The Forward Slice Core Microarchitecture

Kartik Lakshminarasimhan
Ghent University, Belgium

Ajeya Naithani
Ghent University, Belgium

Josué Feliu
Universidad de Murcia, Spain

Lieven Eeckhout
Ghent University, Belgium

ABSTRACT

Superscalar out-of-order cores deliver high performance at the cost of increased complexity and power budget. In-order cores, in contrast, are less complex and have a smaller power budget, but offer low performance. A processor architecture should ideally provide high performance in a power- and cost-efficient manner. Recently proposed slice-out-of-order (sOoO) cores identify backward slices of memory operations which they execute out-of-order with respect to the rest of the dynamic instruction stream for increased instruction-level and memory-hierarchy parallelism. Unfortunately, constructing backward slices is imprecise and hardware-inefficient, leaving performance on the table.

In this paper, we propose Forward Slice Core (FSC), a novel core microarchitecture that builds on a stall-on-use in-order core and extracts more instruction-level and memory-hierarchy parallelism than slice-out-of-order cores. FSC does so by identifying and steering forward slices (rather than backward slices) to dedicated in-order FIFO queues. Moreover, FSC puts load-consumers that depend on L1 D-cache misses on the side to enable younger independent load-consumers to execute faster. Finally, FSC eliminates the need for dynamic memory disambiguation by replicating store-address instructions across queues. FSC improves performance by 9.7% on average compared to Freeway, the state-of-the-art sOoO core, across the SPEC CPU2017 benchmarks, while incurring reduced hardware complexity and a similar power budget.

ACM Reference Format:
Kartik Lakshminarasimhan, Ajeya Naithani, Josué Feliu, and Lieven Eeckhout. 2020. The Forward Slice Core Microarchitecture. In *2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3410463.3414629

1 INTRODUCTION

Modern processors are designed to either deliver high performance or provide high energy efficiency. The two ends of the spectrum are represented by superscalar out-of-order (OoO) and in-order (InO) cores, respectively. To deliver high performance, OoO cores are power-hungry due to their higher design complexity and bigger chip area. InO cores on the other hand, consume significantly less power as a consequence of their much simpler design and smaller chip area. An ideal processor design, however, should deliver high performance at a small chip area and power overhead.

Table 1: Comparing FSC against prior slice-out-of-order microarchitectures in terms of ILP, MHP and hardware complexity. ***FSC offers higher ILP and MHP than LSC and Freeway at lower hardware overhead.***

Processor design	ILP	MHP	Hardware Complexity
InO	–	–	+
LSC	+	+	+++
Freeway	++	++	+++
FSC	+++	+++	++
OOO	++++	++++	+++++++

Although in-order cores are highly energy-efficient, their in-program order execution model severely restricts performance compared to OoO cores. Recently, *slice-out-of-order* (sOoO) core microarchitectures have been proposed to address the in-order issue bottleneck by allowing the execution of load and store instructions, plus their backward slices (i.e., the address-generating sequence of instructions leading up to these memory operations) to bypass arithmetic instructions in the dynamic instruction stream. The sOoO cores are restricted out-of-order machines that add modest hardware overhead upon a stall-on-use in-order core to improve instruction-level parallelism (ILP) as well as memory-hierarchy parallelism (MHP).[1] *Load Slice Core (LSC)* [5] was the first work to propose an sOoO core; Freeway [10] builds upon the LSC proposal and exposes more MHP than LSC by adding one more in-order queue for uncovering additional independent loads.

LSC and Freeway identify the *address-generating instructions (AGIs)* of loads and stores in an iterative manner using a hardware mechanism called *Iterative Backward Dependence Analysis (IBDA).* The loads, store-address operations and AGIs execute through a separate bypass queue (B-queue), while all other instructions execute from the main, arithmetic queue (A-queue). Kumar et al. [10] observe that, in LSC, an independent load may be stuck behind a load that depends on an older long-latency memory load, unnecessarily limiting the exploitable MHP. They therefore propose the *Freeway* microarchitecture, which adds one more in-order queue for putting dependent loads on the side so that younger independent loads can go ahead and execute.

PACT '20, October 3–7, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7092-9/20/08…$15.00
https://doi.org/10.1145/3410463.3414629

[1] We refer to MHP to denote parallelism across the memory hierarchy including the various levels of cache and main memory. Formally, MHP is defined as the average number of overlapping memory accesses that hit anywhere in the cache hierarchy, including main memory.

Table 1 compares the InO, LSC, Freeway and OoO microarchitectures in terms of ILP, MHP and design complexity. Since both LSC and Freeway execute instructions from multiple queues, the degree of ILP exposed by sOoO cores is higher than an in-order core. sOoO cores expose substantially higher MHP by allowing independent loads and their backward slices to execute ahead of older (possibly stalled) instructions. Unfortunately, IBDA requires dedicated hardware. More specifically, LSC and Freeway rely on two structures, namely the *Register Dependence Table (RDT)* and the *Instruction Slice Table (IST)*, for identifying backward slices. Not only do the RDT and IST incur hardware overhead, IBDA is also imperfect. In particular, a workload for which the code footprint is too big to fit within the IST may lead to IST misses. Moreover, iteratively constructing backward slices leads to imprecise backward slices, further limiting the exploitable MHP. Another source of increased complexity lies in memory disambiguation. The LSC eliminates the need for dynamic memory disambiguation by executing all memory instructions in program order from the B-queue. In Freeway on the other hand, memory instructions can execute out-of-order, which requires expensive content-addressable hardware support for correctly handling memory dependences. In summary, backward slice analysis is imprecise and adds hardware complexity; Freeway further increases complexity by requiring hardware support for dynamic memory disambiguation.

In this paper, we propose *Forward Slice Core (FSC)*, a novel core microarchitecture, that builds on a stall-on-use in-order core but achieves higher ILP and MHP compared to prior sOoO cores at lower hardware cost. In contrast to sOoO cores which target backward slices of both loads and stores, the FSC targets *forward slices* of *only loads*. A forward slice consists of the direct and indirect dependents of a load that is yet to be executed. At dispatch time, all instructions — including loads — that are not part of a forward slice are steered to an in-order FIFO queue, the so-called *Main Lane (ML)*. This enables the execution of instructions that are independent of older loads to execute as soon as possible. Forward-slice instructions are steered to dedicated FIFO queues: loads are sent to the *Dependent Load Lane (DLL)* and non-loads are sent to the *Dependent Execute Lane (DEL)*. Instructions that wait at the head of the DEL for more than a preset number of cycles (i.e., the L1 D-cache access time) are sent to the *Holding Lane (HL)* to enable instructions that are independent of L1 D-cache misses to be selected for execution. The four in-order queues (ML, DEL, DLL and HL) issue instructions in program order, but operate out-of-order with respect to each other, i.e., instructions at the head to the queues are selected for execution on a functional unit as soon as their register dependences have been resolved.

There are three key differences that set the FSC microarchitecture apart from prior work. First, FSC operates on forward slices rather than backward slices. This simplifies the hardware: building forward slices can be done in a single pass whereas constructing backward slices requires multiple passes using dedicated RDT and IST hardware structures. Second, FSC drains all DEL instructions waiting on an L1 D-cache miss to a separate Holding Lane. These instructions cause the longest performance stalls in an in-order core, and re-directing them to a separate queue accelerates the execution of younger independent instructions. Third, FSC performs memory disambiguation in a novel manner through *store-address replication (SAR)*. When dispatching a store instruction, FSC replicates the store-address micro-op across all FIFO queues. The store-address micro-op is issued to a functional unit only when all copies of the micro-op are at the heads of their respective queues. This ensures that loads can never bypass older stores, i.e., loads always execute in program order with respect to older stores. SAR greatly simplifies dynamic memory disambiguation, in contrast to Freeway, which requires expensive content-addressable memory (CAM) look-ups prior to the execution of a load to verify there are no prior unresolved and aliasing stores.

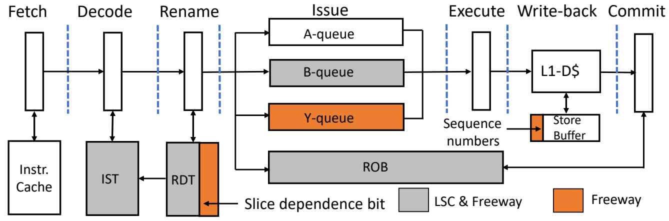


Figure 1: Slice-out-of-order cores: LSC and Freeway add a number of new structures color-coded in gray and orange, respectively. *LSC steers loads and their backward-slices to the B-queue. Freeway in addition steers dependent backward slices to the Y-queue. Non-backward-slice instructions are steered to the A-queue in both LSC and Freeway.*

As summarized in Table 1, FSC achieves higher ILP and MHP at less hardware complexity compared to the previously proposed sOoO processors. *FSC achieves higher MHP* by focusing on forward slices rather than backwards slices. Backward slice construction is an iterative and imperfect process, in contrast to identifying forward slices. *FSC achieves higher ILP* by steering arithmetic instructions across multiple lanes, and by re-directing instructions that depend on L1 D-cache misses to the Holding Lane, paving the way for younger independent arithmetic instructions to execute. *FSC reduces hardware complexity* by eliminating the need for dedicated RDT and IST hardware tables and by eliminating the need for expensive memory disambiguation support.

We experimentally evaluate the proposed FSC microarchitecture through detailed cycle-level simulation using the SPEC CPU2017 benchmarks. We report that FSC outperforms the state-of-art sOoO core microarchitecture, Freeway, by 9.7% on average while being more hardware-efficient (requiring 1408 less bytes). Compared to an in-order core baseline, FSC achieves 64% higher performance on average, versus 44% for LSC and 50% for Freeway. FSC is within 6.9% of an out-of-order processor, while consuming 56% less power.

2 BACKGROUND AND MOTIVATION

In this section, we briefly cover the background on the two prior sOoO cores — LSC [5] and Freeway [10] — and we elaborate on their shortcomings. Figure 1 provides a schematic overview of the two sOoO core microarchitectures.

2.1 Load Slice Core

In a stall-on-use in-order core, an instruction that depends on a load miss stalls the head of the issue queue. The processor is stalled for tens up to hundreds of cycles depending on where the miss is

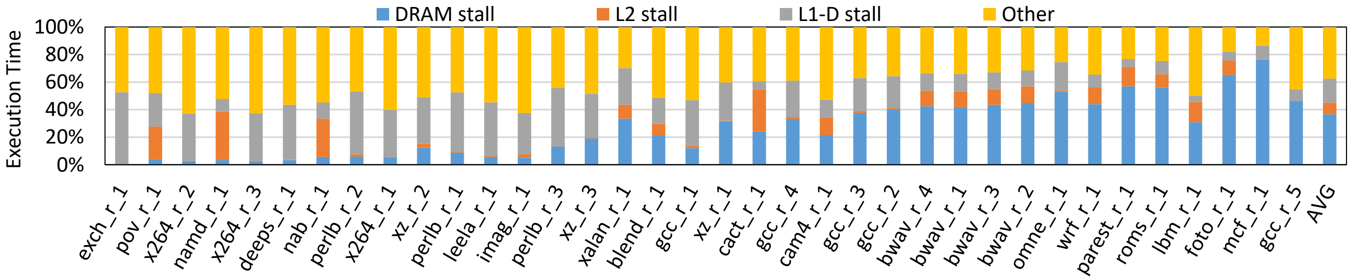


Figure 2: CPI stacks for the SPEC CPU2017 benchmarks (sorted by LLC MPKI from left to right) on a stall-on-use in-order core. *Compute-intensive workloads frequently stall on L1/L2 load consumers, whereas memory-intensive workloads frequently stall on memory access consumers.*

serviced, i.e., the next level of cache or main memory. This hinders future independent loads from accessing the memory hierarchy. To be able to issue independent loads as soon as possible, LSC separates loads and their backward slices, i.e., the *Address-Generating Instructions (AGIs)*, into a separate in-order queue, called the *bypass queue* or B-queue. All other instructions — primarily arithmetic instructions — are issued from the *arithmetic queue* or A-queue. Store instructions are broken down into *store-address (STA)* and *store-data (STD)* micro-ops; the STA micro-op is dispatched to the B-queue (along with its AGIs), whereas the STD micro-op is dispatched to the A-queue. Sending the instructions that depend on a load to the A-queue enables LSC to extract more MHP from the instruction stream compared to an in-order core, i.e., multiple independent loads can issue in parallel from the B-queue even if there are load-dependent instructions in-between the loads in the instruction stream. Although instructions are issued in program order from the A and B-queues, they can be issued out-of-order with respect to each other.

Identifying backward slices incurs additional hardware. LSC uses a mechanism called *Iterative Backward Dependency Analysis (IBDA)* to do so: backward slices are identified iteratively across multiple executions of the same code (e.g., multiple iterations of the same loop or multiple invocations of the same function). IBDA relies on two dedicated hardware structures as shown in Figure 1. LSC piggybacks on register renaming by adding a new hardware structure, called the *Register Dependence Table (RDT)*, to identify the AGIs leading to a load instruction in an iterative manner. The backward-slice instructions identified by IBDA are stored in a dedicated cache, called the *Instruction Slice Table (IST)*. Future occurrences of AGI instructions in the instruction stream are identified by consulting the IST: an instruction is considered a backward-slice instruction if it hits in the IST — if so, the instruction is steered to the B-queue. According to our experimental results, LSC achieves 44% higher performance than an InO core.

2.2 Freeway

While steering load-dependent instructions to a separate A-queue paves the way for more loads to access the memory hierarchy in parallel, LSC still serializes all loads from the B-queue. In particular, in case of load-dependent loads, i.e., a load that depends on a older load, the head of the B-queue stalls on the dependent load. Therefore, younger independent loads behind the dependent load cannot issue to the memory hierarchy, hindering the opportunity to expose MHP.

Kumar et al. [10] propose *Freeway*, a core microarchitecture that overcomes LSC's MHP bottleneck caused by load-dependent loads. Freeway splits a slice that contains multiple loads into two types: a producer slice and a dependent slice. The producer slice ends with a load; the dependent slice starts after the load, and ends on another load. Freeway steers the dependent slice to a new in-order queue called the *yielding queue* or Y-queue. Parking dependent slices in the Y-queue enables Freeway to issue independent slices from the B-queue, exposing more MHP than LSC. The dependent slices from the Y-queue are issued to the memory hierarchy when their producer slices finish execution. By exposing more MHP, Freeway achieves 4% higher performance than LSC (according to our experimental results). Freeway adds complexity over LSC by adding a third queue and, more importantly, by requiring memory disambiguation to allow out-of-order execution of loads and stores.

2.3 Shortcomings of Slice-Out-of-Order Cores

There are three major shortcomings with sOoO cores which we address in this work.

Limited Instruction-Level Parallelism. The sOoO cores are fundamentally limited in the way they can extract instruction-level parallelism (ILP) from the dynamic instruction stream. The reason is that younger independent instructions may be stuck behind load-consumers. In particular, an instruction waiting for a load to return from the memory hierarchy may stall the head of the A-queue for a few cycles (in case of an on-chip cache hit) or for many cycles (in case of an off-chip memory access). None of the instructions in the A-queue can be issued until the stall resolves, even if the instructions are independent of the instruction stalling the A-queue.

Figure 2 supports this by showing normalized CPI stacks for the SPEC CPU2017 benchmarks on a stall-on-use in-order core. (Please refer to Section 4 for details regarding the experimental setup.) These normalized CPI stacks report the fraction stall cycles due to a consumer of an L1 D-cache access, an L2/LLC D-cache access[2] or a main memory access; the remaining cycles are classified as 'other'. (The benchmarks in Figure 2 are sorted from left to right by increasing number of LLC misses per-kilo instructions (MPKI).) Memory-intensive benchmarks appearing in the right half of the figure, spend the majority of their time waiting for data to return

[2] We consider a two-level hierarchy, so L2 is the last-level cache (LLC).

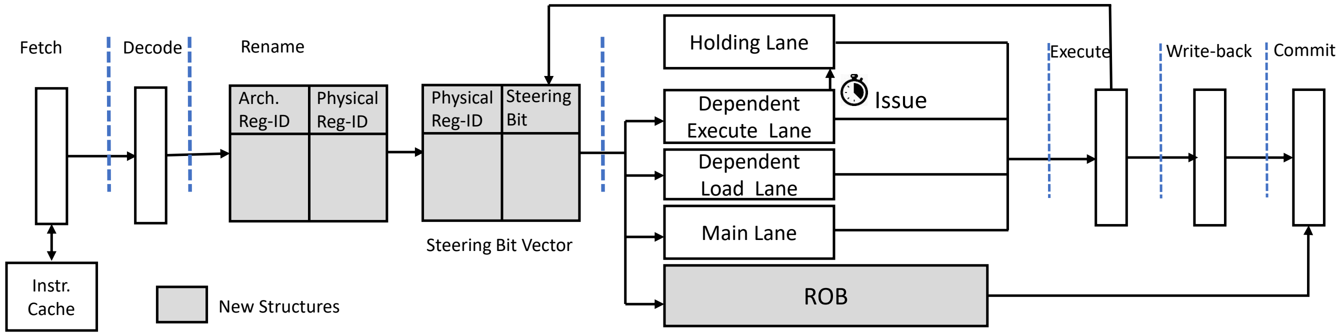


Figure 3: Forward Slice Core (FSC) architecture. FSC adds a number of new structures (gray components) over an stall-on-use in-order core. *FSC consists of four in-order queues: non-forward-slice instructions are steered to the Main Lane (ML), forward-slice instructions are steered to the Dependent Execute Lane (DEL) and the Dependent Load Lane (DLL); DEL instructions that stall on a L1 D-cache miss are re-directed to the Holding Lane (HL) so that independent forward-slice instructions can execute as soon as possible.*

from main memory. Compute-intensive benchmarks, on the left-hand side of the figure, spend almost half of their total execution time waiting for L1 and L2 D-cache accesses. This suggests that allowing instructions that are independent of on-chip cache hits and their consumers to execute ahead, can significantly improve ILP for the compute-intensive workloads.

Limited Memory-Hierarchy Parallelism. sOoO cores expose higher degrees of MHP compared to InO cores, which is beneficial for memory-intensive benchmarks. However, the MHP on sOoO cores is still limited by at least two factors. First, sOoO cores rely on the IBDA mechanism to identify AGIs. IBDA is supported by the IST hardware structure which is of limited size. Hence, if the total set of AGIs for a particular workload (i.e., a workload with a large code footprint) exceeds the size of the IST, this may lead to IST misses which will cause AGIs to be sent to the A-queue and which will hinder the level of MHP that can be extracted. Moreover, backward slice analysis is an iterative and imprecise process. Hence, while building up the backward slices, AGIs will be sent to the A-queue, which also hinders the exploitable MHP. Second, and more specifically to Freeway, dependent slices serialize in the Y-queue. Hence, a dependent slice which gets stalled in the Y-queue may hinder a younger independent slice in the Y-queue to execute. So, in conclusion, even though sOoO cores significantly improve the exploitable MHP over an in-order core, there is still room for improvement.

Hardware Complexity. sOoO cores incur hardware overhead over in-order cores. First, sOoO cores require dedicated hardware structures to dynamically compute backward slices. sOoO cores do so by using the RDT and IST structures. The RDT is 512 Bytes in size while the IST incurs a hardware overhead of 768 Bytes. Moreover, the IST needs to have N read and N write ports to support an N-wide superscalar pipeline, *and* the IST needs to be accessed within a single clock cycle. This may be challenging (or even problematic) for wide and high-frequency pipelines.

Second, and more specifically to Freeway, memory disambiguation incurs a non-trivial hardware cost. In particular, to guarantee that all memory dependences are respected, Freeway marks all loads and stores with a sequence number in program order. When issuing a load, a look-up is performed in the store buffer to verify whether all older stores have computed their addresses. A load proceeds only if there are no unresolved and aliasing stores. This operation requires (i) comparing the sequence number of the load against all stores in the store buffer, and (ii) an associative comparison of the memory addresses. Overall, the complexity for handling memory disambiguation in Freeway is close to that of an OoO core.

3 FORWARD SLICE CORE

We propose the *Forward Slice Core (FSC)* microarchitecture to address the aforementioned shortcomings of sOoO cores. Figure 3 provides an overview of the FSC microarchitecture. The general intuition of the FSC microarchitecture is to steer instructions to different in-order FIFO queues depending on whether an instruction is a load-consumer, i.e., whether an instruction depends (directly or indirectly) on an older load. In addition, instructions that depend on an L1 D-cache miss are re-directed to a separate queue to enable younger independent load-consumers to make forward progress. We now discuss the various unique components of the FSC microarchitecture.

3.1 Identifying Forward Slices

We define a *forward slice* as the sequence of instructions that depend (directly or indirectly) on a load instruction. The FSC microarchitecture identifies forward-slice instructions dynamically in hardware using a bit vector called the *Steering Bit Vector (SBV)*. The SBV is indexed by a physical register tag and initially all bits of the vector are cleared. Upon register-renaming a load in the front-end of the pipeline, the SBV bit corresponding to the destination physical register of the load is set. When a younger instruction reads (consumes) a physical register for which the corresponding SBV bit is (still) set, the SBV bit corresponding to the destination physical register of this instruction is also set. This process propagates the dependence chain of a load forward in the dynamic instruction stream, hence the name 'forward slice'.

An SBV bit is cleared when the instruction executes and has computed its destination physical register. Clearing a steering bit

Algorithm 1: FSC Instruction Steering.

```
if a forward-slice instruction then
    if a load instruction then
        if $\exists$ a free entry in DLL then
            Dispatch to DLL
        else
            Stall dispatch
    else
        if $\exists$ a free entry in DEL then
            Dispatch to DEL
        else
            Stall dispatch
else
    if $\exists$ a free entry in ML then
        Dispatch to ML
    else
        Stall dispatch
```

in the SBV indicates that a future instruction reading the corresponding physical register does not need to wait for the execution of the instruction, i.e., its input register is available and the future instruction can immediately read the value from the physical register file. In other words, the SBV keeps track of the forward-slice instructions that are still waiting for their input registers to be computed. Or, more precisely, the SBV keeps track of the physical registers that are yet to be written along the loads' forward slices.

3.2 Instruction Steering

We make a distinction when steering or dispatching forward-slice versus non-forward-slice instructions. A *forward-slice instruction* is an instruction for which at least one of the input physical registers has the SBV bit set. When none of the SBV bits are set, the instruction does not belong to a forward slice and is therefore a *non-forward-slice instruction*. Note that the first load of a chain of dependent instructions is a non-forward-slice instruction; all instructions that (indirectly) depend on the load are forward-slice instructions.

Non-forward-slice instructions are sent to the 'main' in-order FIFO queue, called the *Main Lane (ML)*. FSC includes two more FIFO queues for handling forward-slice instructions: a *Dependent Load Lane (DLL)* and a *Dependent Execute Lane (DEL)*. Loads among the forward-slice instructions are dispatched to the DLL, while all other forward-slice instructions are dispatched to the DEL. The reason for steering load and non-load forward-slice instructions to different queues is to enable younger independent non-load instructions to execute ahead of older load-dependent loads.

The mechanism for steering instructions in FSC is presented in Algorithm 1. Forward-slice instructions are steered to the DLL in case of a load, and to the DEL in case of a non-load instruction. Non-forward-slice instructions are steered to the main lane. When a lane is full upon steering a new instruction, dispatch is stalled. Back-pressure causes the rest of the front-end pipeline to stall.

3.3 Holding Lane

The forward-slice instructions, by definition, wait for data to return from the memory hierarchy. The number of cycles that forward-slice instructions stall depends on whether and where the load hits in the memory hierarchy. In case of a hit in the on-chip cache hierarchy, the forward-slice instructions have to wait for only a couple cycles or at most a dozen cycles. In case of an LLC miss on the other hand, the forward-slice instructions need to wait on the order of a hundred or more cycles. In other words, forward-slice instructions that depend on L1 D-cache misses stall the DEL/DLL queues for a (large) number of cycles, preventing younger independent forward-slice instructions to execute, severely limiting performance.

We therefore introduce the *Holding Lane (HL)*. The basic intuition is to gradually filter out instructions that belong to the forward slices of L1-missing loads and move those instructions to the HL to allow younger independent forward-slice instructions to execute earlier. In particular, an instruction at the head of the DEL is moved to the HL when its producer load misses in the L1 D-cache. This is implemented by setting a counter to a pre-set value (i.e., the L1 D-cache access time) whenever a new instruction reaches the DEL head. The counter is decremented every cycle and when the counter reaches zero — this denotes an L1 D-cache miss — the instruction at the DEL head is moved to the HL. The instructions in the DEL are then moved up one place and the counter is reset to its pre-set value after which it starts decrementing again. A forward slice that depends on an L1 D-cache miss thus gradually migrates from the DEL to the HL, so that other independent forward-slice instructions can execute sooner. Note that FSC does not move instructions from the HL back to the DEL.

The pre-set value is set such that it enables identifying L1 misses in a cost-effective way. In our experimental setup, we set this value to 4, i.e., the access time to the L1 D-cache. A forward-slice instruction that waits at the DEL head for four cycles implies that it depends on an L1 miss and FSC moves the instruction to the HL. This is a hardware-efficient implementation. An alternative implementation would be to notify the core upon an L1 D-cache miss. The counter implementation allows for determining an L1 miss locally within the core at low overhead.

FSC does not re-direct instructions from the DLL to the HL — only DEL instructions are moved to the HL. We recognize that the DLL head may also stall on long-latency loads in case of miss-dependent misses. This may happen in case of pointer-chasing code patterns. Nevertheless, we experimentally observe a negligible performance impact from filtering out instructions from the DLL to the HL, in contrast to the DEL. There are two reasons. First, this is an infrequent scenario because the DLL is stalled less frequently compared to the DEL. For our set of workloads, only 10% of the instructions are steered to the DLL on average, out of which only a minority depend on L1 D-cache misses; hence, re-directing these miss-dependent loads to the HL has limited impact. Second, the opportunity to improve performance is less. Putting a miss-dependent miss out of the way quickly leads to the next miss-dependent miss, which does not improve performance.

Note that the instructions are never steered to the HL from the front-end; instructions are only steered to the ML, DLL and DEL, and forward-slice instructions in the DEL can be re-directed to the HL. However, instructions are selected for execution on a functional unit from the four lanes. At most two instructions can be issued per cycle. We use an oldest-first policy for selecting instructions when there are multiple instructions ready at the heads of the lanes in a given cycle.

3.4 Store-Address Replication

In FSC, a load instruction is steered to the ML or DLL. A store instruction is broken up in a store-address (STA) micro-op that computes the memory address and a store-data (STD) micro-op that performs the actual store operation. The STD micro-op is steered to the ML or DEL, and may be dynamically re-directed to HL. A load instruction may therefore bypass an older store. While executing a load ahead of an earlier store helps improve performance, one has to be careful and respect through-memory dependences at all times. In particular, a load that executes before an older (unresolved) store may possibly read an old value if the load and store reference the same (or overlapping) memory address(es). Correctly handling memory dependences while executing loads and stores out of program order requires complex memory disambiguation logic.

We propose *Store-Address Replication (SAR)* as a simple yet elegant solution to the memory disambiguation problem; SAR is applicable to any multi-queue architecture, including FSC. SAR replicates the STA micro-op across all four lanes upon instruction steering. An STA micro-op is selected for execution if its input register operands are available *and* if it is at the head of *all* the lanes. FSC executes the STA micro-op from the ML and discards the duplicate copies from the other lanes. SAR guarantees that younger loads after the STA micro-op in program order effectively execute after the STA micro-op. Note that SAR guarantees that STA micro-ops are ordered with respect to loads, and loads are ordered with respect to STA micro-ops, however, SAR does not impose any ordering among loads in-between two consecutive STA micro-ops.

3.5 Code Example

Figure 4 illustrates the difference between the various sOoO cores using a hot loop taken from the SPEC CPU2017 mcf_r_1 benchmark. The code includes one AGI, 5 loads (L), 2 arithmetic instructions (E) and 2 stores (S). The subscripts indicate program order. LSC steers the AGI and all the loads to the B-queue; all other instructions are steered to the A-queue. Freeway further steers the dependent load slices (L_4 and L_6) to the Y-queue. FSC steers the non-forward-slice instructions to the ML and the forward-slice instructions (i.e., the load-consumers) to the DEL and DLL. Assuming now that L_2 is an L1 D-cache miss or an LLC miss, FSC will re-direct E_7 and S_8 to the HL, paving the way for E_9 and S_{10} to execute. LSC and Freeway on other hand do not allow E_9 and S_{10} to move ahead, limiting performance.

3.6 Hardware Complexity

Overall, the hardware cost and complexity is less for FSC compared to Freeway, the state-of-the-art sOoO core microarchitecture. Compared to Freeway, FSC removes the IST (768 bytes), the RDT (512 bytes), and the load sequence numbers in the store buffer (48 bytes).

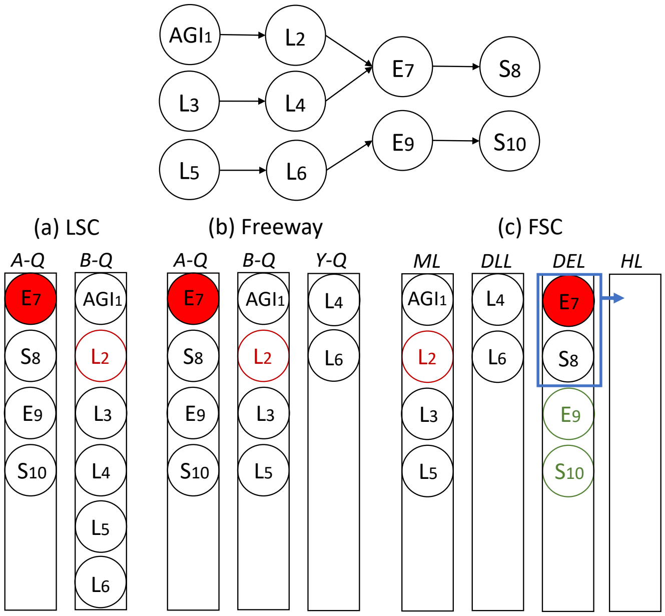


Figure 4: Code example taken from mcf_r_1**'s hot loop illustrating how instructions are steered (and re-directed) to queues.** *FSC steers non-forward-slice instructions to the ML and the forward-slice instructions (i.e., load-consumers) to the DEL and DLL. Instructions in the DEL that depend on an L1 D-cache miss (e.g., E_7 and S_8) are re-directed to the HL.*

In our implementation, FSC features four queues with 8 entries each versus three queues with 12 entries each for Freeway — a reduction by 88 bytes. FSC requires new structures with minimal hardware cost: the Steering Bit Vector (8 bytes) and a count-down timer at the DEL head for moving instructions to the HL (3 bits). In addition, some logic is required for duplicating STA micro-ops at instruction steering and for re-directing instructions from the DEL to the HL. The steering logic has similar complexity for FSC and Freeway because both architectures dispatch instructions into three lanes. The total hardware cost for FSC over an in-order core amounts to 2,388 bytes, versus 3,796 bytes for Freeway — a reduction by 1408 bytes compared to Freeway. Note that this calculation does not account for the content-addressable logic needed to support dynamic memory disambiguation in Freeway, which is significant. We conclude that FSC incurs (significantly) less hardware compared to Freeway.

4 EXPERIMENTAL SETUP

We evaluate FSC using the most detailed, cycle-level, and hardware-validated core model in Sniper v6.0 [4]. The configurations for the InO, FSC and OoO cores are provided in Table 2. We evaluate LSC and Freeway following the configurations by Kumar et al. [10]. The size of the A and B queues in LSC is 16-entries each. Freeway has three queues of size 12 entries each. FSC has four lanes with 8 entries each. For fair comparison, the total number of in-flight instructions equals 32 for all core microarchitectures evaluated in this work. All simulated cores are 2-wide superscalar processors as we target small embedded and mobile processors. LSC, Freeway and

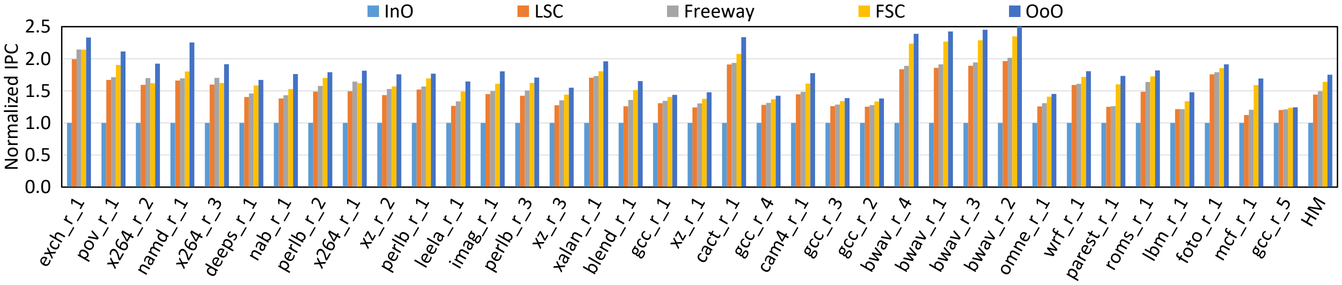


Figure 5: Performance of LSC, Freeway, FSC and OoO cores normalized to the baseline InO core. *FSC improves performance by 64% on average compared to an InO core, versus 44% for LSC and 50% for Freeway. The OoO core improves performance by 75% on average.*

Table 2: Simulated InO, FSC and OoO configurations.

	InO core	*FSC*	*OoO core*
Frequency	2.0 GHz		
Branch Predictor	1.5 KB hybrid local/global/loop predictor and BTB		
Scorebrd/ROB	32		
Issue queue	—	8-entry ML/DEL/DLL/HL	32-entry
Store queue	16		
Pipeline width	2		
Pipeline depth	5 front-end pipeline stages		
Register file	32 int, 32 fp		
ALUs	2 int add (1 c), 1 int mul (3 c), 1 int div (18 c) 1 fp add (3 c), 1 fp mul (5 c), 1 fp div (6 c)		
MMU ports	2 ld/sta, 1 std, 1 sta		
L1 I-cache	4-way 32 KB, 2 cycles		
L1 D-cache	8-way 32 KB, 4 cycles (1-cycle tag look-up)		
L2	8-way 512 KB, 8 cycles (3-cycle tag look-up)		
Memory	3.8 GB/s, 45 ns		

FSC deploy an oldest-first issue policy, which selects up to two of the oldest operand-ready instructions from the two, three and four queues, respectively; up to two instructions can be selected from the same queue. We assume perfect memory disambiguation for the OoO core (i.e., assuming a perfect memory-dependence predictor); in Freeway, a load waits for unresolved and aliasing older stores; LSC and FSC execute loads and stores in program order. The IST is modeled the same way for both LSC and Freeway — we assume a 128-entry 2-way set-associative cache with 2/2 read/write ports.

We estimate power consumption and chip area using McPAT [11] and CACTI v6.5 [12] assuming a 22 nm technology node. Area and per-access power estimates for the newly added FSC hardware structures are calculated using CACTI. We compute chip area and per-component static power consumption and per-access power values from CACTI. Dynamic power is calculated by combining the per-access power values with the activity factors obtained from the timing model, which are then added to the power consumption numbers provided by McPAT.

We create representative 1B-instruction SimPoints [17] for the SPEC CPU2017 benchmarks. We sort the benchmarks by increasing number of last-level cache (LLC) misses per-kilo instructions (MPKI). We notice the maximum MPKI of 48 for gcc_r_5.

5 EVALUATION

We evaluate the following processor cores:

- **InO:** The baseline stall-on-use in-order core, which is modeled to resemble an ARM Cortex-A7 [2].
- **LSC:** The load slice core, as discussed in Section 2.1.
- **Freeway:** The Freeway microarchitecture, as described in Section 2.2.
- **FSC:** The Forward Slice Core microarchitecture proposed in this work.
- **OoO:** The out-of-order core from Table 2.

We compute per-application performance as *instructions per cycle (IPC)* and the overall performance across all the benchmarks is calculated using the harmonic mean IPC [7].

5.1 Overall Performance Results

Figure 5 reports performance for all the evaluated cores across the SPEC CPU2017 benchmarks. Overall, FSC achieves substantially higher performance than InO, LSC and Freeway. Relative to the baseline InO core, LSC and Freeway improve performance by 44% and 50% on average, respectively. By generating both higher ILP and MHP, FSC outperforms the InO core by 64%; this is an additional gain of 9.7% (or 14 percentage point) over Freeway, the state-of-the-art sOoO core. Furthermore, it is interesting to note that FSC performance gets close to that of the OoO core, which achieves a performance gain of 75% over the InO core. In other words, FSC performs within 6.9% (or 11 percentage point) of the OoO core. In conclusion, by steering instructions to the different FIFO in-order queues based on the notion of a forward-slice and by dynamically re-directing instructions that depend on L1 D-cache misses to a separate holding lane, the FSC is able to bridge a large fraction of the performance gap between an InO and OoO core.

5.2 CPI Stack Analysis

We now analyze the performance for four representative benchmarks from the SPEC CPU2017 suite to provide further insight into why FSC outperforms LSC and Freeway. We select one highly compute-intensive benchmark (povray_r_1), one highly memory-intensive benchmark (mcf_r_1), and two benchmarks (blender_r_1 and omnetpp_r_1) with characteristics in-between the two opposite ends. Figure 6 provides CPI stacks. A CPI stack visualizes where time is spent and is represented as a stacked bar in which the base component denotes the fraction of time during which useful work is done. The other CPI components are added on top of the base component and represent lost cycles due to branch mispredictions, L1-D, L2-D and DRAM accesses, plus other stall events.

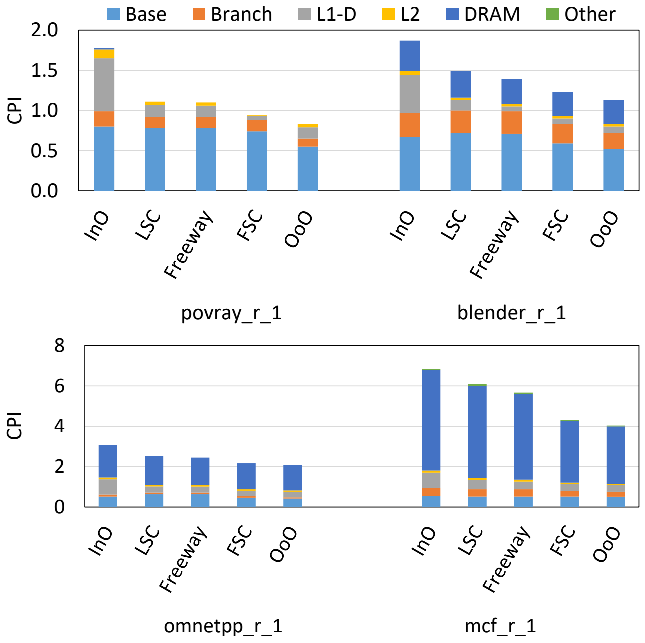


Figure 6: CPI stacks for select benchmarks comparing InO, LSC, Freeway, FSC and OoO. *FSC improves performance by attacking different performance bottlenecks for different workloads.*

povray_r_1 is a compute-intensive benchmark that spends almost half of its time waiting for L1 D-cache accesses on an in-order core. LSC and Freeway steer the L1 D-cache hit accesses and their consumers to separate queues, i.e., loads go to the B-queue and the load consumers go to the A-queue. This dramatically reduces the L1-D component for LSC and Freeway. FSC bridges the gap between LSC/Freeway and OoO by further reducing the L1-D and L2-D components. The L1-D component is improved by getting rid of imperfect backward slice identification in LSC/Freeway. Re-directing instructions that depend on L2 D-cache accesses to the HL enables younger independent DEL instructions to execute sooner.

blender_r_1 spends 20% of its time on DRAM stalls. LSC significantly reduces the L1-D component, which Freeway reduces further by adding one additional queue over LSC. FSC improves the base component compared to LSC and Freeway by exposing more ILP.

omnetpp_r_1 is a memory-intensive benchmark that spends about half of its execution time waiting for DRAM. LSC and Freeway reduce the DRAM stalls by 10% and 15%, respectively, relative to the baseline in-order core. FSC reduces the DRAM CPI component by 20% by virtue of re-directing DEL instructions that wait for long-latency loads to the HL. Moreover, omnetpp_r_1 also benefits from improved ILP compared to the other sOoO cores.

mcf_r_1 is a highly memory-intensive benchmark that spends 72% of its execution time waiting for DRAM. LSC, Freeway and FSC reduce the DRAM CPI component by 9%, 15% and 39%, respectively. Re-directing instructions that depend on long-latency loads from the DEL to the HL clears the way for independent instructions in the DEL. This, in turn, expedites the execution of independent load instructions in the DLL. This explains the significant reduction in the DRAM component for FSC compared to Freeway and LSC.

5.3 Detailed Analysis

We now investigate where the performance benefit is coming from in terms of the Dependent (DEL and DLL) Lanes and the Holding Lane. We therefore consider a number of FSC variants, see also Figure 7 for their normalized performance:

- **ML+DL:** We consider FSC with only two lanes, the main lane (ML) and a unified Dependent Lane (DL), i.e., DEL and DLL are unified in a single lane. There is no Holding Lane.
- **ML+DEL+DLL:** We consider FSC with three lanes: ML, DEL and DLL. There is no Holding Lane.
- **FSC:** The final FSC microarchitecture with four lanes: ML, DEL, DLL and HL.

The ML+DL configuration improves performance by 44% on average compared to an in-order core. Load-consumers are dispatched to the dependent lane which prevents stalling the main lane as in a stall-on-use in-order core. This allows executing younger independent instructions earlier, which in turn results in higher ILP and MHP, and thus higher performance. Unfortunately, in the ML+DL configuration, consumers of long-latency loads may prevent younger independent load-consumers that depend on shorter-latency loads from being executed. This constraints the performance benefits.

The ML+DEL+DLL configuration improves performance by an additional 14 percentage point on average over the ML+DL configuration by steering forward-slice instructions to separate queues (i.e., loads are sent to the DLL and non-loads are sent to the DEL). This split-steering enables young independent arithmetic instructions to execute ahead of older load-dependent loads. Vice versa, loads may issue from the DLL before older load-consumers that are stalled in the DEL. Unfortunately, independent instructions may be stuck behind older instructions in the DEL.

The final FSC proposal overcomes the latter issue by adding the Holding Lane to which instructions are re-directed if they have been waiting for more than 4 cycles at the DEL head. Adding the HL improves performance by an additional 6 percentage point on average. The introduction of the Holding Lane enables independent instructions to execute ahead of older DEL instructions that depend on an L1 D-cache miss. This is particularly helpful for some of the memory-intensive benchmarks (bwaves_r_x, parest_r_1, cam4_r_1 and foto_r_1), for which the Holding Lane improves performance by up to 20%.

5.4 Lane Distribution

Figure 8 reports how instructions (or more precisely, micro-ops) are distributed across the four FSC lanes. On average, 47% of the instructions are steered to the ML, 10% to the DLL and 43% to the DEL; 28% of the DEL instructions (or 12% of the total number of instructions) are re-directed to the HL. For a number of benchmarks we note that a substantial fraction (more than 40%) of instructions is steered to the DEL, see for example deeps_r_1 (45%), exch_r_1 (59%) and x264_r_3 (61%). These instructions have long forward slices containing mostly arithmetic instructions. The benchmarks with the largest fraction instructions steered to the DLL are xalan_r_1

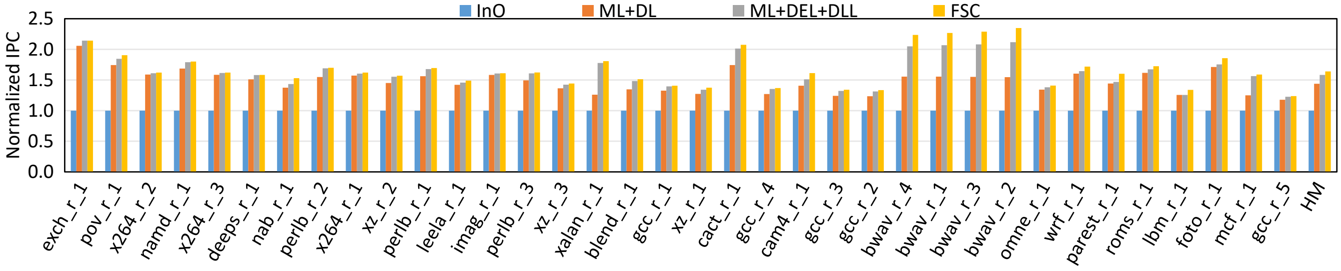


Figure 7: Normalized performance for the following FSC variants: ML+DL, ML+DEL+DLL and FSC with four queues (ML, DEL, DLL and HL). *All benchmarks significantly benefit from discerning forward slices (i.e., ML+DL variant). Many benchmarks benefit from splitting up into the DEL and DLL queues. Several memory-intensive benchmarks benefit from adding the HL.*

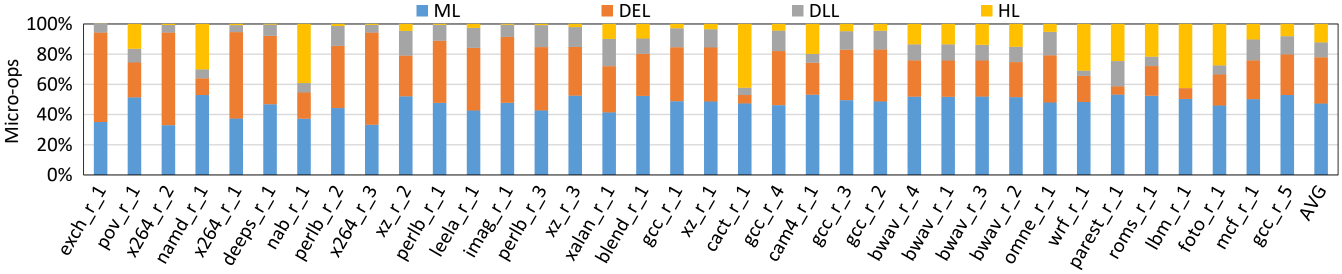


Figure 8: Distribution of micro-ops across the FSC lanes. *The majority of instructions are sent to the ML and DEL lanes; a significant number of instructions are re-directed from the DEL to the HL for several benchmarks.*

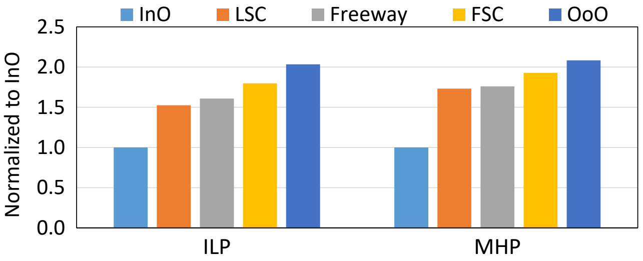


Figure 9: MHP and ILP normalized to an InO core for LSC, Freeway, FSC and OoO. *FSC significantly improves MHP and ILP over LSC and Freeway.*

(18%), parest_r_1 (17%) and xz_r_2 (16%); these benchmarks have a relatively large fraction of load-dependent loads. A substantial fraction of instructions are re-directed from the DEL to the HL for several benchmarks, see for example cact_r_1 (42%), lbm_r_1 (42%) and nab_r_1 (39%). These benchmarks have a fairly large number of arithmetic instructions that depend on L1 D-cache misses.

5.5 ILP and MHP

We now quantify how FSC improves ILP and MHP compared to LSC and Freeway. We compute ILP as IPC assuming a perfect L1 D-cache; that is, all memory accesses hit in the L1 D-cache (there are cold misses but no conflict or capacity misses in the L1 D-cache). We define MHP as the number of outstanding cache misses if at least one is outstanding; note MHP accounts for both on-chip and off-chip misses. Figure 9 reports ILP and MHP normalized to the in-order core for the different core microarchitectures. The overall conclusion is that FSC significantly improves both ILP and MHP.

FSC improves ILP by 80%, 17% and 12% compared to InO, LSC and Freeway, respectively. FSC steers arithmetic instructions to two queues (ML and DEL), unlike InO. In addition, FSC re-directs arithmetic forward-slice instructions that depend on an L1 D-cache miss to the Holding Lane so that younger independent arithmetic instructions can execute out-of-order, unlike LSC and Freeway. FSC's ILP is within 13% of an OoO core. OoO outperforms FSC in terms of ILP because an OoO core selects instructions for execution on a functional unit as soon as their input register dependences have been resolved.

FSC improves MHP by 93%, 11% and 10% compared to InO, LSC and Freeway, respectively. FSC steers forward-slice loads to the DLL, so that younger independent loads can issue from the ML. In contrast, InO and LSC steer all loads to a single queue. Freeway also steers loads to multiple queues, but suffers from imprecise backward slice identification, as previously discussed. FSC's MHP is within 8% of an OoO core. The reason for the remaining gap is that independent loads in the DLL may be stuck behind loads that depend on a long-latency load.

5.6 Hardware Overhead

FSC adds hardware structures compared to a baseline in-order core. Table 3 lists the size of these hardware structures (in number of bytes and mm^2). We add a register allocation table (RAT) for register renaming, a reorder buffer for a maximum of 32 instructions, and an 8-entry store queue (SQ). The steering bit vector has 64 entries which amounts to 8 bytes. FSC implements four 8-entry instruction

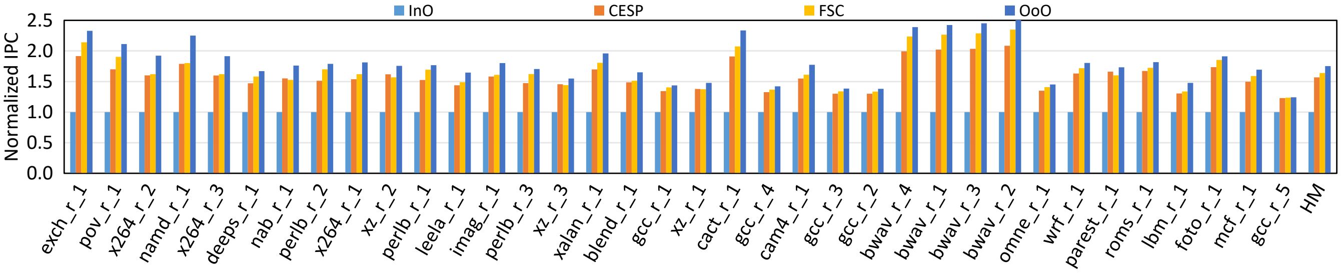


Figure 10: Normalized performance for CESP, FSC and OoO. *FSC improves performance by 4.5% on average, and up to 12.6%, compared to CESP.*

Table 3: Area and power overhead for the various FSC structures over a baseline in-order core.

Structure	*Details*	*Overhead* (Bytes)	*Area* (mm^2)	*Power* (mW)
RAT	32 entries	24	0.0061	4.15
PRF	32 int / 32 fp	768	0.0195	5.67
SBV	64 entries	8	0.0001	<0.01
ML	8 entries	176	0.0044	2.60
DEL	8 entries	176	0.0044	1.76
DLL	8 entries	176	0.0044	0.64
HL	8 entries	176	0.0044	0.74
ROB	32 entries	320	0.0081	6.44
MSHR	8 entries	52	0.0056	0.48
SQ	8 entries	512	0.0039	1.15
Total		2,388	0.0609	19.48

queues (ML, DEL, DLL and HL). We assume a physical register file (PRF) with 32 integer and 32 floating-point registers. The MSHR is extended to support 8 outstanding misses.

We use CACTI [12] to estimate chip area. CACTI accounts for the area of circuit-level structures such as hierarchically repeated wires, arrays, logic and the clock distribution network. The chip area incurred by FSC over a baseline in-order core amounts to 0.061 mm^2. Relative to our baseline InO core which occupies approximately 5.98 mm^2, we find that the FSC chip area overhead is around 1%. When compared to our baseline OoO core with a chip area of 8.29 mm^2, FSC occupies 37% less chip area.

5.7 Power Consumption

We use McPAT [11] to calculate InO and OoO core power consumption. The power consumed by the additional FSC hardware structures is modeled using CACTI [12]. Table 3 reports power consumption for the newly added components.

Overall, the added hardware structures increase power consumption by 19.48 mW compared to our baseline in-order core. When added to the power consumption of the baseline in-order core (2.99 W), the total power consumption for FSC amounts to 3.01 W. Freeway increases power consumption over an InO core by 39.83 mW. This is because of the extra hardware structures (IST and RDT) to support IBDA.

The total power consumption of the OoO core amounts to 6.95 W. Therefore, FSC consumes 56% less power than the OoO core. This large saving in power comes from eliminating out-of-order wakeup, select and issue logic required for the OoO core. FSC replaces the complex OoO logic with simple FIFO queues.

5.8 Comparison Against CESP

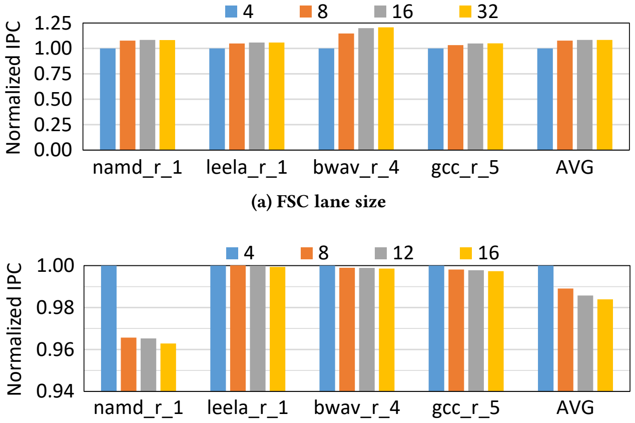
Palacharla et al. [13] propose the complexity-effective superscalar processor (CESP) architecture which steers chains of dependent instructions into generic in-order queues. Figure 10 compares FSC against CESP with four queues. It is important to note that CESP's steering logic is more complex than FSC for at least two reasons: (1) CESP steers instructions into four queues as opposed to FSC which steers instructions into three queues, and (2) CESP requires a table access to find out in which queue the producer instruction resides so that chains of dependent instructions are steered to the same queue — in contrast, FSC simply steers instructions to the appropriate queue based on a single SBV bit and instruction type (load vs. non-load). In spite of its smaller hardware complexity, we find that FSC outperforms CESP by 4.5% on average, and up to 12.6%. The reason is that CESP stalls dispatch when an independent instruction cannot be steered to an empty queue. In contrast, FSC steers instructions to queues based on whether an instruction belongs to a forward slice or not. Moreover, instructions that depend on L1 D-cache misses are re-directed to the HL so that younger instructions can go ahead and execute.

5.9 Sensitivity Analyses

We conduct a couple sensitivity analyses to comprehensively explore the design space.

Lane Size. Figure 11(a) reports performance sensitivity to lane size while keeping the maximum number of in-flight instructions constant at 32. (All lanes have equal sizes.) We show performance results for a couple representative benchmarks along with the average across all. We conclude that 8-entry lanes are close to optimal on average, which is what we assume throughout the paper.

Waiting Cycles. Figure 11(b) reports performance sensitivity to the number of waiting cycles before re-directing an instruction from the DEL to the HL. We find that performance is relatively insensitive on average, although we note a 1% performance degradation for more than 8 cycles compared to 4 cycles. Especially compute-intensive benchmarks, such as namd_r_1, seem to suffer more. The reason is that independent load-consumers are stuck behind other load-consumers for a longer period of time. A waiting



(a) FSC lane size

(b) Number of waiting cycles

Figure 11: Sensitivity analyses with respect to (a) FSC lane size and (b) number of waiting cycles before re-directing an DEL instruction to the HL. *A lane size of 8 entries is optimal on average. Four waiting cycles is optimal on average.*

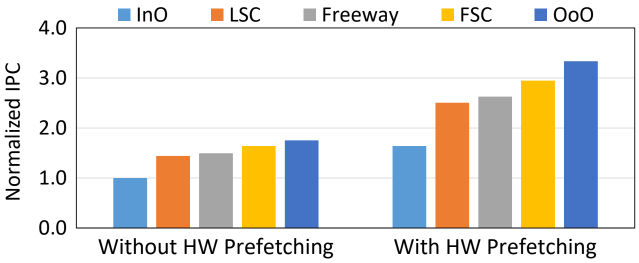


Figure 12: Normalized performance for LSC, Freeway, FSC and OoO without and with a 16-stream stride-based hardware prefetcher at L1. Performance is normalized to the InO core without prefetcher in all configurations. *FSC improves performance by 75% compared to an InO baseline with hardware prefetching.*

time of 4 cycles (i.e., L1 D-cache access time) is optimal, which is what we assume throughput the paper.

Hardware Prefetching. We did not assume hardware prefetching in our setup so far, for ease of analysis. We now consider a baseline architecture with hardware prefetching by adding a stride-based prefetcher at L1, which tracks up to 16 independent streams. Figure 12 reports normalized performance without and with the stride-based prefetcher enabled for all the core microarchitectures. For all configurations, performance is normalized to the InO core without hardware prefetching. It is interesting to note that the performance improvement achieved by the sOoO cores compared to the InO baseline is higher when hardware prefetching is enabled versus disabled. The reason is that as performance improves with hardware prefetching enabled, a smaller fraction of time is spent on stalls due to cache misses, hence a similar improvement in instruction scheduling leads to a higher impact on performance. This is especially the case for the memory-intensive benchmarks. On average, we report that FSC performs 76% better than InO, versus 51% and 59% for LSC and Freeway, respectively, with hardware prefetching enabled.

Memory Disambiguation. We further find that SAR degrades performance by 1% compared to perfect memory disambiguation (no results shown here due to space constraints). The performance impact of SAR mainly comes from the last STA micro-op arriving at the head of the four lanes. We find that the last STA micro-op typically resides in the DEL. As FSC periodically re-directs instructions from the DEL to the HL, it does not take long for the STA to reach the head of the DEL, resulting in a minimal performance impact.

Four-Wide Superscalar Pipeline. We considered a two-wide superscalar baseline throughout the paper. For a 4-wide superscalar pipeline with a 192-entry reorder buffer size, we find that FSC outperforms Freeway by 15.5% on average (again, no results shown due to space constraints). Note that this relative performance improvement is larger than for the two-wide superscalar baseline (9.7% average improvement). The performance benefit primarily comes from extracting ILP across the four lanes. In contrast, Freeway has limited capabilities to extract ILP which widens the performance gap for wide superscalar architectures.

6 RELATED WORK

A significant body of prior work has contributed to making processors more complexity-effective and power-efficient. We now point out the most closely related work.

Complexity-Effective Architectures. Palacharla et al. [13] propose the complexity-effective superscalar processors (CESP) architecture which steers chains of dependent instructions to in-order queues. Dispatch stalls when an independent instruction cannot be steered to an empty queue. Salverda and Zilles [14] evaluate CESP in the context of a realistic baseline and point out a large performance gap with a traditional OoO core because of frequent dispatch stalls. A similar steering policy is used by Kim et al. [9] in their Instruction-Level Distributed Processing (ILDP) work, which proposes an ISA with in-order accumulator-based execution units. Our experimental results show that FSC outperforms CESP.

Salverda and Zilles [15] analyze the fundamental challenges of fusing small in-order cores on demand into larger cores. They find that fusing small cores is not appealing if those cores support in-order execution only; some form of out-of-order execution capability is needed to achieve high performance. In particular, they propose a cost-based steering policy that uses a complex load-latency predictor such that instructions do not get stuck behind long-latency loads. In contrast, FSC moves instructions to the Holding Lane based on a simple down-counter. Overall, FSC features a low-cost and effective instruction steering policy that enables out-of-order execution capabilities among in-order queues.

Decoupled Access-Execute. DAE [19] is the first work to separate access and execute phases of a program through coordinated queues. Proposals such as speculative-slice execution [23], flea-flicker multipass pipelining [3], braid processing [22] and OUTRIDER [6] also exploit critical instruction slices [24] for improving performance. More recently, Clairvoyance [20] and SWOOP [21] exploit the decoupled nature of access and execute phases for improving energy efficiency. These compiler-based techniques involve new instructions, advanced profiling information, or binary translation for separating critical instruction slices, unlike FSC.

Restricted Out-of-Order Microarchitectures. We extensively discussed the Load Slice Core [5] and Freeway [10] throughout the paper. Shioya et al. [18] propose the front-end execution architecture which executes instructions that have their operands ready in the front-end of the pipeline; other non-ready instructions are dispatched to the out-of-order back-end. CASINO [8] pursues a similar goal by augmenting an in-order core with an additional speculative queue from which ready instructions are executed ahead of a traditional in-order instruction queue. CASINO adds significant complexity over an in-order core because of the CAM-based selection logic in the speculative queue and dynamic memory disambiguation.

A number of proposals take an OoO core as a starting point and reduce complexity by bypassing some of the out-of-order structures. FSC eliminates all out-of-order structures and is therefore more area- and power-efficient. Long-term parking [16] saves power in an OoO core by allocating back-end resources for critical instructions while buffering non-critical instructions in the front-end. More recently, Alipour et al. [1] leverage instruction criticality and readiness to bypass the out-of-order back-end. Instructions that do not benefit from out-of-order scheduling and instructions that do not suffer from being delayed are sent to an in-order FIFO queue.

7 CONCLUSION

Slice-out-of-order cores were recently proposed to tackle the in-order issue bottleneck in stall-on-use in-order processors by allowing loads and stores, plus their backward slices, to bypass older instructions in the dynamic instruction stream. sOoO cores improve ILP and MHP, yet they leave significant performance on the table. In particular, backward slice analysis is imprecise while incurring a non-trivial hardware cost.

In this paper, we propose *Forward Slice Core (FSC)*, a novel core microarchitecture that steers instructions to in-order FIFO queues based on the notion of forward slices of loads (i.e., the direct and indirect load-consumers). Forward slices are constructed in a single pass as opposed to backward slice analysis, which is iterative, imprecise and hardware-inefficient. In addition, FSC re-directs instructions waiting for an L1 D-cache miss to the Holding Lane. Finally, FSC implements store-address replication to alleviate the need for expensive dynamic memory disambiguation logic. FSC outperforms the state-of-the-art sOoO core, Freeway, by 9.7% on average across the SPEC CPU2017 benchmarks while being more hardware-efficient.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable comments. This work was supported in part by European Research Council (ERC) Advanced Grant agreement no. 741097, and FWO projects G.0434.16N and G.0144.17N. Josué Feliu was supported through a postdoctoral fellowship by the Ministerio de Ciencia, Innovación y Universidades (FJC2018-036021-I).

REFERENCES

[1] M. Alipour, S. Kaxiras, D. Black-Schaffer, and R. Kumar. Delay and bypass: Ready and criticality aware instruction scheduling in out-of-order processors. In *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 424–434, 2020.

[2] ARM. ARM Cortex-A7 processor. http://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[3] R. D. Barnes, S. Ryoo, and W. W. Hwu. "flea-flicker" multipass pipelining: an alternative to the high-power out-of-order offense. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 319–330, 2005.

[4] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):28, 2014.

[5] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. The load slice core microarchitecture. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 272–284, 2015.

[6] N. C. Crago and S. J. Patel. OUTRIDER: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 117–128, 2011.

[7] L. Eeckhout. *Computer Architecture Performance Evaluation Methods.* Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2010.

[8] I. Jeong, S. Park, C. Lee, and W. W. Ro. CASINO core microarchitecture: Generating out-of-order schedules using cascaded in-order scheduling windows. In *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 383–396, 2020.

[9] H. Kim and J. E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 71–81, 2002.

[10] R. Kumar, M. Alipour, and D. Black-Schaffer. Freeway: Maximizing mlp for slice-out-of-order execution. In *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–569, 2019.

[11] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec. 2009.

[12] 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 *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 694–701, 2011.

[13] 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)*, pages 206–218, June 1997.

[14] P. Salverda and C. Zilles. Dependence-based scheduling revisited: A tale of two baselines. In *Proceedings of the Sixth Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD), held in conjunction with ISCA*, 2007.

[15] P. Salverda and C. Zilles. Fundamental performance constraints in horizontal fusion of in-order cores. In *Proceedings of the 14th Annual International Symposium on High Performance Computer Architecture (HPCA)*, pages 252–263, 2008.

[16] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 334–346, 2015.

[17] 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)*, pages 45–57, 2002.

[18] R. Shioya, M. Goshima, and H. Ando. A front-end execution architecture for high energy efficiency. In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, pages 419–431, 2014.

[19] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture (ISCA)*, pages 112–119, 1982.

[20] K. A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean. Clairvoyance: Look-ahead compile-time scheduling. In *Proceedings of the International Conference on Code Generation and Optimization (CGO)*, pages 171–184, 2017.

[21] K. A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras. SWOOP: Software-hardware co-design for non-speculative, execute-ahead, in-order cores. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 328–343, 2018.

[22] F. Tseng and Y. N. Patt. Achieving out-of-order performance with almost in-order complexity. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 3–12, 2008.

[23] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, July 2001.

[24] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 172–181, June 2000.