Dependence-aware Slice Execution to Boost MLP in Slice-out-of-order Cores

RAKESH KUMAR, Norwegian University of Science and Technology (NTNU), Norway
MEHDI ALIPOUR, Ericsson Research, Sweden
DAVID BLACK-SCHAFFER, Uppsala University, Sweden

Exploiting memory-level parallelism (MLP) is crucial to hide long memory and last-level cache access latencies. While out-of-order (OoO) cores, and techniques building on them, are effective at exploiting MLP, they deliver poor energy efficiency due to their complex and energy-hungry hardware. This work revisits slice-out-of-order (sOoO) cores as an energy-efficient alternative for MLP exploitation. sOoO cores achieve energy efficiency by constructing and executing *slices* of MLP-generating instructions out-of-order only with respect to the rest of instructions; the slices and the remaining instructions, by themselves, execute in-order. However, we observe that existing sOoO cores miss significant MLP opportunities due to their dependence-oblivious in-order slice execution, which causes dependent slices to frequently block MLP generation. To boost MLP generation, we introduce Freeway, a sOoO core based on a new dependence-aware slice execution policy that tracks dependent slices and keeps them from blocking subsequent independent slices and MLP extrac-

Extension of Conference Paper. The article is an extension of our conference paper entitled "Freeway: Maximizing MLP for Slice-out-of-order Execution," which was presented at the 25th IEEE International Symposium on High Performance Computer Architecture (HPCA 2019) [24]. The key contributions of this article, over the conference version, are as follows:

- Prior work evaluated slice-out-of-order (sOoO) cores only at small instruction windows. This work analyzes their scalability across the full spectrum of window sizes (i.e., from wimpy to brawny cores). We show that sOoO cores are less effective at large (wide/deep) instruction windows, because MLP contributes less to the overall performance as the window size grows. This analysis, together with the fact that Freeway captures the bulk of available MLP, implies that the future research in scaling sOoO cores should focus on ILP rather than capturing the missed MLP opportunities. (Section 6.6)
- We assess the resource utilization and L1 cache latency tolerance of Freeway over the previous state-of-the-art, Load Slice Core (LSC). We show that Freeway requires one-sixth as many instruction queue entries and can tolerate 1.5× higher L1 latencies while providing same or better performance than LSC. These results imply that Freeway is a better choice than prior sOoO cores in resource constrained environments. (Sections 6.4 and 6.5)
- We analyze the contribution of different sources of MLP, i.e., Freeway, our proposed core design, and LLC prefetcher. Our results show that they mostly complement each other, although Freeway generates much more MLP than the LLC perfetcher. These results also imply that in area constrained designs it is better to invest area in Freeway rather than in an LLC prefetcher. (Section 6.3)

This work was done while Mehdi Alipour was at Uppsala University.
This work was supported by the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant No 715283), and the Research Council of Norway (NFR) Grant No. 302279 to NTNU.
Authors' addresses: R. Kumar, Norwegian University of Science and Technology (NTNU), Sem Sælands vei 9, 7034 Trondheim, Norway; email: rakesh.kumar@ntnu.no; M. Alipour, Ericsson Research, Mobilvägen 12, 22362 Lund, Sweden; email: mehdi.alipour@ericsson.com; D. Black-Schaffer, Uppsala University, Lägerhyddsvägen 2, hus 1 75237 Uppsala, Sweden; email: david.black-schaffer@it.uu.se.

1544-3566/2022/03-ART25 $15.00
https://doi.org/10.1145/3506704

tion. The proposed core incurs minimal area and power overheads, yet approaches the MLP benefits of fully OoO cores. Our evaluation shows that Freeway delivers 12% better performance than the state-of-the-art sOoO core and is within 7% of the MLP limits of full OoO execution.

CCS Concepts: • **Computer systems organization → Architectures;**

Additional Key Words and Phrases: Microarchitecture, memory level parallelism, instruction scheduling

ACM Reference format:
Rakesh Kumar, Mehdi Alipour, and David Black-Schaffer. 2022. Dependence-aware Slice Execution to Boost MLP in Slice-out-of-order Cores. *ACM Trans. Arch. Code Optim.* 19, 2, Article 25 (March 2022), 28 pages.
https://doi.org/10.1145/3506704

1 INTRODUCTION

Today's power-constrained systems face challenges in generating memory-level parallelism (MLP) to hide the increasing access latencies across the memory hierarchy [48]. Historically, memory latency has been addressed through multilevel cache hierarchies to keep the frequently used data closer to the core. While cache hierarchies provide lower-latency in L1 caches, they have grown in complexity to the point where the 40–60 cycles it takes to access the last-level cache has itself become a bottleneck. Therefore, exploiting MLP, by overlapping cache/memory accesses to hide the latency of later requests in the "shadow" of earlier requests, across the entire hierarchy is crucial for performance. However, the traditional approaches to extract MLP, such as out-of-order (OoO) or run-ahead execution, are not energy-efficient.

The standard means of extracting MLP is OoO execution, as it enables parallel memory accesses by executing independent memory instructions from anywhere in the issue window. However, the ability to identify, select, and execute independent instructions in an arbitrary order, while maintaining program semantics, requires complex and energy-hungry hardware structures. For example, one of the key enablers of OoO execution, the OoO issue queue, is typically built using content addressable memories (CAMs), whose power consumption grows super-linearly with queue depth and issue width.

State-of-the-art MLP extraction techniques aim to improve performance by increasing the amount of MLP extraction beyond the OoO execution. However, they fail to deliver energy efficiency primarily, because they build upon already energy-hungry OoO execution and further introduce significant additional complexity of their own. For example, Runahead Execution [33], which was originally proposed to improve the data cache performance in in-order cores [11], continues to extract MLP after an OoO core stalls, but requires additional resources for checkpointing and restoring states, tracking valid and invalid results, psuedo-instruction retirement, and a runahead cache. This additional complexity entails a significant energy overhead.

To minimize the energy cost of MLP exploitation, a new class of cores, called *slice-out-of-order* (sOoO) cores, builds on energy efficient in-order execution and adds just enough support for MLP extraction. These cores first construct groups, or *slices*, of MLP-generating instructions. A slice consists of one memory access instruction, i.e., load or store, and all the instructions required for its address generation. The slices and non-slice instructions are then dispatched to and scheduled from separate in-order instruction queues (IQs). As the instructions in one queue can bypass the instructions in the other queue, the slices and non-slice instructions execute out-of-order with respect to each other. However, by themselves, they still execute in-order as younger instructions cannot bypass the older instructions in the same in-order queue. Thus, by enabling MLP-generating slices to bypass the rest of the potentially stalled instructions, sOoO cores extract significant MLP. Yet, since they only support limited, coarser-grained out-of-order execution, they incur only a fraction of the energy cost of full, per-instruction out-of-order execution.

The state-of-the-art sOoO core, the Load Slice Core (LSC) [6], builds on an in-order stall-on-use core. LSC identifies MLP-generating instructions using a small hardware table. To enable these instructions to execute out-of-order with respect to the rest of the instructions, LSC adds an additional in-order instruction queue, called the bypass queue (B-IQ). By restricting the out-of-order execution to choosing between the heads of two in-order instruction queues (the main, or A-IQ, and the bypass B-IQ), LSC minimizes energy requirements while still providing MLP.

Though sOoO cores are highly energy efficient, they fall noticeably behind OoO cores in terms of MLP extraction. Our key observation is that *dependent slices limit MLP extraction opportunities.* We define a dependent slice to be the one that contains at least one instruction that depends on the load instruction of another slice, called *producer slice*. Dependent-slices limit MLP opportunities, because, for example, when a dependent slice reaches the head of the in-order B-IQ in LSC, it blocks any further MLP generation by stalling the execution of subsequent, possibly independent, slices until the load instruction of its producer slice receives data from the memory hierarchy. Our analysis reveals that, in LSC, dependent slices block MLP generation for up to 83% of the execution time (average 23%). More importantly, the MLP loss is not just caused by the long stalling dependent slices whose producers miss in the on-chip caches. We demonstrate that, counter-intuitively, the dependent slices cause significant MLP loss even if they only stall for a few cycles: our results show that about 65% of the dependent slice-induced MLP loss is caused by slices whose producers hit in the L1 cache. Together, these results demonstrate that dependent slices are a serious bottleneck in LSC.

This work addresses the fundamental limitation of the state-of-the-art sOoO core's ability to extract MLP: *dependence-oblivious* first-in first-out (FIFO) slice execution causes dependent slices to delay the execution of subsequent independent slices. We propose to abandon the FIFO slice execution model in favor of a *dependence-aware* slice scheduling model. Our proposed model tracks slice dependencies in hardware to identify dependent slices and steers them out of the way of the independent ones. As a result, the independent slices execute without stalling and expose more MLP.

To achieve this, we introduce Freeway, an energy efficient core design powered by a dependence-aware slice scheduling policy for boosting MLP and performance. Freeway tracks inter-slice dependencies with minimum additional hardware (one bit per entry in Register Dependence Table) to filter out the dependent slices. These slices are then steered to a new in-order queue, called the yielding queue (Y-IQ), where they wait until their producers finish execution. Such slice segregation clears the way for independent slices to generate more MLP as they no longer stall behind the dependent slices. Overall, Freeway delivers a substantial MLP boost by unblocking independent slice execution with minimal additional hardware resources. Our main contributions include:

- Identifying that the dependence-oblivious FIFO slice execution is a major bottleneck to MLP generation in existing sOoO cores. We further demonstrate that dependent slices limit MLP even if they stall only for a few cycles (i.e., their producers hit in the L1 cache).
- Proposing a new dependence-aware slice execution policy that executes independent slices unobstructed by tracking and keeping the dependent slices out of their way, hence boosting MLP.
- Introducing the Freeway core design that employs minimal additional hardware to implement the dependence-aware slice execution: one bit per entry in Register Dependence Table, 7-bits per entry in Store Buffer, a FIFO instruction queue, and some combinational logic.
- Demonstrating that Freeway provides 12% more performance than the state-of-the-art sOoO core and is within 7% of the MLP limits of full OoO execution. We also analyze the

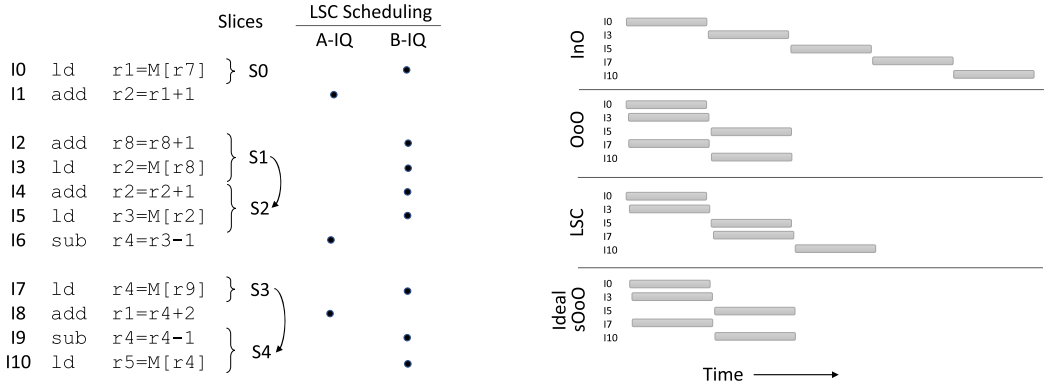

Fig. 1. Overlapping memory accesses in InO, OoO, LSC, and Ideal sOoO. The arrows show inter-slice dependencies.

remaining bottlenecks that cause this 7% performance gap and show that mitigating them brings minimal performance returns on resource investment.

- Showing better resource utilization and latency tolerance for Freeway than LSC. Freeway needs six times fewer IQ entries and tolerates 1.5× higher L1 cache latency while providing better or same performance as LSC.
- Analyzing the scalability of sOoO cores over the full spectrum of instruction window sizes from wimpy to brawny cores. We show that sOoO cores are less effective at large (wide/deep) instruction windows, because the MLP contributes less to overall performance as the window size grows. This result, along with the fact that Freeway captures the bulk of MLP opportunity, implies that the future research in scaling sOoO cores should focus on ILP rather than capturing the missed MLP opportunities.

2 BACKGROUND AND MOTIVATION

2.1 MLP Versus Energy: InO, OoO, and Slice-OoO Cores

Existing core designs force a trade-off between MLP and energy efficiency. For example, an in-order (InO) core can be highly energy efficient, but it is unable to generate significant MLP, and therefore delivers poor performance. In contrast, OoO cores are generally good at extracting MLP, but at the cost of (much) lower energy efficiency. To exploit MLP while delivering high energy efficiency, a recent design, the LSC [6], proposed a new approach of sOoO execution. LSC builds on an efficient in-order core and employs separate instruction queues, AIQ and B-IQ, for non-MLP and MLP-generating instructions, respectively. This enables MLP-generating instructions in the B-IQ to bypass the potentially stalled load consumers in the A-IQ. By exposing MLP in this way, LSC avoids much of the complexity of full OoO architectures.

MLP Extraction: Figure 1 shows how the sOoO execution of LSC fairs against InO and OoO cores in exploiting MLP. As a stall-on-use InO core stalls on the first use of the value being loaded from memory, it serializes all the loads in this example, resulting in no MLP. The OoO core is able to extract the maximum MLP by overlapping the execution of independent load instructions (I0, I3 and I7). When these loads' data returns, their dependent loads (I5 and I10) are also overlapped. The sOoO execution of LSC falls between the InO and OoO cores. LSC overlaps the execution of the first two load instructions (I0 and I3) as the B-IQ enables I2 and I3 to bypass the stalled instructions (I1) in the A-IQ.

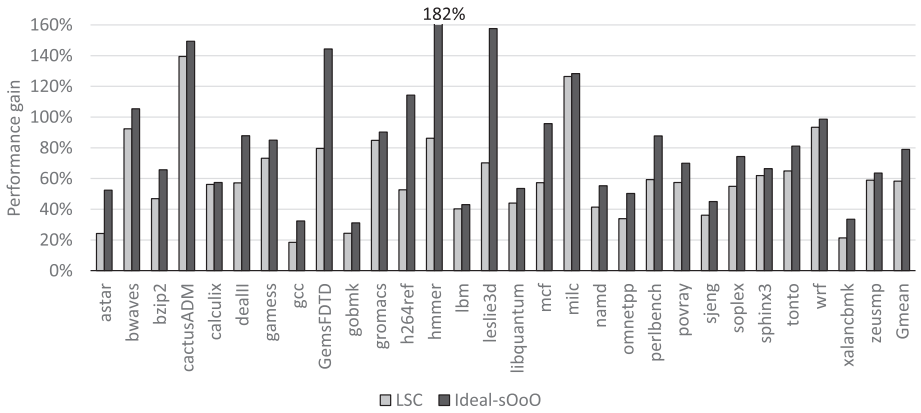

Fig. 2. Performance gain for LSC and Ideal-sOoO over InO execution.

Figure 1 also demonstrates a major limitation of LSC: it is effective in extracting MLP only when the *slices are independent*. A dependent slice at the head of the B-IQ stalls MLP extraction by blocking the execution of the subsequent independent slices. In Figure 1, slice S2, a dependent slice as it depends on the load in S1, stalls the B-IQ and delays the execution of the next independent slice S3 until its producer slice S1 receives data from the memory hierarchy. Such slice dependencies limit MLP and overall performance. However, an Ideal sOoO core, that allows fully out-of-order execution among slices, would eliminate this limitation. As shown in Figure 1, an ideal sOoO core matches the MLP generation of a full OoO core.

Energy Consumption: LSC's sOoO execution is implemented using simple hardware components: FIFO queues and small tables for tracking slices. As a result, it only slightly increases the area and power consumption compared to an already small InO core. In contrast, the energy requirements of OoO cores are substantially higher due to the use of complex structures, such as CAMs. Indeed, previous research [35] has shown that the ability to select arbitrary instructions from IQ is one of the most energy consuming tasks in OoO cores. Carlson et. al. [6] concluded that LSC incurs only 15% area and 22% power overheads over an InO core (ARM Cortex-A7), whereas an out-of-order core (ARM Cortex-A9) requires 2.5× area and 12.5× power compared to the same in-order core.

The slice-out-of-order execution in LSC is a promising step towards energy efficient MLP extraction. However, LSC's strict FIFO execution of slices limits its potential to extract MLP in the case of dependent slices. To understand this limitation, we next explore its impact on performance.

2.2 Potential for MLP Extraction

To quantify the potential MLP available in a sOoO core, we compare LSC, with its in-order B-IQ, to a LSC with a fully out-of-order B-IQ (Ideal-sOoO). While an out-of-order B-IQ would be impractical (it would defeat the efficiency goal of avoiding the complexity of out-of-order instruction selection), it allows us to observe the maximum MLP gains possible if the independent slices can bypass other stalled slices. (Our simulation methodology, including microarchitectural parameters, is detailed in Section 5.)

Figure 2 shows the performance gains obtained by LSC and Ideal-sOoO (LSC with a fully out-of-order B-IQ) over an InO core. The relative difference between the two shows the opportunity missed by LSC due to its FIFO slice execution. As the figure shows, the performance difference between Ideal-sOoO and LSC is 20%, geometric mean, and more than 50% on `GemsFDTD`, `h264ref`,

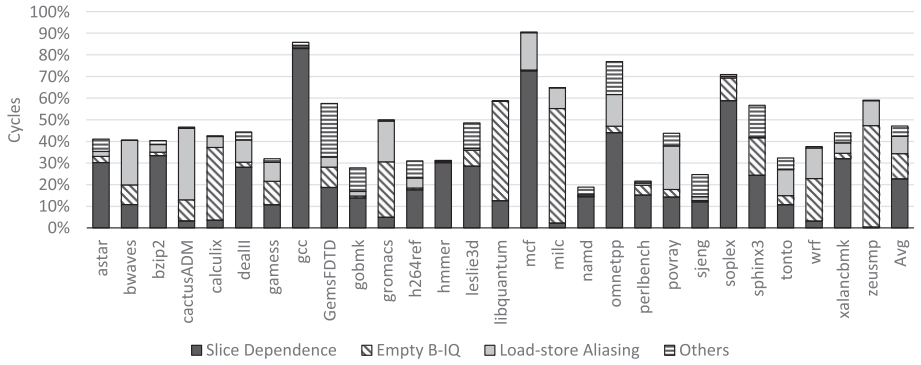

Fig. 3. Percentage of execution time the issue stage is stalled in LSC and the breakdown of stall sources.

`hmmer`, and `leslie3d`, due to relatively larger numbers of dependent slices. However, there is little gain for `calculix`, `lbm`, and `milc`, as they have fewer dependent slices (see Section 2.3). Overall, the majority of the workloads demonstrate considerable performance opportunity if we can eliminate the dependent slice bottleneck.

2.3 Sources of Stalls in the Bypass Queue

For a deeper understanding of the microarchitectural bottlenecks limiting MLP extraction in LSC, we examine the stall sources afflicting the B-IQ and categorize them as follows:

- **Slice Dependence Stalls:** A dependent slice at the B-IQ head is waiting for its producer to receive data from the memory hierarchy.
- **Empty B-IQ Stalls:** There are no instructions (slices) in the B-IQ.
- **Load-store Aliasing Stalls:** A load at B-IQ head cannot be issued, because an older store in the A-IQ is waiting to write to the same address (true alias). This might happen because, in LSC, store data calculations and the store operations themselves go to the A-IQ, whereas, the store address calculation goes to the B-IQ.
- **Other Stalls:** Intra-slice dependencies, unresolved store addresses blocking younger loads, and so on.

For this study, we assume an ideal core front-end (no instruction cache or BTB misses and a perfect branch predictor) to isolate the slice execution bottlenecks.

Figure 3 shows the breakdown of stall cycles (when no instruction is issued from either the A-IQ or the B-IQ) as a fraction of overall execution time. The figure reveals that instruction issue is stalled for 47% of the execution time on average, and *Slice Dependence Stalls* are responsible for almost half of these stalls. The *Slice Dependence Stalls* are particularly significant in `gcc`, `mcf`, `soplex`, and `hmmer`, where they account for more than 80% of all stalls. Notice that `gcc` and `mcf` are the most severely affected workloads, yet they are not the ones that show the highest performance opportunity with Ideal sOoO execution (Figure 2). The reason is that the performance opportunity is a function of not only the number of stalls caused by dependent slices but also where their producer slices hit in the memory hierarchy. As shown in Figure 4, the majority of producer slices, in `gcc` and `mcf`, miss in the on-chip cache hierarchy and must be loaded from memory. This long memory latency stalls instruction retirement, and therefore causes the instruction window to fill, which blocks further MLP generation and limits performance. In this work, we use the term instruction window to refer to the window of all in-flight instructions, i.e., the instructions that have been

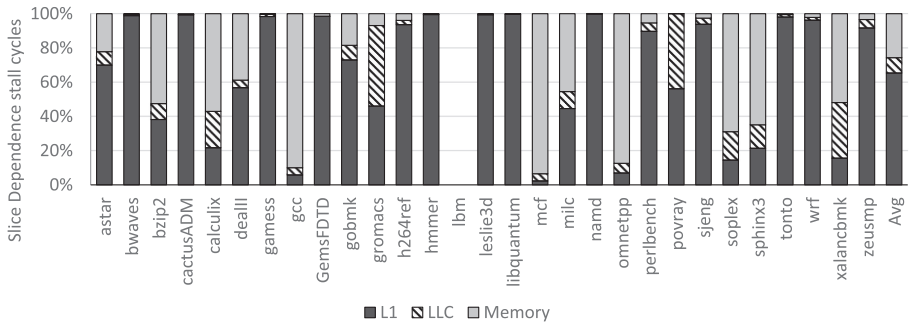

Fig. 4. Breakdown of *Slice Dependence Stall* cycles based on the producer slice hit site in the memory hierarchy. `lbm` does not have any dependent slices.

dispatched but not yet committed. Scoreboard and reorder buffer (ROB) are the typical hardware implementations of instruction window in sOoO and OoO cores, respectively.

Empty B-IQ Stalls are the second largest source of stalls. We observe that the primary reason for the B-IQ to be empty is a full instruction window and the oldest instruction is not ready to retire. As a result, no new instructions can enter either instruction queue. This could be remedied through larger instruction windows or methods such as Runahead Execution [33]. The third largest source of stalls are *Load-store Aliasing Stalls*, and they are particularly severe in `bwaves`, `cactusADM`, `gromacs`, `lbm`, and `povray`. However, they account for only 8% of execution time.

Looking at the potential of an Ideal-sOoO, we see that LSC misses about 20% performance opportunity. The majority of this loss is due to Slice Dependence Stalls. Next, we analyze memory slice behaviour to mitigate this bottleneck.

3 ADDRESSING SLICE DEPENDENCE

A generic approach to handle dependent slices is to get them out of the way of the independent slices by buffering them outside of the B-IQ. To this end, we next study the slice behaviour to understand which dependent slices should be buffered and where they should be buffered.

3.1 Which Dependent Slices to Buffer?

Intuitively, only the dependent slices that stall the B-IQ for many cycles need to be buffered. Such long stalls are typically due to slice's producers hitting in the LLC or memory. However, unintuitively, we found that 65% of the *Slice Dependence Stalls* are caused by dependent slices that stall only for a few cycles as their producers *hit* in the L1 cache. This demonstrates that even the relatively short L1 hit latency (four cycles in our simulation) can significantly limit the MLP and performance in a sOoO core with strict FIFO slice execution.

Figure 4 shows the breakdown of *Slice Dependence Stall* cycles based on the producer slice hit site in the memory hierarchy. The results are especially interesting for workloads such as `hmmer`, where producer slices almost always hit in the L1 cache, and yet dependent slices are responsible for more than 96% of all stall cycles, which accounts for about 31% of the execution time (Figure 3).

These results suggest that it is important to buffer all dependent slices, even those that only stall for the duration of an L1 hit. Interestingly, this also suggests that in many cases, we only need to buffer the dependent slices for a few cycles (to cover L1 latency) to achieve much of the MLP benefit. If such limited buffering is sufficient, then it would suggest we can achieve these benefits at a low implementation cost.

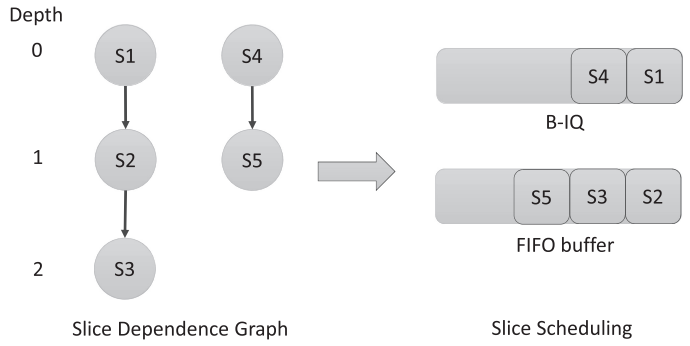

Fig. 5. Slice dependence graph with dependence depth (left) and slice scheduling to different queues (right). S1–S5 are slices.

3.2 Where to Buffer?

To mitigate the slice dependence bottleneck, the dependent slices need to be kept in a separate buffer to prevent them from stalling the B-IQ. However, traditional instruction buffers, such as the Waiting Instruction Buffer [27], are complex, energy intensive, and are designed to buffer instructions for longer time intervals, such as during LLC misses. In addition, those designs require the instructions to be inserted back to the main IQ before issuing them for execution [27, 38]. The extra energy and latency of re-inserting instructions is particularly costly for instruction slices that will only be buffered for a few cycles.

A simple FIFO queue is an attractive alternative instruction buffer due to its low complexity and energy cost. However, as instructions can only be picked from the head of the FIFO queue, buffering all dependent slices in a single queue might cause a bottleneck if the younger slices become ready for execution before the older slices. This occurs for primarily two reasons: first, if a younger slice has fewer slices before it in its dependence chain than a slice in an older chain; or, second, if the producer of a younger slice hits closer to the core in the memory hierarchy than the producer of an older slice. To understand the implications of these effects, we analyze potential stall sources to determine if a single, cheap, FIFO queue is appropriate for buffering dependent slices.

Slice dependence depth: Slices further down their dependence chains can potentially stall the execution of slices in a younger chain. To better understand this, we define the *dependence depth* of a slice as the number of slices in the dependent slice chain leading up to it. For example, in Figure 5, S1 and S4 are independent slices and start the dependent slice chain, hence their dependence depth is 0. Next, S2 and S5 are at dependence depth 1 as they have one slice ahead of them, S1 and S4, respectively.

Using this definition, we observe that a younger slice with a lower dependence depth is likely to become ready before an older slice with higher dependence depth. For example, in Figure 5, S3 (depth 2) will be ready only after both S2 and S1 have received their data, whereas S5 (depth 1) needs to wait only for S4. If all the slices hit at the same level in memory hierarchy, leading to similar execution times, then S5, the younger slice, will be ready for execution before S3. However, it will be stalled behind S3 in the FIFO queue, thereby limiting MLP extraction.

To understand the potential bottleneck due to such stalls, we study the slice dependence depth in our workloads in Figure 6. As the figure shows, 78% of all slices are independent slices (depth 0) and do not need to be buffered. Of the remaining slices that do need to be buffered, more than 72% are at dependence depth 1. Therefore, as the majority of dependent slices are at the smallest

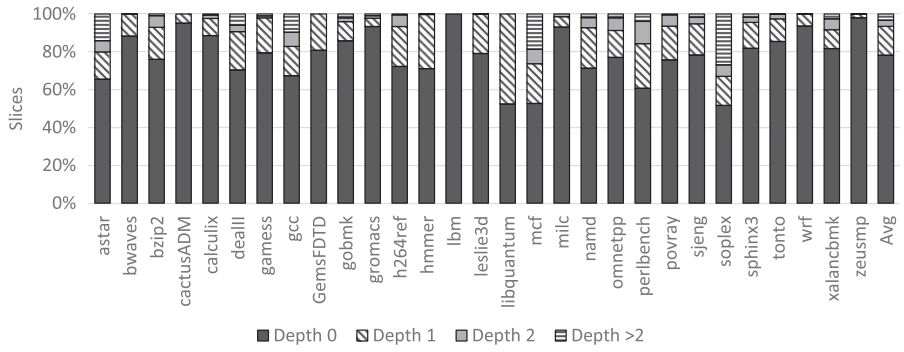

Fig. 6. Slice dependence depth distribution.

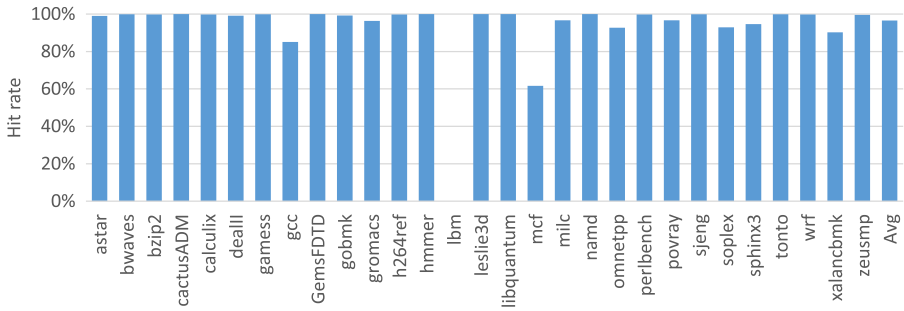

Fig. 7. L1 cache hit rate for producer slices at dependence depth 0. `lbm` does not have any dependent or producer slices.

depth of 1, the stalls caused by slices at larger dependence depths (6% of all slices) are likely to be minimal.

Producer slice hit site: Even if dependent slices are at the same dependence depth, a younger slice can still become ready earlier than an older slice if its producer hits closer to the core in the memory hierarchy than the producer of the older slice. For the example in Figure 5, S2 and S5 both are at dependence depth 1, but S5 may become ready earlier if its producer S4 hits in L1 and S2's producer S1 hits in LLC or farther. In this scenario, S5 will be stalled as S2 is blocking the head of the FIFO queue.

To understand the extent of the potential bottleneck, we study the hit site of producer slices with at least one dependent slice. For this study, we only consider the producer slices at dependence depth 0, because the majority of dependent slices are at depth 1 (i.e., dependence chain lengths of two slices). Figure 7 shows that more than 96% of these producer slices hit in the L1 cache. Therefore, as the majority of producer slices hit at the same level, L1, the dependent slices are likely to become ready in the program order, and, hence, incur minimal stalls due to ready younger slices waiting behind stalled older ones.

Overall, Figures 6 and 7 suggest that a single FIFO queue for all dependent slices is sufficient to expose most of the potential MLP available from out-of-order slice execution. This is because most of the dependence slices are at the dependence depth one and the majority of producer slices hit in L1, indicating that it is unlikely that dependence slices will stall behind each other. (Results in Section 6.2 validate that additional queues bring only minimal performance gains).

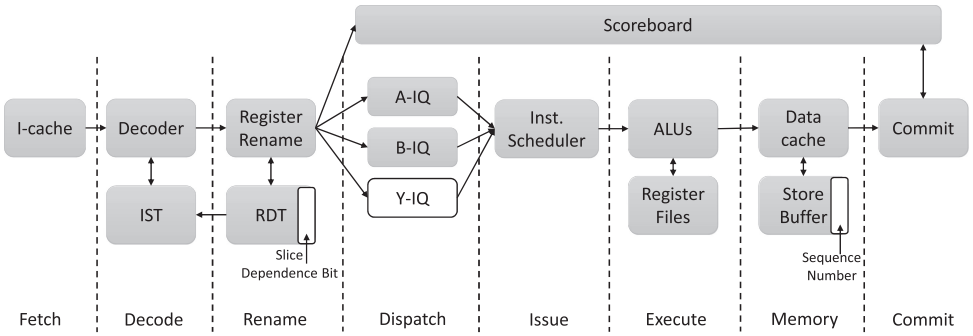

Fig. 8. Freeway microarchitecture. New components are in white; Instruction scheduler issues from Y-IQ as well.

4 FREEWAY

Freeway is a new sOoO core designed to achieve the MLP benefits of full out-or-order execution. Freeway goes beyond prior work by addressing the sources of sOoO stalls identified in our analysis (Section 3) to execute the majority of slices without stalling on dependent slices. Freeway requires only small changes over the baseline sOoO design (LSC), and thereby retains its low complexity. As a result, Freeway is able to substantially increase the exposed MLP and performance, while retaining a simple, and energy efficient design.

An overview of the Freeway microarchitecture is presented in Figure 8. The components common to both LSC and Freeway, such as the B-IQ, are shown in light gray. The additions required for Freeway are in white. As Freeway builds upon LSC, we first describe the baseline LSC microarchitecture before providing an overview of the Freeway design and, finally, a detailed discussion of the key design issues.

4.1 Baseline sOoO

The first sOoO design, the LSC, builds upon an energy efficient in-order, stall-on-use core. LSC identifies MLP-generating instructions (slices) in hardware and executes them early (via the B-IQ) with minimum additional resources, thereby achieving both MLP and energy efficiency.

Slice construction: To identify MLP-generating instructions efficiently, LSC leverages applications' loop behaviour to construct memory slices using iterative backward dependency analysis (IDBA) [6], starting with the memory access instructions. In each loop iteration, the producers of the instructions identified in the previous iteration are added to the slice. LSC identifies the producers of an instruction through the Register Dependence Table (RDT), which maps each physical register to the instruction that last wrote to it. LSC then uses a simple PC-indexed Instruction Slice Table (IST) to track the instructions in slices. By building up slices via simple RDT look-ups over multiple loop iterations, LSC avoids the complexity and energy overheads of explicit slice generation techniques [8, 32].

Figure 1 shows an example of slice construction. As soon as the decoder detects a load or store instruction, it starts slice construction. The construction of slice S1 in this figure starts with the load instruction I3. Once the load is detected, IBDA consults RDT to find its producers. In this case, RDT will report I2 to be a producer as it wrote to the input register, r8, of the load instruction I3. Further, I2 will be inserted into the IST as it has been identified to be a part of the slice. During the next iteration, I2 will hit in IST and IBDA will consult RDT to find its producers. However, I2 does not have any producers in the instruction window as its operands are already available when it is

decoded. Therefore, IBDA will stop backwards traversal implying completion of slice construction. Notice that a slice includes only data flow but not the control flow required to generate the address for memory access instruction.

Slice Execution: To exploit MLP, LSC adds an additional in-order *bypass queue (B-IQ)* to the baseline InO core. The MLP-generating instructions (slices), as identified by IST, are dispatched to the B-IQ, which enables them to bypass the instruction in the main instruction queue (A-IQ). For store instructions, only the address calculation is dispatched to the B-IQ so that their addresses are available earlier, and subsequent loads can be disambiguated. The data calculation and the store operation itself proceed to the regular instruction queue (A-IQ), as they rarely limit MLP. By allowing such limited out-of-order execution, MLP-generating instructions can bypass (via the B-IQ) stalled instructions in the main instruction flow (the A-IQ). As a result, the sOoO execution of the LSC allows it to extract considerable MLP, without the energy cost of the full out-of-order execution.

4.2 Freeway: Overview

While LSC is effective in exploiting MLP when the memory slices are independent, dependent slices cause a serious bottleneck due to LSC's strict FIFO slice execution. As LSC mixes dependent slices with the independent ones in the B-IQ, it limits MLP by delaying the execution of independent slices stalled behind the dependent ones. To increase MLP, Freeway abandons LSC's FIFO slice execution and allows independent slices to execute out-of-order with respect to dependent ones with minimum additional hardware.

To enable out-of-order execution among slices, Freeway *tracks* slice dependencies in hardware and *separates* dependent slices from independent ones. Freeway uses the slice dependency information to *accelerate* independent slice execution by reserving the B-IQ exclusively for them. To handle the dependent slices, Freeway introduces a new FIFO instruction queue called the *yielding queue* (Y-IQ). Dependent slices can then wait in the Y-IQ, yielding execution to the independent slices, until they become ready for execution. As our analysis in Section 3 demonstrated, the dependent slices mostly become ready in program order, therefore, ready dependent slices should rarely stall behind the non-ready ones. This characteristic allows us to use simple hardware to boost MLP by executing the majority of slices from both the B-IQ and Y-IQ without any stalls.

Freeway requires only minimal additional hardware, as shown in Figure 8, to support out-of-order slice execution: extending the RDT entries with one bit to track whether an instruction belongs to a dependent slice; adding a FIFO instruction queue (Y-IQ) for dependent slices; extending each store buffer entry with 7 bits and comparators to maintain memory ordering; and adding logic to issue instructions from the Y-IQ in addition to from the A-IQ and B-IQ.

4.3 Freeway: Details

We first describe the mechanism for tracking slice dependence and then provide details of instruction flow through Freeway, before discussing memory ordering requirements.

4.3.1 Tracking Dependent Slices. We classify a memory slice as a dependent slice if it contains at least one instruction that depends on the *load instruction* of an older slice. However, a slice is not classified as dependent if it depends on a non-load instruction of a older slice. Freeway detects dependent slices in the *Register Rename* stage by leveraging the existing data dependence analysis of the baseline core. These analyses are required by LSC to identify the instructions belonging to memory slices. The first instruction of a dependent slice can be identified trivially using the data dependence analysis as it is the instruction that receives at least one of its operands from a

load instruction. Identifying the remainder of the dependent slice instructions is more involved as they may not be directly dependent on the load. Therefore, the dependence information must be propagated from the first dependent instruction to the memory access instruction terminating the slice.

Freeway extends LSC's RDT with a *slice dependence bit* to propagate the dependence information through a slice. The dependence bit indicates whether the instruction reading a register would belong to a dependent slice or not. Initially the slice dependence bits are 0 for all RDT entries. When Freeway detects a load instruction, it sets the slice dependence bit of its destination register's RDT entry to 1, as any slice instruction reading this register would belong to a dependent slice. Subsequently, if the slice dependence bit for any of the source registers of an instruction is found to be 1, the instruction is marked as a dependent slice instruction. In addition, the dependent slice bit of its destination register's RDT entry is set to 1. This propagates the dependence information through the slice. As such, Freeway only requires 1 additional bit per RDT entry to identify the chain of dependent instructions constituting a dependent slice.

4.3.2 Instruction Flow Through Freeway. **Front-end:** The Freeway front-end is very simliar to that of LSC, but with the addition of dependent slice identification and tracking. As with LSC, after instruction fetch and pre-decode, the IST is accessed with the instruction pointer to check if an instruction belongs to a memory slice or not. This information is propagated down the pipeline to assist instruction dispatch. Next, register renaming identifies true data dependencies among instructions so that dependent instructions wait until their producers finish execution. At this point Freeway consults the RDT to determine if a memory slice instruction also belongs to a dependent slice, and passes on this information to the dispatch stage.

Instruction Dispatch: Freeway dispatches an instruction to one of the three FIFO instruction queues (A-IQ, B-IQ, or Y-IQ) based on the slice and dependence information received from the IST and RDT. Loads, stores, and their address generating instructions, as identified by the IST are dispatched to the B-IQ if they belong to *independent* memory slices. In contrast, if the RDT classifies them as part of a dependent slice, they are dispatched to the Y-IQ, where they wait until their producer slices finish execution. The rest of the instructions are dispatched to the A-IQ. For Stores, as with LSC, the data calculation and the store operation itself are dispatched to the A-IQ. Whereas the address calculation goes to either the B-IQ or Y-IQ, based on its dependence status. Such split dispatching for stores enables their addresses to be available early so that the subsequent loads can be disambiguated against them and continue execution, if they access non-overlapping memory locations.

Back-end: The Freeway back-end selects instructions for execution from its three IQs (i.e., A-IQ, B-IQ, and Y-IQ), executes them, and ensures that they update architectural state in program order. As Freeway employs FIFO instruction queues, only the instructions at their heads can be scheduled for execution. If multiple IQs have ready instructions at their heads, then instructions are scheduled using an age-based policy, i.e., older instructions are prioritized over the younger ones. We also studied prioritizing slices over non-slice instructions; however, performance was similar to the age-based policy. Further, in each cycle, not all scheduled instructions need to come from the same IQ, rather they can come from any combination of IQs to fill the issue width.

Though Freeway employs only FIFO IQs for instruction scheduling, having multiple of them enables younger instructions in one IQ to bypass the older instructions stalled in the other IQs. As a result, instructions can be scheduled and executed out of program order. Therefore, Freeway needs to track instruction order to ensure that the instructions update the architectural state in program

order. Like LSC, Freeway employs a Scoreboard to track instruction order. As shown in Figure 8, instruction are inserted into Scoreboard in program order at dispatch stage. As instructions finish their execution, which might be out of program order, they do not update the architectural state immediately, rather they record their completion in the Scoreboard. Only once an instruction reaches the head of scoreboard, it updates the architectural state and is taken off the Scoreboard. This ensures that the architectural state is always updated in program order even though instructions can execute out of order. To track a sufficient number of instructions, Freeway and LSC increase the size of Scoreboard over what is typical in an in-order core.

4.3.3 Memory Ordering. Before describing Freeway's mechanism to maintain memory ordering, we first discuss how the baseline LSC maintains this order. LSC computes memory addresses strictly in program order as all address calculations are performed via the FIFO B-IQ. Despite the FIFO address generation, younger loads can still bypass the older stores that are waiting in the A-IQ (recall that only the address calculation for stores is performed via B-IQ, whereas the store operation itself passes through the A-IQ). Therefore, to avoid loads from bypassing the aliased stores, LSC incorporates a *store buffer*. It inserts store addresses in to the store buffer so that they can be used to disambiguate the subsequent loads. LSC then issues loads to memory only if their address does not match any store address in the store buffer, thereby ensuring memory ordering.

This mechanism cannot be directly ported to Freeway to maintain memory ordering. This is because the strict FIFO address generation in LSC guarantees that all previous outstanding stores have their addresses in the store buffer when a load is about to be issued. Freeway, in contrast, allows independent memory slices to bypass the dependent ones waiting in the Y-IQ. As a result, a load may not check against an older store whose address calculation is still waiting in the Y-IQ and the address has not yet been written to the store buffer. To avoid this scenario, Freeway marks all loads and stores with a sequence number in program order. In addition, stores are allocated an entry in the store buffer at dispatch and the entry is later updated with the store address when available. As a result, loads that are about to be issued can look in the store buffer to check if all previous stores have computed their addresses. They only proceed to execution if there are no unresolved and aliasing stores. This simple store buffer extension maintains memory ordering while only requiring the addition of a small (depending on instruction window size) sequence number to the store buffer entries.

It is worth noting that, as with LSC, Freeway issues stores to the memory only when they are the oldest instruction in the instruction window. Therefore, such stores do not violate memory ordering even though they can bypass older loads waiting in the Y-IQ that access the same memory location. When such a bypassed load becomes ready, it checks the store buffer and finds a store with the same memory address. However, instead of forwarding data from the store, the load is issued to memory as the store is younger.

5 METHODOLOGY

To evaluate Freeway, we use the Sniper [7] simulator configured with a cycle-accurate core model [5]. Sniper works by extending Intel's PIN tool [31] with models for the core, memory hierarchy, and on-chip networks. Area and power estimates are obtained from CACTI 6.5 [28] using its most advanced technology node, 32nm. We use the SPEC CPU2006 [43] workloads with reference inputs. Furthermore, we use multiple inputs per workload to evaluate performance, energy, and area. We use SimPoint [39] to choose a single representative region of 1B instructions in each application.

The key microarchitectural parameters are presented in Table 1, with all core designs being two-wide superscalar with 64-entry instruction window and a cache hierarchy employing hardware

Table 1. Microarchitectural Parameters

Core	2 GHz, 2-wide issue, 64-entry instruction window
Branch Predictor	Intel Pentium M-style [45]
Branch Penalty	9 cycles (7 cycles for in-order core)
Functional Units	2 Int, 1 VPU, 1 branch, 2 Ld/St (1+1)
L1-I	32 KB, 4-way LRU
L1-D	32 KB, 8-way LRU, four cycle, 8 MSHRs
LLC	512 KB per core, 16-way LRU, avg. 30-cycle round-trip latency
LLC Prefetcher	stride-based, 16 independent streams
Main Memory	4 GB/s, 45 ns access latency

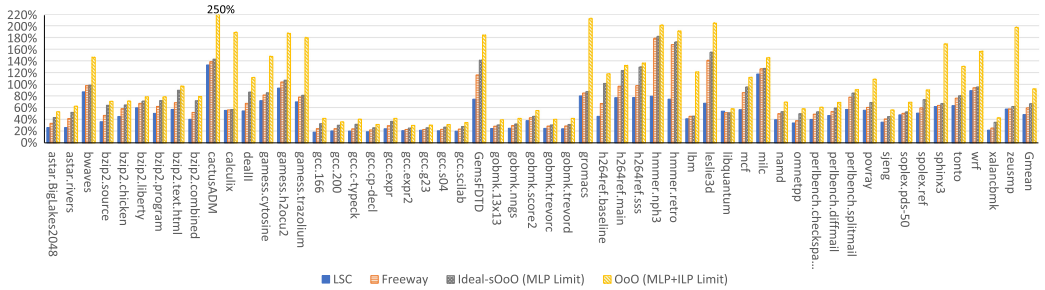

Fig. 9. Performance gain of different core designs over an in-order core.

prefetchers. We compare the MLP, performance, energy, and area overheads for the following four core designs:

In-order Core: We use an in-order stall-on-use core, resembling ARM Cortex-A7 [4], as a baseline.

Load Slice Core: LSC, as proposed by Carlson et al. [6], with strict in-order memory slice execution. We model the A-IQ and B-IQ as each having 64 entries.

Freeway: Our proposed design with dependent slice tracking and a FIFO yielding queue (Y-IQ) to enable out-of-order execution among slices. To keep the total number of instruction queue entries same as in LSC, we model A-IQ, B-IQ and Y-IQ with 64-entries, 32-entries, and 32-entries, respectively.

Ideal-sOoO: LSC with a fully out-of-order B-IQ. This design provides an upper bound on the performance limits of MLP in sOoO cores as it can execute MLP-generating instructions from anywhere in the B-IQ, thus preventing stalled slices from blocking MLP exploitation.

Out-of-order Core: We use a fully out-of-order core, resembling ARM Cortex-A9, as MLP+ILP limit on performance.

6 EVALUATION

6.1 Performance

Figure 9 presents the performance gains of LSC and Freeway over the baseline in-order core. The figure also shows the performance limits of Ideal-sOoO (MLP limit) and full OoO (MLP+ILP limit) execution. The geometric mean (Gmean) performance difference between Freeway and LSC is 12% as Freeway attains 60% speedup over the in-order execution compared to 48% for LSC. More importantly, Freeway is within 7% of the performance of Ideal-sOoO, which is the upper bound on the performance achievable via MLP exploitation (MLP limit). An OoO core delivers 33% more

performance gain than Freeway, over an InO core, as it exploits both ILP and MLP, whereas Freeway targets only MLP. However, this additional performance comes with high area and power overheads (Section 6.7).

Freeway Versus LSC: On individual workloads, Freeway performs significantly better than LSC on workloads such as `hmmer`, `leslie3d`, and `GemsFDTD` where dependent slices stall the B-IQ of LSC for a significant fraction of execution time (Figure 3). Freeway eliminates these stalls by steering the dependent slices to the newly added Y-IQ, thereby executing the subsequent independent slice without stalling and boosting performance.

A closer inspection reveals that Freeway's performance advantage over LSC is a function of not only the number of stalls caused by dependent slices but also where producer slices hit in the memory hierarchy. For example, workloads such as `gcc`, `soplex`, and `omnetpp`, where slice dependencies stall execution for more than 45% of time in LSC, benefit only moderately from Freeway. The reason for this behaviour is that more than 70% of the producer slices in these workloads miss in the on-chip cache hierarchy and must be loaded from memory (Figure 4). This latency stalls instruction retirement, and therefore causes the instruction window to fill, which blocks further MLP generation and limits performance. In contrast, Freeway delivers significantly more performance (95% and 76%, respectively) for `hmmer` and `leslie3d`, despite LSC stalling on slice dependencies for only 30% of the execution time. This is because almost all of the producer slices in these workloads hit in the L1 cache. Therefore, the instruction window is rarely full and Freeway can continuously exploit MLP and improve performance.

Finally, Figure 9 also shows that both Freeway and LSC perform similarly on workloads such as `zeusmp`, `milc`, `lbm`, and `calculix`. These workloads do not have many dependent slices and the corresponding stalls, as shown in Figure 3, are minimal. As a result, Freeway does not have much opportunity for improvement over LSC.

Freeway Versus Ideal-sOoO Versus OoO: Figure 9 shows that the majority of the benefits of full OoO execution can be obtained primarily by exploiting MLP as Ideal-sOoO (MLP limit) achieves about 72% of the performance benefits of full OoO (MLP+ILP) execution. Furthermore, the figure also shows that Freeway captures the bulk of this MLP opportunity and reaches within 7% of the performance delivered by idealized MLP extraction (Ideal-sOoO design). Compared to OoO execution, which targets both ILP and MLP, Freeway falls short on workloads that present significant ILP opportunity, such as `calculix`, `gromacs`, `zeusmp`, and so on, as it exclusively aims for MLP. However, on workloads that offer little ILP, such as `sjeng`, `perlbench`, `xalancbmk`, and so on, Freeway is within 15% of the full OoO performance. Overall, full OoO execution provides 93% performance gain over in-order execution compared to the 60% gain of Freeway. However, the additional performance of OoO comes at a significantly higher area and power costs as discussed in Section 6.7.

Performance impact of load speculation: For the results presented in Figure 9, load instructions are issued only when all the older store instructions have their addresses available and there is no aliasing. Though such conservative scheduling reduces the complexity of load-store unit, it also loses performance opportunities. To understand the performance opportunity in load speculation for different core designs, we assume an oracle memory dependence predictor and issue a load if it is not predicted to be aliased with older stores even if their addresses are not yet computed. Notice that, in addition to OoO core, Freeway also benefits from load speculation. This is because, as discussed in Section 4.3.3, a load in Freeway might have its address available, i.e., it is ready for execution, before all the older stores compute their addresses. Finally, Ideal-sOoO core would also benefit from load speculation as its B-IQ is fully out-of-order. The in-order core and

LSC, in contrast, do not benefit from load speculation, because they compute all load and store addresses in program order. Our results shows that load speculation enables Freeway, Ideal-sOoO, and OoO cores to provide, repsectively, 63%, 70%, and 97% performance gain over in-order core; whereas their respective performance gains without load speculation are 60%, 67%, and 93%. On individual applications, the performance gain from load speculation is as high as 20%, for example, on bzip2.program. These results suggest that load speculation helps Freeway, Ideal-sOoO, and OoO about equally. For the rest of the evaluation, we use the conservative scheduling policy, i.e., without load speculation.

6.2 Analysis of the Remaining Opportunity

Figure 9 shows that despite its dependence-aware slice execution, Freeway lags behind the optimal performance achievable via MLP exploitation ("Ideal-sOoO (MLP limit)") by 7%. We observe that there are two main factors that cause this gap: First, Freeway addresses only the slice dependence related stalls but not the other stall sources such as Load-store Aliasing, Empty B-IQ, and so on (described in Section 2.3). Second, buffering all dependent slices in a single Y-IQ leads to stalls when a younger slice becomes ready earlier than an older slice, although this is infrequent. Here, we analyze the performance loss due to these factors and explore potential solutions.

As Figure 3 shows, *Empty B-IQ* is the largest source of stalls after slice dependence stalls. However, mitigating them requires improvements in core components other than instruction scheduling. For example, it either requires a better front-end, if the B-IQ is empty due to branch mispredictions or instruction cache misses, or it requires a larger instruction window, if the window is full when the B-IQ is empty. As the focus of this work is instruction scheduling, we do not explore techniques to mitigate these stalls.

The next largest source of stalls, as Figure 3 shows, is the stalls coming from load-store aliasing. Such load-store aliasing causes LSC to stall for 8% of the execution time. Furthermore, as Freeway enables early execution of independent slices, it can potentially expose more aliasing if some of the aliased loads were earlier hidden behind the dependent slices in the B-IQ of LSC. To quantify the resulting performance loss, we simulate skipping the aliased loads and issuing the subsequent instructions if they are ready. Though impractical, such scheduling shows the potential benefits of eliminating the load-store aliasing related stalls. The *skip_Aliased_Load* bar in Figure 10 shows that Freeway obtains only 2% additional performance by eliminating all such stalls. As the *Other* stall sources contribute even less, we do not quantify their impact on performance loss. From this analysis, we see that even completely addressing the load-store aliasing related stalls would result in little performance gain.

As discussed in Section 3.2, buffering all dependent slices in a single Y-IQ might lead to stalls if younger slices become ready before the older ones (either the younger slices are at a lower dependence depth or their producers hit closer to the core in the memory hierarchy). As almost all producers hit in the L1 cache (Figure 7), we only consider mitigating stalls due to slice dependence depth by adding a single additional Y-IQ. Here, we explore the benefits of having the first Y-IQ buffer only the dependent slices at dependence depth 1, while the rest of the dependent slices go to a second Y-IQ. As a result, the stalls in the first Y-IQ will be reduced as all the slices are at the same dependence depth. The *skip_Aliased_Load+Additional_Y-IQ* bar in Figure 10 shows that adding a second Y-IQ brings only 1.5% additional performance. These results also confirm our hypothesis that a single Y-IQ is enough to capture the most of the opportunity in out-of-order slice execution.

If combined, then the above optimizations would bring performance to within 3.5% of the optimal. However, individually they provide only minimum performance returns on the resource investment.

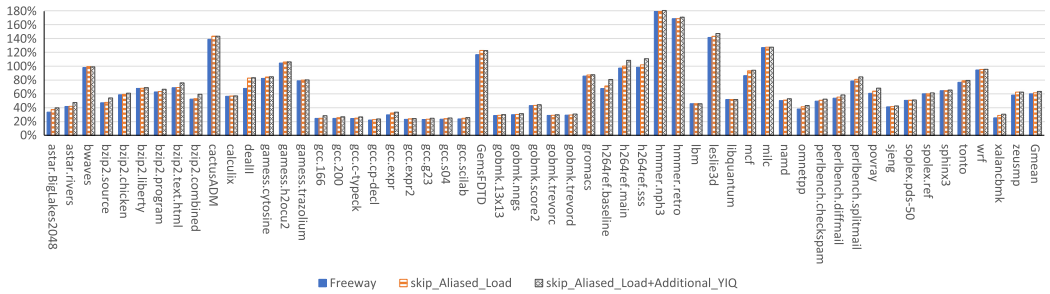

Fig. 10. Freeway performance gain achieved by skipping the aliased loads and adding another Y-IQ.

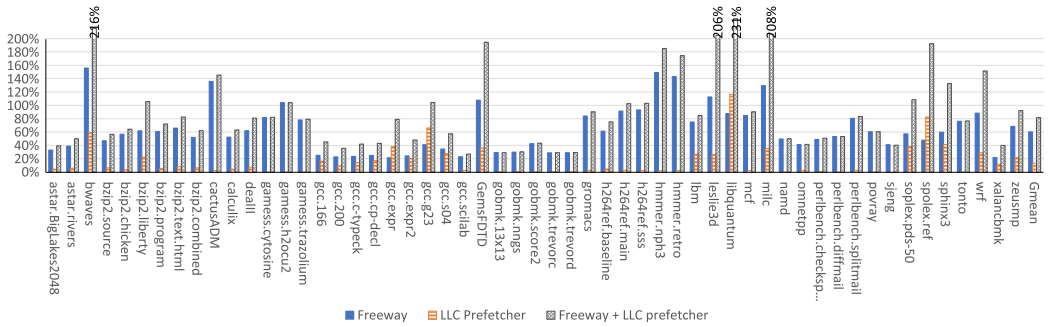

Fig. 11. Performance breakdown: gains achieved individually by Freeway and LLC prefetcher and their combined gain over a prefetcherless in-order core. (The baseline core in all other studies does include a LLC prefetcher).

6.3 Breaking Down the Performance Contribution of Different MLP Generators

Our proposed core design, Freeway, generates MLP by enabling slices to bypass the stalled non-slice instructions and enabling independent slices to bypass the stalled dependent ones. In addition, we employ a LLC prefetcher that increases the number of overlapping memory requests, hence MLP, by speculatively generating accesses for cache blocks that are likely to be required soon. This section quantifies the contribution of each of these components towards overall performance.

Figure 11 presents the performance gain achieved by the individual MLP-generating components, i.e., Freeway and the LLC prefetcher, as well as the gain when they work together. For this study, we do not model prefetcher in the baseline in-order core to help evaluate the performance gains of the LLC prefetcher. In all other studies, however, our baseline in-order core (and all other evaluated cores including Freeway) does include a LLC prefetcher.

As the figure shows, Freeway provides a much higher performance gain (61%) than the LLC prefetcher (14%). This is because Freeway hides L1, LLC, and to some extent memory latency by generating parallel memory accesses and overlapping their latency. LLC prefetcher, in contrast, hides only the latency between LLC and memory; however, the LLC (and even L1) latency is still exposed due to the serial accesses up until LLC. We also evaluated aiding the LLC prefetcher with an L1 prefetcher (stride prefetcher with four independent streams); however, it provides only about 1.5% additional Gmean performance.

Looking closely at individual workloads, there are three applications, `gcc` (expr and g23 inputs), `libquantum`, and `soplex` (ref input), where the LLC prefetcher outperforms Freeway. Figure 12 shows that all of these applications have high LLC MPKI (25–34) and thus get most of their data

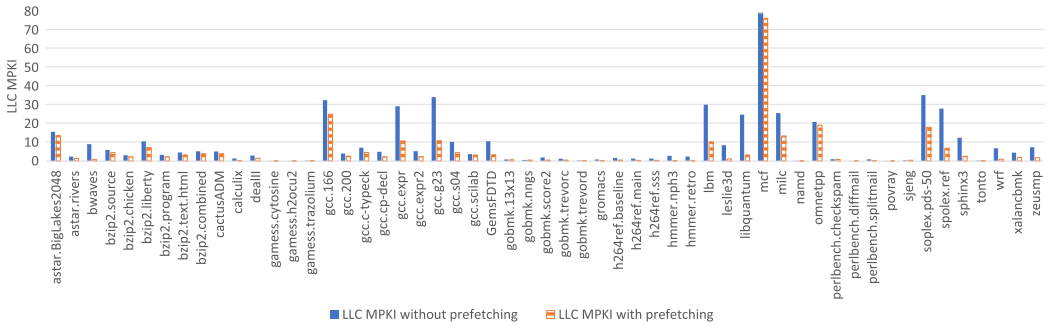

Fig. 12. LLC MPKI (misses per kilo instructions) without and with LLC prefetcher.

from memory. The LLC prefetcher delivers higher performance due to their prefetch friendly access pattern as it is able to reduce the LLC MPKI to 3–11. In contrast, Freeway is not able to generate enough MLP mainly due to a combination of two factors. First, the high LLC MPKI leads to frequent full instruction window stalls (long memory latency on LLC misses causes instruction window to fill up), which block further MLP generation. Second, and more importantly, as shown in Figure 6, these applications have a larger number of dependent slices that cannot be executed until their parent slices receive data, thus further lowering MLP.

For the other benchmarks with high LLC MPKI, Freeway outperforms the LLC prefetcher either because the benchmarks do not have many dependent slices, and therefore Freeway generates enough MLP, or their access patterns are not prefetch friendly, and therefore the LLC prefetcher does not perform well. For example, `lbm` does not have any dependent slices and, therefore, Freeway generates significant MLP. Conversely, `mcf`'s access pattern is not prefetch friendly, therefore, resulting in poor LLC prefetcher performance.

Another important finding from Figure 11 is that combining Freeway with LLC prefetcher provides considerably higher performance gain (82%) than that delivered individually by Freeway (61%) and the LLC prefetcher (14%). This is because, when combined, more memory accesses generated by Freeway are served from LLC due to the prefetched blocks, thereby reducing average memory access time. In addition, if these accesses were causing full instruction window stalls by blocking the instruction commit, then their early completion will also enable subsequent memory accesses to enter the instruction window and execute early, thus further improving MLP and performance.

Overall the results in Figure 11 show that MLP generated by Freeway provides significantly higher performance than the LLC prefetcher. These results also imply that if designers must pick one technique among several competing candidates for MLP generation, for example, in area-constrained designs, then it is better to invest real estate in Freeway rather than in LLC prefetcher.

6.4 Freeway's Efficacy at Smaller Instruction Queues

As Section 6.1 detailed, Freeway performs significantly better than LSC when both are given equal number of instruction queue entries (128). This section investigates the designs' relative sensitivity to reducing the number of instruction queue entries and explores how few entries Freeway can work with while still delivering similar performance to LSC.

Figure 13 presents the performance gains achieved by LSC and Freeway with a total of 128, 32, and 20 instruction queue entries. We observed that the performance with 128 and 64 entries is similar (<1% difference); therefore, we do not show 64-entry design point in the figure. The ratio of entries among different queues remains same as in the 128-entry case, i.e., 1:1 (A-IQ:B-IQ) in

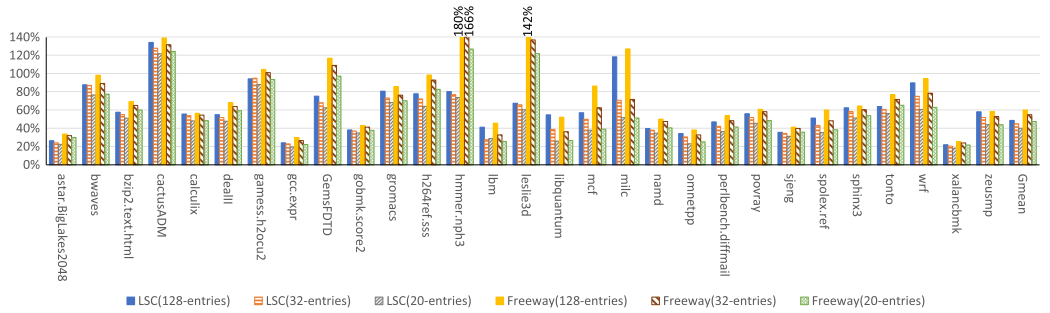

Fig. 13. Performance gain with different number of instruction queue entries. The ratio of entries among queues is 1:1 (A-IQ:B-IQ) in LSC and 2:1:1 (A-IQ:B-IQ:Y-IQ) in Freeway. (For readability, we show only one input per benchmark).

LSC and 2:1:1 (A-IQ:B-IQ:Y-IQ) in Freeway. The results show that Freeway always performs better than LSC when they both feature equal number of queue entries regardless of whether it is 128, 32, or 20-entries. Furthermore, Freeway with just 32 entries provides about 6% more gain than 128-entry LSC, over InO core, and Freeway requires barely 20 instruction queue entries to provide similar performance to that of the 128-entry LSC.

Looking at individual benchmarks, there is a set of applications where the 20-entry Freeway performs significantly better the 128-entry LSC. For example, Freeway performs about 50% better on `hmmer` and `leslie3d` and about 20% better on `GemsFDTD`. These are the applications where letting younger independent slices to bypass the older dependent ones offers significant performance opportunities as shown in Figure 2. The results in Figure 13 imply that even a small Y-IQ, with only five entries, enables significant number of independent slices to bypass the dependent ones, thus providing a high performance gain. Whereas a larger B-IQ in LSC does not provide additional performance as it simply results in more independent slices sitting behind the stalled dependent ones.

However, the 20-entry Freeway performs worse than the 128-entry LSC for some applications like `milc`, `wrf`, `zeusmp`, and so on. These applications do not present much performance opportunity as shown in Figure 2. In the absence of performance opportunity, the frequent dispatch (insertion into instruction queues) stalls caused by smaller instruction queues result in 20-entry Freeway performance dropping below that of the 128-entry LSC. The dispatch stalls are more frequent with smaller queues as they fill up quickly and the dispatch stage stalls as soon as any one of the queues is full and the next instruction also needs to be dispatched to that same queue. Though the combination of low opportunity and frequent dispatch stalls causes a performance drop in these applications, Freeway still performs slightly better than LSC when both are given same number of queue entries.

Overall, the results in Figure 13 show that Freeway can tolerate more than 6× reduction in the instruction queue entries (128 to 20) while delivering similar performance as 128-entry LSC due to its dependence-aware slice execution.

6.5 Sensitivity to L1 Cache Latency

As the majority of stalls in LSC occurs while producer slices fetch data from L1 cache (Figure 4), we study the sensitivity of Freeway's performance gain to the L1 cache latency. In addition, we also study how close Freeway comes to the Ideal-sOoO and OoO cores at different L1 latencies. For this study, we vary the L1 latency (load-to-use) from two to eight cycles. The default L1 latency used for the other experiments is four cycles.

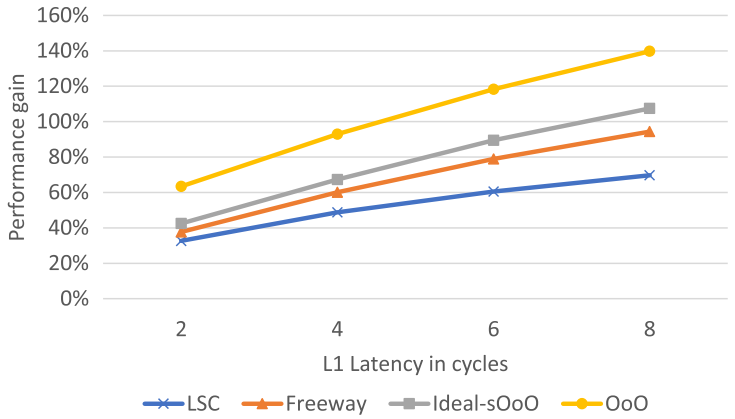

Fig. 14. Performance gain at different L1 cache latencies.

Figure 14 presents the performance gain of different cores over an in-order core as a function of L1 latency. The first point to notice is that the gain of all cores is smaller at lower latencies. For example, Freeway delivers 60% performance gain over in-order core at four-cycle latency, whereas it drops down to 38% at two-cycle latency. This is because the baseline in-order core itself performs increasingly better at lower latencies as the dependent instructions stall for less time on L1 hits. As a result, the performance opportunity, and the achieved gain, for other cores is reduced.

An important finding from Figure 14 is that even though the performance gap between Freeway and LSC is very small (5%) at two-cycle latency, it widens quickly at larger latencies: 12% at four-cycle, 18% at six-cycle, and 25% at eight-cycle latency. This is because the stalled dependent slices prevent the subsequent independent slices from executing for longer intervals at larger latencies in LSC, thereby limiting MLP. In contrast, Freeway keeps executing independent slices from B-IQ while the dependent ones wait longer in the Y-IQ. Overall, these results demonstrate that Freeway is better than LSC in tolerating higher L1 latencies.

The figure also shows that the performance gap between Freeway and Ideal-sOoO (which represents the MLP limit) grows at a significantly lower rate compared to the gap between Freeway and LSC. This gap increases from about 5% at two-cycle L1 latency to 13% at the eight-cycle latency. As Ideal-sOoO executes slice instructions fully out-of-order, it is able to cover the MLP opportunities missed by Freeway, as discussed in Section 6.2. The performance gap between Ideal-sOoO and fully OoO also increases while going from two-cycle to eight-cycle latency, because OoO can hide the increased latency better by executing more ILP-generating instructions.

Freeway Versus LSC (with fixed L1 latency for the baseline core): To understand how much additional L1 latency Freeway can tolerate compared to LSC, we keep the baseline in-order core's L1 latency constant at four cycles (the default value) and vary LSC and Freeway L1 latency to four, six, and eight cycles. Notice that this is in contrast to the results in Figure 14, where we vary the L1 latency for the baseline in-order core along with other core designs.

The results in Figure 15 show that Freeway not only performs better than LSC while considering equal L1 latency for both but also when it has to tolerate higher latency than LSC. For example, LSC delivers about 48% performance gain over the baseline InO core with a four-cycle L1 latency, whereas Freeway achieves 50% gain even at a higher latency of six cycles. Similarly, LSC provides 35% gain at six-cycle L1 latency while Freeway attains 41% gain even with eight-cycle latency. These results further validates Freeway's advantage over LSC in tolerating L1 latency.

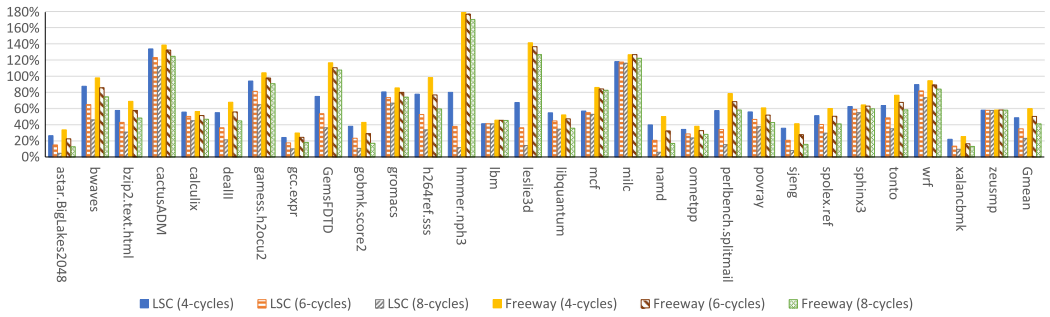

Fig. 15. Freeway and LSC performance gain at different L1 cache latencies while the baseline in-order core L1 latency fixed at four cycles. (For readability, we show only one input per benchmark).

These results also imply that Freeway is likely to benefit more from a larger L1 cache (with correspondingly higher access latency) than LSC because of its better L1 latency tolerance.

6.6 Scalability Analysis

The instruction window size, e.g., scoreboard size in Freeway, limits the number of in-flight instructions. Consequently, the number of outstanding memory accesses, hence MLP, is also bounded by the size of instruction window. While prior work [6, 24] evaluated sOoO cores only at small instruction windows, we analyze their scalability across the full spectrum of instruction window widths and depths (i.e., from wimpy to brawny cores) by varying the instruction window size from 32 to 224 entries. We also size other structures (instruction queues, execution units, etc.) appropriately in accordance with the instruction window size. Furthermore, we use an issue width of 2 for the instruction windows of 32 and 64 entries, while 128-, 168-, 192-, and 224-entry instruction windows use issue widths of 3, 4, 6, and 8, respectively.

The performance results (geometric mean) across the benchmark suite are presented in Figure 16. The results show that Freeway's performance advantage over LSC increases from about 10% to 14% while moving from 32- to 224-entry instruction window. Their performance difference at the 32-entry window is small, because there are not many MLP opportunities in such a small window due to fewer in-flight instructions. However, as the number of in-flight instructions increases with window size, Freeway exposes increasingly more MLP by executing independent slices unobstructed, whereas in LSC most of the additional slices simply sit behind the stalled ones. The performance gap between Freeway and Ideal-sOoO also increases with window size, though Freeway is still within 16% of Ideal-sOoO (limits of MLP) even at a large 8-wide 224-entry instruction window.

The performance gains of fully out-of-order execution increase even more with the larger windows due to its ability to utilize both MLP and ILP. An important finding from the results of Figure 16 is that MLP contributes less to overall performance at large instruction windows. For example, Ideal-sOoO core, which exploits the maximum MLP, is within 18% of full OoO performance at a small 32-entry instruction window. However, the difference grows with instruction window size. This is because larger instruction windows offer more ILP opportunities to OoO execution, thus widening the performance gap with Ideal-sOoO, which only extracts MLP.

These results highlight that further research is required for scaling sOoO cores to larger instruction windows. Furthermore, given that MLP's contribution to overall performance reduces at large windows and that Freeway already extracts a large portion of the available MLP, the research in scaling sOoO cores should focus on ILP rather than capturing the MLP opportunities missed by Freeway.

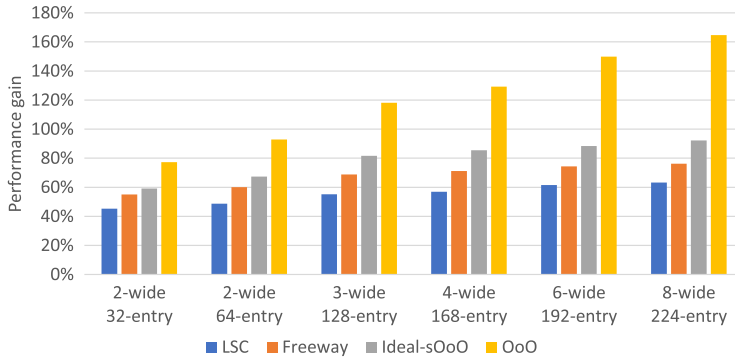

Fig. 16. Performance sensitivity to core size. x-axis is labeled with issue-width (x-wide) and instruction window size (y-entry).

6.7 Area and Power Overheads

To evaluate the area and power overheads of different core designs, we use CACTI 6.5 to compute the area and power consumption of each of their major components in 32nm technology. The area overhead of LSC is about 15% over the baseline in-order core. Freeway requires very little additional hardware over LSC: one bit per entry in RDT, 7 bits per entry in the store buffer, and the Y-IQ and logic to issue instructions from it. As a result, it needs only 1.5% additional area. In contrast, as reported by Reference [6], the OoO core incurs an area overhead of 154% over the baseline in-order core.

For power calculations, we use static power consumption and per-access energy values from CACTI and combine them with activity factors obtained from the timing simulations to compute power requirements of each component. Our evaluation, together with prior results [6], show that LSC, Freeway, and OoO core increase the average power consumption by 1.22×, 1.24×, and 12.6×, respectively, over an in-order core. These results demonstrate the exorbitant area and power costs of the moderate performance benefits achieved by OoO core over more efficient slice-out-of-order core designs such as Freeway.

7 RELATED WORK

7.1 Tolerating Long Memory Latency

Existing research on tolerating memory access latency to prevent cores from stalling can be divided into two broad categories: techniques that *extract MLP* by overlapping the execution of multiple memory requests, and techniques that *prefetch data* proactively into caches by predicting future memory addresses. To some extent, these categories are complementary as prefetching can increase the number of overlapping memory requests by speculatively generating accesses that would otherwise be serialized due to dependencies or lack of resources. However, the energy efficiency of the majority of these techniques is bounded by the underlying energy intensive OoO core. Freeway, in contrast, provides an energy efficient alternative that these techniques can build on to potentially raise overall efficiency.

MLP Extraction: OoO execution is the most generic approach for generating MLP. However, it is limited by the instruction window size. The following techniques break this size barrier to boost MLP extraction:

Runahead Execution: Runahead Execution [33] improves MLP by pre-executing instructions beyond a full instruction window. Once the OoO core stalls due to a full ROB, Runahead checkpoints

the processor state, tosses out the ROB stalling instruction, and continues to fetch subsequent instructions. These new instructions are executed if their source data is available, thereby generating additional memory accesses and boosting MLP. Recent work has improved several aspects of runahead execution. Filtered runahead [14] clock- and/or power-gates the core front-end while supplying instructions from a buffer to reduce energy consumption. Precise Runahead [34] reduces the overhead of restarting regular execution after coming out of the runahead mode. Continuous Runahead [13] continuously executes MLP-generating instructions in parallel with core, instead of waiting for the core to stall, to improve runhaed coverage.

Helper Threads: These techniques rely on pre-executing "helper threads" or code segments to generate MLP. A helper thread is a stripped down version of the main thread that only includes the necessary instructions to generate memory accesses, including control flow instructions. However, helper threads require an independent execution context (SMT or a CMP core) for their execution. As they generate memory accesses in parallel with the main thread, they increase MLP and/or prefetch data.

Helper threads can be generated either in software or dynamically in hardware. On the software side, many prior works have proposed compiler/programmer driven approaches for helper thread generation [9, 19–21, 23, 30, 50] while others have proposed dynamic compilation techniques [29, 49]. These techniques either execute the helpers threads on an available SMT context [9, 21] or require a dedicated core [19].

Collins et al. [10] explored helper thread generation in hardware by tracking dependent instruction chains in the back-end. To keep the helper thread generation off the critical path, they introduced large, post-retirement, hardware structures to filter the desired instructions. Once the helper threads were generated, they were stored in a large cache and run on a free SMT context. Annavaram et al. [3] also extracted the dependent chains of operations that were likely to result in a cache miss in hardware, though from the front-end during instruction decode, and added a dedicated back-end for the execution of such chains.

To summarize, helper threads incur significant overhead as they require: (1) an independent execution context (SMT, a CMP core, or dedicate hardware) for their execution, (2) a mechanism to construct them either in hardware or software, (3) duplicated instruction execution in the main thread and helper thread.

Prefetching: Prefetchers predict future addresses based on prior memory access patterns. However, they either have limited coverage due to being limited to simple access patterns or require extensive hardware. For example, stride and stream prefetchers [18, 36] require only simple hardware but are limited to regular access patterns. Advanced prefetchers, such as Correlation Prefetchers [17, 25, 37], enable complex access pattern prefetching at the cost of large tables to link the past miss addresses to future miss addresses. Spatial and temporal streaming-based prefetching has also been explored in server domain [41, 42, 46], though they still incur significant storage and energy overhead. Recently, co-design of prefetching with replacement policies has also been explored in References [22, 47].

7.2 Energy-efficient Core Design

Instruction scheduling is one of the most energy hungry operations in modern OoO cores [35]. Therefore, researchers have proposed a number of techniques to reduce its energy requirements. Recent research in this domains exploits two properties, instruction readiness and instruction criticality, to minimize scheduling energy overhead.

Shioya et al. [40] observed that a large fraction to total dynamic instructions is either ready for execution at dispatch stage or becomes ready within a few cycles of dispatching to the issue queue.

These instructions do not benefit from OoO scheduling as they would execute without stalls even in an in-order pipeline. Therefore, they propose an architecture that attempts to execute all instructions via in-order pipeline stages before dispatching the unexecuted ones to OoO pipeline, thereby reducing scheduling energy. FIFOrder [2], instead of trying to execute all instruction via in-order stages, dispatches ready instructions to a FIFO issue queue and non-ready instructions to an OoO (content addressable memory-based) issue queue. As the OoO queue handles fewer instructions, FIFOrder reduces its depth and width, thus reducing the scheduling energy cost. Another recent architecture, CASINO core [16], also targets ready instructions to simplify instruction scheduling.

Other researchers have targeted instruction criticality to cut the energy cost of scheduling. For example, Long-term Parking (LTP) [38], at dispatch stage, allocate issue queue entries only to critical instructions, whereas non-critical instructions are *parked* in a FIFO parking queue. Parked instructions are allocated issue queue entries only when they reach close to the head of reorder-buffer. As the parking queue reduces pressure on issue queue, LTP reduces its dimensions to reduce its energy requirements. A recent work, Delay and Bypass [1], advocates exploiting both readiness and criticality simultaneously to further improve energy savings.

Though all of these designs reduce instruction scheduling energy, they still feature CAM-based instruction queues, albeit smaller than OoO cores. Therefore, their energy savings are not as high as those of sOoO cores. A recent design, Forward Slice Core (FSC) [26], is the closest approach to sOoO cores in that it builds on a stall-on-use in-order core and uses only FIFO queues for instruction scheduling. Unlike sOoO cores, which specifically target MLP, FSC aims to extract generic ILP. Therefore, some of its features can be borrowed for ILP extraction in sOoO cores. Specifically, compared to sOoO cores, FSC enables non-slice instructions that do not depend on load instructions to bypass the load-dependent instructions via a dedicated IQ, thereby extracting ILP among non-slice instructions. In sOoO cores, in contrast, as all non-slice instructions, whether load-dependent or not, share the same IQ, they miss the ILP extraction opportunity. sOoO cores can borrow FSC's mechanism to split load-dependent and independent non-slice instructions into different queues to improve ILP extraction. However, this is only one of the many possible ways of splitting non-slice instructions. Other splitting criteria could include dispatching integer and floating-point instructions to different queues, creating and dispatching slices of branch instructions to dedicated queues to potentially reduce the branch misprediction penalty, or dispatching consumers of high- versus low-latency loads to different queues, or splitting instructions based on fanout. Further investigation is needed to understand the trade-offs provided by these criteria.

FSC also shares some features with sOoO cores. For example, its mechanism to create forward slices is very similar to Freeway's mechanism of identifying dependent slices. Also, as FSC does not explicitly identify address generating instructions (AGIs) and mixes them with non-AGIs in its Main Lane, load/store address generation is likely to be delayed, compared to sOoO cores, which would limit MLP extraction.

Besides instruction scheduling, register renaming is also an energy intensive operation [35] that has received researchers' attention. Gonzalez et al. [12] proposed to delay the register allocation until a late pipeline stage to reduce register file pressure, thus delivering same performance with a smaller and less energy hungry register file. Tabani et al. [44] proposed a technique that releases physical registers earlier than conventional techniques, thus reducing register file pressure, size, and energy requirements. A recent work, STRAIGHT [15], proposes an instruction set architecture that eliminates the register renaming altogether.

8 CONCLUSION

Tolerating long memory and LLC access latencies is critical for performance. MPL exploitation techniques such as out-of-order execution, runahead execution, and so on, have been successful

in hiding these latencies, however, at the cost of large energy overheads. Recent attempts to address these in an energy-efficient manner have led to sOoO cores. These cores construct slices of MLP-generating instructions and execute them out-of-order with respect to the rest of instructions. However, the slices and the remaining instructions, by themselves, still execute in-order. By limiting the out-of-order execution to only between slice and non-slice instructions, sOoO cores are able to achieve much of the MLP benefits of OoO processor with far less hardware overhead.

This work introduces Freeway, a highly energy-efficient core that approaches the MLP benefits of full out-of-order execution. To keep the energy overhead low, Freeway builds upon a modern sOoO core. We show that, though energy-efficient, state-of-the-art sOoO cores miss significant MLP opportunities due to inter-slice dependencies. Freeway addresses this bottleneck by *identifying* dependent slices and introducing an efficient *dependence-aware* slice execution policy based on a detailed analysis of slice behaviour. Freeway's policy forces dependent slice to *yield* to independent slices, thereby boosting MLP and performance. Moreover, as shown through our analysis and simulation, Freeway's policy can be implemented with a simple FIFO queue, which requires only minimum additional hardware over the baseline sOoO core. Our results show that Freeway is able to attain 12% better performance than previous sOoO designs and delivers performance within 7% of the MLP limits of the ideal sOoO execution.

Our analysis shows that Freeway is also more flexible than previous sOoO designs, as it requires one-sixth as many instruction queue entries to deliver similar performance and can tolerate 1.5x higher L1 cache latencies at similar performance. Further, while Freeway's MLP generation is complementary to that of an LLC prefetcher, we find that the Freeway generates significantly more MLP in general. However, our results also show that sOoO cores are less effective at larger instruction window sizes as MLP provides a smaller portion of the overall performance, with ILP becoming more important. Based on these results, and our data showing that Freeway captures the bulk of the available MLP, we conclude that future sOoO core research should focus on ILP, rather than capturing the limited MLP opportunities missed by Freeway.

REFERENCES

[1] 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 *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. 424–434.

[2] Mehdi Alipour, Rakesh Kumar, Stefanos Kaxiras, and David Black-Schaffer. 2019. FIFOrder MicroArchitecture: Ready-aware instruction scheduling for OoO processors. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'19)*. 716–721. https://doi.org/10.23919/DATE.2019.8715034

[3] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. 2001. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. 52–61. https://doi.org/10.1145/379240.379251

[4] ARM. ARM Cortex-A7 Processor. [n.d.]. Retrieved from ttp://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[5] Trevor E. Carlson, W. Heirman, Stijn Eyerman, I. Hur, and L. Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.* 11, 3, Article 28 (Aug. 2014), 25 pages. https://doi.org/10.1145/2629677

[6] Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout. 2015. The load slice core microarchitecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 272–284.

[7] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. ACM, New York, NY, Article 52, 12 pages. https://doi.org/10.1145/2063384.2063454

[8] Jamison D. Collinsy, Dean M. Tullseny, Hong Wangz, and John P. Shen. 2001. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'01)*. IEEE Computer Society, Washington, DC, USA, 306–317.

[9] J. D. Collins, Hong Wang, D. M. Tullsen, C. Hughes, Yong-Fong Lee, D. Lavery, and J. P. Shen. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. 14–25. https://doi.org/10.1145/379240.379248

[10] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. 2001. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'01)*. 306–317. Retrieved from http://dl.acm.org/citation.cfm?id=563998.564037

[11] James Dundas and Trevor Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing (ICS'97)*. Association for Computing Machinery, New York, NY, 68–75.

[12] A. Gonzalez, J. Gonzalez, and M. Valero. 1998. Virtual-physical registers. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. 175–184. https://doi.org/10.1109/HPCA.1998.650557

[13] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. Article 61, 12 pages.

[14] Milad Hashemi and Yale N. Patt. 2015. Filtered runahead execution with a runahead buffer. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. 358–369. https://doi.org/10.1145/2830772.2830812

[15] Hidetsugu Irie, Toru Koizumi, Akifumi Fukuda, Seiya Akaki, Satoshi Nakae, Yutaro Bessho, Ryota Shioya, Takahiro Notsu, Katsuhiro Yoda, Teruo Ishihara, and Shuichi Sakai. 2018. STRAIGHT: Hazardless processor architecture without register renaming. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE Press, 121–133. https://doi.org/10.1109/MICRO.2018.00019

[16] 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 *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. 383–396.

[17] Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*. 252–263. https://doi.org/10.1145/264107.264207

[18] Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. 364–373. https://doi.org/10.1145/325164.325162

[19] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2011. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 393–404. https://doi.org/10.1145/1950365.1950411

[20] Dongkeun Kim and Donald Yeung. 2002. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. 159–170.

[21] Dongkeun Kim and Donald Yeung. 2004. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Trans. Comput. Syst.* 22, 3 (Aug. 2004), 326–379. https://doi.org/10.1145/1012268.1012270

[22] Jinchun Kim, Elvira Teran, Paul V. Gratz, D. Jiménez, Seth H. Pugsley, and C. Wilkerson. 2017. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 737–749.

[23] K. Koukos, D. Black-Schaffer, Vasileios Spiliopoulos, and S. Kaxiras. 2013. Towards more efficient execution: A decoupled access-execute approach. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS'13)*. ACM, New York, NY, 253–262. https://doi.org/10.1145/2464996.2465012

[24] R. Kumar, M. Alipour, and D. Black-Schaffer. 2019. Freeway: Maximizing MLP for slice-out-of-order execution. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. 558–569. https://doi.org/10.1109/HPCA.2019.00009

[25] An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. 144–154. https://doi.org/10.1145/379240.379259

[26] 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 (PACT'20)*. 361–372.

[27] A. Lebeck, Tong Li, E. Rotenberg, J. Koppanalil, and J. Patwardhan. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*. IEEE Computer Society, Washington, DC, 59–70. Retrieved from http://dl.acm.org/citation.cfm?id=545215.545223.

[28] Sheng Li, Ke Chen, J. Ahn, J. Brockman, and N. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'11)*. 694–701. Retrieved from http://dl.acm.org/citation.cfm?id=2132325.2132479.

[29] Jiwei Lu, A. Das, Wei-Chung Hsu, Khoa Nguyen, and S. G. Abraham. 2005. Dynamic helper threaded prefetching on the Sun UltraSPARC CMP processor. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. 93–104. https://doi.org/10.1109/MICRO.2005.18

[30] Chi-Keung Luk. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. 40–51.

[31] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. 190–200. https://doi.org/10.1145/1065010.1065034

[32] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. 2001. Slice-processors: An implementation of operation-based prediction. In *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*. ACM, New York, NY, 321–334.

[33] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'03)*.

[34] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. 2020. Precise runahead execution. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA'20)*. 12.

[35] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. 1997. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*. ACM, New York, NY, 206–218. https://doi.org/10.1145/264107.264201

[36] S. Palacharla and R. E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*. 24–33. https://doi.org/10.1145/191995.192014

[37] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*. 115–126. https://doi.org/10.1145/291069.291034

[38] Andreas Sembrant, Trevor E. Carlson, Erik Hagersten, D. Black-Schaffer, Arthur Perais, André Seznec, and P. Michaud. 2015. Long-term Parking (LTP): Criticality-aware resource allocation in OOO processors. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. ACM, New York, NY, 334–346. https://doi.org/10.1145/2830772.2830815

[39] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. 45–57. https://doi.org/10.1145/605397.605403

[40] Ryota Shioya, Masahiro Goshima, and Hideki Ando. 2014. A front-end execution architecture for high energy efficiency. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 419–431. https://doi.org/10.1109/MICRO.2014.35

[41] Michael Ferdman, Stephen Somogyi, and Babak Falsafi. 2006. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA'06)*. 252–263. https://doi.org/10.1109/ISCA.2006.38

[42] Stephen Somogyi, T. Wenisch, A. Ailamaki, and B. Falsafi. 2009. Spatio-temporal memory streaming. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 69–80. https://doi.org/10.1145/1555754.1555766

[43] SPEC. [n.d.]. SPEC CPU2006. Retrieved from http://www.spec.org/cpu2006/.

[44] Hamid Tabani, José-María Arnau, Jordi Tubella, and Antonio González. 2018. A novel register renaming technique for out-of-order processors. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 259–270. https://doi.org/10.1109/HPCA.2018.00031

[45] V. Uzelac and A. Milenkovic. 2009. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 207–217. https://doi.org/10.1109/ISPASS.2009.4919652

[46] T. Wenisch, Stephen Somogyi, N. Hardavellas, Jangwoo Kim, Anastassia G. Ailamaki, and B. Falsafi. 2005. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. 222–233.

[47] Carole-Jean Wu, A. Jaleel, M. Martonosi, S. Steely, and J. Emer. 2011. PACMan: Prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44'11)*. 442–453.

[48] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (Mar. 1995), 20–24. https://doi.org/10.1145/216585.216588

[49] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. 2007. Accelerating and adapting precomputation threads for efficient prefetching. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. 85–95. https://doi.org/10.1109/HPCA.2007.346187

[50] Craig Zilles and Gurindar Sohi. 2001. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. 2–13. https://doi.org/10.1145/379240.379246

Received May 2021; revised November 2021; accepted December 2021