2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)

Freeway: Maximizing MLP for Slice-Out-of-Order Execution

Rakesh Kumar
Norwegian University of Science and Technology (NTNU)
rakesh.kumar@ntnu.no

Mehdi Alipour, David Black-Schaffer
Uppsala University
{mehdi.alipour, david.black-schaffer}@it.uu.se

***Abstract*—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 hardware and the resulting energy overheads. As energy efficiency becomes the prime design constraint, we investigate low complexity/energy mechanisms to exploit MLP.**

This work revisits slice-out-of-order (sOoO) cores as an energy efficient alternative to OoO cores for MLP exploitation. 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, execute in-order. Though their energy overhead is low compared to full OoO cores, sOoO cores fall considerably behind in terms of MLP extraction. We observe that their dependence-oblivious in-order slice execution causes dependent slices to frequently block MLP generation.

To boost MLP generation in sOoO cores, we introduce Freeway, a sOoO core based on a new dependence-aware slice execution policy that tracks dependent slices and keeps them out of the way of MLP extraction. The proposed core incurs minimal area and power overheads, yet approaches the MLP benefits of fully OoO cores. Our evaluation shows that Freeway outperforms the state-of-the-art sOoO core by 12% and is within 7% of the MLP limits of full OoO execution.

I. Introduction

Today's power-constrained systems face challenges in generating memory level parallelism (MLP) to hide the increasing access latencies across the memory hierarchy [1]. 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 across the entire hierarchy, by overlapping memory accesses to hide the latency of later requests in the "shadow" of earlier requests, 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 out-of-order (OoO) execution, as it enables parallel memory accesses by executing independent memory instructions from anywhere in the instruction 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 instruction 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 [2] 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 OoO support for MLP extraction. These cores first construct groups, or *slices*, of MLP generating instructions that contain the address generating instructions leading up to loads and/or stores. The slices are executed out-of-order with respect to the rest of the instructions. However, the slices and the remaining instructions, by themselves, still execute in-order. As the MLP generating slices bypass the rest of the potentially stalled instructions, sOoO cores extract significant MLP. Yet since they only support limited out-of-order execution, they incur only a fraction of the energy cost of the full out-of-order execution.

The state-of-the-art sOoO core, the Load Slice Core (LSC) [3], builds on an in-order stall-on-use core. LSC learns MLP generating instructions using a small hardware table. To enable these instructions to execute out-of-order as regards 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 B-IQ), LSC minimizes the energy requirements while still exploiting MLP.

Though highly energy efficient, existing sOoO cores fall noticeably behind OoO execution in terms of MLP extraction. Our key observation is that *inter-slice dependencies limit MLP extraction opportunities.* For example, when a dependent

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00009

memory slice[1] 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, the 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: its *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 model in favor of a *dependence-aware* slice scheduling model. The 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 unveil 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, via detailed simulations, that Freeway outperforms state-of-the-art sOoO core by 12% and is within 7% of the MLP limits of full OoO execution. We also analyze the remaining bottlenecks that cause this 7% performance gap and show that mitigating them brings minimal performance returns on resource investment.

[1]A *dependent slice* is one that contains at least one instructions that depends on the load instruction of another slice, called *producer slice*.

II. Background and Motivation

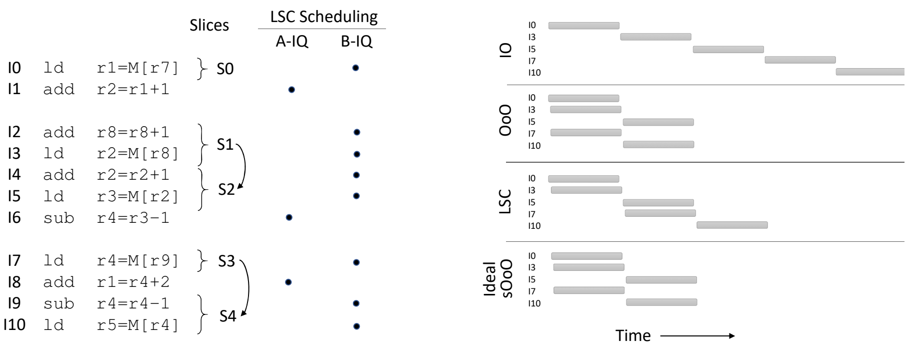
A. MLP vs Energy: IO, OoO, and slice-OoO cores

Existing core designs force a trade-off between MLP and energy efficiency. For example, an in-order (IO) core can be highly energy efficient, but 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 Load Slice Core (LSC) [3], proposed a new approach of *slice-out-of-order* (sOoO) execution. LSC builds on an efficient in-order core and employs separate instruction queues, A-IQ 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 IO and OoO cores in exploiting MLP. As a stall-on-use IO 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 IO 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.

Figure 1 also demonstrates a major limitation of LSC: it is effective in extracting MLP only when the *memory 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 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 with 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 IO 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 [4] has shown that the ability to select arbitrary instructions from IQ is one of the



```
I0  ld   r1=M[r7]
I1  add  r2=r1+1

I2  add  r8=r8+1
I3  ld   r2=M[r8]
I4  add  r2=r2+1
I5  ld   r3=M[r2]
I6  sub  r4=r3-1

I7  ld   r4=M[r9]
I8  add  r1=r4+2
I9  sub  r4=r4-1
I10 ld   r5=M[r4]
```

Figure 1: Overlapping memory accesses in IO, OoO, LSC, and Ideal sOoO. The arrows show inter-slice dependencies.

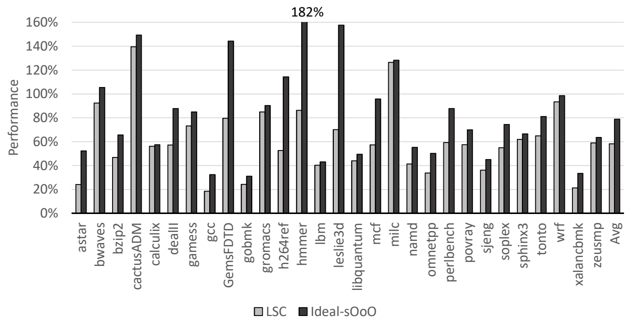


Figure 2: Performance gain for LSC and Ideal-sOoO over IO execution. Prefetching is enabled in all designs.

most energy consuming tasks in OoO cores. Carlson et. al. [3] concluded that LSC incurs only 15% area and 22% power overheads over an IO core (ARM Cortex-A7), whereas an out-of-order core (ARM Cortex-A9) requires 2.5x area and 12.5x 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 memory slices limits its potential to extract MLP in the case of dependent memory slices. To understand this limitation, we next explore its impact on performance.

B. 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 independent slices can bypass other stalled slices. (Our simulation methodology, including microarchitectural parameters, is detailed in Section V.)

Figure 2 shows the performance gains obtained by LSC and Ideal-sOoO (LSC with a fully out-of-order B-IQ) over an IO core. The relative difference between the two shows the opportunity missed by LSC due to its FIFO slice execution. The average performance gain of Ideal-sOoO over LSC is 20%, and more than 50% on `GemsFDTD`, `h264ref`, `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 II-C). Overall, the majority of the workloads demonstrate considerable performance opportunity if we can eliminate the dependent slice bottleneck.

C. 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 (memory 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)[2].
- **Other Stalls:** Intra-slice dependencies, unresolved store addresses blocking younger loads, etc.

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 neither the A-IQ nor B-IQ) as a fraction of overall execution time. The figure reveals that instruction issue is stalled for, on average, 47% of the execution time, 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

[2]In LSC, store data calculation and store operation itself go to the A-IQ, whereas, the store address calculation goes to the B-IQ.

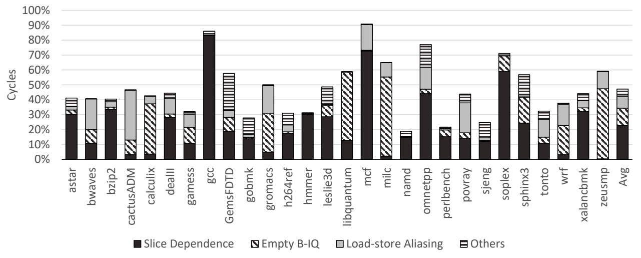


Figure 3: Percentage of execution time the issue stage is stalled in LSC, and the breakdown of stall sources.

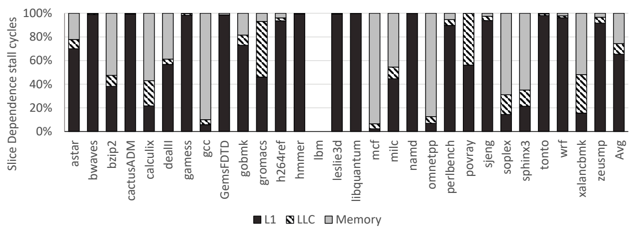


Figure 4: Breakdown of *Slice Dependence Stall* cycles based on the producer slice hit site in the memory hierarchy. `lbm` does not have any dependent slices.

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.

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 [2]. The third largest source of stalls are *Load-store Aliasing Stalls*, and they are particularly severe in `bwaves`, `cactusADM`, `gromacs`, `lbm`, and `povray`. However, on average, they only stall the execution for about 8% of the overall execution time.

Looking at the potential of a fully out-of-order B-IQ, we see that there is roughly a 20% performance gain possible over LSC's in-order B-IQ. The majority of this loss is due to Slice Dependence Stalls, where the B-IQ is blocked by dependent slices. Next, we analyze memory slice behaviour to mitigate this bottleneck.

III. Addressing Slice Dependence

A generic approach to handling 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 memory slice behaviour to understand which dependent slices should be buffered and where they should be buffered.

A. 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 the 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 (4 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.

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 should 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, it would suggest we can achieve these benefits at a low implementation cost.

B. 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 [5], 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 [5], [6]. The extra energy and latency of re-inserting instructions is particularly costly for instruction slices which 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 will cause a bottleneck if the younger slices become ready for execution before the older slices. This may occur primarily for 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

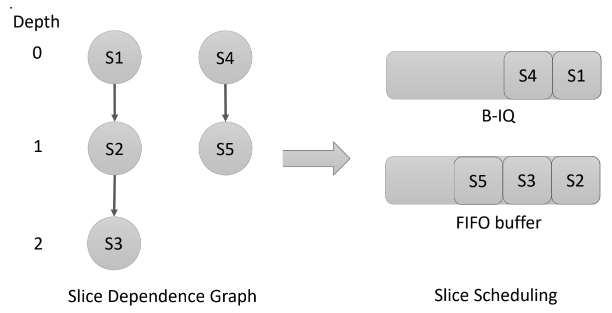


Figure 5: Slice dependence graph with dependence depth (left) and slice scheduling to different queues (right). S1 - S5 are memory slices.

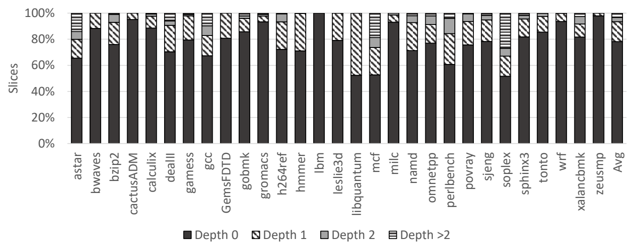


Figure 6: Slice dependence depth distribution.

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 because 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[3], 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 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

[3]Slice execution time is a function of number of instructions in the slice and the hit site of the load ending the slice. However, we discovered that it mostly varies due to the load hit site as most of the slices have similar instruction count.

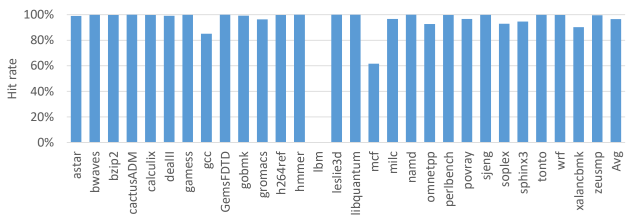


Figure 7: L1 cache hit rate for producer slices at dependence depth 0. `lbm` does not have any dependent or producer slices.

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 (e.g., dependence chain lengths of 2 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. (Our results in Section VI-C validate that additional queues bring only minimal performance gains.)

IV. Freeway

Freeway is a new slice-out-of-order (sOoO) core designed to achieve the MLP benefits of full out-or-order execution. Freeway goes beyond previous work by addressing the sources of sOoO stalls identified in our analysis (Section III) to execute the majority of memory 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.

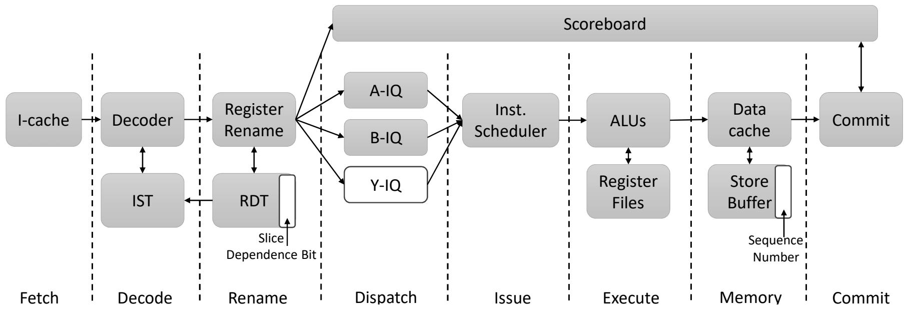


Figure 8: Freeway microarchitecture. New components are in white.

A. Baseline sOoO

The first sOoO design, the Load Slice Core, 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.

To identify MLP generating instructions efficiently, LSC leverages applications' loop behaviour to construct memory slices in a backward iterative manner, 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 memory 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 [7], [8].

To exploit MLP, LSC adds an additional in-order *bypass queue (B-IQ)* to the baseline IO 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.

B. 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, leveraging the insights from Section III, 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 III 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.

C. 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.

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[4]. 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

[4]A slice is not classified as dependent if it depends on a non-load instruction of a older slice.

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.

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 instruction scheduler selects up to two instructions from the heads of the FIFO instruction queues and issues them to the execution units. The instructions can be selected from different queues or from a single queue using an age based policy (Prioritizing slice delivers similar performance). This scheduling policy enables restricted out-of-order execution as younger instructions in one queue can bypass the older instructions waiting the other queues, despite the instructions queues themselves being FIFO. However, such out-of-order execution necessitates tracking the instruction ordering to ensure in-order commit. Freeway, like LSC, employs a Scoreboard for tracking instruction order from dispatch. Instructions record their completion in the Scoreboard as they are executed. When the oldest instruction finishes, it is removed from the Scoreboard in program 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.

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

Core	2GHz, 2-wide issue, 64-entry scoreboard
Branch Predictor	Intel Pentium M-style [9]
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, 4 cycle, 8 MSHRs
LLC	512KB per core, 16-way LRU, avg 30-cycle round-trip latency
Prefetcher	stride-based, 16 independent streams
Main Memory	4 GB/s, 45 ns access latency

Table I: Microarchitectural parameters

is issued to memory as the store is younger.

V. Methodology

To evaluate Freeway, we use the Sniper [10] simulator configured with a cycle-accurate core model [11]. Sniper works by extending Intel's PIN tool [12] with models for the core, memory hierarchy, and on-chip networks. Area and power estimates are obtained from CACTI 6.5 [13] using its most advanced technology node, 32nm. We use the SPEC CPU2006 [14] workloads with reference inputs. Furthermore, we use multiple inputs per workload to evaluate performance, energy, and area, though the results presented in Section II and III use only a single representative input. To keep the simulation time reasonable, SimPoint methodology [15] is used to choose a single most representative region of 1 billion instructions in each application.

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 [16], as a baseline.

Load Slice Core: LSC, as proposed by Carlson et al. [3], with strict in-order memory slice execution.

Freeway: Our proposed design with dependent slice tracking and a FIFO yielding queue (Y-IQ) to enable out-of-order execution among slices.

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 a limit on performance through ILP and MLP exploitation.

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

VI. Evaluation

In this section, we first evaluate the performance benefits from Freeway's dependence aware slice execution, in comparison to LSC, Ideal-sOoO, and full OoO. Next, we breakdown the performance to understand where Freeway's benefit comes from. We then compare against the Ideal-sOoO to analyze the opportunity missed by Freeway, and finally present the area and power requirements of Freeway.

A. 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. On average, Freeway delivers 12% higher performance than LSC as it 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 than Freeway as it exploits both ILP and MLP, whereas Freeway targets only MLP. However, this additional performance comes with high area and power overheads (Section VI-D).

Freeway vs LSC: On individual workloads, Freeway comprehensively outperforms LSC on workloads like `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 gain 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, 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 similar 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 minimum. As a result, Freeway does not have much opportunity for improvement and delivers similar performance as LSC.

Freeway vs Ideal-sOoO vs 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 ideal-

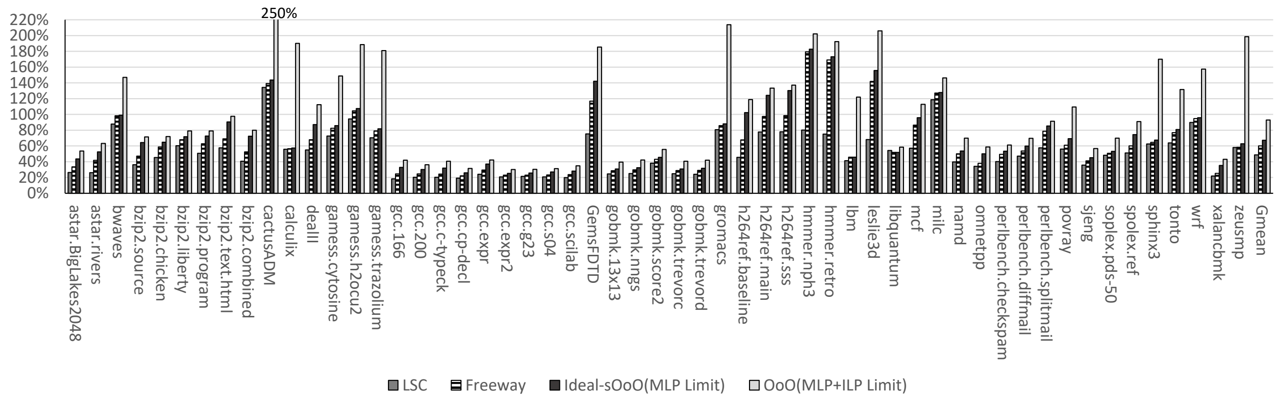


Figure 9: Performance gain of different core designs over in-order core.

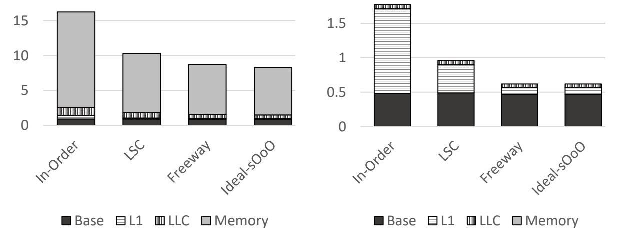


Figure 10: CPI stack for: (a) mcf and (b) hmmer.

ized MLP extraction (Ideal-sOoO design). Comparing to OoO execution, which targets both ILP and MLP, Freeway falls short on workloads that present significant ILP opportunity, such as `calculix`, `gromacs`, `zeusmp` etc., as it exclusively aims for MLP. However, on workloads that offer little ILP, such as `sjeng`, `perlbench`, `xalancbmk`, etc., 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, resulting in a performance difference of 33%. However, this additional performance comes at a significantly higher area and power costs as discussed in Section VI-D.

B. Understanding the Performance Benefits

To better understand the performance gains, we present CPI stacks in Figure 10 that break down the execution time across the memory hierarchy and base components (instruction cache and branch behaviour etc.). For this analysis, we pick two representative workloads, `mcf` and `hmmer`. In `mcf`, the majority of execution time is spent in waiting for data from the off-chip memory, whereas most accesses in `hmmer` hit in the L1 cache.

As expected, Freeway benefits from reducing the memory access time in both workloads as shown in Figure 10. However, as the workloads spend their time waiting on data from different levels in the memory hierarchy, the benefits show up in different parts of the CPI stack. In `mcf`, Freeway reduces average off-chip memory access latency, whereas in `hmmer`, the average L1 access time is shortened. Also, Freeway provides relatively less CPI reduction for `mcf` (1.8x) compared to `hmmer` (3x), over IO execution. This is because, as alluded in Section VI-A, off-chip memory accesses in `mcf` lead to frequent full instruction window stalls that block MLP exploitation. Overall, these results show Freeway's efficacy in tolerating the access latency across the whole memory hierarchy, especially compared to LSC, irrespective of where the bottleneck lies.

C. Analysis of the Remaining Opportunity

Figure 9 shows that despite its dependence aware slice execution, Freeway lags behind the optimal performance "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 detailed in Section II-C (Load-Store Aliasing, Empty B-IQ, etc). 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 the potential solutions to avoid it.

As Figure 3 shows, load-store aliasing is the largest source of potentially mitigable stalls after slice dependence. (*Empty B-IQ* stalls require physical or virtual expansion of the instruction window, and are not considered.) 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 related 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 11 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

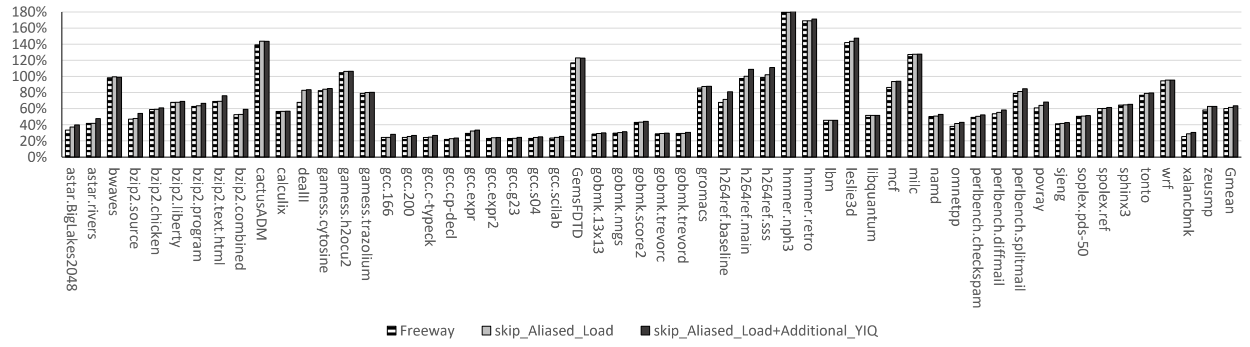


Figure 11: Freeway performance gain achieved by skipping the aliased loads and adding another Y-IQ.

completely addressing these sources of stalls would result in little performance gain.

As discussed in Section III-B, 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 smaller dependence depth or their producers hit closer to the core in the memory hierarchy). As almost all the 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 to 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 11 shows that an additional Y-IQ brings only 1.5% more performance. These results 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, 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.

D. 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% more area than LSC. In contrast, as shown by Carlson et al. [3], 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 [3], show that LSC, Freeway, and OoO core increase the average power consumption by 1.22x, 1.24x, and 12.6x respectively over an in-order core. These results reveal the exorbitant area and power costs of the moderate performance benefits achieved by OoO core over Freeway.

VII. Related Work

A large body of prior research focuses on tolerating memory access latency to prevent cores from stalling. This work 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 [2] 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. Hashemi et al. [17] observed that Runahead incurs significant energy overhead due to the core front-end being operational during the entire runhead duration. They proposed to filter out the instructions leading up to the memory accesses and buffer them in a *Runahead Buffer*. This allowed them to save energy by power- or clock-gating the core front-end while supplying instructions from the Runahead Buffer. Hashemi et al. [18] further observed that Runahead's ability to generate MLP is restricted by the limited duration of runahead intervals, when core is stalled due to full ROB. To address this, they proposed a Continuous Runahead Engine, located at the memory controller, that continuously executes memory slices in a loop irrespective of whether the core is stalled or not.

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 [19], [20], [21], [22], [23], [24], [25] while others have proposed dynamic compilation techniques [26], [27]. These techniques either execute the helpers threads on an available SMT context [19], [24] or require a dedicated core [23].

Collins et al. [28] 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. [29] 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.

Slipstream [30] uses two cores to execute both a filtered version of the application, called the A-stream, ahead of the full application, called R-stream. The A-stream communicates performance hints such as branch-directions or memory addresses for prefetching back to the R-stream. However, the power and area overhead of using two cores to execute a single application is significant.

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. Freeway does not incur such overheads.

Prefetching: Prefetchers predict future addresses based on the 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 [31], [32] require only simple hardware but are limited to regular access patterns. Advanced prefetchers, such as Correlation Prefetchers [33], [34], [35], 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 to prefetch the huge datasets of server applications [36], [37], [38], though they still incur significant storage and energy overhead. Recently, co-design of prefetching with replacement policies has also been explored in [39], [40].

However, the majority of MLP extraction and prefetching techniques build on an energy exhaustive OoO core which necessitates high energy budget. In contrast, Freeway offers a low energy cost alternative to OoO cores that these techniques can leverage to improve overall energy efficiency.

VIII. Conclusion

Tolerating long memory and LLC access latencies is critical for performance. MPL exploitation techniques such as out-of-order execution, runahead execution, etc., 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 *slice-out-of-order* (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 this way, 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, hence 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 outperform previous sOoO designs by 12% and delivers performance within 7% of the MLP limits of the ideal sOoO execution.

IX. Acknowledgments

This work was supported by the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant No 715283).

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[2] O. Mutlu *et al.*, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *International Symposium on High-Performance Computer Architecture*, HPCA '03, 2003.

[3] T. E. Carlson *et al.*, "The load slice core microarchitecture," in *International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 272–284, ACM, 2015.

[4] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, (New York, NY, USA), pp. 206–218, ACM, 1997.

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

[6] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud, "Long term parking (ltp): Criticality-aware resource allocation in ooo processors," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 334–346, ACM, 2015.

[7] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, (Washington, DC, USA), pp. 306–317, IEEE Computer Society, 2001.

[8] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi, "Slice-processors: An implementation of operation-based prediction," in *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, (New York, NY, USA), pp. 321–334, ACM, 2001.

[9] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 207–217, April 2009.

[10] T. E. Carlson, W. Heirman, and L. Eeckhout, "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, (New York, NY, USA), pp. 52:1–52:12, ACM, 2011.

[11] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, pp. 28:1–28:25, Aug. 2014.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pp. 190–200, 2005.

[13] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pp. 694–701, 2011.

[14] SPEC, "Spec cpu2006." http://www.spec.org/cpu2006/.

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

[16] ARM, "Arm cortex-a7 processor." ttp://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[17] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pp. 358–369, 2015.

[18] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pp. 61:1–61:12, 2016.

[19] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pp. 14–25, 2001.

[20] D. Kim and D. Yeung, "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 X, pp. 159–170, 2002.

[21] C.-K. Luk, "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, pp. 40–51, 2001.

[22] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pp. 2–13, 2001.

[23] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pp. 393–404, 2011.

[24] D. Kim and D. Yeung, "A study of source-level compiler algorithms for automatic construction of pre-execution code," *ACM Trans. Comput. Syst.*, vol. 22, pp. 326–379, Aug. 2004.

[25] K. Koukos, D. Black-Schaffer, V. Spiliopoulos, and S. Kaxiras, "Towards more efficient execution: A decoupled access-execute approach," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 253–262, ACM, 2013.

[26] W. Zhang, D. M. Tullsen, and B. Calder, "Accelerating and adapting precomputation threads for effcient prefetching," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pp. 85–95, 2007.

[27] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham, "Dynamic helper threaded prefetching on the sun ultrasparc cmp processor," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pp. 93–104, 2005.

[28] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pp. 306–317, 2001.

[29] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pp. 52–61, 2001.

[30] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," *SIGPLAN Not.*, vol. 35, pp. 257–268, Nov. 2000.

[31] N. P. Jouppi, "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, pp. 364–373, 1990.

[32] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pp. 24–33, 1994.

[33] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pp. 252–263, 1997.

[34] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction &amp; dead-block correlating prefetchers," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pp. 144–154, 2001.

[35] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pp. 115–126, 1998.

[36] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pp. 252–263, 2006.

[37] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pp. 222–233, 2005.

[38] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 69–80, 2009.

[39] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, "Pacman: Prefetch-aware cache management for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 442–453, 2011.

[40] J. Kim *et al.*, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pp. 737–749, 2017.