

[6] D. Folegnani and A. González, "Energy-effective issue logic," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001, pp. 230–239.

[7] N. H. Weste and D. Harris, *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.

[8] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 272–284.

[9] R. Kumar, M. Alipour, and D. Black-Schaffer, "Freeway: Maximizing mlp for slice-out-of-order execution," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 558–569.

[10] K. Lakshminarasimhan, A. Naithani, J. Feliu, and L. Eeckhout, "The forward slice core microarchitecture," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 361–372.

[11] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 142–153, 1998.

[12] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 1997, pp. 235–245.

[13] J. Stark, M. D. Brown, and Y. N. Patt, "On pipelining dynamic instruction scheduling logic," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, 2000, pp. 57–66.

[14] K. Yamaguchi, Y. Kora, and H. Ando, "Evaluation of issue queue delay: Banking tag ram and identifying correct critical path," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*. IEEE, 2011, pp. 313–319.

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

[16] P. G. Sassone, J. Rupley, E. Brekelbaum, G. H. Loh, and B. Black, "Matrix scheduler reloaded," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 335–346, 2007.

[17] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue *et al.*, "An open source fpga-optimized out-of-order risc-v soft processor," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 63–71.

[18] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Quantifying the complexity of superscalar processors," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1996.

[19] A. Buyuktosunoglu, A. El-Moursy, and D. H. Albonesi, "An oldest-first selection logic implementation for non-compacting issue queues [microprocessor power reduction]," in *15th Annual IEEE International ASIC/SOC Conference*. IEEE, 2002, pp. 31–35.

[20] T. Liu and S.-L. Lu, "Performance improvement with circuit-level speculation," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*. IEEE, 2000, pp. 348–355.

[21] J. A. Farrell and T. C. Fischer, "Issue logic for a 600-mhz out-of-order execution microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 5, pp. 707–712, 1998.

[22] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell, L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix, R. Gammack, V. Germini *et al.*, "Design of an 8-wide superscalar risc microprocessor with simultaneous multithreading," in *2002 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No. 02CH37315)*, vol. 1. IEEE, 2002, pp. 334–472.

[23] M. Golden, S. Arekapudi, and J. Vinh, "40-entry unified out-of-order scheduler and integer execution unit for the amd bulldozer x86–64 core," in *2011 IEEE International Solid-State Circuits Conference*. IEEE, 2011, pp. 80–82.

[24] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: New microarchitecture code-named skylake," *IEEE Micro*, vol. 37, no. 2, pp. 52–62, 2017.

[25] M. Alipour, S. Kaxiras, D. Black-Schaffer, and R. Kumar, "Delay and bypass: Ready and criticality aware instruction scheduling in out-of-order processors," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 424–434.

[26] J.-M. Parcerisa and A. González, "Reducing wire delay penalty through value prediction," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 317–326.

[27] P. Salverda and C. Zilles, "Dependence-based scheduling revisited: A tale of two baselines," in *6th Annual Workshop on Duplicating, Deconstructing, and Debunking*. Citeseer, 2007.

[28] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[29] J. E. Thornton, *Design of a computer—the control data 6600*. Scott Foresman & Co, 1970.

[30] V. Petric, T. Sha, and A. Roth, "Reno: a rename-based instruction optimizer," in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 98–109.

[31] E. Safi, A. Moshovos, and A. Veneris, "Two-stage, pipelined register renaming," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 10, pp. 1926–1931, 2010.

[32] B. Bowhill, B. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, O. Mendoza, C. Morganti, C. Houghton, D. Krueger, O. Franza *et al.*, "The xeon® processor e5-2600 v3: A 22 nm 18-core product family," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 92–104, 2015.

[33] B. Sinharoy, J. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. E. Moreira *et al.*, "Ibm power8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 2–1, 2015.

[34] D. Suggs, M. Subramony, and D. Bouvier, "The amd "zen 2" processor," *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.

[35] M. Evers, L. Barnes, and M. Clark, "The amd next-generation "zen 3" core," *IEEE Micro*, vol. 42, no. 3, pp. 7–12, 2022.

[36] A. ARM, "Cortex®-a72 mpcore processor technical reference manual (revision r0p2)."

[37] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, " Multi2Sim: A Simulation Framework for CPU-GPU Computing ," in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[38] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2016.

[39] C. D. Spradling, "SPEC CPU2006 Benchmark Tools," *SIGARCH Computer Architecture News*, vol. 35, March 2007.

[40] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[41] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 45–57.

[42] 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 ACM/IEEE International Symposium on Microarchitecture*, 2009, pp. 469–480.

[43] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in mcpat and potential impacts on architectural studies," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 577–589.

[44] "Ice lake (client) - microarchitectures - intel." [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/ice_lake_(client)

[45] J. C. Wright, C. Schmidt, B. Keller, D. P. Dabbelt, J. Kwak, V. Iyer, N. Mehta, P.-F. Chiu, S. Bailey, K. Asanović *et al.*, "A dual-core risc-v vector processor with on-chip fine-grain power management in 28-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 12, pp. 2721–2725, 2020.

[46] 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*, 2015, pp. 334–346.

[47] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. USA: IEEE Computer Society, 2002, p. 59–70.

[48] M. Alipour, R. Kumar, S. Kaxiras, and D. Black-Schaffer, "Fiforder microarchitecture: Ready-aware instruction scheduling for ooo processors," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 716–721.

[49] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu, "Beating in-order stalls with "flea-flicker" two-pass pipelining," in *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, 2003, pp. 387–.

[50] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?" in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[51] R. Nair and M. E. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 13–25.

[52] E. Talpes and D. Marculescu, "Execution cache-based microarchitecture for power-efficient superscalar processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 1, pp. 14–26, 2005.

[53] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, "Dynamos: dynamic schedule migration for heterogeneous cores," in *Proceedings of the 48th Annual ACM/IEEE International Symposium on Microarchitecture*, 2015, pp. 322–333.

[54] C. Fallin, C. Wilkerson, and O. Mutlu, "The heterogeneous block architecture," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, 2014, pp. 386–393.

[55] A. Diavastos and T. E. Carlson, "Efficient instruction scheduling using real-time load delay tracking," *arXiv preprint arXiv:2109.03112*, 2021.

[56] C. E. Leiserson, "Systolic priority queues." CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1979.