Orinoco: Ordered Issue and Unordered Commit with Non-Collapsible Queues

Dibei Chen
Tsinghua University
chendb17@mails.tsinghua.edu.cn

Tairan Zhang
Tsinghua University
zhangtr19@mails.tsinghua.edu.cn

Yi Huang
Tsinghua University
y-huang21@mails.tsinghua.edu.cn

Jianfeng Zhu
Tsinghua University
jfzhu@tsinghua.edu.cn

Yang Liu
Innovation Institute of
High Performance Server
yang.liu@shingroup.cn

Pengfei Gou
Innovation Institute of
High Performance Server
pengfei.gou@shingroup.cn

Chunyang Feng
HEXIN Technologies
chunyang.feng@shingroup.cn

Binghua Li
HEXIN Technologies
binghua.li@shingroup.cn

Shaojun Wei
Tsinghua University
wsj@tsinghua.edu.cn

Leibo Liu*
Tsinghua University
liulb@tsinghua.edu.cn

ABSTRACT

Modern out-of-order processors call for more aggressive scheduling techniques such as priority scheduling and out-of-order commit to make use of increasing core resources. Since these approaches prioritize the issue or commit of certain instructions, they face the conundrum of providing the capacity efficiency of scheduling structures while preserving the ideal ordering of instructions. Traditional collapsible queues are too expensive for today's processors, while state-of-the-art queue designs compromise with the pseudo-ordering of instructions, leading to performance degradation as well as other limitations.

In this paper, we present Orinoco, a microarchitecture/circuit co-design that supports ordered issue and unordered commit with non-collapsible queues. We decouple the temporal ordering of instructions from their queue positions by introducing an age matrix with the bit count encoding, along with a commit dependency matrix and a memory disambiguation matrix to determine instructions to prioritize issue or commit. We leverage the Processing-in-Memory (PIM) approach and efficiently implement the matrix schedulers as 8T SRAM arrays. Orinoco achieves an average IPC improvement of 14.8% over the baseline in-order commit core with the state-of-the-art scheduler while incurring overhead equivalent to a few kilobytes of SRAM.

*Corresponding author: Leibo Liu (liulb@tsinghua.edu.cn)

ISCA '23, June 17–21, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589046

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**.

KEYWORDS

microarchitecture, out-of-order execution, instruction scheduling, out-of-order commit, processing-in-memory

ACM Reference Format:
Dibei Chen, Tairan Zhang, Yi Huang, Jianfeng Zhu, Yang Liu, Pengfei Gou, Chunyang Feng, Binghua Li, Shaojun Wei, and Leibo Liu. 2023. Orinoco: Ordered Issue and Unordered Commit with Non-Collapsible Queues. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579371.3589046

1 INTRODUCTION

To many people's surprise, Moore's Law is still alive and well [38]. Whether it is TSMC's 5 nm or Intel 7 process, with an increasing transistor budget, modern core microarchitectures of major CPU vendors become deeper and wider to bring instructions per cycle (IPC) improvements on top of clock speed gains. The issue width (IW) has increased around 50% and the reorder buffer (ROB) goes all the way up from 224 entries to 512 entries across generations [19, 64]. Although larger out-of-order windows and wider back-end facilitate out-of-order (OoO) execution, their capabilities are hindered by the inefficient use of these resources. Top-down microarchitecture analyses of Google and Facebook's services [40, 72] expose that only 20-40% of instructions can be retired without stalling and only one-third of the theoretical execution bandwidth is used, leaving the performance of those ultra wide and deep microarchitectures on the table. Therefore, in addition to widen and deepen the cores, more aggressive scheduling strategies are needed to make the most of the core resources.

The allocation and reclamation of the queue structures is fundamental to the dynamic scheduling in OoO processors. An instruction allocates its entries in the instruction queue (IQ) and ROB at the dispatch stage, then releases them at the issue stage and the commit stage, respectively. The latter also involves the reclamation of physical registers and load/store queue (LSQ) entries for memory operations. What is implicit in these queue designs is the instruction ordering in dynamic scheduling. For example, the schedulers generally prefer to issue the oldest ready instructions in the IQ to avoid stalling, and only instructions at the head of ROB are checked for commit to enforce precise exceptions. In other words, the temporal ordering of instructions is determined implicitly by their positions in the IQ and ROB, i.e., the closer to the head of the queues, the older the instruction is. Based on OoO execution, prior approaches such as criticality-based scheduling [4, 5, 17, 48, 54, 67] and out-of-order commit [15, 27, 28, 49, 50, 62] have shown performance benefits by prioritizing the issue or commit of certain instructions. However, they face the conundrum of recycling the gaps left in the queues by the issued or committed instructions while preserving the ideal ordering of instructions. Hereafter we say the instruction ordering is ideal if it follows the temporal ordering. For instance, a traditional collapsible queue (SHIFT) [20] shifts younger instructions down to fill the gaps (1), which introduces significant hardware complexity that is only feasible when the queue size is small. The crux of the problem is the physical ordering of instructions from the head to the tail of the queues. To get rid of this intrinsic characteristic of queues, we need to explicitly track the order between instructions, allowing arbitrary allocation and removal of entries without jeopardizing the ability to emit instructions. For instance, linked lists may seem intuitive, but the traversal of linked lists takes O(n) time, where n is the number of instructions in an IQ or the commit width of an ROB, which is untenable for instruction scheduling with a huge n. Likewise, timestamps are inadequate since the complexity of the sorting algorithm is O(log n) at best. The random queues (RAND) combined with the age matrix (AGE) [66] represent the state-of-the-art scheduler designs in commercial processors [22, 70]. Although RAND provides capacity efficiency without the overhead of compacting in SHIFT, only the single oldest instruction is prioritized in AGE, which leads to performance degradation, especially for ultra wide and deep core designs. Recently proposed architectures [4, 17, 27] tend to employ multiple sub-queues to differentiate between instructions at the issue or commit stage. Since each individual queue still suffers from the same issue, these designs only satisfy the pseudo-ordering of instructions, which sacrifices performance and imposes practical limitations to the system, such as precise exceptions and memory consistency models.

In this paper, we manifest that the matrix-based scheme itself is not the problem, but the implementation is, which can be solved with the emergent circuit techniques. To this end, we present Orinoco, a microarchitecture/circuit co-design that supports ordered issue and unordered commit with non-collapsible IQ, ROB, and LQ. We redefine the semantics of the matrix scheduling and extend beyond IQ to the organization of ROB and LQ. By tracking the temporal ordering of instructions in the age matrix, the relative age between instructions is decoupled from their positions in the IQ or ROB, allowing random allocation of entries at dispatch. With the aid of the bit count encoding, the IQ's age matrix can select up to IW oldest instructions to issue, where only one was possible before. By tracking the commit conditions between instructions in the ROB's commit dependency matrix, instructions can commit out-of-order non-speculatively without the complexities of post-commit speculation in prior works. Meanwhile, memory dependencies can be resolved early in the LSQ's memory disambiguation matrix, allowing early commit of loads without compromising the memory consistency model. Since OoO commit allows for the early reclamation of critical resources such as IQ, ROB, LQ, and physical registers, more instructions can be scheduled beforehand, which in return, contributes to further performance benefits of priority scheduling.

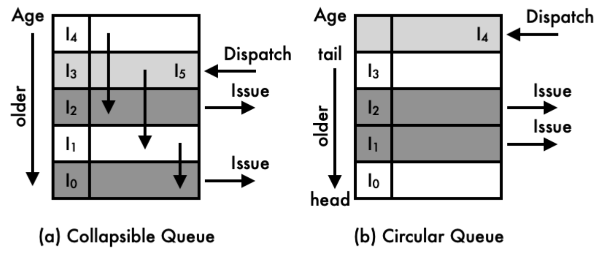

Figure 1: Example of a collapsible queue and a circular queue.

Another challenge arises from the efficient implementation of the matrix schedulers. Matrix scheduling involves a large amount of logic operations, whereas existing solutions, whether implemented with static or dynamic logic, suffer from circuit complexities that can neither fulfill the magnitude of queue structures nor the clock frequencies of modern processors. To address such complexity, we observe that the computation of matrix scheduling matches the in-SRAM logical operations [35] and leverage the Processing-in-Memory (PIM) approach to implement the matrix schedulers as 8T SRAMs. Besides, the bit count encoding can be sensed as the voltage drop on the bit lines during PIM operations. By customizing the bit cells and the peripheral circuits, all the required operations can be performed inside the SRAM arrays in an all-digital manner, whose latency is comparable to the processor's clock speed. Therefore, the timing constraints of the pipeline stages are met, and our SRAM-based design also provides higher density and better scalability over traditional matrix schedulers.

In summary, the contributions of Orinoco are as follows:

- We examine the existing queue designs for instruction scheduling in OoO processors and recognize that the physical ordering inherent in queue structures hinders further performance benefits of priority scheduling and out-of-order commit.
- We propose the age matrix with the bit count encoding that decouples the temporal ordering of instructions from IQ or ROB positions while determining instructions to prioritize issue or commit in O(1) time.
- We propose the commit dependency matrix and the memory disambiguation matrix to track the commit conditions and the memory dependencies between instructions to facilitate out-of-order commit with non-collapsible queues.

- We leverage the Processing-in-Memory (PIM) approach and modify the 8T SRAM arrays to implement the matrix schedulers, which directly support bit count encoding, vertical access, and superscalar dispatch at a low cost.

Orinoco achieves an average IPC improvement of 6.5% with priority scheduling and 13.6% with out-of-order commit, which synergistically increases performance by 14.8% compared to the baseline OoO core with AGE and in-order commit. It incurs 0.3% area overhead and 0.6% power overhead, which is equivalent to a few kilobytes of SRAM.

In the remainder of this paper, we first introduce the background on priority scheduling and out-of-order commit in Section 2. In Section 3, we propose the matrix scheduling for instruction issue and commit, and in Section 4 we propose the PIM-based implementation of the matrix schedulers. Section 5 outlines Orinoco's microarchitecture. Section 6 discusses the evaluation methodology and presents the results. We review the related works in Section 7.

2 BACKGROUND

Contemporary processors are generally built upon superscalar out-of-order execution [74]. OoO processors exploit instruction-level parallelism (ILP) and memory-level parallelism (MLP) to improve IPC by executing instructions when their operands are available rather than in program order. The structures where instructions are dispatched and waiting to become ready are known as instruction queues (IQ) or reservation stations (RS) in some nomenclature. When the number of ready instructions exceeds the number of execution units, which accounts for 18% of total cycles, arbitration is required to decide which instructions to issue. On the other hand, in-flight instructions are tracked in the reorder buffer (ROB) and commit (retire) in program order to have the illusion of sequential, atomic instruction execution as intended by the program [71]. Although in-order commit enforces precise exceptions, it is overly conservative, given that for the rare cases of exceptions, younger instructions may be blocked by older long-latency instructions that were initially bypassed during dynamic scheduling. Critical resources are held for far longer than needed, which may lead to full window stalls when any of them are exhausted. In the following subsections, we briefly introduce the concept of priority scheduling and out-of-order commit, their techniques, and shortcomings.

2.1 Priority Scheduling

Two notable attributes of instruction scheduling are its suboptimum and timeliness. First, instruction scheduling is an NP-hard problem[9], meaning it is challenging to derive the perfect scheduling order without oracle knowledge. Second, since scheduling logic is on the critical path of the processor pipeline, it should be performed with minimum latency without complex hardware. Therefore, most processors adopt the oldest-first scheduling policy by prioritizing the issue of older instructions. The rationale behind this heuristic is to increase the likelihood of committing when an instruction reaches the head of the ROB and potentially facilitate the execution of the critical path, e.g., a long dependency chain. For a physically ordered IQ, it also allows for the select logic that is based on the position of instructions, such as a tree arbiter[58].

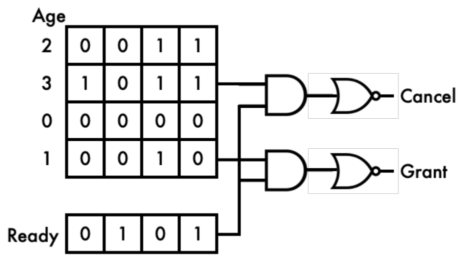

Figure 2: Example of an age matrix.

However, the oldest ready instruction does not necessarily mean the oldest instruction. To fill the gaps created by the issued instructions, the collapsible queue (SHIFT)[20] issue utilizes a compacting circuit to determine vacancies and shift instructions downward, as shown in Figure 1(a). Although the collapsible queue achieves capacity efficiency while maintaining the temporal ordering, each compacting takes O(mn) shifts, where m is the depth and n is the width of the IQ, rendering the compacting circuit not only complicated in design but also considerable in power consumption. It was used in Alpha 21264 [42] when the IQ only had 20 entries, but it is no longer realistic in today's processors. In terms of non-collapsible designs, the circular queue (CIRC) [11] and the random queue (RAND) [78] represent the two ends of the spectrum: the former remaining gaps that causes capacity inefficiency (Figure 1(b)), while the latter randomly dispatching instructions into gaps that perturbs the temporal ordering — both degrading performance. Therefore, contemporary processors augment RAND with an age matrix (AGE), as deployed by AMD Bulldozer [22] and IBM POWER8 [70]. As demonstrated in Figure 2, each row and column of AGE is associated with an instruction in the IQ, while each bit represents the relative age order. To determine the oldest, each ready instruction performs bitwise AND between its age mask with the request vector, then reduction NOR the result. Only for the oldest, the result will be zero, which will be granted for execution. However, determining multiple oldest instructions demands O(n) iterations of the age matrix, where n is the issue width of the IQ. The latency would be unacceptable for pipelining at high clock speeds, especially for an ultra wide design. Therefore, in AGE, the remaining issue width is selected randomly in terms of age, which limits the effectiveness of the priority scheduling. While the most recent work, SWQUE [6], dynamically configures the IQ as CIRC or AGE according to the capacity demand, it still cannot have the best of both worlds.

Recent works have leveraged instruction criticality to improve the quality and efficiency of dynamic scheduling. For example, Long term parking [67], FIFOrder [5], and Delay and Bypass [4] classify instructions based on their criticality and readiness, and allocate them into smaller structures to reduce the depth and width of the IQ, thus reducing scheduling energy. Precise Runahead [54], Criticality Driven Fetch [17], and Critical Slice Prefetching [48] identify critical instructions and their dependency chains via online or offline profiling and dataflow analysis, and prioritize their execution through pipeline modifications. However, as long as these approaches still somehow use IQs for instruction scheduling rather than in-order FIFOs, they suffer from the same ordering problem mentioned above, which is exactly what we address in Orinoco.

2.2 Out-of-Order Commit

Knowing the limitations of in-order commit, Bell and Lipasti [8] have deconstructed the notion of commit and scrutinized the necessary conditions under which instructions can commit out of program order in a non-speculative way. These conditions are:

(1) The instruction is completed;
(2) Register write-after-read (WAR) hazards are resolved;
(3) The instruction is on the correct path of the predicted branches;
(4) No prior instruction will raise a potential exception;
(5) The instruction is not involved in memory replay traps;
(6) The global order is not perturbed in a manner that violates the memory consistency model.

None of them imply that the instruction should be at the head of the ROB, thus allowing for the out-of-order commit of younger instructions without waiting for older instructions to complete. Our analysis shows that a non-trivial portion of in-flight instructions meet all the conditions but cannot commit only because they are not at the head of the ROB. They appear in 72% of cycles completely stalled on instruction commit, even 76% of cycles during full window stalls. Allowing these instructions to commit out-of-order and release resources early may potentially improve performance by reducing stalls due to resource exhaustion. However, similar to the IQ, it also entails the management of the gaps created by the committed instructions in the ROB and the LQ [3]. In addition, compared to in-order commit that determines the commit of the head instruction based solely on the local state, OoO commit conditions require examining the states of older instructions, such as branches, memory aliasing, and exceptions (Conditions 3, 4, and 5). Therefore, early works proposed a collapsible ROB [8] not only to recycle the gaps but also to reduce the propagation of the global state. However, it disallows the use of the ROB position to identify an instruction, and as mentioned earlier, the collapsible ROB is not viable, especially for an ultra deep design.

To avoid collapsing the queues, prior works try to commit instructions where one or more commit conditions are not preserved in a speculative or non-speculative manner. For example, Cherry [50] and Out-of-order commit processors [15] commit instructions speculatively but demand expensive checkpointing and rollback mechanisms in case the speculation fails. NOREBA [27] relaxes Condition 5 and commits instructions non-speculatively beyond the reconvergence point of a branch, and proposes a Selective ROB consisting of multiple queues to separate them and commit early. However, since each queue commits in program order, they may still be blocked by long-latency instructions other than branches. Validation Buffer [49] relaxes Condition 1 and commits instructions as soon as they are guaranteed to be non-speculative and leave the ROB in program order without waiting for completion. However, there are several limitations with the post-execution of committed instructions. First, it complicates exception handling and context switching in terms of register status. Physical registers mapped to older instructions cannot be reclaimed immediately since they may still be busy with consumers when an exception is encountered. In other words, the register status is not precise before the canceled instructions are drained to restore the register mappings, which could be prohibitive for a large ROB. Second, it is incompatible with state-of-the-art performance profilers such as Intel PEBS [77], which attribute execution time to instructions from ROB sampling at commit [25], because instructions may leave the ROB before they complete. Lastly, it only supports weak or relaxed memory consistency model [28] due to the fact that speculative loads may commit before the reordering could be caught by other cores, violating the load→load ordering of Total Store Order (TSO) [68]. Ros et al. [62] propose to irrevocably bind reordered loads and delay conflicting stores in its store buffer to preserve TSO. Interestingly, not only do they necessitate a collapsible ROB but also a collapsible LQ to fill the gaps and keep program order. We address exactly these problems in Orinoco.

3 MATRIX SCHEDULING

The goal of Orinoco is to preserve the ideal ordering of instructions with non-collapsible queues while determining instructions to prioritize issue or commit in O(1) time. To this end, we propose the age matrix with the bit-count encoding that decouples the temporal ordering of instructions from IQ or ROB positions and determines multiple oldest instructions simultaneously (Section 3.1). In non-collapsible queues, to track the out-of-order commit conditions between instructions in the ROB and the memory dependencies between loads and stores in the LQ/SQ, the commit dependency matrix and the memory disambiguation matrix are proposed (Section 3.2 and 3.3). Moreover, we revisit the idea of the wakeup matrix as an alternative to the CAM-based IQ (Section 3.4), which also benefits from the PIM implementation of the matrix schedulers, as we will elaborate in Section 4.

3.1 Age Matrix

To begin with, the age matrix tracks the relative age between instructions in a bit matrix, which has as many rows and columns as entries in the IQ/ROB. Each row and column of the matrix is associated with an instruction in the IQ/ROB, and the intersection represents the age order between instructions. After instructions are decoded and renamed, they are dispatched to the IQ and the ROB. A VLD vector tracks the valid entries in the IQ/ROB. Since the front-end executes in-order, when an instruction is dispatched, it must be younger than any existing instructions in the IQ/ROB. Therefore, the row vector corresponding to its entry is set to all ones while the column vector is cleared, as illustrated in Figure 3 and Figure 4.

When an instruction is woken up in the IQ and ready to schedule (Section 3.4), it sets the corresponding bit in a BID vector. Each ready instruction then performs bitwise AND between its row vector and the BID vector. Instead of reduction NOR the result vector in previous methods, we count the number of ones in the vector. The insight is that for n oldest ready instructions, there are at most n-1 instructions older than them, therefore the result should contain no more than n-1 ones. By comparing the numbers to the issue width (IW), the bit count encoding is able to pick up to IW oldest instructions to issue, as illustrated in Figure 3. Since there is no dependency between the select logic of each ready instruction, arbitration can be done in parallel in O(1) time.

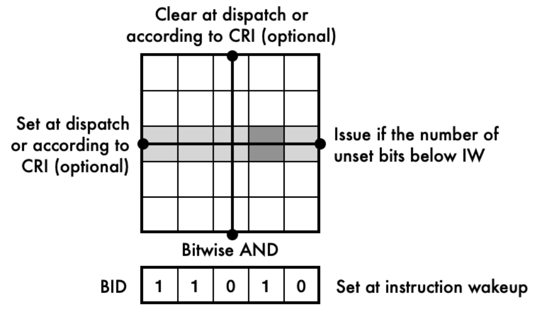

Figure 3: Example of an age matrix for IQ. An instruction sets its row and clears its column at dispatch. Critical instructions are always "older" than non-critical ones. Each ready instruction performs bitwise AND between its row vector and the BID vector, whose result contains ones less than IW is granted to issue.

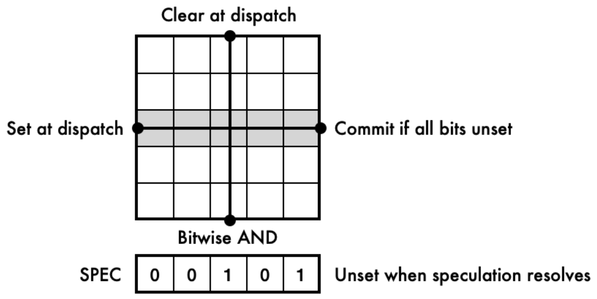

Figure 4: Example of an age matrix for ROB. An instruction sets its row and clears its column at dispatch, and clears its bit in the SPEC vector when not speculative. Each completed instruction performs bitwise AND between its row vector and the SPEC vector, and commits if the result is all zeros.

Although non-speculative out-of-order commit maintains the precise architectural state in the presence of misspeculation or exceptions, precise exception handling is non-trivial since they may occur anywhere in a non-collapsible ROB. To locate the oldest exception, which must be raised by the oldest instruction left in the ROB (Section 3.2), each in-flight instruction performs bitwise AND between its row vector and the VLD vector. Only for the oldest, the result will be all zeros. After reduction NORed to a single bit, the one-hot result vector will indicate the instruction that causes the exception event.

Criticality-based Scheduling Prior works that consider instruction criticality generally identify critical instructions via hardware performance counters (HPC) at runtime [26] and construct their dependency chains via iterative backward dependency analysis (IBDA) [13] performed in hardware or software. The instructions on the critical paths are prioritized by either tagging them in IQ [48] or splitting the IQ to differentiate them [17]. In addition to the ordering problem, it may incur load unbalancing and complicate the arbitration between critical and non-critical instructions in the IQ.

To leverage the criticality information, the critical instructions in the IQ are tracked in a CRI vector. When a critical-tagged instruction is dispatched to the IQ, only bits corresponding to critical instructions are set in its row vector and cleared in its column vector, as illustrated in Figure 3. Since it makes critical instructions appear "older" than non-critical ones, taking into account the bit count encoding, the oldest critical instructions are prioritized to issue, along with the oldest non-critical instructions if the issue width remains, which are arbitrated at the same time. Nevertheless, the actual performance of the age matrix depends on the implementation of the bit count encoding, whose logic is much more complex than the boolean algebra used in AGE, which we will address in Section 4.

3.2 Commit Dependency Matrix

From the discussion in Section 2.2, the commit conditions of an instruction can be categorized into two types, those related to the local state and those based on the global state of older instructions. Specifically, the local state is concerned with whether the instruction has completed, raised an exception, or flushed, similar to the in-order commit scenario. The global state, on the other hand, refers to whether older instructions may raise misspeculation or exceptions and cause subsequent instructions to abort, which could be identified early in the pipeline before an instruction completes. For example, page faults for memory operations occur when resolving their addresses and accessing the TLB [28], as opposed to waiting for memory access to complete, whose latency frequently induces pipeline stalls. Long-latency arithmetic operations such as floating-point instructions can either be free of exceptions or accrue status for future checks without causing traps [56], as in some instruction sets (ISA) for performance reasons. For that reason, the opportunity to commit instructions out of order in a non-speculative way arises. A straightforward way to track the global state is to implement scoreboarding in ROB, similar to how data dependencies are tracked in IQ. However, it requires structuring the ROB as content-addressable memory (CAM) and associatively searching the entire ROB at commit, which would be too expensive since ROB is much larger than IQ or LSQ that only keeps the subset of instructions.

We observe that the commit conditions, like the temporal ordering, are essentially a form of dependency between instructions, which can also be tracked in a commit dependency matrix. The commit dependency matrix has as many rows and columns as entries in the ROB, where each row and column is associated with an in-flight instruction. The intersection, in this case, indicates the commit dependencies between instructions. As illustrated in Figure 5, when an instruction is dispatched to the ROB, the bits corresponding to older instructions that may raise exceptions or misspeculation are set in its row vector, such as memory operations, branches, synchronization barriers, etc. Once these instructions are guaranteed to be safe, for example, the memory operation successfully accesses the page table or the branch is correctly predicted, the corresponding column vectors are cleared. When an instruction completes, it waits for all the older instructions to become non-speculative. Therefore, the global state is checked by reduction NOR the row vector, and a result of "0" means the instruction is safe to commit, even if it is not at the head of the ROB.

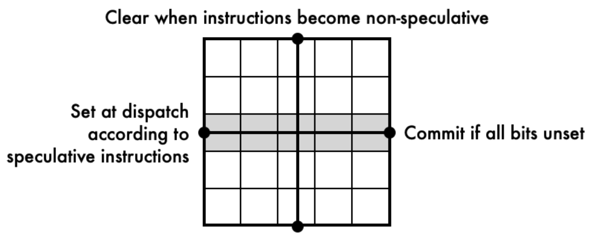
Clear when instructions become non-speculative
Set at dispatch according to speculative instructions
Commit if all bits unset

Figure 5: Example of a commit dependency matrix. An instruction sets its row according to older speculative instructions in ROB at dispatch, and clears its column when not speculative. Each completed instruction commits when all the bits in its row vector is zero.

Precise Exception Handling In case of misspeculation due to branches or memory disambiguation, the delinquent and all subsequent instructions are squashed. The subsequent instructions can be located by checking the column vector of the delinquent instruction in the ROB's age matrix. Since none of them have committed yet, the same recovery mechanism in an in-order commit core is used to revert the architectural state of the processor. To provide precise exception handling, when none of the instructions can commit, the oldest instruction either has not resolved its speculation or has raised an exception, whose position can be determined as described in Section 3.1. Unlike post-commit speculation, all the older instructions have completed their execution, therefore the architectural state is precise without further ado, and the program counter (PC) is redirected to the appropriate exception handler. Note that instructions such as atomic RMWs [7] that cannot execute speculatively are handled similarly.

Matrix Merging Both the commit dependency matrix and the ROB's age matrix have the complexity of O(n^2), where n is the number of entries in the ROB. Since n could be large for deep OoO designs, to reduce the size of the age matrix, Sassone et al. [66] propose to group instructions and only track age between instruction groups, not within a group. However, it produces a pseudo-ordering of instructions, which would prevent precise exceptions when applied to ROB. Luckily, the commit dependency matrix and the ROB's age matrix can be merged by exploiting the fact that the global state is the same for all instructions in ROB, and those belong to older instructions constitute the commit dependencies of an instruction. Specifically, a SPEC vector tracks which instructions may raise misspeculation or exceptions, where the corresponding bit is set when such an instruction is dispatched to the ROB and cleared when the instruction becomes non-speculative. At commit time, the row vector of each completed instruction in the ROB's age matrix is bitwise ANDed with the SPEC vector, as illustrated in Figure 4. The result is the same as the original commit dependency matrix, which is then reduction NORed to acquire the commit grant. Since ROB has many more entries than other queue structures involved in matrix scheduling, the merging cuts down the area overhead by 40% for the configuration in our evaluation.

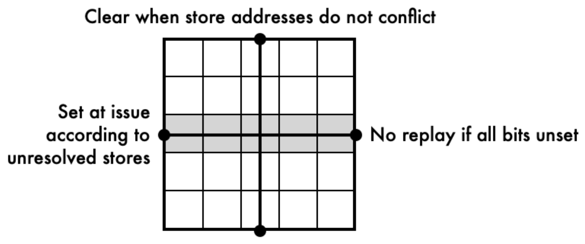
Clear when store addresses do not conflict
Set at issue according to unresolved stores
No replay if all bits unset

Figure 6: Example of a memory disambiguation matrix for LQ/SQ. A load in LQ sets its row according to older unresolved stores in SQ at issue, and a store clears bits in its column when its address does not conflict. A load becomes non-speculative when all the bits in its row vector is zero.

3.3 Memory Disambiguation Matrix

OoO processors speculatively issue loads over older loads or stores with unknown addresses to improve performance. To guarantee correctness, when resolving its address, a load searches SQ/SB for the latest store to the same address, and if it exists, the value is forward; a store searches LQ for younger speculative loads to the same address and squashes the loads if there is a conflict. Similar to IQ and ROB, the relative ordering of loads and stores is defined by their positions in the LQ/SQ, and in-order commit ensures all previous addresses have been resolved thus all conflicts have been detected. However, waiting for all previous memory instructions to perform (we say a load is performed when data is in the cache) results in significant performance degradation for out-of-order commit. To decide the absence of conflicts as early as possible in the absence of the collapsible LQ, the order and dependencies between loads and stores need to be explicitly tracked. Although it could be accomplished by assigning timestamps to memory instructions, timestamp storage and comparison incurs large overhead and frequent wraparounds due to counter limits [65].

We propose the memory disambiguation matrix, which combines the role of the age matrix and the dependency matrix. The memory disambiguation matrix has as many rows as entries in the LQ, each associated with a load instruction, and as many columns as entries in the SQ, each associated with a store instruction. The intersection indicates the undetermined dependency between the load-store pair. Specifically, when a load issues, it searches the SQ/SB for possible aliases and unresolved addresses. Note that the SQ/SB are FIFOs since stores commit in program order. The bits related to older stores with unresolved addresses are set in the row vector, as illustrated in Figure 6. Therefore, younger speculative loads with respect to an unresolved store are tracked in the corresponding column vector. When a store resolves its address, it searches the LQ according to its column vector and clears the corresponding bits when there is no conflict. A load becomes non-speculative as soon as all the previous stores have resolved their addresses and no replay is required, which can be determined by reduction NOR its row vector. Consequently, the corresponding bit in the SPEC vector is cleared.

Enforcing Total Store Order Since loads are committed out of order before it is determined that the reordering has not been

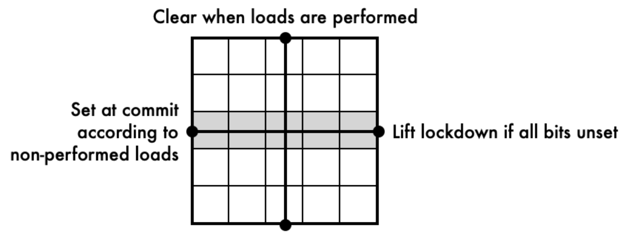
Clear when loads are performed
Set at commit according to non-performed loads
Lift lockdown if all bits unset

Figure 7: Example of a lockdown matrix. A committed load sets its row according to older unperformed loads in LQ, and a performed load clears its column. A lockdown to the address of a committed load is lifted when all the bits in its row vector is zero.

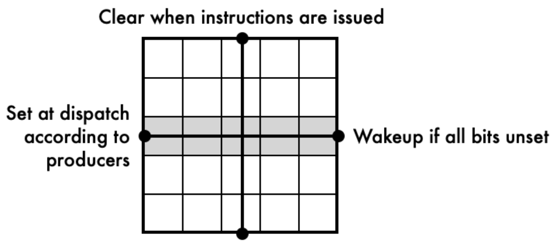
Clear when instructions are issued
Set at dispatch according to producers
Wakeup if all bits unset

Figure 8: Example of a wakeup matrix. An instruction sets its row according to its producers in IQ at dispatch, and clears its column at issue. An instruction is woken up when all the bits in its row vector is zero.

caught by anyone, it transgresses the ordering rules of Total Store Order (TSO) [68] and can only resort to a weak memory model [28]. Ros et al. [62] propose to irrevocably bind reordered loads to allow them to commit out of order while TSO is preserved at the coherence layer. In particular, when a load is performed out of order with respect to older loads, it locks down and withholds any invalidation acknowledgment or cache eviction to its address until all previous loads are performed. When a speculative load commits out of order, its lockdown is transferred to a Lockdown Table (LDT). Since their approach is based on a collapsible LQ, the responsibility of releasing its lockdown is assigned to the closest older non-performed load until the oldest non-performed load is completed and lifts the lockdown. However, prior works with post-commit speculation fail to track non-performed loads since they have already committed and been removed from the LQ.

To achieve non-speculative load reordering with a non-collapsible LQ, we use a Lockdown Matrix to track each speculative load to its older non-performed loads. Each row is associated with a committed load and each column is associated with a load in the LQ. The relative age between loads is initially tracked in the LQ's age matrix. When a load commits over older non-performed loads, it sets their corresponding bits in the row vector of the lockdown matrix, as illustrated in Figure 7. A performed load then clears the corresponding column vector. A speculative load becomes ordered when all previous loads are performed, which can be determined by reduction NOR its row vector, and the lockdown for its address is lifted. Since multiple lockdowns are allowed for the same address, the acknowledgement to invalidation or eviction is returned only when all the lockdowns for that address are released. Furthermore, Ros et al. [63] propose to eliminate the LQ by delaying stores in the store buffer (SB) until the speculative loads are validated and become ordered at commit. Out-of-order commit would be more beneficial for an implementation without LQ, where ROB becomes the predominant bottleneck. When a load issues, their approach uses the ROB positions to identify unresolved stores or older non-performed loads and set sentinels, which could also be achieved on a non-collapsible ROB with instruction type masks for the ROB's age matrix. Although it has no benefit for the uniprocessor model in our evaluation, Orinoco should work with TSO in a multiprocessor as described above, and we leave further evaluation for future work.

3.4 Wakeup Matrix

Historically, the matrix scheduling was first proposed to perform the wakeup logic of IQ [24] as an alternative to the content-addressable memory (CAM) based implementation. In a CAM-based IQ, each entry of CAM holds the source operand tags of an instruction and is compared with the broadcast destination tags of issued instructions. However, its complexity grows linearly with the depth of the IQ and logarithmically with the number of tags, which is further exacerbated by the number of search ports required according to the issue width. Therefore, IQ is one of the most complex and power-hungry components in OoO cores [1].

To ameliorate such overhead, Goshima et al. [24] observe that since the dependencies between instructions have been detected by the register renaming in the front-end, they can be represented as the positions assigned to the instructions in the IQ, thus eliminating the associative search needed for the wakeup process. The wakeup matrix has as many rows and columns as entries in the IQ, each associated with a non-ready instruction. When an instruction is dispatched to the IQ, the bits for the instructions that produce its source operands are set in the row vector, as illustrated in Figure 8. In the wakeup process, each issued instruction clears its column vector. An instruction is ready when all the bits in its row are zero, which can be determined by reduction NOR its row vector. The woken-up instructions are then scheduled as described in Section 3.1.

In contrast to the original design that uses a separate wakeup matrix for each source operand, we show that the wakeup matrix can be implemented as a whole using the PIM approach described in the following section. In addition, prior works [24, 66] exploit the scarcity (number of dependencies) or locality (distances between dependent instructions) in the scheduler to narrow down the wakeup matrix, i.e., reduce the number of columns. Although we do not include these optimizations in our evaluation, they are compatible with our implementation and should also benefit from it.

4 PROCESSING-IN-MEMORY IMPLEMENTATION

The aforementioned matrix scheduling fulfills O(1) time scheduling in theory but comes at the cost of increased logic complexity.

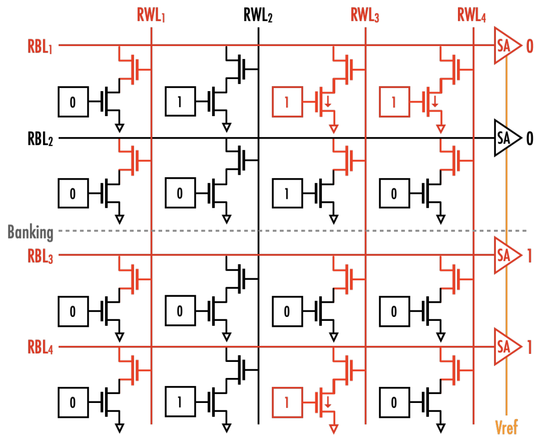

Figure 9: The PIM-based matrix scheduler. The RBLs are precharged beforehand, then the RWLs are activated. The voltage drop on the RBLs is compared against the reference voltage of the SAs.

Therefore, its actual performance is subject to the efficient implementation of the matrix schedulers and the ability to meet the timing constraints of the pipeline stages while producing reasonable area overhead for core designs. The age matrix, for example, performs n^2 AND operations and n n-bit NOR operations in a given cycle, even without considering the bit count encoding. Given the timing and area constraints, it is unrealistic to implement such a large-scale matrix scheduler with static logic, i.e., register files in combination with logic gates. In contrast, prior works design the matrix schedulers based on the 12 transistors (12T) cells in dynamic logic, 8 of which are used for dependency storage and the rest for logic operations [24, 66]. However, even after careful layout design, the transistor density remains double its SRAM counterpart, while non-trivial modifications are required to support the bit count encoding.

In Orinoco, we leverage the Processing-in-Memory (PIM) approach and offer two insights to implement the matrix schedulers efficiently. First, the computation of matrix scheduling matches the element-wise logical operations performed by PIM, rendering the matrix schedulers to be implemented as 8T SRAM arrays. Second, during PIM operations, the bit count encoding can be sensed as the voltage drop on the bit lines, which is detected by adjusting the threshold of sense amplifiers (Section 4.1). Since standard SRAM arrays only support horizontal data access, in addition to a vertical read scheme, a vertical write mechanism based on dual supply voltages is adopted to directly clear the column vector during scheduling (Section 4.2). Finally, in superscalar processors, multiple instructions are dispatched to the matrix schedulers every cycle, which requires multiple write ports of the SRAM arrays. We show how to support multiple memory banks with minimal area overhead without perturbing the total ordering of instructions (Section 4.3).

4.1 Bit Line Computing

PIM dissolves the line that distinguishes memory from computational units by repurposing memory arrays as computational units [21]. It can be implemented by activating multiple word lines simultaneously and sensing on the pair of bit lines to perform element-wise logical operations (e.g., AND, OR, NOR) on the stored data [35]. Taking into account its similarity to the computation of matrix scheduling, the mapping from the matrix schedulers to the memory arrays is straightforward. As demonstrated in Figure 9, each matrix scheduler is implemented as an 8T SRAM array of the exact dimensions, where the dependency information between instructions is stored. To adapt to the bitwise AND and the reduction NOR on the row vector, the read bit lines (RBL) and the read word lines (RWL) are transposed. According to the functionality of each matrix scheduler, the vector indicating rows to reduction (BID/VLD) is applied to the RBLs, while the vector to AND (BID/SPEC) is applied to the RWLs. The RBLs corresponding to the set bits are precharged beforehand, and then the RWLs are activated accordingly. If any of the activated cells stores "1", the RBL will be pulled down, otherwise it will remain at the precharged value. Essentially, the bitwise AND is performed by activating the RWLs while the reduction NOR is performed by precharging the RBLs, hence the results are obtained by sensing the voltage on RBL with a single-ended sense amplifier (SA).

Implementing the bit count encoding as actual counter logic has an overwhelming cost in area and performance. To leverage the same PIM fabric, we observe that the bit count encoding can be sensed as the voltage drop on the bit lines during PIM operations [80]. Since a "0" in any activated cell would discharge the RBL with a current I, currents from all the activated cells in a row flow into the RBL, and the sum of currents leads to the voltage drop. The amount of the voltage drop is proportional to the number of discharge paths, which equals the number of "1" in a row. The results can be sensed by regulating the reference voltage of the SAs, which are determined beforehand according to the issue width of the IQ. As demonstrated in Figure 9, only for RBLs with voltages higher than the threshold, the output will be high, hence the bit count encoding of the age matrix is carried out. Note that unlike computation in the analog domain [41], SAs produce binary results without the severe area and energy overhead incurred by Analog-to-Digital Converters (ADC) [69].

4.2 Vertical Access

Column-wise Write The update of the matrix schedulers requires clearing the particular column of the memory arrays, which challenges standard one-direction SRAM designs that write each row sequentially. Although dual-port SRAM using 10T cells [47] should do the job, we observe that since only zero needs to be written in the column vector (because dependencies are one-way), merely column-wise clear is required instead of full vertical access. Therefore, we adopt a direct column-wise clear mechanism based on dual supply voltages [35], as demonstrated in Figure 10. To clear a column, the write word lines (WWL) are activated, and the write bit line (WBL) and the write bit line bar (WBLB) of the specific column are appropriately driven to write "0". Multiple columns can be cleared at the same time. However, as multiple word lines are activated simultaneously, with the pseudo-write behavior, disturbance may occur between cells in columns that are not under write. To prevent data corruption during the column-wise write,

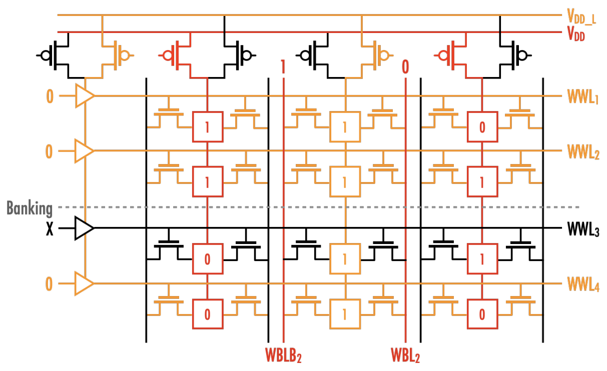

Figure 10: Column-wise write mechanism. To clear the second column, it switches to a lowered supply voltage and the WWLs are under driven.

the WWLs are under driven to weaken the access transistors, and the supply voltage of the cross-coupled inverters in the column to be written is also lowered to allow write with the low WWL voltage. Meanwhile, data in other columns are protected by keeping their supply voltage high.

Column-wise Read To support the column-wise read required by the memory disambiguation and the instruction squash, it can be performed by bitwise ANDing each column vector with the one-hot column select (CS) vector, then reduction OR the result. Similarly, the CS vector is applied to the RWLs, which are activated accordingly. If any cell stores "1", the precharged RBL will discharge. Since this computes the reduction NOR, the sensed result is inverted to get the final read.

4.3 Multibanking

To sustain the pipeline width of superscalar processors, multiple instructions may enter the matrix schedulers every cycle, putting pressure on the write ports of the memory arrays. Since true multi-ported SRAM is too expensive for a large pipeline width, we opt for multibanking that partitions a matrix scheduler into several small arrays (banks), similar to the ROB for a modern superscalar processor [23]. As illustrated in Figure 9 and 10, the SRAM array is divided horizontally into n single-ported banks, where n is the dispatch width of the schedulers. Each instruction is steered to a different bank in a load-balancing manner. Due to the split of WBLs and RWLs, the vectors applied to them during memory operations are broadcast to each bank. The results do not need to be further aggregated since the RBLs remain integrated. Otherwise, the reduction of partial results would greatly complicate the implementation of the bit count encoding.

5 MICROARCHITECTURE OVERVIEW

Figure 11 provides an overview of Orinoco's microarchitecture, with the additional structures relative to a baseline OoO core in the shade. Since they are involved in various pipeline stages, we also outline the changes to the pipeline in Figure 12.

Rename After instructions are fetched and decoded, since out-of-order commit releases the physical registers early, they are renamed

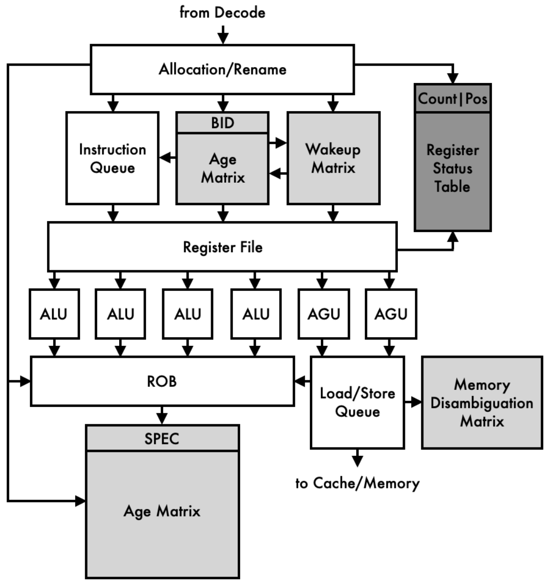

Figure 11: The overview of Orinoco's microarchitecture.

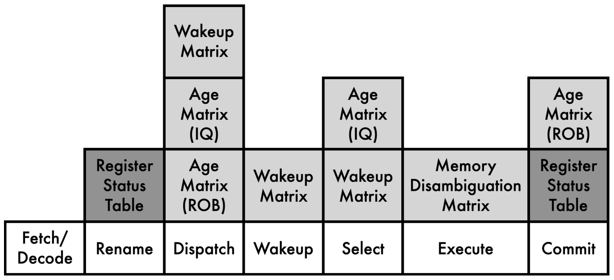

Figure 12: Changes to the pipeline stages.

with a counter-based method [49] to avoid WAR hazards. A register status table (RST) records the producer's position in the IQ and the number of consumers for each physical register.

Dispatch While instructions are dispatched to the IQ and the ROB, the wakeup matrix, the age matrices and the SPEC vector is set. A shortcut may be taken to appropriately handle the dependencies inside instruction groups due to superscalar. Orinoco can be implemented with a unified IQ shared by all the execution units or separate IQs for specific execution units (integer, floating point, memory). The latter divide and conquer the monolithic complexity by decentralizing the wakeup matrix and the age matrix at the cost of capacity efficiency. We assume a unified IQ here. A mask of dispatched instructions in the IQ is maintained for each instruction type, representing all the older instructions, as illustrated in Figure 13. It is used to set the age matrix when an instruction is dispatched and also helps switch the reference voltages of the SAs according to the number of specific execution units, which could be further adjusted regarding their availability. Although this creates a partial ordering of instructions in the IQ, the scheduling order is unaffected since arbitration is only between instructions that require the same execution units.

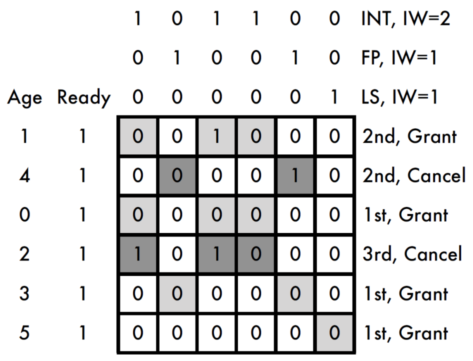
1 0 1 1 0 0 INT, IW=2
0 1 0 0 1 0 FP, IW=1
Age Ready 0 0 0 0 0 1 LS, IW=1
1 1 0 0 1 0 0 0 2nd, Grant
4 1 0 0 0 0 1 0 2nd, Cancel
0 1 0 0 0 0 0 0 1st, Grant
2 1 1 0 1 0 0 0 3rd, Cancel
3 1 0 0 0 0 0 0 1st, Grant
5 1 0 0 0 0 0 0 1st, Grant

Figure 13: Arbitration between the partial ordering.

Issue The wakeup matrix is traversed to discover ready instructions. Once instructions are ready, they assert their request signals in the BID vector and the IQ's age matrix is traversed. Due to the partial ordering, the arbitration for different execution units is performed at once, and up to IW oldest instructions of each type are granted to issue, as demonstrated in Figure 13. These instructions then update the wakeup matrix to wake up the dependent instructions and are removed from the IQ.

Execute Loads and stores set and traverse the memory disambiguation matrix to detect store→load dependences. The count values are decremented when an instruction reads the source registers. As instructions execute, the SPEC vector is updated according to the state in the ROB entries. The ROB's age matrix is accessed under misspeculation for instruction squash.

Commit As instructions finish execution, they are marked in the ROB entries, and the ROB's age matrix is traversed with the SPEC vector. Instructions granted to commit are removed from the ROB. A physical register is reclaimed when the associated logical register has been irrevocably remapped and all the consumers have read its value. The ROB's age matrix is traversed under an exception, and the count value in RST is reset.

6 EVALUATION

6.1 Methodology

Simulation Environment We use the gem5 simulator [10] in syscall emulation mode with the execution-driven cycle-level O3CPU model, where necessary modifications are made to support priority scheduling and out-of-order commit. We use the emergent RISC-V ISA [76], which limits exceptions to floating-point instructions and memory operations, along with the RVWMO memory consistency model, which relaxes the store→load constraint and maintains load→load constraint. The baseline core configurations are modeled after the Intel Skylake microarchitecture [18], as listed in Table 1.

Benchmarks We use the workloads from SPEC CPU2017 benchmark suites [12] and use SimPoint [29] to create checkpoints for representative regions containing 1B instructions for each application.

Memory Modeling We custom design the 8T SRAM arrays on the 28 nm process technology for the matrix schedulers. Compared to traditional 6T SRAM cell, 8T SRAM cell enhances read margins under low supply voltages by decoupling the read and write path [2], hence it is widely used in modern processors. It also allows more word lines to be activated simultaneously during PIM operations, which is critical in the matrix schedulers. The transposition of RBLs and RWLs incurs negligible area penalty since we do not change the cell layout and the push-rule layers remain intact. The sizes and banks of the SRAM arrays are chosen according to the pipeline width of the target processor. The Vref is generated by a self-bias circuit, and the VDD_L is brought in as an additional power supply, similar to providing separate voltages for core and cache via CPU voltage regulators. The robustness of the PIM operations and the column-wise write is verified by the Monte Carlo simulation, showing more than six sigma stability. Detailed memory parameters from the SPICE simulation are listed in Table 2, which we will elaborate on later.

Table 1: Microarchitecture Configurations

Clock Frequency	3.2 GHz		
Branch Predicator	TAGE-SC-L-8KB		
Prefetcher	64 Streams		
L1 Cache	32KB, 8-way, 4-cycle		
L2 Cache	256KB, 8-way, 12-cycle		
LLC	1MB, 16-way, 36-cycle		
Memory	DDR4-2400		
Size	Base	Pro	Ultra
IW/CW	4/4	6/6	8/8
ROB	224	256	512
IQ	97	160	224
LQ/SQ	72/56	128/72	128/72
RF	180	280	380
FU	8	8	11

6.2 Performance

Priority Scheduling We evaluate the IPC improvements of priority scheduling over the following IQ configurations of the baseline core:

- AGE: only the single oldest ready instruction is prioritized
- MULT: the single oldest ready instruction of each type is prioritized
- Orinoco: ideal instruction ordering, where multiple oldest ready instructions are prioritized
- CRI: ideal instruction ordering considering instruction criticality

As shown in Figure 14, Orinoco achieves an average speedup of 6.5%, up to 11.8% over AGE from prioritizing multiple oldest instructions rather than the single one. MULT improves on AGE by preparing multiple age matrices for each type of instructions, yet still incurs 3.2% performance degradation relative to Orinoco due to the persistence of pseudo-ordering within each type of instructions.

Criticality-based Scheduling We also evaluate the impact of priority scheduling on instruction criticality. We use a 64-entry critical count table to identify the most frequent cache missing loads and mispredicted branches, and a 1024-entry instruction slice table

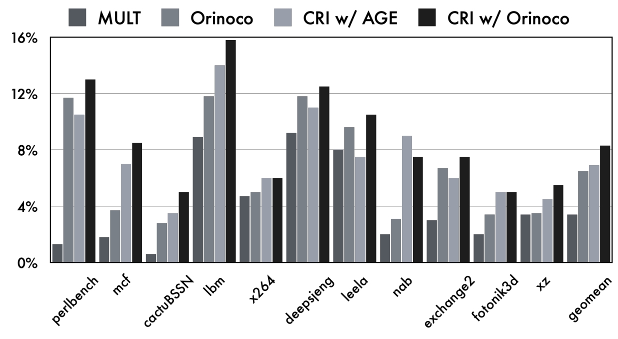

Figure 14: IPC improvements of priority scheduling.

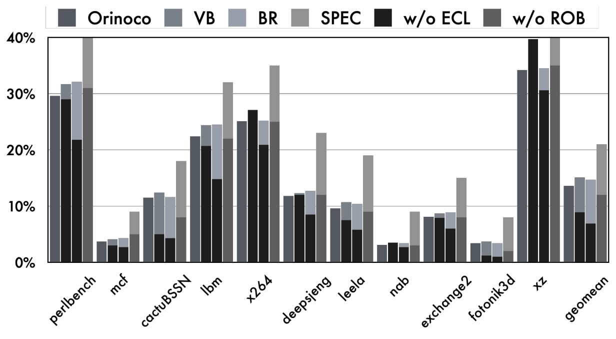

Figure 15: IPC improvements of out-of-order commit.

(IST) to perform IBDA [13] and mark critical instructions, which are prioritized before the non-critical instructions. As labeled by CRI w/ AGE and CRI w/ Orinoco in Figure 14, Orinoco provides an additional 2.1% performance gains in criticality scheduling compared to AGE, which comes from the ideal ordering both inside and between critical and non-critical instructions.

Out-of-Order Commit We evaluate the IPC improvements of out-of-order commit over the following ROB configurations of the baseline core:

- IOC: instructions commit in order
- Orinoco: instructions commit out of order once all the commit conditions are satisfied
- VB: refers to Validation Buffer [49], where non-completed instructions commit when guaranteed to be non-speculative
- BR: refers to NOREBA [27], where instructions commit past oracle branches, represents the upper bound of any solution that relaxes the branch condition
- SPEC: refers to Cherry [50], where oracle instructions commit speculatively without rollback cost, represents the upper bound of speculative out-of-order commit
- ECL: refers to DeSC [28], where non-performed loads commit when guaranteed to be safe, thus only supports weak memory consistency model
- ROB: ROB entries are reclaimed out of order

As shown in Figure 15, Orinoco achieves an average speedup of 13.6%, up to 34.2% over the in-order commit baseline from reducing 65% of full window stalls. Specifically, 67% of ROB exhaustion is unclogged, which contributes to 83% of full window stalls, while LQ is unclogged by 55% and REG is now barely clogged. The most significant difference between Orinoco and previous works lies in the fact that although other resources are recovered out of order, ROB entries are still reclaimed in order because of gap management, which may be clogged by unresolved instructions. The workaround is to release their ROB entries early, yet at the cost of the complexities induced by post-commit speculation (Section 2.2). On the other hand, Orinoco disallows the commit of non-completed instructions but allows the commit of instructions that pass them, which reclaims their ROB entries out of order and achieves 90% performance of VB. We demonstrate the merits of the scheme by simply disabling ECL for VB and BR in the context of a stronger consistency model. As labeled by VB w/o ECL and BR w/o ECL in Figure 15, the performance of both VB and BR is severely degraded by 41% and 53% due to long-latency loads blocking at the head of the FIFO-based ROB. Similarly, Cherry, as labeled by SPEC w/o ROB in Figure 15, reserves ROB entries, thus limiting the amount of further work a core can do. Note that given a limited commit depth, i.e, how far the core can scan to find instructions to commit out of order, reserving ROB entries also hinders reaping the maximum performance benefits of OoO commit. The non-collapsible ROB and the unlimited commit window in Orinoco address these problems, hence it is orthogonal to BR, SPEC, or future designs that relax other conditions to improve performance further.

6.3 Overhead

Performance Table 2 shows the performance of memory operations in the matrix schedulers. Considering the worst case, i.e., ROB's age matrix, we set the clock frequency to 2 GHz to meet the timing constraints of the pipeline stages. Since PIM operations usually activate multiple word lines thus speeding up the bit line discharge, it should be at least as fast as access to SRAM arrays, which is up to 4 GHz in modern processes with better process nodes [31]. Since the matrix schedulers replace some existing logic with close or worse latency, they should fulfill the timing of the critical path.

Area Overhead Table 2 also shows the area overhead of the matrix schedulers. Since vectors for PIM operations are directly applied to word lines/bit lines, no additional decoder except for regular access is required. In addition, since the RBLs are transposed, no duplication of SAs is needed for banking. Both contribute a lot to the overhead of memory peripherals, hence the area efficiency of the banked SRAM arrays is greatly improved.

Power Consumption The power consumption of the matrix schedulers depends on the runtime behavior of the pipeline stages. For example, the number of PIM operations in the IQ's age matrix is determined by the number of ready instructions, while in the ROB's age matrix it is determined by the number of completed instructions and speculative instructions. To accurately estimate the power consumption, we collect statistics from the simulated pipeline and feed them into the SPICE simulation.

Overall, Orinoco incurs 0.3% area overhead and 0.6% power overhead compared to the baseline OoO core measured by McPAT [46] assuming 22 nm technology, which is equivalent to a few kilobytes of SRAM. Compared to the traditional matrix schedulers we implemented with dynamic logic, the PIM-based matrix schedulers

Table 2: Memory Parameters

Parameter	Age Matrix (IQ)	Age Matrix (ROB)	Memory Disambiguation Matrix	Wakeup Matrix
Size	96 x 96	224 x 224	72 x 56	96 x 96
Bank	4	4	4	4
Area	0.0036 mm^2	0.014 mm^2	0.002 mm^2	0.0036 mm^2
Latency	429 ps	493 ps	364 ps	429 ps
Row Write	350 ps	406 ps	305 ps	350 ps
Column Clear	350 ps	406 ps	305 ps	350 ps
Power	0.03 W	0.02 W	0.06 W	0.03 W
VDD = 0.9 V, VDD_L = 0.4 V, Vref = 0.48 V				

reduce the transistor count by a third, coupled with the dense push-rule SRAM cells that double the transistor density, resulting in 3.75 times area reduction under the same size. That is to say, a 224 x 224 PIM-based matrix scheduler is as big as a 96 x 96 traditional matrix scheduler while achieving nearly the same latency. We also observe that when implemented with more common static logic, the timing of the matrix schedulers becomes extremely hard to constrain when the size exceeds 64 x 64 due to the logarithmic complexity of the reduction tree it requires. In addition, we model a theoretical collapsible IQ with 96 entries. Since potentially every entry is being read and written every cycle, it results in a staggering 2.1 W power consumption, which is 70 times the age matrix we proposed. All of the above demonstrate the advantages of the PIM-based matrix schedulers as an alternative to preserving instruction ordering in modern OoO processors.

6.4 Scalability

Sensitivity Study Figure 16 shows the sensitivity of priority scheduling and out-of-order commit and their synergistic performance gains to more aggressive microarchitectures, as listed in Table 1. Overall, Orinoco achieves an average speedup of 14.8% over the baseline with AGE and IOC, up to 25.6% for large cores. Smaller cores are more likely to suffer from issue conflicts and full window stalls due to the lack of core resources, which is advantageous to priority scheduling and out-of-order commit. Although larger cores have fewer issue conflicts in IQ, more intrinsic parallelism is extracted from increased in-flight instructions in ROB, leaving more performance opportunities, especially in criticality scheduling.

Comparing Table 1 and Table 2 shows that most of the matrix schedulers should scale well and meet critical path timing with larger core resources except for the age matrix of a huge ROB, e.g, 512. Although we could add an extra cycle to the commit stage, we find that since the bit count encoding is not required by the ROB's age matrix, its SRAM array could be further split vertically in addition to horizontal banking. The partial results could be reduced with a simple 2-input NOR gate.

Lastly, a scaled OoO core with an area comparable to our Orinoco implementation provides little to no IPC improvement. On the contrary, out-of-order commit utilizes core resources more efficiently, which in fact, increases their effective sizes and allows more instructions to enter the core and be prioritized.

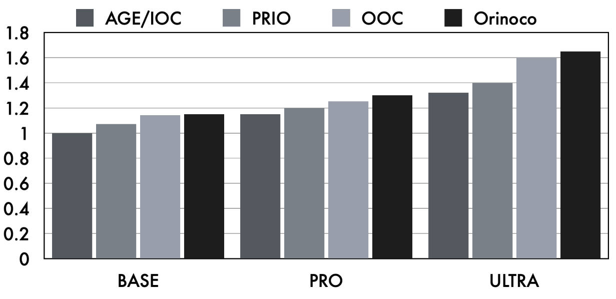

Figure 16: Normalized performance sensitivity.

7 RELATED WORK

In addition to prior works already covered in the paper, criticality-based scheduling and out-of-order commit are also related to dynamic scheduling with in-order cores [13, 36, 37, 43, 44, 51, 75] and runahead execution that sits in between [30, 52–54]. A comparison between the effects of Runahead and Orinoco would be interesting, which we leave for future work. The implementation of Orinoco is inspired by several memory-centric architectures [14, 73, 80].

Priority scheduling, in a broad sense, also involves tasks in ordered parallelism [16, 34, 59, 60], threads in the operating system [32, 57, 61], or requests in network stacks [33, 39, 55]. Whether the goal is to improve performance, throughput, work efficiency, or tail latency, the fundamental trade-off is communication and synchronization overhead versus perfect priority ordering [45, 79], similar to the relation between entropy and energy in physics. We believe it is closely related to the problem we discuss in this paper. Given that the age matrix is essentially a hardware priority queue, we envision pervasive priority scheduling as a potential direction for Orinoco.

ACKNOWLEDGMENTS

This work is supported in part by the National Key Research and Development Program of China (2021YFB3100903, 2018YFB2202101), and in part by the National Natural Science Foundation of China (Grant No. 61834002).

REFERENCES

[1] Jaume Abella Ferrer, Ramon Canal Corretger, and Antonio María González Colás. 2003. Power-and complexity-aware issue queue designs. *IEEE micro* 23, 5 (2003), 50–58.

[2] Amogh Agrawal, Akhilesh Jaiswal, Chankyu Lee, and Kaushik Roy. 2018. X-SRAM: Enabling in-memory Boolean computations in CMOS static random access memories. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 12 (2018), 4219–4232.

[3] Mehdi Alipour, Trevor E Carlson, and Stefanos Kaxiras. 2017. Exploring the performance limits of out-of-order commit. In *Proceedings of the Computing Frontiers Conference.* 211–220.

[4] Mehdi Alipour, Stefanos Kaxiras, David Black-Schaffer, and Rakesh Kumar. 2020. 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, 424–434.

[5] Mehdi Alipour, Rakesh Kumar, Stefanos Kaxiras, and David Black-Schaffer. 2019. FIFOrder MicroArchitecture: Ready-aware instruction scheduling for OoO processors. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 716–721.

[6] Hideki Ando. 2019. SWQUE: A mode switching issue queue with priority-correcting circular queue. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 506–518.

[7] Ashkan Asgharzadeh, Juan M Cebrian, Arthur Perais, Stefanos Kaxiras, and Alberto Ros. 2022. Free atomics: hardware atomic operations without fences.. In *ISCA.* 14–26.

[8] Gordon B Bell and Mikko H Lipasti. 2004. Deconstructing commit. In *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software, 2004.* IEEE, 68–77.

[9] David Bernstein, Michael Rodeh, and Izidor Gertner. 1989. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Transactions on computers* 38, 9 (1989), 1308–1313.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[11] Edward Brekelbaum, Jeff Rupley, Chris Wilkerson, and Bryan Black. 2002. Hierarchical scheduling windows. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.* IEEE, 27–36.

[12] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering.* 41–42.

[13] Trevor E Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The load slice core microarchitecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA).* IEEE, 272–284.

[14] Dibei Chen, Zhaoshi Li, Tianzhu Xiong, Zhiwei Liu, Jun Yang, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. CATCAM: Constant-time Alteration Ternary CAM with Scalable In-Memory Architecture. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 342–355.

[15] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. 2004. Out-of-order commit processors. In *10th International Symposium on High Performance Computer Architecture (HPCA'04).* IEEE, 48–59.

[16] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. Polygraph: Exposing the value of flexibility for graph processing accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 595–608.

[17] Aniket Deshmukh and Yale N Patt. 2021. Criticality Driven Fetch. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture.* 380–391.

[18] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro* 37, 2 (2017), 52–62.

[19] Mark Evers, Leslie Barnes, and Mike Clark. 2022. The AMD Next-Generation "Zen 3" Core. *IEEE Micro* 42, 3 (2022), 7–12.

[20] James A Farrell and Timothy C Fischer. 1998. Issue logic for a 600-mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits* 33, 5 (1998), 707–712.

[21] Daichi Fujiki, Xiaowei Wang, Arun Subramaniyan, and Reetuparna Das. 2021. In-/near-memory Computing. *Synthesis Lectures on Computer Architecture* 16, 2 (2021), 1–140.

[22] Michael Golden, Srikanth Arekapudi, and James Vinh. 2011. 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, 80–82.

[23] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. 2010. Processor microarchitecture: An implementation perspective. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–116.

[24] Mashiro Goshima, Kengo Nishino, Yasuhiko Nakashima, Shin-inchiro Mori, Toshiaki Kitamura, and Shinji Tomita. 2001. A high-speed dynamic instruction scheduling scheme for superscalar processors. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34.* IEEE Computer Society, 225–225.

[25] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. 2021. TIP: Time-Proportional Instruction Profiling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture.* 15–27.

[26] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011).

[27] Ali Hajiabadi, Andreas Diavastos, and Trevor E Carlson. 2021. NOREBA: a compiler-informed non-speculative out-of-order commit processor. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* 182–193.

[28] Tae Jun Ham, Juan L Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 191–203.

[29] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[30] Milad Hashemi, Onur Mutlu, and Yale N Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 1–12.

[31] Min Huang, Moty Mehalel, Ramesh Arvapalli, and Songnian He. 2013. An energy efficient 32-nm 20-mb shared on-die L3 cache for intel® xeon® processor E5 family. *IEEE Journal of Solid-State Circuits* 48, 8 (2013), 1954–1962.

[32] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.* 588–604.

[33] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A Nanosecond Network Stack for Datacenters.. In *OSDI.* 239–256.

[34] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proceedings of the 48th international symposium on microarchitecture.* 228–241.

[35] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. 2016. A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits* 51, 4 (2016), 1009–1021.

[36] Ipoom Jeong, Jiwon Lee, Myung Kuk Yoon, and Won Woo Ro. 2022. Reconstructing Out-of-Order Issue Queue. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 144–161.

[37] Ipoom Jeong, Seihoon Park, Changmin Lee, and Won Woo Ro. 2020. 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, 383–396.

[38] Lizy Kurian John and Vijaykrishnan Narayanan. 2021. Microprocessor at 50: Industry Leaders Speak. *IEEE Micro* 41, 06 (2021), 13–15.

[39] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).* 345–360.

[40] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture.* 158–169.

[41] Mingu Kang, Sujan K Gonugondla, Ameya Patil, and Naresh R Shanbhag. 2018. A multi-functional in-memory inference processor using a standard 6T SRAM array. *IEEE Journal of Solid-State Circuits* 53, 2 (2018), 642–655.

[42] Richard E Kessler. 1999. The alpha 21264 microprocessor. *IEEE micro* 19, 2 (1999), 24–36.

[43] Rakesh Kumar, Mehdi Alipour, and David Black-Schaffer. 2019. Freeway: Maximizing MLP for slice-out-of-order execution. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 558–569.

[44] Kartik Lakshminarasimhan, Ajeya Naithani, Josué Feliu, and Lieven Eeckhout. 2020. The forward slice core microarchitecture. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques.* 361–372.

[45] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2015. Priority queues are not good concurrent priority schedulers. In *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings 21.* Springer, 209–221.

[46] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture.* 469–480.

[47] Zhiting Lin, Zhiyong Zhu, Honglan Zhan, Chunyu Peng, Xiulong Wu, Yuan Yao, Jianchao Niu, and Junning Chen. 2021. Two-direction in-memory computing based on 10T SRAM with horizontal and vertical decoupled read ports. *IEEE Journal of Solid-State Circuits* 56, 9 (2021), 2832–2844.

[48] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: critical slice prefetching. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 300–313.

[49] Salvador Petit Marti, Julio Sahuquillo Borras, Pedro Lopez Rodriguez, Rafael Ubal Tena, and Jose Duato Marin. 2009. A complexity-effective out-of-order retirement microarchitecture. *IEEE Transactions on computers* 58, 12 (2009), 1626–1639.

[50] José F Martínez, Jose Renau, Michael C Huang, and Milos Prvulovic. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings*. IEEE, 3–14.

[51] Daniel S McFarlin, Charles Tucker, and Craig Zilles. 2013. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 241–252.

[52] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. IEEE, 129–140.

[53] Ajeya Naithani, Sam Ainsworth, Timothy M Jones, and Lieven Eeckhout. 2021. Vector runahead. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 195–208.

[54] Ajeya Naithani, Josué Feliu, Almutaz Adileh, and Lieven Eeckhout. 2020. Precise runahead execution. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 397–410.

[55] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads.. In *NSDI*, Vol. 19. 361–378.

[56] Michael L Overton. 2001. *Numerical computing with IEEE floating point arithmetic*. SIAM.

[57] Chandandeep Singh Pabla. 2009. Completely fair scheduler. *Linux Journal* 2009, 184 (2009), 4.

[58] Subbarao Palacharla, Norman P Jouppi, and James E Smith. 1997. Complexity-effective superscalar processors. In *Proceedings of the 24th annual international symposium on Computer architecture*. 206–218.

[59] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 12–25.

[60] Gilead Posluns, Yan Zhu, Guowei Zhang, and Mark C Jeffrey. 2022. A scalable architecture for reprioritizing ordered parallelism. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 437–453.

[61] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-aware thread management. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 145–160.

[62] Alberto Ros, Trevor E Carlson, Mehdi Alipour, and Stefanos Kaxiras. 2017. Non-speculative load-load reordering in TSO. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 187–200.

[63] Alberto Ros and Stefanos Kaxiras. 2018. The superfluous load queue. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 95–107.

[64] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, et al. 2022. Intel Alder Lake CPU Architectures. *IEEE Micro* 42, 3 (2022), 13–19.

[65] Amir Roth. 2005. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 458–468.

[66] Peter G Sassone, Jeff Rupley, Edward Brekelbaum, Gabriel H Loh, and Bryan Black. 2007. Matrix scheduler reloaded. *ACM SIGARCH Computer Architecture News* 35, 2 (2007), 335–346.

[67] Andreas Sembrant, Trevor Carlson, Erik Hagersten, David Black-Shaffer, Arthur Perais, André Seznec, and Pierre Michaud. 2015. Long term parking (LTP) criticality-aware resource allocation in OOO processors. In *Proceedings of the 48th International Symposium on Microarchitecture*. 334–346.

[68] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

[69] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.

[70] Balaram Sinharoy, JA Van Norstrand, Richard J Eickemeyer, Hung Q Le, Jens Leenstra, Dung Q Nguyen, B Konigsburg, K Ward, MD Brown, José E Moreira, et al. 2015. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development* 59, 1 (2015), 2–1.

[71] James E. Smith and Andrew R. Pleszkun. 1988. Implementing precise interrupts in pipelined processors. *IEEE Transactions on computers* 37, 5 (1988), 562–573.

[72] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

[73] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 259–272.

[74] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development* 11, 1 (1967), 25–33.

[75] Kim-Anh Tran, Alexandra Jimborean, Trevor E Carlson, Konstantinos Koukos, Magnus Själander, and Stefanos Kaxiras. 2018. SWOOP: Software-hardware co-design for non-speculative, execute-ahead, in-order cores. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 328–343.

[76] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. 2014. The RISC-V instruction set manual. *Volume I: User-Level ISA', version* 2 (2014).

[77] Vincent M Weaver. 2016. *Advanced hardware profiling and sampling (PEBS, IBS, etc.): creating a new PAPI sampling interface*. Technical Report. Technical Report UMAINE-VMWTR-PEBS-IBS-SAMPLING-2016-08. University of Maine.

[78] Kenneth C Yeager. 1996. The MIPS R10000 superscalar microprocessor. *IEEE micro* 16, 2 (1996), 28–41.

[79] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2019. Understanding priority-based scheduling of graph algorithms on a shared-memory platform. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[80] Jintao Zhang, Zhuo Wang, and Naveen Verma. 2016. A machine-learning classifier implemented in a standard 6T SRAM array. In *2016 ieee symposium on vlsi circuits (vlsi-circuits)*. IEEE, 1–2.