The Forward Slice Core: A High-Performance, Yet Low-Complexity Microarchitecture

KARTIK LAKSHMINARASIMHAN and AJEYA NAITHANI, Ghent University, Belgium
JOSUÉ FELIU, Universidad de Murcia, Spain
LIEVEN EECKHOUT, Ghent University, Belgium

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 article, 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. Considering 3-wide pipeline configurations, we find that FSC improves performance by 27.1%, 21.1%, and 14.6% on average compared to Freeway, the state-of-the-art sOoO core, across SPEC CPU2017, GAP, and DaCapo, respectively,

This article is an extension of the conference paper 'The Forward Slice Core Microarchitecture' by the same authors published at the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 361–372, Oct 2020. This submission extends the conference paper in the following ways:

(1) We provide benchmark results for GAP (graph analytics) and DaCapo (Java managed language workloads), in addition to the SPEC CPU2017 results presented in the conference paper. We analyze how FSC performs across the three benchmark suites (SPEC CPU2017, DaCapo and GAP) and show that FSC is more robust across workload types than prior sOoO core microarchitectures by exploiting both ILP and MHP.
(2) We provide additional results for a 3-wide baseline superscalar core in addition to the 2-wide configuration reported in the conference paper. We demonstrate that FSC's performance improvement over the InO baseline and prior sOoO microarchitectures increases with increasing pipeline width.
(3) We analyze the effectiveness of the Instruction Slice Table (IST) in prior sOoO microarchitectures to construct backward slices and find that IST miss rate increases with larger code footprint. FSC circumvents the IST performance issue by constructing forward slices rather than backward slices.
(4) Additional discussion is provided regarding the scheduling policy from the Holding Lane and regarding memory disambiguation complexity.

This work is supported by the European Research Council (ERC) Advanced Grant agreement no. 741097, and FWO project G.0144.17N. Josué Feliu is supported by a Juan de la Cierva Formación Contract (FJC2018-036021-I).
Authors' addresses: K. Lakshminarasimhan, A. Naithani, and L. Eeckhout, Ghent University, Technologypark 126, Zwijnaarde, 9052 Ghent, Belgium; emails: {kartik.lakshminarasimhan, ajeya.naithani, lieven.eeckhout}@ugent.be; J. Feliu, Universidad de Murcia, C. Campus Universitario, Edificio 32, 30100 Murcia, Spain; email: josue.f.p@um.es.

1544-3566/2022/01-ART17 $15.00
https://doi.org/10.1145/3499424

while at the same time incurring reduced hardware complexity. Compared to an OoO core, FSC reduces power consumption by 61.3% and chip area by 47%, providing a microarchitecture with high performance at low complexity.

CCS Concepts: • **Computer systems organization → Superscalar architectures;**

Additional Key Words and Phrases: Superscalar microarchitecture, slice-out-of-order, dynamic instruction scheduling

ACM Reference format:
Kartik Lakshminarasimhan, Ajeya Naithani, Josué Feliu, and Lieven Eeckhout. 2022. The Forward Slice Core: A High-Performance, Yet Low-Complexity Microarchitecture. *ACM Trans. Archit. Code Optim.* 19, 2, Article 17 (January 2022), 25 pages.
https://doi.org/10.1145/3499424

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 high design complexity and large 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. This is of particular importance in the huge and continuously growing mobile and embedded markets. In particular, the number of smartphone users is continuously increasing reaching close to 4 billion users around the world this year [39]; further, projections estimate 50 billion **Internet-of-Things (IoT)** devices by 2030 [38]; finally, the 5G market is expected to involve 666 million devices [30]. Mobile and edge devices need increasingly high performance at low cost and low power consumption.

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)*** [9] was the first work to propose an sOoO core; Freeway [20] 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. [20] observe that, in LSC, an independent load may be stuck behind a load

[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. Comparing FSC Against Prior Slice-out-of-order Microarchitectures in Terms of ILP, MHP and Hardware Complexity

Processor Design	ILP	MHP	Hardware Complexity
InO	–	–	+
LSC	+	+	+++
Freeway	++	++	+++
FSC	+++	+++	++
OoO	++++	++++	+++++++

FSC offers higher ILP and MHP than LSC and Freeway at lower hardware overhead.

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.

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 article, 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 are 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 the 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.

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 backward 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, GAP (graph analytics), and Java DaCapo benchmarks across microarchitecture configurations representative of (high-end) superscalar embedded and mobile processors. The overall key conclusion is that FSC outperforms Freeway, the state-of-art sOoO core microarchitecture, by a significant margin. We report an average 15.7%, 6.9%, and 10.1% improvement over Freeway for SPEC CPU2017, GAP, and DaCapo, respectively, assuming a 2-wide superscalar configuration. The performance improvement increases to 27.1%, 21.1%, and 14.6%, respectively, for a 3-wide superscalar configuration. We further find that the performance gain through FSC over an InO baseline is more robust across the three benchmark suites, i.e., compared to a 3-wide InO baseline, FSC yields a consistent 76.8% to 79.2% performance improvement (2.4 percentage point variance) across the three benchmark suites, versus 41.1% to 54.1% (13 percentage point variance) for Freeway. The fundamental reason for the consistent performance gain is that FSC improves both ILP and MHP. While prior sOoO microarchitectures primarily improve performance for MHP-intensive workloads, FSC is effective for a broader range of workloads with varying characteristics in ILP and MHP, i.e., different workloads benefit in different ways. We argue that FSC is more hardware-efficient (requiring 1,408 fewer bytes) than the state-of-the-art sOoO microarchitecture Freeway. We further find that FSC performs within 6.0% and 12.8% of a 2-wide and 3-wide out-of-order processor, while consuming 56.7% and 61.3% less power and while incurring 37% and 47% less chip overhead, respectively. Finally, we find that FSC's performance gain over an InO baseline increases with hardware prefetching enabled.

2 BACKGROUND AND MOTIVATION

In this section, we briefly cover the background on the two prior sOoO cores—LSC [9] and Freeway [20]—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

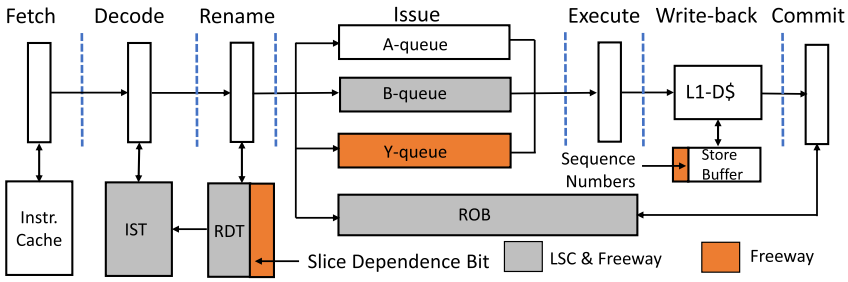

Fig. 1. Slice-out-of-order cores: LSC and Freeway add a number of new structures color-coded in gray and orange, respectively. *LSC steers loads and their backward-slices to the B-queue. Freeway in addition steers dependent backward slices to the Y-queue. Non-backward-slice instructions are steered to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

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 those 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 using SPEC CPU2017, LSC achieves 35% 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. [20] 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

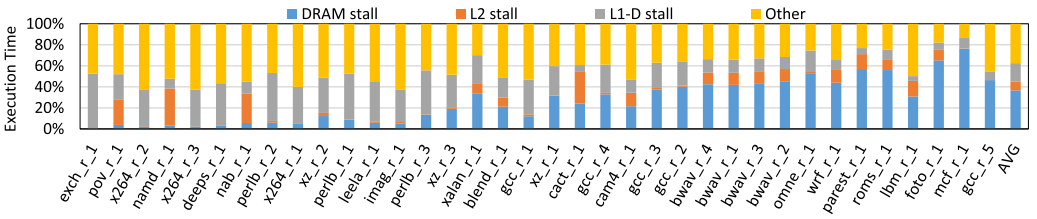

Fig. 2. CPI stacks for the SPEC CPU2017 benchmarks (sorted by LLC MPKI from left to right) on a stall-on-use in-order core. *Compute-intensive workloads frequently stall on L1/L2 load consumers, whereas memory-intensive workloads frequently stall on memory-access consumers.*

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, as we will discuss in the next section.

2.3 Shortcomings of Slice-Out-of-Order Cores

There are four major shortcomings with prior 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 at the head of 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 on the right-hand side of the figure, spend the majority of their time waiting for data to return 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

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

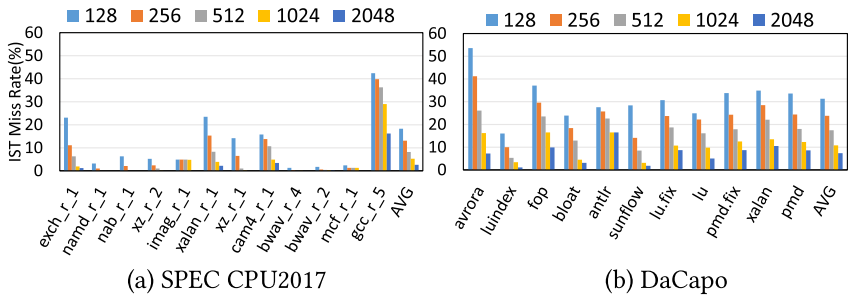

Fig. 3. IST miss rate as a function of IST size for (a) SPEC CPU2017 (random selection is reported; average computed across all benchmarks) and (b) DaCapo. *The higher IST miss rate for the DaCapo workloads is a result of their larger code footprint compared to SPEC CPU2017.*

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.

Second, backward slice analysis is an iterative and imprecise process. As a result, while building up the backward slices, AGIs will be sent to the A-queue, which also hinders the exploitable MHP.

Finally, 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, the bottom line is that even though sOoO cores significantly improve the achievable MHP over an in-order core, there is still room for improvement.

IST Misses. The IST employed in LSC and Freeway to identify backward slices is a cache structure of limited size which may limit performance. Figure 3(a) reports the IST miss rate as a function of its size for SPEC CPU2017 and the Java DaCapo benchmarks in Freeway.[3] We observe a significantly higher IST miss rate for DaCapo versus SPEC CPU2017: we report an average miss rate of 31% versus 18% for DaCapo and SPEC, respectively, for a 128-entry 2-way set-associative IST (our baseline); likewise, we report an average miss rate of 7.4% versus 2.6%, respectively, for a 2048-entry IST. The higher IST miss rate is a result from the larger code footprint for DaCapo — we report an approximately 3× higher I-cache miss rate for DaCapo versus SPEC CPU2017. The high IST miss rate negatively affects performance as shown in Figure 4 which reports performance degradation for DaCapo of a 128-entry IST versus a 2048-entry IST as a function of IST miss rate. This leads to two observations: (i) IST miss rate correlates negatively with performance, i.e., increased IST miss rate leads to performance degradation, and (ii) even an unrealistically large (2K-entry) IST does not dramatically improve performance (by a couple percent only for this set of benchmarks), which further motivates the need for a different sOoO architecture paradigm.

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. Moreover, the IST needs 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, and a store entry is allocated in the store buffer upon dispatch (instruction steering into one of the queues). When issuing a load,

[3]The GAP benchmarks are not considered here because of their small (less than 1%) IST miss rate and small code footprint.

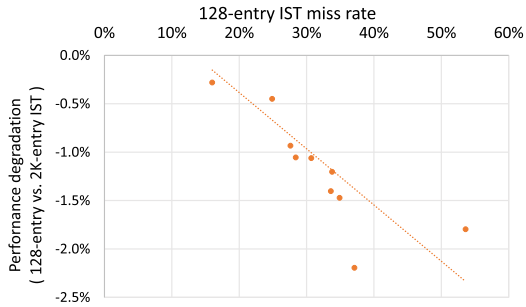

Fig. 4. IPC performance degradation for the DaCapo benchmarks when comparing an unrealistically large 2K-entry IST versus the baseline 128-entry IST as a function of (128-entry) IST miss rate. *High IST miss rate leads to performance degradation.*

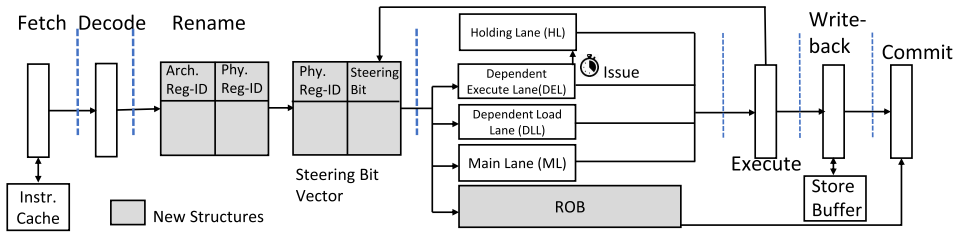

Fig. 5. Forward Slice Core (FSC) architecture. FSC adds a number of new structures (gray components) over an stall-on-use in-order core. *FSC consists of four in-order queues: non-forward-slice instructions are steered to the Main Lane (ML), forward-slice instructions are steered to the Dependent Execute Lane (DEL) and the Dependent Load Lane (DLL); DEL instructions that stall on an L1 D-cache miss are re-directed to the Holding Lane (HL) so that independent forward-slice instructions can execute as soon as possible.*

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 significant.

3 FORWARD SLICE CORE

We propose the ***Forward Slice Core (FSC)*** microarchitecture to address the aforementioned shortcomings of sOoO cores. Figure 5 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 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 a load's forward slice.

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 directly or 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 straightforward. 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: the counter implementation allows for determining an L1 miss locally within the core at low overhead. An alternative implementation would be to notify the core upon an L1 D-cache miss through a dedicated signal from the L1 D-cache to the core. If the core could be notified upon an L1 miss (possibly with shorter latency, i.e., tag access latency), FSC could simply redirect instructions to the Holding Lane based on this notification, thereby eliminating the need for the down counter.

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, which may happen in pointer-chasing code. 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 limited. Putting a miss-dependent miss out of the way quickly leads to the next miss-dependent miss, which does not significantly 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 or three instructions can be issued per cycle in a two-wide and three-wide FSC implementation, respectively. 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.

We recognize that multiple independent forward slices may reside in the Holding Lane, which may limit performance as the Holding Lane is an in-order queue. In particular, if the first forward slice depends on a long-latency load, it may prevent other forward slices from being executed. We found this to be the case for some of our workloads. Figure 6(b) shows a code example from SPEC CPU2017's cact_r_1: the code consists of two loads (L_1 and L_5) and their forward slices ($E_2 - E_3 - S_4$ and $E_6 - E_7 - S_8$, respectively) with E arithmetic instructions and S store instructions; the subscripts denote program order. The forward slices in this example have already been moved to the Holding Lane. If load L_1 incurs a long-latency miss while L_5 is an L1 D-cache hit, this will force the forward slice of L_5 to wait until the first load and its forward slice have executed. This results in unnecessarily holding up the younger slice for tens of cycles to reach the HL head. In this particular example, the younger slice has to wait for 36 cycles in total: 28 cycles for E_2 to execute, plus 5 for E_2 and 3 for E_3. The fundamental reason for this delay is the fact that the Holding Lane is an in-order queue. This observation suggests an opportunity to further improve performance by providing out-of-order issue logic in the Holding Lane. We implemented and evaluated an out-of-order HL while keeping the other lanes in-order, and found that it improves performance by 3% on average (and at most 25%) for the 3-wide configurations. Because an out-of-order HL increases

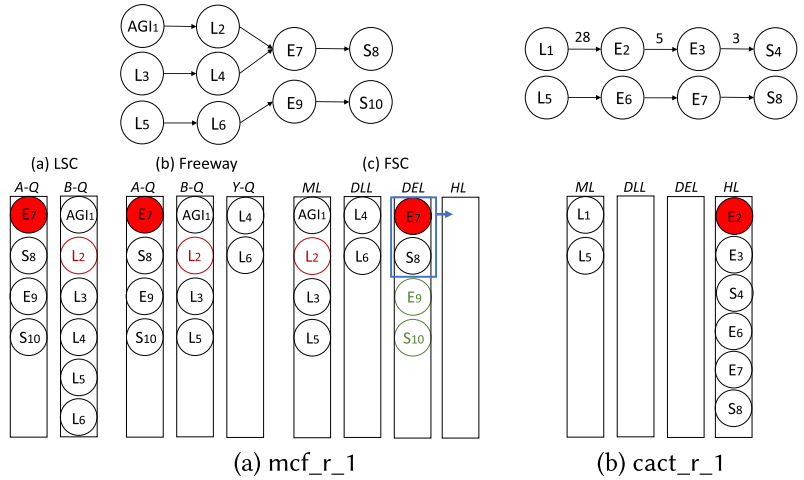

Fig. 6. (a) Instruction dependence graph taken from mcf_r_1's hot loop illustrating how instructions are steered (and re-directed) to queues. *FSC steers non-forward-slice instructions to the ML and the forward-slice instructions (i.e., load-consumers) to the DEL and DLL. Instructions in the DEL that depend on an L1 D-cache miss (e.g., E_7 and S_8) are re-directed to the HL,* and (b) Instruction dependence graph taken from cact_r_1 to illustrate the limitation of an in-order Holding Lane. *There are two load forward slices in the HL where $E_6 - E_7 - S_8$ has to wait until the younger slice $E_2 - E_3 - S_4$ has executed.*

hardware complexity by a significant margin, we conclude that this is not a favorable design point.

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 depends on the STA through a read-after-write register dependence. The STD micro-op is steered to the ML or DEL, and may be dynamically re-directed to the 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 three lanes (ML, DEL, DLL) upon instruction steering. An STA micro-op at the head of the DEL may be re-directed to the HL, just like any other instruction. An STA micro-op is selected for execution if its input register operands are available *and* if it is at the head of *all* three lanes; i.e., the STA micro-op needs to be at the heads of the ML *and* the DLL *and* the DEL or HL. 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 Hardware Complexity

Overall, the hardware cost and complexity is less for FSC compared to Freeway, the state-of-the-art sOoO core microarchitecture. In our baseline 3-wide configuration, compared to Freeway, FSC removes the IST (768 bytes), the RDT (1024 bytes), and the loads and store sequence numbers in the store buffer (32 bytes). FSC requires new structures with minimal hardware cost: the Steering Bit Vector (16 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 3,540 bytes, versus 5,348 bytes for Freeway—a reduction by 1,808 bytes compared to Freeway.

The above calculation does not account for the **content-addressable memory (CAM)** logic needed to support dynamic memory disambiguation in Freeway, which is significant. Recall that LSC executes loads and stores in program order. Yet, it requires one CAM search in the store buffer to make sure that loads read the most recent value written to memory, i.e., younger loads need to compare their addresses against the addresses of the older stores in the store buffer, and, in case of a match, the data value is read from the store buffer. Freeway, in contrast, allows for out-of-order execution of loads and stores, i.e., loads may execute ahead of older non-aliasing stores. Freeway therefore incurs additional fields in the store buffer and issue queues for storing the sequence numbers of loads and stores (98 bytes extra in total). As mentioned before, Freeway allocates an entry in the store buffer when dispatching a store and the store's sequence number is inserted. The address and data value fields are filled in when the store is executed. When issuing a load, the load's sequence number needs to be compared against the sequence numbers in the store buffer. A load can only go ahead and execute if all older stores have computed their addresses. In case there is an address match between the load and an older store, the load needs to wait for the data value to be computed. This incurs a significant hardware cost because two CAM searches are needed: (i) the load's sequence number needs to be compared against the stores' sequence numbers in the store buffer, and (ii) the load address needs to be compared against the store addresses. Overall, memory disambiguation in Freeway incurs significant hardware complexity over LSC.

In contrast, FSC incurs similar hardware complexity as LSC when it comes to memory disambiguation, as shown in Figure 7. Because STA micro-ops are replicated across the different queues so that younger load instructions can never bypass an older store, a load instruction only needs to compare its address to the addresses in the store buffer. (There is no need to compare sequence numbers among loads and stores as in Freeway, because loads execute in program order with respect to older stores.) If a match is found and the data is available, the load reads the data from the store buffer; if the data is not available yet, the load stalls waiting for the store data. In case there is no match, the load goes ahead and obtains the data value from the memory hierarchy. The hardware complexity of FSC's memory disambiguation logic is comparable to LSC.

4 EXPERIMENTAL SETUP

Simulation Setup. We faithfully model and evaluate FSC using the most detailed, cycle-level and hardware-validated core model in Sniper v6.0 [10]. Our baseline configurations are 2-wide and 3-wide superscalar processors as we target embedded and (high-end) mobile processors, see Table 2. The two-wide in-order baseline core is configured after the ARM Cortex-A7 [4]. We follow the LSC and Freeway configurations by Kumar et al. [20]. Note that we keep total aggregate issue queue size constant across all architectures for fair comparison, i.e., 32 and 48 entries for the two-wide and three-wide configurations, respectively. The FSC lanes (ML, DEL, DLL, and HL) contain

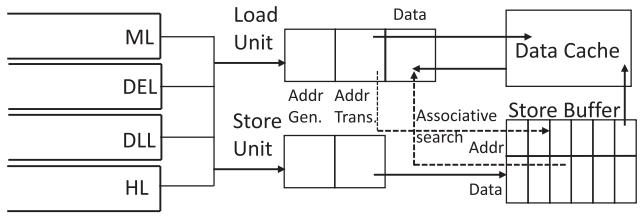

Fig. 7. Non-speculative memory disambiguation in FSC. *Loads compare their addresses against the addresses in the store buffer. If a match occurs, the load reads the data value from the store buffer; if not, the load obtains the data value from the memory hierarchy.*

Table 2. Simulated Two-wide and Three-wide InO, LSC, Freeway, FSC and OoO Configurations

	Two-wide Configuration					**Three-wide Configuration**				
	InO	**LSC**	**Freeway**	**FSC**	**OoO**	**InO**	**LSC**	**Freeway**	**FSC**	**OoO**
Scoreboard/ROB	32					64				
Register File	32 int, 32 fp					64 int, 64 fp				
Store Buffer	16					24				
Issue Queue	—	2 × 16	3 × 12	4 × 8	1 × 32	—	2 × 24	3 × 16	4 × 12	1 × 48
Frequency	2.0 GHz									
Branch Predictor	1.5 KB hybrid local/global/loop predictor and BTB									
Pipeline Depth	5 front-end pipeline stages									
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 L1 D-cache L2	4-way 32 KB, 2 cycles 8-way 32 KB, 4 cycles (1-cycle tag look-up) 8-way 512 KB, 8 cycles (3-cycle tag look-up)									
Memory	3.8 GB/s, 45 ns									

8 and 12 entries for the 2-wide and 3-wide configurations, respectively. LSC, Freeway, and 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., perfect memory-dependence prediction); 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, and 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 [23] and CACTI v6.5 [24] 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.

Benchmarks. We create representative 1B-instruction SimPoints [34] for the SPEC CPU2017 benchmarks with reference inputs. We sort the benchmarks by increasing number of **last-level cache (LLC) misses per-kilo instructions (MPKI)**; we note the highest MPKI of 48 for gcc_r_5.

We further consider six graph analytics applications from the GAP benchmark suite [6]: **Betweenness Centrality (**bc**)**, **Breadth-First Search (**bfs**)**, **Connected Components (**cc**)**, **PageRank (**pr**)**, **Single-Source Shortest Path (**sssp**),** and **Triangle Count (**tc**)**. These graph

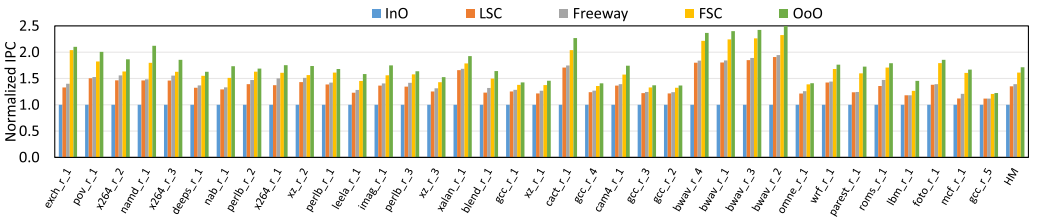

Fig. 8. Performance for the 2-wide LSC, Freeway, FSC, and OoO cores normalized to the baseline InO core for the SPEC CPU2017 benchmarks. *FSC improves performance by 61.1% on average compared to an InO core, versus 35.1% for LSC and 39.2% for Freeway. The OoO core improves performance by 71.4% on average.*

workloads exhibit very different execution characteristics compared to SPEC CPU2017 because of their pointer-chasing code patterns and low-ILP characteristics. We simulate the graph algorithms in two iterations where the first iteration is used to warm up the caches, and we report performance for the second iteration obtained through detailed simulation. Also, we skip the initialization and preprocessing steps during the formation of the graph using an in-built graph generator with a size of 2^{18} nodes, formed according to the Kronecker distribution satisfying the Graph500 specifications.

In addition to the native language SPEC CPU2017 and GAP workloads, we also include a set of managed-language workloads. These workloads typically feature a larger code footprint and higher I-cache miss rate than SPEC CPU2017, which makes backward slice detection harder, as previously discussed in Section 2. We use eleven Java workloads from DaCapo [7] using the large inputs running on top of the Jikes 3.1.2 virtual machine [3]: nine benchmarks are taken from the DaCapo-9.12-bach benchmark suite, plus an updated version of lusearch which eliminates useless allocation (lu.fix) [43], and an updated version of pmd which eliminates a scaling bottleneck due to a large input file (pmd.fix) [13]. We follow best practice in Java performance evaluation by using replay compilation [8, 17] to eliminate non-determinism introduced by the just-in-time compiler. During a profiling run, the optimization level of each method is recorded for the run with the shortest execution time. The JIT compiler then uses this optimization plan in the measurement run, optimizing to the correct level the first time it sees each method. To eliminate the perturbation of the compiler, we measure results during the second invocation, which represents application steady-state behavior.

5 EVALUATION

We evaluate the following five processor cores: (i) InO, the baseline stall-on-use in-order core, (ii) LSC, (iii) Freeway, (iv) FSC, and finally (v) OoO, the out-of-order core. We compute per-application performance as ***instructions per cycle (IPC)*** and the average performance across benchmarks is calculated using the harmonic mean IPC [15]. The below evaluation primarily focuses on 2-wide versus 3-wide performance results as well as contrasting FSC performance across the different benchmark suites. The original conference paper [21] provides additional analyses including CPI stack analysis for individual workloads, lane distribution statistics, ILP and MHP analysis, and various sensitivity analyses to lane configuration, lane size, number of waiting cycles for HL re-direction, and memory disambiguation.

5.1 2-Wide Baseline Performance Results

Figure 8 reports performance for all the evaluated 2-wide cores across the SPEC CPU2017 benchmarks. Overall, FSC achieves substantially higher performance than InO, LSC and Freeway.

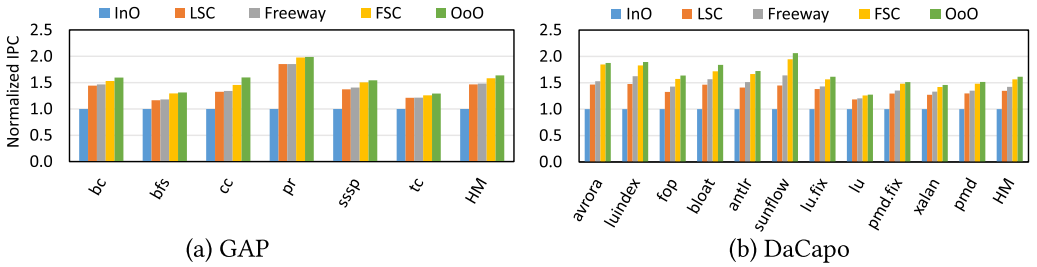

Fig. 9. Performance for the 2-wide cores for (a) GAP and (b) DaCapo. *For GAP, FSC outperforms InO by 58.4% versus 46.5% for LSC and 48.1% for Freeway. For DaCapo, FSC improves performance by 56.6% compared to the InO baseline, versus 34.6% for LSC and 42.2% for Freeway.*

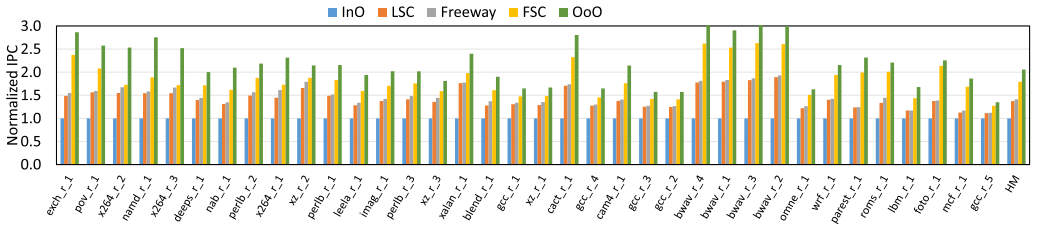

Fig. 10. Performance for 3-wide LSC, Freeway, FSC, and OoO cores normalized to the baseline InO core for SPEC CPU2017. *FSC improves performance by 79.2% on average compared to an InO core, versus 37.2% for LSC and 41.1% for Freeway.*

Relative to the baseline InO core, LSC and Freeway improve performance by 35.1% and 39.2% on average, whereas FSC outperforms the InO core by 61.1%; this is an additional gain of 15.7% (or 21.9 percentage point) over Freeway, the state-of-the-art sOoO core.[4] Furthermore, FSC performs within 6.0% (or 10.3 percentage points) of the OoO core, which improves performance by 71.4% over the InO baseline. 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, FSC is able to bridge a large fraction of the performance gap between an InO and OoO core.

We obtain similarly good results for the DaCapo and GAP benchmarks, see Figure 9. For DaCapo, LSC and Freeway improve performance by 34.6% and 42.2% on average compared the InO baseline, respectively. FSC outperforms the InO core by 56.6%; this is an additional gain of 10.1% (or 14.4 percentage point) over Freeway. For GAP, LSC and Freeway improve performance by 46.5% and 48.1% on average, respectively, while FSC outperforms the InO core by 58.4%; this is an additional gain of 6.9% (or 10.3 percentage point) over Freeway. FSC is within 3.1% and 3.4% of the OoO core for the DaCapo and GAP workloads, respectively.

5.2 Increased Pipeline Width

The performance improvement over InO as well as prior sOoO cores increases with increasing pipeline width. More specifically, FSC yields higher performance improvements for the 3-wide configurations compared to the 2-wide configurations, see Figure 10 for SPEC CPU2017 and

[4]The performance for our baseline 2-wide configuration and SPEC CPU2017 benchmarks has changed slightly compared to the results reported in the original conference paper [21] due to improved modeling of the store-address micro-op, its dependences and impact on issue port contention.

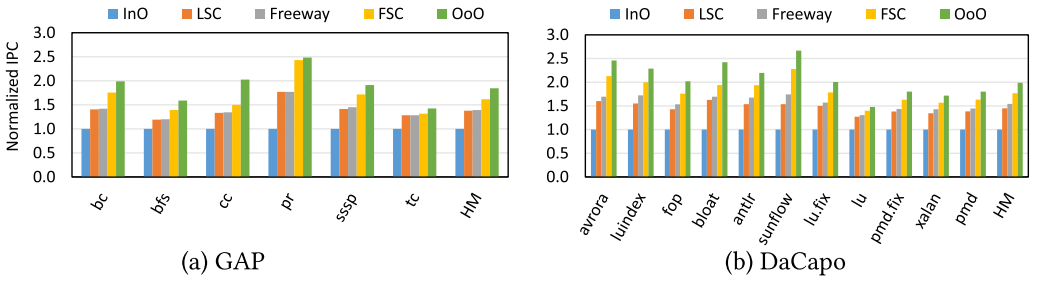

Fig. 11. Performance for the 3-wide cores for (a) GAP and (b) DaCapo. *For GAP, FSC improves performance by 78.4% versus 46.1% for LSC and 47.4% for Freeway. For DaCapo, FSC improves performance by 76.8% on average compared to the InO baseline, versus 44.9% for LSC and 54.1% for Freeway.*

Figure 11 for DaCapo and GAP. For SPEC CPU2017, LSC and Freeway improve performance by 37.2% and 41.1% on average, respectively, compared to the InO baseline. FSC in contrast outperforms the InO core by 79.2%; this is an additional gain of 27.1% (or 38.1 percentage point) over Freeway. FSC performs within 12.8% of the OoO core. The performance benefit for DaCapo and GAP also increases compared to the 2-wide configurations: FSC outperforms the InO baseline by on average 76.8% and 78.4%, and outperforms Freeway by 14.6% and 21.1%, respectively. FSC performs within 11.1% and 11.8% of the OoO core for DaCapo and GAP, respectively.

It is interesting to contrast the results for the 3-wide configurations against the 2-wide configurations. We note that FSC's performance improvement over our InO baseline significantly increases with increasing pipeline width: from 61.1% for 2-wide to 79.2% for 3-wide for SPEC CPU2017; similarly, from 56.6% to 76.8% for DaCapo, and from 58.4% to 78.4% for GAP. Moreover, the performance improvement over prior work (Freeway) also widens with increasing pipeline width. FSC outperforms Freeway by 15.7% for the 2-wide configuration versus 27.1% for the 3-wide configuration for SPEC CPU2017; similarly for DaCapo and GAP, FSC performance advantage over Freeway increases from 10.1% to 14.6%, and from 6.9% to 21.1%, respectively.

The fundamental reason why the performance improvement of FSC over InO and the prior sOoO cores increases with increasing pipeline width is twofold. First, FSC better exploits MHP. In particular, the three-wide configurations have a larger ROB, i.e., there are more instructions in flight, and hence there is more opportunity to exploit MHP from the dynamic instruction stream. A necessary condition for exploitable MHP is that independent misses coincide in the processor pipeline, which increases with a larger ROB. FSC is hence better capable of extracting MHP that is inherently available in the instruction stream. Second, FSC better exploits ILP. A wider pipeline is more sensitive to ILP because there is more opportunity to dynamically schedule instructions and hide latencies. Because FSC is better able at extracting ILP by steering arithmetic instructions to multiple queues as opposed to a single queue in the InO and sOoO cores, FSC is able to extract more ILP. The next section details how FSC extracts high degrees of ILP and MHP.

5.3 CPI Stack Analysis

The fundamental reason why FSC outperforms Freeway (and LSC) is because it improves both ILP and MHP. This is illustrated in Figure 12 which reports average CPI stacks for the 3-wide configurations; CPI stacks break up the average number of cycles executed per instruction in a base component (useful computation) and miss event components due to branch mispredicts, cache misses, etc. There are several interesting observations to be made here. First, all three sOoO architectures effectively improve MHP. This is evident from the large reduction we observe for the combined

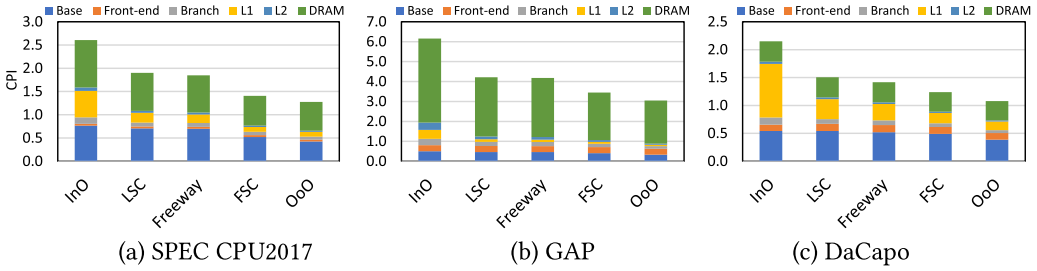

Fig. 12. Average CPI stacks for SPEC CPU2017, GAP and DaCapo comparing the 3-wide InO, LSC, Freeway, FSC and OoO microarchitectures. *Compared to LSC and Freeway, FSC improves both the base and memory-related (L1, L2 and/or DRAM) CPI components across all three benchmark suites.*

memory-related CPI components (L1-D, L2, and DRAM) compared to the InO baseline: average reduction of 38% for LSC, 41% for Freeway and 53% for FSC. As mentioned before, sOoO cores achieve higher MHP compared to an InO core by moving load-dependent instructions to another in-order queue so that younger independent load instructions can soon issue as well, exposing higher degrees of MHP.

Second, FSC yields the highest MHP improvement of all sOoO architectures. In particular, compared to the in-order baseline, FSC reduces the memory-related CPI components by 53% on average and, more specifically, by 53%, 49%, and 59% for the SPEC CPU2017, GAP, and DaCapo benchmarks suites, respectively. Compared to Freeway, these components reduce by 21% on average across the three benchmark suites with a reduction of 24% (SPEC CPU2017), 20% (GAP), and 19% (DaCapo), respectively. The reason is that FSC does not depend on imprecise backward slice analysis, and, more importantly, because it moves instructions that depend on load instructions that miss in L1 or beyond to the Holding Lane so that younger independent load instructions can execute faster.

Finally, FSC significantly improves ILP. LSC and Freeway do not improve the compute CPI Base component for SPEC CPU2017 and GAP considerably, whereas FSC decreases the compute component by 32% and 18%, respectively. For DaCapo, we find that LSC and Freeway reduce the compute component by less than 0.1% and 4.3%, respectively, versus 9.8% for FSC. In other words, FSC consistently improves ILP across all workload types. The reason is increased opportunity to expose ILP to the functional units. FSC steers non-load instructions to the ML and DEL lanes, while DEL instructions may be redirected to the HL. In other words, non-load instructions may be issued to a functional unit from possibly three lanes (ML, DEL, and HL). In contrast, LSC and Freeway steer non-load instructions that do not contribute to address generation to a single queue only, limiting the degree of exploitable ILP.

5.4 Performance Analysis Across Workloads

It is worth further analyzing how FSC performs across the different benchmark suites. One particularly interesting observation is that the performance improvement through FSC is more robust or consistent across the three benchmark suites than Freeway. In particular, for the 2-wide configurations, we observe that the performance improvement for Freeway relative to the InO baseline ranges from 39.2% (SPEC CPU2017) to 48.1% (GAP), which denotes a spread in improved performance of 8.9 percentage points. For FSC on the other hand, we observe that the spread in improved performance is almost twice as small at 4.5 percentage points, ranging between 56.6% (DaCapo) and 61.1% (SPEC CPU2017). The situation is even more pronounced for the 3-wide configurations, for which the spread in performance improvement is more than five times smaller for FSC (between 76.8% and 79.2%, or 2.4 percentage points) than for Freeway (from 41.1% to 54.1% or

Table 3. Area and Power Overhead for the Various FSC Structures Over a Baseline Three-wide In-order Core

Structure	*Details*	*Overhead* (Bytes)	*Area* (mm^2)	*Power* (mW)
RAT	64 entries	48	0.0079	5.01
PRF	64 int/64 fp	1,536	0.0652	9.51
SBV	128 entries	16	0.0004	<0.01
ML	12 entries	264	0.0086	2.92
DEL	12 entries	264	0.0086	1.99
DLL	12 entries	264	0.0086	0.73
HL	12 entries	264	0.0086	0.84
ROB	64 entries	640	0.0138	9.15
MSHR	8 entries	52	0.0056	0.55
SQ	24 entries	192	0.0046	1.30
Total		3,540	0.1356	32.00

13 percentage points). The fundamental reason is that FSC improves both ILP and MHP, whereas Freeway primarily improves MHP only, as discussed above. In other words, while prior sOoO microarchitectures are primarily effective for MHP-intensive workloads, FSC is effective for a broader range of workload types with varying characteristics in ILP and MHP, which leads to consistent performance improvements.

Another observation worth making is that the performance improvement for FSC compared to Freeway is higher for SPEC CPU2017 than DaCapo and GAP. Indeed, for the 2-wide cores, FSC achieves 15.7% higher performance than Freeway for SPEC CPU2017, versus 10.1% for DaCapo and 6.9% for GAP. This holds even stronger on the 3-wide cores, where FSC achieves 27.1% higher performance compared to Freeway for SPEC CPU2017 versus 14.6% and 21.1% for DaCapo and GAP, respectively. The reason is that Freeway achieves a higher performance improvement over the InO baseline for DaCapo (54.1% for the 3-wide configuration) and GAP (47.4%) than for SPEC CPU2017 (41.1%), so that, intuitively speaking, there is less headroom left for FSC to improve. More fundamentally, DaCapo and GAP feature many chains of instructions that depend on L1 D-cache hits and memory accesses, respectively. Freeway (as well as LSC) effectively steers load-dependent instructions to another queue for improved performance. SPEC CPU2017 exhibits fewer such dependence chains while featuring more complex dependence graphs between arithmetic instructions, which explains why FSC achieves higher ILP improvements than Freeway (and LSC) for these workloads.

5.5 Hardware Overhead

FSC adds hardware structures over a baseline in-order core. Table 3 lists the size of these hardware structures (in number of bytes and mm^2) over a three-wide baseline. We add a **register allocation table (RAT)** for register renaming, a reorder buffer for a maximum of 64 instructions, and a 24-entry **store queue (SQ)**. The steering bit vector has 128 entries which amounts to 16 bytes. FSC implements four 12-entry instruction queues (ML, DEL, DLL, and HL). We assume a **physical register file (PRF)** with 64 integer and 64 floating-point registers. The MSHR is extended to support 8 outstanding misses.

We use CACTI [24] to estimate chip area overhead. CACTI accounts for the area of circuit-level structures such as hierarchically repeated wires, arrays, logic and the clock distribution network. For a 3-wide core, the chip area incurred by the additional FSC structures over a baseline 3-wide in-order core amounts to 0.14 mm^2, or a 2.2% increase over a 6.15 mm^2 baseline InO core. For our

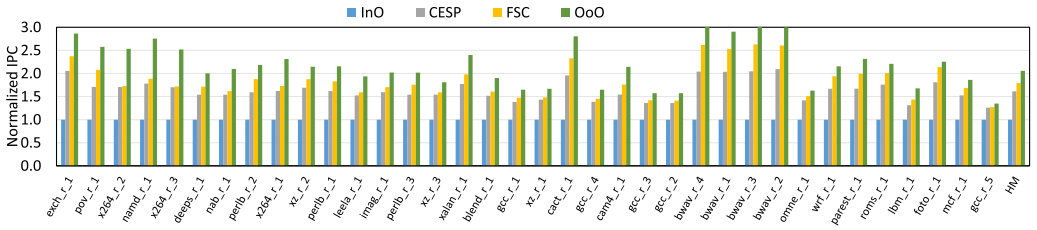

Fig. 13. Normalized performance for CESP, FSC, and OoO for SPEC CPU2017 assuming 3-wide core configurations. *FSC improves performance by 11% on average, and up to 28%, compared to CESP.*

2-wide setup, see our conference paper [21], the additional FSC structures amount to 0.06 mm^2, or a 1.0% increase over a 5.98 mm^2 InO core. As we scale the superscalar pipeline width from 2 to 3, FSC incurs a 3.9% increase in chip core area. In contrast, scaling the OoO core from a 2-wide to 3-wide configuration increases chip area by 11.8% due to more costly CAM structures. Relative to the 3-wide OoO core, we find that the FSC core occupies 47% less chip overhead. This is a more significant saving in chip area as for the 2-wide configurations, i.e., we reported a 37% reduction in chip area for the 2-wide configurations in the conference paper [21]. The bottom line is that the reduction in hardware overhead for FSC relative to an OoO baseline increases with increasing pipeline width.

5.6 Power Consumption

We use McPAT [23] to calculate InO and OoO core power consumption. The power consumed by the additional FSC hardware structures is modeled using CACTI [24]. Table 3 reports power consumption for the newly added components for SPEC CPU2017 benchmarks.

For a 3-wide core and SPEC CPU2017, the additional power consumption incurred by the additional FSC structures over a baseline in-order core amounts to 32 mW relative to our baseline InO core which consumes 3.12 W versus 8.15 W for the OoO core. Similarly, for a 2-wide setup reported in the conference paper [21], the additional FSC structures account for 19.4 mW, versus 2.99 W and 6.95 W for the InO and OoO cores, respectively. As we scale the superscalar pipeline width from 2 to 3, FSC incurs a 4.7% increase in power consumption, versus 17.3% for the OoO core. Relative to an OoO core, FSC significantly reduces power consumption. Moreover, the reduction increases with increasing pipeline width, from 56.7% for the 2-wide configuration to 61.3% for the 3-wide configuration. In other words, the FSC microarchitecture is more power-efficient for wider pipelines than an OoO core, relatively speaking.

5.7 Comparison Against CESP

Palacharla et al. [29] propose the **complexity-effective superscalar processor (CESP)** architecture which steers chains of dependent instructions into generic in-order queues. Figures 13 and 14 compare FSC against CESP with four queues for SPEC CPU2017, and GAP and DaCapo workloads, respectively. 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 incurs even less hardware as it simply steers instructions to the appropriate queue based on a single SBV bit and instruction type (load vs. non-load).

In spite of its lower hardware complexity, we find that FSC outperforms CESP by 11%, 7.7%, and 16.1% on average (18, 13, and 25 percentage point), and up to 28%, 41%, and 33% for a 3-wide

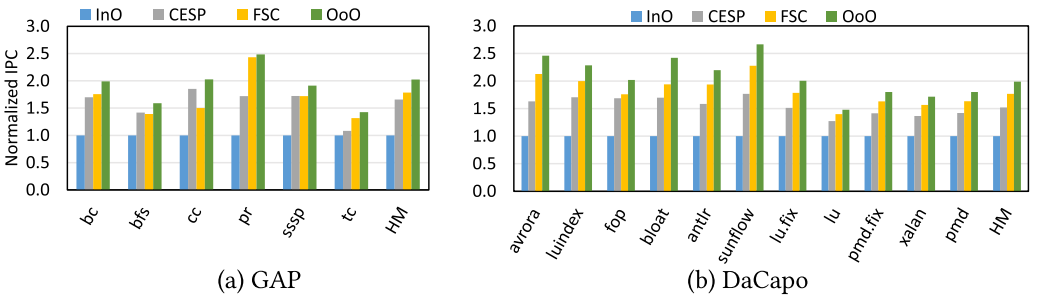

Fig. 14. Performance for 3-wide CESP, FSC, and OoO configurations for (a) GAP and (b) DaCapo. *FSC improves performance by 7.7% and 16.1% on average over CESP for the GAP and DaCapo benchmark suites, respectively.*

configuration running the SPEC CPU2017, GAP, and DaCapo workloads, respectively. The reason why FSC outperforms CESP 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, i.e., it does not need to wait for an empty queue in case of an independent instruction. This is particularly the case for DaCapo: the average 52% performance improvement for CESP compared to the in-order baseline is relatively small compared to SPEC CPU2017 (61%) and GAP (65%). FSC is able to overcome the dispatch stalls due to non-empty queues when dispatching independent instructions, thereby yielding the highest performance over CESP for DaCapo relative to SPEC CPU2017 and GAP. We further observe two anomaly benchmarks, namely GAP's cc and bfs, for which FSC performs (slightly) worse than CESP. Detailed analysis reveals that in the case of cc, a large fraction of instructions are steered to the dependent lanes (i.e., 53% of the instructions are steered to the DEL, out of which 48% are redirected to the HL, and 35.5% of the instructions are steered to the DLL), resulting in an imbalance across the lanes, up to the point that the dependent lanes are getting full. CESP steers load instructions and their consumers to the same queue, which leads to a better overall balance across the queues. Finally, it is interesting to note that the performance improvement over CESP increases with increasing pipeline width—we reported an average 4.5% (and up to 12.6%) improvement for a 2-wide FSC configuration and SPEC CPU2017, see the conference paper [21], while we now report an 11% average improvement for the three-wide configuration.

5.8 Hardware Prefetching

We did not assume hardware prefetching 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 15 reports normalized performance with the stride-based prefetcher enabled for all the 3-wide core microarchitectures. For all configurations, performance is normalized to the InO core with hardware prefetching. It is interesting to note that the performance improvement achieved by the sOoO cores 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, for SPEC CPU2017, GAP, and DaCapo, we report that with hardware prefetching enabled, FSC improves performance by 38%, 21%, and 19% over Freeway, respectively. For comparison, without a hardware prefetcher, FSC's performance improvement over Freeway equals 27%, 21%, and 15%, respectively, as previously reported in Figures 10 and 11.

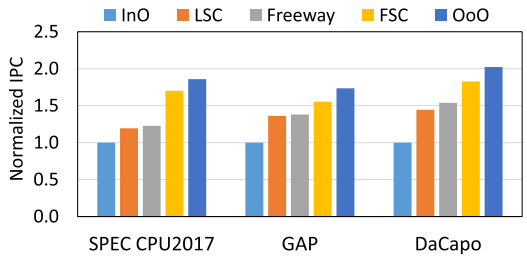

Fig. 15. Normalized performance for 3-wide LSC, Freeway, FSC, and OoO with a 16-stream stride-based hardware prefetcher at L1. Performance is normalized to the InO core with prefetcher in all configurations. *FSC improves performance by 38%, 21%, and 19% compared to Freeway baseline with hardware prefetching for SPEC CPU2017, GAP, and DaCapo, respectively.*

We therefore conclude that FSC's relative performance gain over other sOoO cores improves with the addition of a hardware prefetcher.

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. [29] 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 [31] 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. [19] 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 [32] 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.

Latency-Tolerant Architectures. Prior work proposed various mechanisms to tolerate or hide memory accesses in traditional out-of-order and in-order cores by exploiting the notion of forward slices. Lebeck et al. [22] add a 2K-entry **Waiting Instruction Buffer (WIB)** to an OoO core to temporarily store forward-slice instructions that directly or indirectly depend on a cache miss. When the long-latency operation completes, the instructions are reinserted from the WIB into the issue queue. Runahead execution [14, 25–27] removes the blocking cache miss from the instruction window and continues to speculatively prefetch future memory addresses till the blocking miss returns. **Continuous Flow Pipelines (CFP)** [37] build on top of the **CheckPoint and Renaming (CPR**) proposal [1], releasing scheduler and register file resources for off-chip load-dependent instruction slices. Hilton et al. [16] and Nekkalapu et al. [28] adapt the CFP concept to an in-order architecture by diverting forward-slice miss-dependent instructions to a slice buffer. When the miss

returns, the miss-dependent instructions are merged back from the slice buffer into the pipeline. The Sun Rock processor [11] uses **Deferred Queues (DQs)** to hold forward-slice miss-dependent instructions and their available operand values. These prior works incur significant complexity. In particular, CFP, iCFP and Rock rely on complex checkpointing mechanisms to switch between non-speculative and speculative execution modes. Moreover, memory disambiguation hardware is required to support out-of-order execution of memory operations. FSC in contrast exploits the notion of forward slices in the context of restricted out-of-order microarchitectures, incurring lower complexity compared the above prior works, because (1) it does not rely on checkpointing to defer forward-slice instructions, and (2) it simply replicates store-address micro-ops across in-order FIFO queues to resolve memory addresses in program order, eliminating the need for complex memory disambiguation hardware support.

Decoupled Access-Execute. DAE [36] is the first work to separate access and execute phases of a program through coordinated queues. Proposals such as speculative-slice execution [44], flea-flicker multi-pass pipelining [5], braid processing [42] and OUTRIDER [12] also exploit critical instruction slices [45] for improving performance. More recently, Clairvoyance [40] and SWOOP [41] 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 [9] and Freeway [20] throughout the paper. Shioya et al. [35] 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 [18] 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 [33] 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. [2] 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 article, 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 27.1%, 21.1%, and 14.6%

for SPEC CPU2017, GAP, and DaCapo, respectively, for a 3-wide superscalar configuration, while at the same time being more power- and hardware-efficient.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful feedback and comments.

REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 423–434.

[2] M. Alipour, S. Kaxiras, D. Black-Schaffer, and R. Kumar. 2020. 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)*. 424–434.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (Jan. 2000), 211–238. https://doi.org/10.1147/sj.391.0211

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

[5] R. D. Barnes, S. Ryoo, and W. W. Hwu. 2005. "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)*. 319–330.

[6] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv e-prints*, Article arXiv:1508.03619 (Aug. 2015), arXiv:1508.03619 pages. arXiv:cs.DC/1508.03619.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. Eliot B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 169–190.

[8] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM* 51, 8 (Aug. 2008), 83–89. https://doi.org/10.1145/1378704.1378723

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

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

[11] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. 2009. Simultaneous speculative threading: A novel pipeline architecture implemented in sun's rock processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. Association for Computing Machinery, New York, NY, USA, 484–495. https://doi.org/10.1145/1555754.1555814

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

[13] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. 2013. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 355–372.

[14] J. Dundas and T. Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing (ICS)*. 68–75.

[15] L. Eeckhout. 2010. *Computer Architecture Performance Evaluation Methods*. Morgan and Claypool Publishers.

[16] Andrew Hilton, Santosh Nagarakatte, and Amir Roth. 2009. iCFP: Tolerating all-level cache misses in in-order processors. In *Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 431–442.

[17] S. Blackburn J. Ha, M. Gustafsson and K. S. McKinley. 2008. Microarchitectural characterization of production JVMs and Java workloads. In *IBM CAS Workshop*.

[18] I. Jeong, S. Park, C. Lee, and W. W. Ro. 2020. 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)*. 383–396.

[19] H. Kim and J. E. Smith. 2002. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*. 71–81.
[20] Rakesh Kumar, Mehdi Alipour, and David Black-Schaffer. 2019. Freeway: Maximizing MLP for slice-out-of-order execution. In *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA)*. 558–569.
[21] Kartik Lakshminarasimhan, Ajeya Naithani, Josué Feliu, and Lieven Eeckhout. 2020. The forward slice core microarchitecture. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 361–372.
[22] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*. 59–70.
[23] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 469–480.
[24] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. 2011. 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)*. 694–701.
[25] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*. 129–140.
[26] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout. 2021. Vector runahead. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*. 195–208.
[27] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. 2020. Precise runahead execution. In *Proceedings of the 26th IEEE Symposium on High Performance Computer Architecture (HPCA)*. 397–410.
[28] Satyanarayana Nekkalapu, Haitham Akkary, Komal Jothi, Renjith Retnamma, and Xiaoyu Song. 2008. A simple latency tolerant processor. In *2008 IEEE International Conference on Computer Design*. 384–389. https://doi.org/10.1109/ICCD.2008.4751889
[29] S. Palacharla, N. P. Jouppi, and J. E. Smith. 1997. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*. 206–218.
[30] Research and Markets. 5G Revenue and Devices Forecast. https://www.globenewswire.com/news-release/2020/06/15/2047817/0/en/5G-Devices-Market-Worth-46-Billion-by-2030-Despite-the-Impact-of-COVID-19.html.
[31] P. Salverda and C. Zilles. 2007. 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*.
[32] P. Salverda and C. Zilles. 2008. Fundamental performance constraints in horizontal fusion of in-order cores. In *Proceedings of the 14th Annual International Symposium on High Performance Computer Architecture (HPCA)*. 252–263.
[33] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud. 2015. Long term parking (LTP): Criticality-aware resource allocation in OOO processors. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 334–346.
[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 45–57.
[35] R. Shioya, M. Goshima, and H. Ando. 2014. A front-end execution architecture for high energy efficiency. In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*. 419–431.
[36] J. E. Smith. 1982. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture (ISCA)*. 112–119.
[37] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. 2004. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 107–119.
[38] Statista. IoT Devices Forecast. https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/.
[39] Statista. Smartphone Users. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.
[40] K. A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean. 2017. Clairvoyance: Look-ahead compile-time scheduling. In *Proceedings of the International Conference on Code Generation and Optimization (CGO)*. 171–184.
[41] K. A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras. 2018. SWOOP: Software-hardware co-design for non-speculative, execute-ahead, in-order cores. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 328–343.

[42] F. Tseng and Y. N. Patt. 2008. Achieving out-of-order performance with almost in-order complexity. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*. 3–12.
[43] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. 2011. Why nothing matters: The impact of zeroing. *SIGPLAN Not.* 46, 10 (Oct. 2011), 307–324. https://doi.org/10.1145/2076021.2048092
[44] C. Zilles and G. Sohi. 2001. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*. 2–13.
[45] C. B. Zilles and G. S. Sohi. 2000. Understanding the backward slices of performance degrading instructions. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 172–181.

Received July 2021; revised September 2021; accepted November 2021