Investigating ROB Scaling in Out-Of-Order Processors

Ryan Wang Eliot Yoon Steve Zang
Department of Computer Science
University of California, Los Angeles
Los Angeles, USA
{ryanwang25, eyoon1131, zangbruin007}@ucla.edu

***Abstract*—Traditional out-of-order processors have kept the reorder buffer to a modest size due to complexity, power, and latency constraints. However, recent designs such as the Apple M series chips are rumored to have very large reorder buffers, raising questions of how to leverage the benefits of a large scheduling scope without incurring diminishing returns.**

In this work, we evaluate the effect of scaling ROB size on performance. We also create our own re-implemention of a waiting instruction buffer (WIB), a structure which alleviates ROB pressure from long latency dependent instruction chains by moving them to a larger buffer until the long-latency instruction completes. Performance is evaluated in gem5 on the SPEC 2017 benchmark across a variety of WIB sizes and latency thresholds for a long-latency dependence instruction chain to be moved to the WIB.

The key observation we find is that only scaling ROB size does little to improve IPC across workloads, only allowing more dependent instructions to accumulate behind a long latency operation. We further investigate the effect of a waiting instruction buffer as a mechanism to alleviate this issue. We find that while the WIB alleviates pressure from the reservation station, this does not translate to significant speedups over the default ARM OOO core on the SPEC 2017 benchmark. Our results suggest that the limitation is not just instruction window capacity but also WIB selection logic for which instructions stay in the instruction window.

I. Introduction

The evolution from in-order to out-of-order processors, first introduced in Tomasulo's algorithm [1], has been foundational to the realization and exploitation of instruction-level parallelism latent within programs. Fundamentally, out of order execution works by considering all instructions in a finite instruction window for execution, allowing independent instructions to be executed out of order. Compared to an in-order processor, where only the next instruction can be considered for execution and must stall if its operands are dependent on a previous instruction awaiting completion, out-of-order can schedule any instruction within its instruction window, increasing the chance of finding independent instructions to execute while others are stalled. In modern processors, the instruction window is implemented using multiple structures, the reservation station (also referred to as an instruction queue) and the reorder buffer (ROB). The ROB keeps track of the order in flight instructions should be committed. The bigger your reorder buffer, the more instructions can be in flight at once. When there is a long latency instruction dependent instructions can quickly fill up the ROB preventing further progress even if there are open execution units and independent instructions further downstream. However, if the ROB is large enough, it can tolerate this window pressure allowing forward progress whereas a smaller ROB would have to stall until current in flight instructions are committed and cleared from the ROB. Therefore, a theoretical infinite instruction window can find every independent instruction in the program at any given cycle, thus revealing all possible ILP.

In practice, increasing the size of the ROB comes with practical trade offs. Because of the difference in latency scale where instruction fetches take a single cycle compared to a long latency load which may take thousands of cycles for a read from main memory, to tolerate a single additional long latency load the ROB would need to be able to hold up to thousands of additional instructions to fully tolerate the pressure. Moreover, scaling ROB introduces fundamental hardware challenges. To realize the full benefit of scaling ROB, the reservation station must scale with ROB size. Because reservation stations are implemented in hardware using CAMs, larger reservation stations will increase energy and latency overheads associated with scaling CAMs [2]. Furthermore, the growth of complexity with regard to wiring the wakeup and selection logic relative to ROB size is a critical bottleneck [3]. Importantly, there are also fundamental limits to the amount of ILP any given program has, due to true dependencies. Prior work on the limits of instruction-level parallelism (ILP) consistently finds that achievable speedups are modest, with most studies reporting bounds in the range of 2–3x under realistic assumptions [4]–[11]. This poses some theoretical limits to the scaling of performance with ROB size.

Recent high-performance processors like the Apple M series chips are widely believed to utilize ROBs which are significantly larger than traditional processors, indicating a renewed interest in wide instruction window microarchitectures. However, it is unclear whether wide-instruction buffers alone account for the performance improvements seen in their processors or if other mechanisms are used with large ROBs to maximize their performance.

In this work we investigate the effectiveness of scaling ROB size. We also explore a promising complimentary mechanism, the waiting instruction buffer and evaluate its impact. In doing so, we evaluate several aspects of WIB design, from selection

logic of long-latency instruction chains to the WIB and the size of the WIB. We implement this design in gem5, a cycle level simulator for evaluating the performance of different micro-architectures.

Our first key observation is that naively scaling ROB size does little to improve performance as measured by IPC and simulated time. Our second key observation is that with a WIB as an auxiliary structure, scaling ROB marginally improves performance although limitations to the implementation of the WIB limit its ability to improve performance further. We confirm that across most evaluations utilizing a WIB, the number of events when the instruction queue is full decreases very slightly. However, we also observe that many instructions are being exempted. The number of exempted instructions is very high compared to the amount of IQFull events being mitigated, suggesting the WIB is often failing to do useful work, moving dependent instructions when the IQ is under low pressure but not moving enough instructions to meaningfully prevent long-latency dependency chains from filling the IQ. This ties to our other observation which is that the latency threshold to be considered long-latency had very little impact on performance, likely because the most important instructions to be exempted, such as the ones where the load must be completed from main memory, were of much higher latency than even the thresholds we tested. Finally, because the current WIB is intentionally kept simple there are limitations in its ability to alleviate window pressure. Because it can only exempt one dependent instruction, if another instruction is dependent on this instruction it will sit in the IQ and block progress. Furthermore, because it can't exempt dependent load instructions, a chain of two or more long latency loads will block progress, limiting its applicability to pointer chasing workloads.

II. Background

The out-of-order processor originates with Tomasulo's algorithm [1]. In Tomasulo's algorithm, a hardware structure known as a reservation station is introduced to keep track of a finite window of fetched instructions and their source operand register tags and destination tags. The reservation station is implemented using content-addressable memory (CAM) like structures and determines operand readiness by parallel tag comparison across all entries. However, in order to handle interrupts efficiently a notion of precise state was needed. Therefore, the reorder buffer was introduced which keeps track of all in flight instructions and their program order. Specifically, the reorder buffer keeps a head pointer which points to the next instruction to be committed and a tail pointer which points to the youngest instruction. Every instruction in the ROB can complete independently and write their results to the ROB. However, only the completed instructions at the head be committed, ensuring instructions are committed in-order. This enables precise state because it guarantees all committed instructions have been written to the register file in-order,giving the notion that the precise state of the program is at the last committed instruction, while allowing instructions to complete and their results used out of order.

A. Behavior Under Long-Latency Operations

When a long-latency operation such as a cache miss occurs, all younger instructions in the ROB must wait for this operation to complete and retire, even if all younger instructions are complete. Also, all instructions in the reservation station dependent on the long-latency instruction's result (directly or indirectly through dependence on a instruction dependent on the long latency instruction) will stay in the reservation station until this operation is complete. If one of these structures reaches capacity, the entire pipeline must stall until there is space, meaning long-latency operations cause severe head of line blocking issues. Importantly, downstream there might be completely independent instructions that can't be fetched and executed.

B. Effective Instruction Window Utilization

Importantly, long-latency dependence chains reveal that not all instructions contribute equally to forward progress. As explored above, instructions dependent on a long latency instruction take up window space without hope of execution in the near future, effectively reducing the window size. This motivates smarter strategies to populate the instruction window with instructions likely to make progress or augment the instruction window by delaying instructions that can't make progress.

III. Design

A. Large Reorder Buffer (ROB)

In order to explore the effects of increasing ROB size, we simply modified the `numROBEntries` parameter in the configuration file. We tested a range of ROB sizes from 96 to 512 entries. This testing will be explored in further detail in later sections.

B. Waiting Instruction Buffer (WIB)

The idea of decoupling stalled instructions from the active instruction window is not new. Ideas such as runahead execution [12], have explored different mechanisms for bypassing or exempting long latency instructions and their dependent instructions. Other designs have proposed moving such instructions to a separate buffer, often referred to as a waiting instruction buffer (WIB). The purpose of the WIB is to remove instructions from the issue queue that depend on a previously issued long-latency instruction, such as a load instruction that missed in the cache and requires retrieval from main memory. These removed instructions are stored in the WIB and reinserted back into the instruction queue once they are ready for execution, thus freeing up space in the IQ while these instructions wait. In order to keep the gem5 modifications as minimally invasive as possible, we refrained from actually implementing a new data structure to store the long-latency dependent instructions or moving instructions back and forth between buffers.

Instead, we implemented a pseudo-WIB policy within the O3 CPU: instructions that are dependent on a long-latency load are marked as being exempt from the IQ, but they are not removed. Thus, the instruction is added to and executed in the IQ as it normally would, except that any exempted instruction is not counted towards the number of entries in the IQ. Thus, the issue-execute-writeback (IEW) pipeline is able to continue issuing instructions into the IQ, even if the IQ already contains the maximum number of entries. This models the idea of moving the instruction to the WIB and back to the IQ when it is ready. The instruction remains present in the IQ but is essentially hidden, and when it is ready, it executes and is removed from the IQ. Although this policy does not match the behavior of the WIB exactly, in theory it should result in fewer stalls due to a full IQ. Additionally, in the normal O3 CPU if an instruction is added to the IQ and makes it full, the next instruction will see the full IQ and stall. However, in the pseudo-WIB implementation if the IQ becomes full, a subsequent instruction that is long-latency dependent can be marked as exempt and still issued.

In the initial implementation of the WIB, we did not consider the size of the WIB as a factor, and essentially treated the WIB as infinite. In order to make our results more realistic, we limited the number of exempted instructions that can be issued concurrently to the size of the WIB, simulating a WIB size cap.

C. Code Implementation

In order to mark an instruction as exempt, we added a flag `exemptFromIQ` to each instruction in the `DynInst` class. This flag is false by default, and is only set in the `dispatchInsts()` function in the `IEW` class, which decides where and how to dispatch an instruction based on its type. To keep the dispatch logic consistent with the standard O3 CPU, the structure of this function was kept the same. The standard O3 CPU checks if the IQ is full, stalling the entire pipeline if it is and not dispatching any further instructions. In our implementation, this is only slightly modified so that prior to the IQ size check, the instruction is examined by the function `shouldExemptFromIQ()` to see if it should be exempted, the details of which will be explained shortly. If the instruction is ineligible to be exempted, it is treated as a normal instruction and follows the standard decision path, which in the case of a full IQ, leads to a stall. If the instruction should be exempted, however, the IQ size check is skipped and it is marked as exempt.

The `shouldExemptFromIQ()` function first checks the type of the instruction. If it is a load or store, barrier, atomic, nop, or non-speculative, it returns false. The purpose of this decision will be explained shortly. After this check, it then checks if the instruction depends on a long-latency load. The function examines the source registers of the instruction and looks to see if the registers are waiting on a pending load. In order to track the pending loads, a map from registers to load information (dispatch tick and source instruction) is kept. If the register is dependent on a pending load, it then checks how long it has been waiting by computing the difference between the dispatch tick and the current tick and converting to cycles. If the wait time exceeds the threshold, the instruction is marked as exempt.

The reason for excluding load and store instructions from exemption is to avoid modifying the standard dispatch logic any more than it needs to be modified. In the standard logic, special cases exist that check if the instruction is a load or a store, such as a case that first checks the LSQ and blocks if the LSQ is full. Allowing for loads and stores to be exempted would require modification of this case logic beyond the simple exemption check.

When an instruction is marked as exempt, the instruction count and number of free entries is not modified in the `insert()` function in the `IQUnit` class. Correspondingly, the `remove()` function also does not modify any of these values for exempt instructions.

To implement the WIB size logic, a field was added to the `InstructionQueue` class that tracks the number of concurrently exempted instructions in the IQ. This field is updated in the `insert()` and `remove()` functions if the instruction is exempt. This value is then used in the `dispatchInsts()` function to check if the WIB occupancy is full based on the `maxWIBEntries` field that is set by a parameter passed in the configuration, and if the WIB is full the instruction is marked as not exempt and is handled by the standard logic for non-exempt instructions.

D. Limitations

As our WIB implementation is designed to be as minimally invasive as possible, it has limitations that restrict its ability to improve performance, which will be further explained in the evaluation.

One of the most significant limitations is the fact that loads and stores are not exemptable in our current design. This means that, for example, a load that depends on a long-latency load is not moved to the WIB and is kept in the IQ. Similarly, indirect dependencies are not tracked. For example, an add instruction that depends on the result of another add instruction that depends on a long-latency load would not be treated as a long-latency dependent instruction, even though indirectly it depends on the long-latency load. This can become a significant issue, especially if multiple instructions form a chain that all depend on a single very long-latency instruction to be executed.

Another limitation lies in how instructions are handled when the IQ is full. The standard pipeline simply stalls if the IQ becomes full and a new instruction is attempted to be dispatched. With the WIB, an exempt instruction can still be issued if it is dispatched directly after the WIB becomes full. However, if the next instruction right after the IQ becomes full is not exemptable, the pipeline will follow the standard logic and stall the whole pipeline. Thus, even if there are many exemptable instructions right after this one, they will not be seen until the pipeline is not stalled.

Lastly, if an instruction is marked exempt but is ready very quickly after it is put in the WIB, there is little performance gain to the IQ. For example, if the latency threshold is 20 cycles and an instruction relies on a load that began 20 cycles ago but only takes 25 cycles to complete, once the instruction is actually exempted (20 cycles after the load began), it will only take 5 cycles to be ready. Thus, it will only be removed from the IQ for 5 cycles, meaning that the IQ does not see much benefit. Although not necessarily detrimental to overall performance, if the latency threshold is too low and there are many instructions like these that are ready very quickly after being exempted, the benefit to IQ pressure may not be worth the extra overhead from marking an instruction exempt and ”removing” it from the IQ.

IV. Evaluation

A. Baseline ROB Scaling Analysis

Before evaluating our proposed Waiting Instruction Buffer (WIB), we first establish the performance characteristics of scaling ROB size alone in the baseline ARM O3 core.

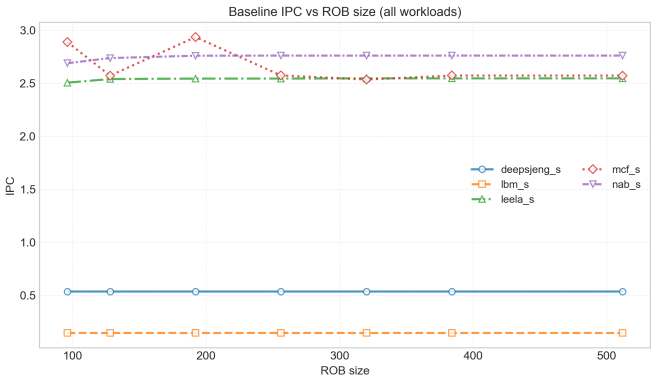

Fig. 1. Baseline IPC vs ROB size

Figure 1 shows IPC as a function of ROB size across five workloads. For all workloads, increasing ROB size from 96 to 512 entries yields virtually no IPC improvement since simply enlarging the reorder buffer allows more instructions to remain in-flight but does nothing to address the bottlenecks that limit instruction-level parallelism.

Across all workloads, increasing ROB size from 96 to 512 entries yields virtually no IPC improvement since simply enlarging the reorder buffer allows more instructions to remain in-flight but does nothing to address the bottlenecks that limit instruction-level parallelism. We can further infer the reason from Figure 2 as ROB size increases, ROB full events decrease—rename-stage stalls are alleviated as expected.

However, from Figure 3, IQ FullEvents increase correspondingly across all workloads. The additional in-flight instructions enabled by a larger ROB simply shift the bottleneck from the front-end (rename) to the back-end (issue), where long-latency loads and their dependents occupy issue queue slots for extended periods, which explains the flat IPC response.

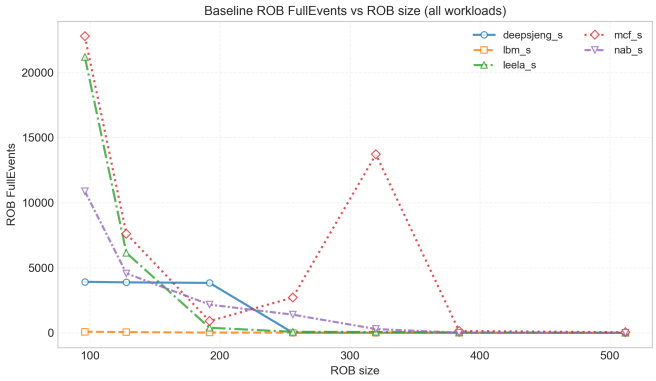

Fig. 2. Baseline ROB FullEvents vs ROB size

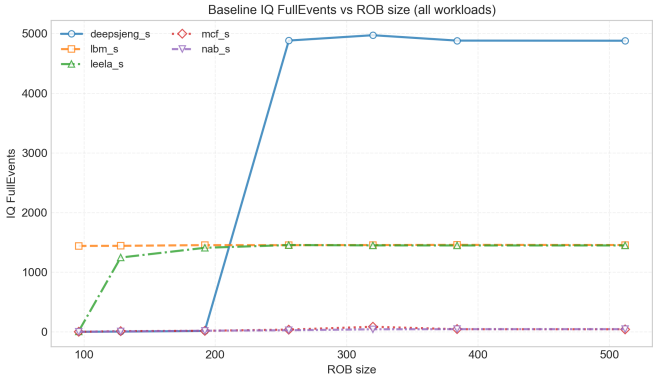

Fig. 3. Baseline IQ FullEvents vs ROB size

B. Waiting Instruction Buffer Analysis

We evaluate two variants **WIB-Infinite** and **WIB-Finite** of our proposed Waiting Instruction Buffer architecture against the baseline ARM O3 core. Both variants are evaluated across four latency thresholds (20, 40, 80, and 160 cycles) that determine which loads are considered ”long-latency.” Instructions exceeding this threshold become candidates for WIB migration.

1) WIB-Infinite:

a) Performance Impact: Figure 4 shows IPC for WIB-Infinite across ROB sizes for each workload. The improvements are marginal. For larger ROB, WIB shows slight IPC improvements across most workloads. As ROB size grows and IQ pressure becomes the bottleneck, removing long-latency dependents provides modest relief, allowing some ready instructions to issue sooner. For smaller ROB, WIB either shows no improvement or slightly degrades performance. Here the ROB itself remains the primary bottleneck, so IQ relief has little effect while the overhead of managing exempted instructions still applies.

b) Latency Distribution: Figure 5 shows IPC remains flat across all latency thresholds (20–160 cycles). This is because long-latency instructions depend on loads exceeding even our highest threshold, so they qualify for exemption. Instructions

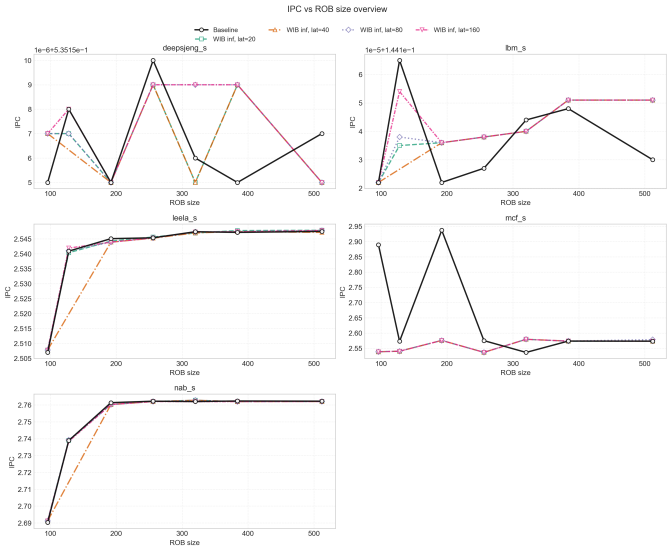

Fig. 4. WIB-Infinite IPC vs ROB size

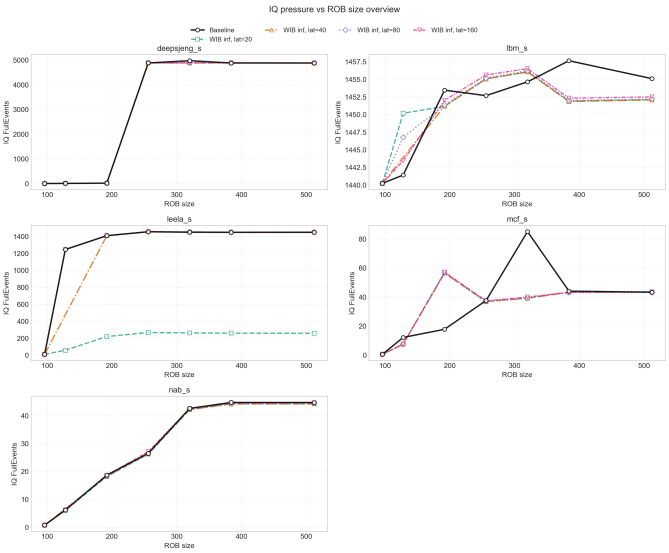

Fig. 6. WIB-Infinite IQ FullEvents vs ROB size

that cross lower thresholds but not higher ones (e.g., 25-cycle loads) would be ready within cycles after exemption, making their removal from the IQ negligible. As a result, latency threshold choice has no meaningful impact on performance.

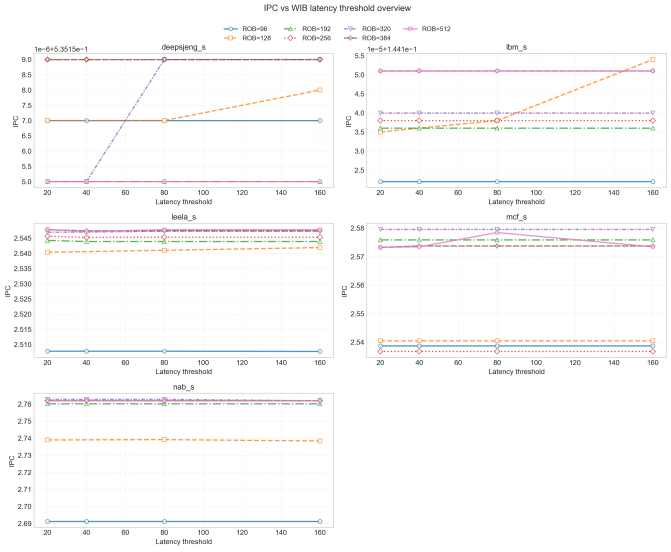

Fig. 5. WIB-Infinite IPC vs Latency

c) Instruction Queue: Figure 6 shows IQ FullEvents for WIB-Infinite across ROB sizes for each workload. The reduction is slight overall, but becomes more noticeable at larger ROB sizes for memory-intensive workloads like `lbm_s` and `mcf_s`. This pattern aligns with expectations: WIB targets issue queue pressure, which only becomes the dominant bottleneck when ROB is large.

d) Exempted Instruction: Figure 7 shows exempted instructions vs actual full IQ bypass events for long-latency equal to 20. Although many instructions are exempted—especially at larger ROB sizes—the number of events where an exempted instruction bypasses a full IQ remains very low. This low exemption-to-bypass ratio reveals a key inefficiency: instructions are removed from the IQ, but the IQ is rarely full when removal occurs. As a result, freed slots often go unused, explaining why the reduction in IQ pressure (Figure 6) translates to only marginal IPC gains.

Fig. 7. WIB-Infinite Exemption

2) WIB-finite:

a) Performance Trade Off: Figure 8 and Figure 9 compare WIB-Finite (16 entries) against WIB-Infinite and baseline across two latency thresholds (20 and 160 cycles). The finite configuration performs identically to its infinite counterpart, as both IPC and IQ FullEvents are nearly the same. This shows that a 16-entry WIB captures all useful benefits; larger

sizes provide no additional gains, making a small WIB a cost-effective design choice in our minimal implementation.

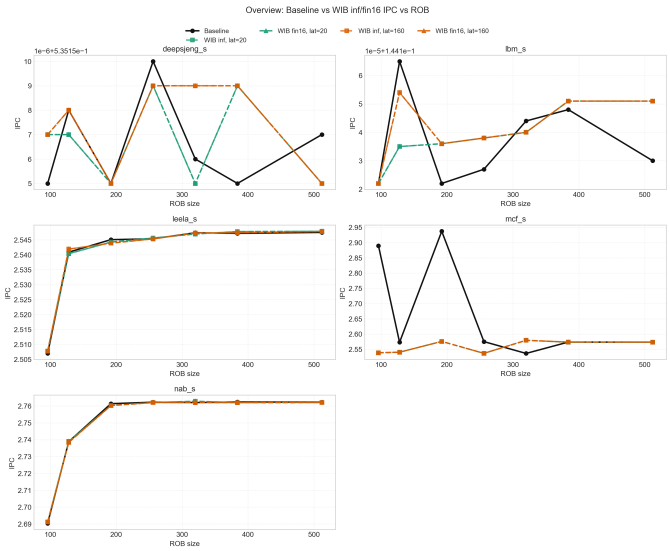

Fig. 8. WIB-Finite16 IPC vs ROB size

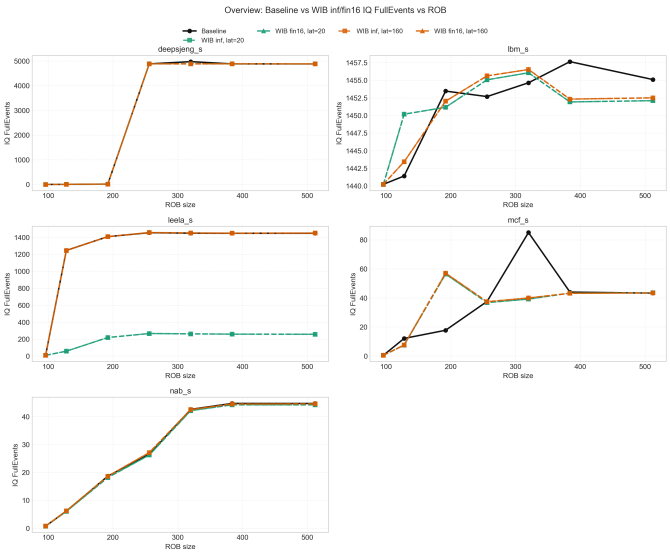

Fig. 9. WIB-Finite16 IQ FullEvents vs ROB size

V. Conclusion

In this work, we evaluated the impact of scaling the reorder buffer (ROB) and explored a Waiting Instruction Buffer (WIB). Our results show that increasing ROB size along provides little improvement in IPC. While the WIB reduces issue queue pressure and slightly improves performance at larger ROB sizes, these gains are limited. We find that the effectiveness of the WIB is constrained by its selection policy, which often fails to target instructions that meaningfully resolve the bottlenecks. In addition, latency thresholds have minimal impact, and with our implementation a small WIB is sufficient to capture the benefits. Overall, our findings suggest that simply increasing instruction window capacity is insufficient. Instead, more effective scheduling mechanisms are required to fully utilize large ROB designs.

VI. Statement of Work

Steve conducted the preliminary setup and evaluation of the ROB scaling experiments. Eliot created the implementation of both the finite and infinite WIB in gem5. All three members contributed equally to the WIB evaluation experiments. Ryan was the primary organizer for the final project paper, compiling the preliminary research to introduce and motivate the problem and outlining the best structure to present the evaluation and conclusion. For the paper writing, Ryan wrote the abstract, introduction, and background sections, Eliot wrote the design section, and Steve wrote the evaluation and conclusion section.

References

[1] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[2] Z. Youssfi and M. Shanblatt, "Instruction-window power reduction using data dependence metric."

[3] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors."

[4] S. Weiss and J. E. Smith, "Instruction issue logic for high-performance, interruptible pipelined processors," *IEEE Transactions on Computers*, vol. C-33, no. 12, pp. 1013–1020, 1984.

[5] G. S. Sohi and V. K. Vajapeyam, "Instruction issue logic for high-performance, interruptible pipelined processors," in *ISCA*, 1987.

[6] G. Tjaden and M. Flynn, "Detection and parallel execution of independent instructions," *IEEE Transactions on Computers*, vol. C-19, no. 10, pp. 889–895, 1970.

[7] B. C. Tjaden and M. J. Flynn, "Evaluation of parallelism in instruction streams," *IEEE Transactions on Computers*, 1973.

[8] A. K. Uht, "Vliw architectures for a trace scheduling compiler," *IEEE Micro*, 1986.

[9] J. E. Smith, "Instruction-level parallelism in programs," *IEEE Micro*, 1989.

[10] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *ASPLOS*, 1988.

[11] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," *SIGARCH Comput. Archit. News*, vol. 17, no. 2, p. 290–302, Apr. 1989. [Online]. Available: https://doi.org/10.1145/68182.68209

[12] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: An effective alternative to large instruction windows," *Micro, IEEE*, vol. 23, pp. 20 – 25, 12 2003.