Efficiently Scaling Out-of-Order Cores for Simultaneous Multithreading

Faissal M. Sleiman and Thomas F. Wenisch
Computer Engineering Laboratory
EECS Department, University of Michigan
Ann Arbor, Michigan, USA
{*sleimanf, twenisch*}*@umich.edu*

***Abstract*—Simultaneous multithreading (SMT) out-of-order cores waste a significant portion of structural out-of-order core resources on instructions that do not need them. These resources eliminate false ordering dependences. However, because thread interleaving spreads dependent instructions, nearly half of instructions dynamically issue in program order after all false dependences have resolved. These *in-sequence* instructions interleave with other *reordered* instructions at a fine granularity within the instruction window. We develop a technique to efficiently scale in-flight instructions through a hybrid out-of-order/in-order microarchitecture, which can dispatch instructions to efficient in-order scheduling mechanisms—using a FIFO issue queue called the *shelf*—on an instruction-by-instruction basis. Instructions dispatched to the shelf do not allocate out-of-order core resources in the reorder buffer, issue queue, physical registers, or load-store queues. We measure opportunity for such hybrid microarchitectures and design and evaluate a practical dispatch mechanism targeted at 4-threaded cores. Adding a shelf to a baseline 4-thread system with 64-entry ROB improves normalized system throughput by 11.5% (up to 19.2% at best) and energy-delay product by 10.9% (up to 17.5% at best).**

***Keywords*-microarchitecture; in-sequence; reorder**

I. Introduction

Modern processors use a variety of microarchitectural techniques to enhance application performance. Out-of-order (OOO) execution and simultaneous multithreading [1] (SMT) are two such techniques, which seek to utilize superscalar execution resources by increasing single-threaded instruction-level parallelism and thread-level parallelism, respectively. By incorporating both OOO and SMT hardware, some designs seek to balance single-threaded performance and throughput. This combination comes at an efficiency cost, as OOO and SMT mechanisms compete to fill the same functional units using different types of parallelism. As such, prior work finds that the throughput of an in-order (INO) core approaches that of an OOO core as the number of SMT threads is increased [2].

OOO hardware enables early issue of instructions that encounter false dependences, for which INO cores must stall. However, in SMT cores, the last-arriving input operand (true dependence) for a significant fraction of instructions arrives after all false dependences have resolved. Such instructions, which we call *in-sequence*, do not stall in INO cores and naturally issue after all elder instructions (i.e., in program order) in OOO cores. Conversely, we refer to instructions that naturally issue out of program order as *reordered*. We find that having more SMT threads increases the fraction of in-sequence instructions observed in a particular OOO instruction window. Figure 1 illustrates the extent of this effect; as the number of threads in a 128-entry OOO instruction window is increased, the fraction of in-sequence instructions more than doubles to more than 50% on average.

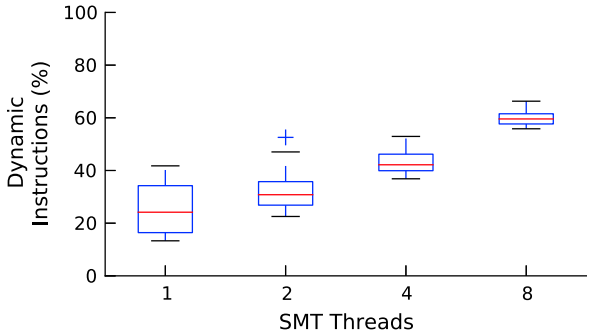

Figure 1: Fraction of instructions wasting OOO resources.

In-sequence instructions gain no benefit from the OOO microarchitecture structures they occupy. In fact, these instructions can be safely executed on schedule *without* allocating in OOO structures, including the reorder buffer, issue queue, load-store queue, and physical register file. However, in-sequence instructions interleave at fine granularity with reordered instructions. We find that groups of consecutive in-sequence or reordered instructions average 5 to 20 instructions per group. So, existing hybrid INO/OOO microarchitectures [3], [4], which switch at 1000-instruction (or higher) granularity, cannot exploit the in-sequence phenomenon without sacrificing performance on reordered instructions.

Instead, we propose a microarchitecture where in-sequence instructions occupy an energy-efficient FIFO queue, which we call the *shelf*[1], from which instructions may issue only in sequence (reordered instructions occupy a conventional, unordered issue queue). By shifting in-

[1] We borrow the naming concept for the *shelf* from the Metaflow architecture [5], as our structure is based on the principle of shelving deferred instructions.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.45

sequence instruction occupancy to the inexpensive shelf, capacity in OOO structures is freed for reordered instructions. As Figure 1 shows that more than half of instructions are in-sequence in a 4-thread SMT, we aim to double the effective instruction scheduling window simply with the allocation of FIFO queues.

This paper makes contributions in three areas. First, we report on the correlation between in-sequence instructions and the effectiveness of OOO hardware—in-sequence instructions gain no benefit from OOO mechanisms. Second, we leverage that insight to design a microarchitecture that integrates a shelf into an SMT-enabled out-of-order core. To our knowledge, this is the first design that enables a modern dynamically scheduled instruction window, with instruction reordering and register renaming, to contain statically scheduled, unreordered instructions, which reuse the same registers, chosen at instruction granularity. We evaluate the opportunity for such microarchitectures to improve energy-delay product under an oracle dispatch mechanism. Finally, we design a simple hardware steering mechanism that determines whether instructions must be steered to the shelf based on whether they are predicted to be in-sequence or reordered in the future schedule. With this practical steering mechanism, a 64-entry shelf improves normalized system throughput (a metric that considers both performance and fairness across threads [6]) by 11.5% (up to 19.2% at best) and energy-delay-product by 10.9% (up to 17.5% at best) over a baseline 4-thread OOO core with a 64-entry ROB.

II. Background and Design Overview

OOO cores *dispatch* instructions into a dynamic scheduling window where they can be selected to *issue* to functional units out of program order. On the other hand, a simple INO core stalls at the issue stage until all ordering dependences resolve. We identify three such dependences that cause simple INO cores to stall, but do not stall OOO cores. These are data, speculation, and structural dependences. We consider an instruction to be *reordered* if it issues to functional units before all three types of dependences are resolved, otherwise the instruction is *in-sequence*.

Data dependences govern the order in which register reads and writes must be performed. These comprise the well-known Read-After-Write (RAW) or true dependence, as well as the Write-After-Write (WAW) and Write-After-Read (WAR) false dependences. The simple INO core stalls for true and false data dependences by issuing instructions in program order, which takes care of WAR hazards, and with the use of a register ready bit-vector, which can detect RAW and WAW dependences. *Speculation dependences* involve speculative execution of instructions, including processor speculation on control flow, such as after a branch or excepting instruction, or on values, such as those returned by loads executed early in memory order. We consider that speculation is bounded by a known maximum latency that

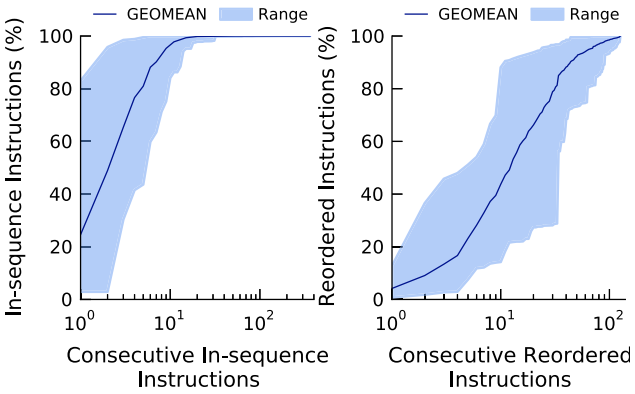

Figure 2: Weighted cumulative distribution of consecutive in-sequence and reordered instruction series lengths.

is a function of the pipeline, especially for relaxed memory models (Section III-D). Speculation dependences can be handled at the issue stage using Smith and Pleszkun's result shift register [7]. *Structural dependences* represent resource constraints that prevent instructions from proceeding to the next stage, as in pipeline stalls in an INO core, or a temporary shortage of functional units for a particular type of operation. Structural dependences are honored automatically by the FIFO nature of the INO issue stage.

Thus, all instructions in a simple INO core are in-sequence, while OOO cores allow some instructions to issue sooner (relative to other instructions) than they would have in an INO core. OOO achieve this dynamic scheduling window by provisioning a number of hardware structures, including a reorder buffer (ROB), issue queue (IQ), load-store queue (LSQ), and physical register file (PRF). These structures alleviate data, speculation, and structural dependences by implementing register renaming, instruction reordering, and associative wakeup. Larger OOO cores scale all structures in a balanced fashion so that no single structure is a dominant bottleneck.

In-sequence and reordered instructions interleave at fine granularity in an OOO core. Figure 2 depicts the cumulative distribution of consecutive in-sequence and reordered series lengths, weighted by the number of instructions in the series (the series length). The plot shows the geometric mean across benchmarks, as well as their range of behavior, for single-threaded benchmarks. We find that 99% of in-sequence instructions occur in series with 30 instructions or fewer, while a series of reordered instructions is bound by the ROB size (128 entries in this case). SMT workload mixes with 2, 4 and 8 threads generally produce similar distributions.

Our main observation is that in-sequence instructions do not need costly OOO structures to execute correctly on schedule. We demonstrate in the coming sections a practical OOO microarchitecture wherein in-sequence instructions

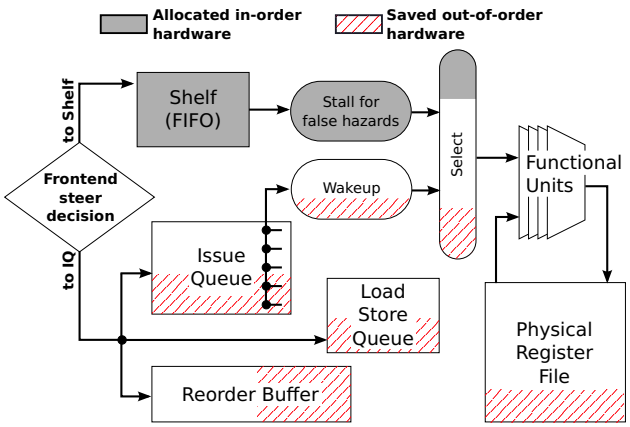

Figure 3: Design overview with FIFO shelf.

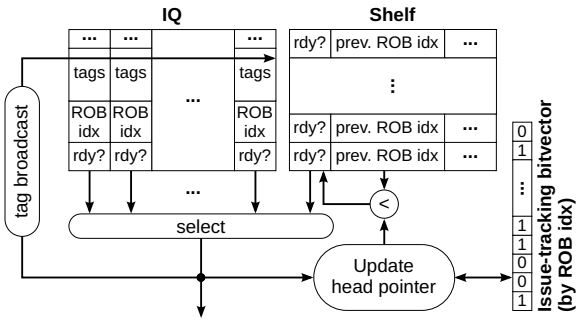

Figure 4: In-order issue of shelf instructions.

can be selected on an instruction-by-instruction basis and skip allocation in OOO structures, *while still executing correctly in the same schedule as a conventional OOO core*.

Nevertheless, in-sequence instructions must still be buffered so as to extend the OOO instruction window. We provide this buffering via a per-thread in-order issue queue, which we call a *shelf*. A shelf is a FIFO buffer that holds instructions in between the dispatch and issue stages much like the (fully associative) IQ. It serves to unblock the dispatch stage to allow reordered instructions to proceed past stalled in-sequence instructions. Shelf instructions are not allocated a new PRF or ROB entry. As such, shelf issue logic must detect and handle false dependences by stalling. Ideally, instructions steered to the Shelf are in-sequence instructions, which do not incur additional stalls for false dependences. We discuss steering instructions to the shelf or IQ in Section IV.

Figure 3 illustrates the shelf within a generic OOO pipeline. It depicts incoming instructions from the dispatch stage as steered to the shelf or to the IQ. The steering mechanism may interleave shelf and IQ instructions on an instruction-by-instruction basis. Once instructions have dispatched to the shelf and IQ, the microarchitecture must correctly and quickly resolve true and false dependences across the two queues to prevent unnecessary stalls. Prior hybrid INO/OOO microarchitectures [3], [4] cannot exploit such fine-grain interleaving, while our design can. As the average series length of in-sequence or reordered is on the order of 10 instructions, the instruction window will simultaneously contain multiple series that interdepend. The next section focuses on the details of our mechanism.

III. A Hybrid Instruction Window

The FIFO shelf is designed to avoid costly associative operations like the tag comparison in the IQ as well as store-to-load forwarding and memory order violation detection in the LSQ. We implement the shelf as a circular buffer with head and tail pointers, much like the ROB. All shelf instructions will block behind a stalled head instruction even if they are ready to issue. Each instruction at the head of the shelf will check for false and true dependences before issuing to the functional units.

Ideally, the shelf would only require mechanisms like the simple INO core to ensure that instructions issue in-sequence. However, in-order issue is complicated by the dynicamically-scheduled OOO instruction window. To detect false dependences inexpensively, we maintain the invariant that instructions issue from the shelf in program order. The main implication of INO shelf issue is that shelf instructions must issue after preceding IQ instructions, which we describe in Section III-A. We also modify the stalling mechanisms for speculation (Section III-B) and data dependences (Section III-C). Finally, we discuss memory ordering in Section III-D.

A. Issuing from the Shelf in Program Order

By virtue of the shelf being a FIFO buffer, its instructions are already ordered with respect to each other. So, an instruction at the head of the shelf need only stall for unissued instructions from the immediately preceding series of IQ instructions (earlier series of IQ instructions must have already issued for the shelf instruction to reach the head). For this reason, we designate that a new *run* of instructions starts when an IQ instruction is steered immediately following a shelf instruction from the same thread. One run consists of a series of IQ instructions followed by a series of shelf instructions. An instruction at the head of the shelf that issues after all IQ instructions in the same run is guaranteed to issue in program order.

Since IQ instructions are dynamically scheduled, the first instruction to dispatch is not necessarily the first one to issue. Additionally, consecutive instructions are generally not allocated adjacent entries in an IQ. To track the issue order of IQ instructions, we allocate a per-thread issue-tracking bitvector with one bit per ROB entry, which represents whether the corresponding instruction has yet to issue (see Figure 4). The bit corresponding to an instruction is cleared upon dispatch,

and set upon issue. A head pointer is maintained to track the oldest unissued IQ instruction, similarly to how the ROB tracks the oldest instruction that has not retired. To be eligible for issue, a shelf instruction must ensure that the head pointer has moved past the last IQ instruction in its run. So, as an instruction is dispatched to the shelf, it records the ROB index of the last preceding IQ instruction (i.e., the tail pointer of the ROB/issue-tracking bitvector for its thread). Once the head pointer advances past this index, the shelf head is the eldest unissued instruction and can proceed to issue in program order.

Critical Path Considerations. In a superscalar machine, we may desire to issue a shelf instruction in the same cycle as the last older IQ instruction. We consider the circuit-level critical path challenges associated with same-cycle issue. An issue cycle consists of selecting a number of ready instructions, followed by waking up their dependents to mark them ready for the next cycle. To determine if the head of the shelf is eligible for issue, the issue-tracking bitvector must be updated to reflect the elder IQ instructions selected for issue this cycle. Same-cycle issue of an IQ instruction and subsequent shelf instructions requires this combinational logic to be placed on the critical path of wakeup and select.

In OOO cores with relatively small issue queues, this additional logic may not affect the processor clock frequency. However, in larger OOO designs, issue logic is often already among the longest paths. Hence, we propose the design depicted in Figure 4, which does not bypass issue-tracking bitvector updates, removing these updates from the wakeup-select critical path. As the shelf extends the instruction window, it effectively competes against larger OOO cores with longer critical paths. We evaluate various OOO sizes under the same clock frequency to isolate microarchitectural effects; nevertheless, we assume that small critical path overheads induced by our shelf design compare favorably to the critical paths in larger, slower designs.

B. Handling Speculation

The ROB is the conventional OOO structure that enables misspeculation recovery by maintaining the program order retirement of architectural state. If an instruction misspeculates, the implementation is able to recover the architectural state prior to that instruction, squashing younger instructions in the process. The PRF buffers the alternative versions of register state needed for this recovery process, and typically scales with the OOO instruction window. An instruction is considered *committed* once it can no longer be squashed, and any state it overwrites is no longer needed for recovery. By orchestrating the in-sequence completion of shelf instructions, we find we are able to forego allocation of ROB entries and overwrite previously allocated PRF entries. Shelf instructions must be delayed at issue until they are guaranteed to be committed. We discuss the shelf delay mechanisms below, then discuss how to squash shelf instructions and prevent them from writing back on a misspeculation. Finally we consider coordinating the ROB retire order with shelf instructions.

Delaying Shelf Instructions for Speculation. We first describe the simplest method to delay shelf instruction writeback correctly. This method is based on the result shift register proposed by Smith and Pleszkun in the context of in-order cores with varying but deterministic instruction execution latencies [7]. We introduce a *speculation shift register* (SSR) per thread, which tracks the maximum remaining resolution cycles for any in-flight instruction. As each speculative instruction issues, it sets the SSR to the maximum of its resolution delay and the current SSR value. Since shelf instructions issue in program order, when the instruction at the head of the shelf is eligible for issue, the SSR will have been updated by all older instructions. A shelf instruction can only issue once its minimum execution delay compares greater than or equal to the value in the SSR. Any earlier and it becomes unsafe to issue the shelf head (i.e., it could overwrite the value in its destination register, which is later needed for recovery).

Although the mechanism we have described thus far maintains precise state, it can unnecessarily delay shelf instructions due to speculative execution of younger reordered instructions; such younger instructions may issue early, merging their resolution time into the SSR. In pathological cases, the shelf head may be the eldest incomplete instruction and yet stall indefinitely, until the issue of all younger instructions becomes blocked due to dependences on the shelf. (This pathology could not arise in Smith and Pleszkun's setting, where issue is in-order [7].) To avoid this pathology, we provision additional SSRs. We could enforce precise speculation stalls by provisioning a separate SSR for each run; however, the number of in-flight runs varies greatly over the course of execution. Moreover, to support per-run SSRs, each IQ instruction would need to track which SSR it must update.

Instead, we propose a design with only two SSRs, an IQ SSR and a shelf SSR as shown in Figure 5. All IQ instructions update only the IQ SSR with their resolution time as they issue. Shelf instructions refer only to the shelf SSR to determine if they are safe to issue. Whenever the first shelf instruction in a particular run becomes eligible for in-order issue, the IQ SSR is first copied to the shelf SSR. At this moment, it is guaranteed that all elder IQ instructions have issued and updated the SSR (as the shelf head is the eldest unissued instruction). The IQ SSR may include the resolution delay of younger instructions that issued early, for which we (unnecessarily, but conservatively) enforce a delay. However, the starvation pathology described above is no longer possible, since no more IQ instructions will affect the shelf SSR until the shelf head issues.

Shelf Retirement and Squashing. Having been delayed sufficiently, a shelf instruction that arrives at the writeback

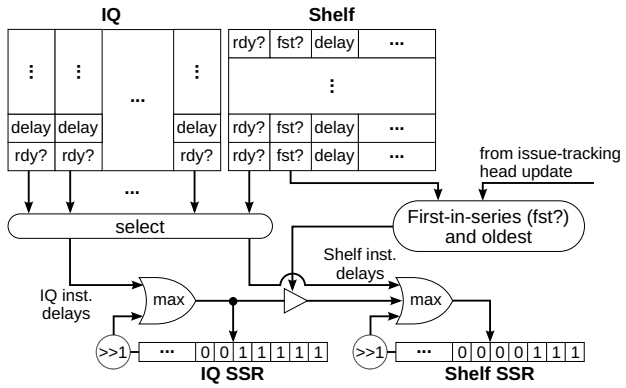

Figure 5: Delaying the shelf for speculation.

stage without being squashed is definitely committed and can be retired. There can be no readers, writers or recoveries to the state that it will overwrite (data dependences are handled in Section III-C). A consequence of this is that shelf instructions may retire out of program order. We discuss coordinating the retire order with the ROB in Section III-B. On a misspeculation, all shelf instructions that need to be squashed are either unissued or still in-flight in execution pipelines. These shelf instructions, possibly including the misspeculating instruction itself, must be prevented from writing back as they complete.

There may also be elder in-flight shelf instructions mixed in, which must *not* be squashed. Hence, a misspeculating instruction must indicate precisely the index of the first shelf instruction to be squashed. This *shelf squash index* can be used to filter out younger shelf instructions as they write back. For misspeculating shelf instructions, identifying the shelf squash index is trivial: it is the misspeculating instruction's own index. For IQ instructions, we store during dispatch the index that the next shelf instruction will be assigned, indicated by the shelf tail pointer.

A consequence of this recovery design is that a shelf index may not be recycled for use by another instruction until its first assignee writes back. In contrast, IQ entries may be recycled immediately once the instruction occupying them issues. A simple solution is to release shelf entries only upon writeback. However, this approach greatly increases shelf occupancy; as our goal is to squeeze the most efficiency out of as little hardware as possible, the increased occupancy is undesirable. We discuss an alternative that decouples the shelf index (which cannot be reused) from the shelf entry (which may then be used by another instruction) below.

ROB Retirement. For IQ instructions, the ROB ensures in-order retirement with respect to other IQ instructions; however, the ROB must also coordinate with the out-of-order retirement of shelf instructions to ensure precise state in the event of a misspeculation—ROB instructions may not retire before older shelf instructions. We address this by tracking shelf instruction retirement in a *shelf retire bitvector*, much like the completion bit associated with each ROB entry. Similar to the head pointer of the ROB, a shelf retire pointer advances over this bitvector, always pointing to the eldest unretired shelf index. Each ROB entry tracks the index of the next shelf instruction to follow it in program order (recall that this is the shelf squash index, discussed above, which we must already track for misspeculation recovery). Once the shelf retire pointer matches or exceeds the stored shelf index, the ROB can retire the next IQ instruction.

Whereas this design ensures correct ordering of ROB retirement with respect to shelf instructions, shelf indices must now be reserved until they are no longer referenced by the ROB. The shelf squash index at the head of the ROB is effectively a shelf reservation pointer, preventing a second shelf instruction from retiring an ambiguous shelf index. This shares the downside noted previously: shelf entries may not be recycled for use by a new instruction, in this case until elder ROB entries retire.

We solve this potential resource shortage by decoupling the allocation and deallocation of the (comparatively) expensive shelf entry from that of the shelf index (a virtual resource). We assume the size of the shelf is a power of two, and allow the shelf index to span a range double the shelf size. The shelf retire and reservation pointers now track shelf indexes in this larger index space, but the most significant bit of the shelf index is not used when accessing shelf entries. Shelf entries may now be reused as soon as the corresponding shelf instruction issues. A single shelf tail pointer is used to allocate a shelf index and the corresponding entry (i.e., ignoring the most significant bit).

C. Handling Data Hazards

To handle data hazards, shelf instructions must stall at issue until it is guaranteed that data dependences are resolved, similar to the simple INO core. Once all data dependences are resolved, in-sequence instructions from the shelf may correctly overwrite the previous value for their destination register. Our strategy, then, is to reuse the previous physical register allocated to the logical identifier for each shelf instruction's destination. We do not allocate new physical registers for shelf instructions, thus reducing the occupancy of the PRF.

Both shelf and IQ instructions translate their source register identifiers in the rename stage to pick up the physical register identifiers (PRI). They also pick up the existing destination register translation; the shelf simply uses it as a destination physical register, while the IQ will retire the identifier back onto the free list as it replaces the translation with a newly allocated physical register mapping. Figure 6 illustrates the life cycle of a PRI. A physical register is first allocated and written by an IQ instruction, and then overwritten by any number of shelf instructions until the next

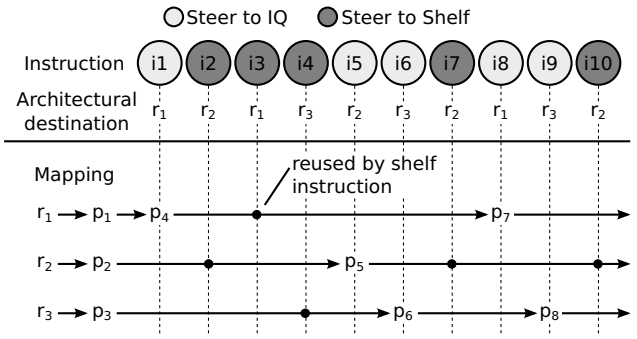

Figure 6: Life-cycle of register alias.

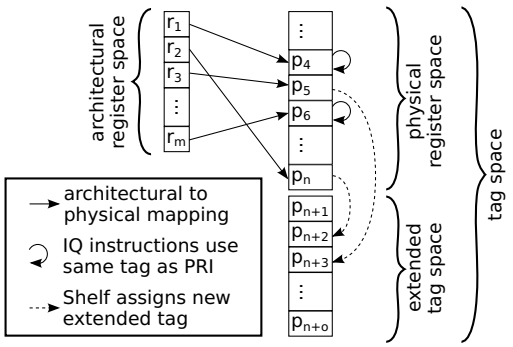

Figure 7: Extended tag space and mapping.

IQ instruction renames the corresponding logical register and eventually retires it.

Instructions at the head of the shelf monitor a ready bitvector for their operand readiness (or may use pipeline interlocks like INO cores) using a conventional scoreboard. The same method can be used to stall shelf instructions for WAW dependences. Nothing additional needs to be done for WAR dependences as the shelf issues in program order. Complications arise, however, when an IQ instruction waits on a true data dependence from an instruction on the shelf; the IQ cannot distinguish the potentially multiple writes to the same physical register by different shelf instructions, which all use the same PRI. In other words, there is an ambiguity in RAW dependences. Shelf instructions avoid this problem because they issue in program order. Once an instruction reaches the head of the shelf, only the last instruction to write a source operand may be outstanding, so there is no ambiguity. Dependent IQ instructions, on the other hand, join a dynamic instruction window, so they observe tag broadcast for multiple shelf writes to the same physical register. The rest of this section describes a mechanism to uniquely identify shelf writes to the same register for the IQ.

Separation of Tag and Physical Register Index. The problem at hand is that the PRI no longer uniquely identifies one instruction in the OOO window, as it does in a conventional PRF-based microarchitecture. Thus, a tag broadcast from one shelf instruction that writes a physical register might incorrectly wake up IQ instructions that depend on a different shelf instruction. To solve this problem while allowing shelf instructions to share a physical register, we must decouple the two traditional roles of the PRI as a destination register and as a unique identifier; each instruction acquires both a PRI and a unique tag from rename. Thus an entry in the mapping table (MT) will now map an architectural register identifier to both a PRI and a tag.

Our implementation expands the tag space in a special way given the life-cycle of an architectural register. For IQ instructions, we retain the original tag space, where each tag corresponds to a particular physical register. When an IQ instruction allocates a new physical register, both its destination PRI and tag are set to that register's index. Shelf instructions allocate a new tag from an extended tag space without allocating a new register, and only change the mapping for the tag. We see that IQ instructions draw only from the original tag space, while shelf instructions draw only from the extended tag space, as depicted in Figure 7. We manage these two portions of the tag space on separate free lists, one *physical* free list for the original tag space and one *extension* free list for the extension.

At rename, IQ instructions read the current mapping for their source operands, noting both the PRI and tag. The PRI is used to index into the PRF, and the tag is used to check readiness and for the wakeup operation. IQ instructions also pick up the current mapping for their destination registers, so as to retire the identifiers from the ROB to their respective free lists. The PRI is retired to the physical free list. If the current PRI and the tag differ, then the tag must be from the tag space extension and is retired to the extension free list. Finally, IQ instructions allocate a new PRI from the physical free list and set both tag and PRI mappings to it.

Shelf instructions similarly record all current mappings. At retire, they only return the tag to the extension free list if it differs from the PRI. Shelf instructions do not retire the PRI as the register remains in use and no new PRI is allocated. Only a tag is allocated from the extension free list, and used to broadcast to the IQ.

Rename Stage. Figure 8 depicts the extended rename stage. Steering is performed during decode, prior to rename, as steering decisions depend only on opcode and the architectural register names of operands and destinations. Depending on whether an instruction is steered to the shelf or to the IQ, its destination register and tag will be different. Tags from the extended tag space are offered by the extended free list (Ext. FL) and register alias table (Ext. RAT), while conventional PRI's are offered by their physical counterparts. The steering decision determines which structures are consulted to allocate a tag.

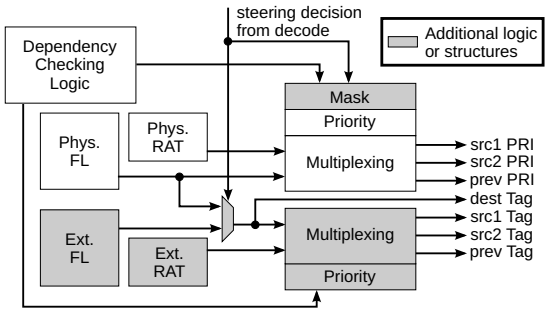

Figure 8: Extended rename stage.

D. Memory Accesses and the LSQ

We first describe the ordering of shelf loads and stores under uniprocessor and relaxed/weak consistency models, which include the ARM v7 memory model used in our evaluation. Shelf loads and stores issue in program order, and thus follow all older loads and stores in the address calculation pipeline. As such, shelf loads and stores do not require their own load or store queue entries; instead, they record the tail pointers of both structures at dispatch to track their relative order.

Shelf loads associatively scan older IQ stores in the store queue, all of which have calculated their addresses and values, and younger IQ loads in the load queue, some of which may have been reordered and issued early to memory. (IQ loads perform the same operations as they execute). The shelf load receives the value from the youngest scanned instruction with a matching address. In particular, it must receive a value from a *younger* matching load to avoid a memory ordering violation [8]. Loads with no matches issue to the cache hierarchy. Loads that issue to the cache wake dependent instructions non-speculatively, resulting in a minimum 2-cycle load-to-use distance for L1 data cache hits. Upon a cache miss, loads (whether from the shelf or IQ) are allocated a miss status holding register, which arbitrates for writeback and tag wakeup when the cache miss returns, unblocking the memory execution pipeline. Memory dependence mispredictions cause a pipeline flush and restart at the mispredicted instruction.

Shelf stores scan younger load instructions for matching addresses to perform store-to-load forwarding, or to squash IQ loads that have speculatively issued early. We use a "store sets" [9] memory dependence predictor to prevent frequent squashes. Shelf stores use their store set identifier to release dependent younger loads, just as IQ stores do. Finally, since uniprocessors and relaxed consistency models support coalescing store buffers and do not require ordering of stores to different addresses, shelf stores scan for the next older matching store and immediately coalesce into its store queue or store buffer entry. It is permissible to skip over older loads in this case because they will have already received a value from the coalescing buffer and taken their place in memory order (non-speculatively). Stores that find no match are released to the cache. We assume memory barriers synchronize the pipeline at the dispatch stage.

Stricter consistency models, like Total Store Order and Sequential Consistency, require in-window speculation [10] for high performance. Amongst other constraints, loads are speculative until all older loads to any address have at least completed (obtained a value from memory). As a consequence, all shelf instructions, including non-memory instructions, that follow a speculative load are speculative and may not writeback/retire until all preceding loads become non-speculative—an uncertain time interval (e.g., duration of a cache miss). Shelf stores additionally need to allocate store queue entries, as strong consistency models often do not permit coalescing in the store buffer. Evaluating the shelf under these models is beyond the scope of this paper. We suggest that steering mechanisms could steer those instructions to the shelf that are predicted to depend on long-latency misses, similarly to recent latency-tolerant designs [11], [12], [13].

IV. Instruction Steering

Instruction steering determines whether an instruction is dispatched to the IQ or the shelf, which directly affects the instruction schedule. Whereas the microarchitecture ensures correct execution under any steering policy, poor steering can result in poor performance. If we steer all instructions to the IQ, then the shelf provides no window size benefit. Conversely, if all instructions are steered to the shelf, the resulting performance will match that of an in-order core.

A. Oracle Steering

To measure the inherent opportunity of a shelf-augmented microarchitecture, we first study an oracle steering mechanism. Unfortunately, a perfect steering mechanism, which steers optimally for maximum performance, is a global optimization requiring complete knowledge of the whole-program critical path. Although mechanisms to predict instruction criticality have been proposed [14], [15], steering compounds the optimization problem: it adds/removes false dependence edges, which changes the very shape of the graph. Hence, even in the context of an offline oracle, perfect steering is intractable.

Instead, we study an oracle that steers each instruction according to whether it would issue earlier from the IQ or the shelf (breaking ties in favor of the shelf). While this determination is made greedily without regard to future (younger) instructions, the greedy oracle steering algorithm requires precise knowledge of the future schedule. Such a mechanism cannot be implemented in practice, since these future arrival times are not always known at dispatch. However, in simulation, we can closely approximate this future schedule using complete knowledge of instruction

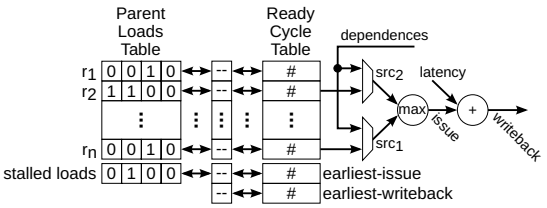

Figure 9: Practical steering.

latencies, dependences, and memory addresses. For memory operations, we functionally query the cache (atomically, instantly and not modifying state) to accurately predict memory latencies.

Note that, because of the complexity of the gem5 simulation model, even with oracle knowledge, we still steer an average of 4% of instructions incorrectly. Though it is highly detailed, our prediction of the future schedule does not account for all corner cases that arise in the simulation. So, our oracle algorithm additionally tracks the actual execution schedule as the simulation progresses to correct its representation of the schedule and recover from mispredictions.

B. Practical Steering

As we will show, our oracle opportunity study reveals relatively limited opportunity for a shelf-augmented microarchitecture in single-threaded executions; as indicated in Figure 1, less than 25% of instructions are in-sequence. In contrast, we demonstrate considerable opportunity as the number of SMT threads increases. Hence, we design a practical steering method targeted for four-threaded execution. The flexibility of the SMT thread fetch policy (ICOUNT in our design [16]) is synergistic with simple instruction steering. When instructions are fetched from a slow-moving thread, they are steered to the shelf, avoiding IQ congestion. Conversely, when an instruction is mis-steered to the shelf, stalling execution, other threads benefit from the available IQ capacity and fill the bubbles with useful execution. This synergy facilitates a steering design without large and power-hungry meta-data structures, which would undermine the energy-efficiency objective of our microarchitecture.

At the heart of steering lies (1) the ability to predict the future execution schedule, and (2) the ability to recover from schedule mispredictions. We describe a practical hardware solution to project instruction completion times and track dependence chains, and show how these mechanisms can be used for steering and misprediction recovery, with reference to Figure 9.

Schedule Prediction. For each architectural register, we maintain a prediction of its future writeback/ready cycle in a Ready Cycle Table (RCT). RCT entries are decremented each cycle to count down how many cycles are left until the register is predicted to be ready.

If we dispatch an instruction to the IQ, we can predict its issue cycle as the maximum ready cycle of its source operands, and its completion cycle as the issue cycle plus the instruction's predicted latency. Instruction latencies are usually available from decode. The prediction ignores structural hazards, such as issue width, and predicts all loads to be L1 hits; the resulting schedule errors are handled via the recovery mechanism. By predicting that all loads hit in L1, we avoid the need for any prediction table.

An instruction dispatched to the shelf will issue after all previously dispatched instructions even if its operands are ready, since the shelf issues in program order. Hence, for the shelf, we maintain an earliest-allowable issue cycle, which is the maximum issue cycle of all previous instructions. Shelf instructions also must stall at writeback while any preceding instruction is speculative. So, we also track an earliest-allowable writeback cycle, which is the maximum speculation resolution cycle for any previous instruction. We can then predict that, if dispatched to the shelf, an instruction will issue at the maximum of its operands' RCT entries and the earliest-allowable issue cycle. Its completion cycle is predicted as the maximum of its predicted issue cycle plus the instruction latency and the earliest-allowable writeback cycle.

With these estimates, we can then steer an instruction by comparing its predicted completion cycle for the shelf and IQ, choosing the earlier of the two and breaking ties in favor of the shelf. Our design exploration shows that it is sufficient to track a range of 32 cycles using 5-bit counters per register.

Schedule Recovery. Our schedule prediction mechanism is approximate; most importantly, it assumes all loads are hits. As schedule errors accumulate, steering accuracy will worsen and performance will suffer. So, we correct schedule misprediction errors by observing the actual execution schedule and using the observed instruction completions to correct predictions for their dependent instructions.

Once a register's RCT counter decrements to zero, the register is predicted to be ready. However, if the instruction took longer than expected (e.g., an L1 miss), the register will not be marked ready in the issue dependency checking logic. In this circumstance, the predicted schedule for all the instruction's dependents is also incorrect. We correct these errors by freezing the decrement of the RCT entry for the destinations of all these dependents. We thereby push back the predicted completion time of the entire dependency tree by one cycle each cycle until the mispredicted instruction ultimately completes.

Maintaining RCT counters as we have just described requires tracking the dependency information among all instructions, which is expensive. Interestingly, we find that tracking the dependents of only a small sample of instructions is sufficient to correct the schedule; a schedule misprediction for an untracked instruction will rapidly be detected when one if its dependents is sampled. Since most

schedule mispredictions are for loads that miss in L1, we track dependents for a sample of loads.

We use a simple bit matrix, the Parent Loads Table (PLT), to track the relationship between sampled loads and their dependents. As loads are steered, each is assigned a column of bits in the PLT, if one is available. Rows in the table correspond to architectural registers; a bit is set if the architectural register depends directly or indirectly on the load. When a load is steered, it sets the bit for its assigned column and destination register row. As further instructions are decoded, they set the row for their destination to the superset of their operands' parent loads (i.e., the bitwise OR of the operands' rows). When loads complete, they reset the bits in their assigned column, freeing the column for reuse. We find it is sufficient to track 4 loads per thread.

If any register's ready cycle reaches zero while its parent loads' bitvector is non-zero, we simply stall the decrement operation for all other registers that share those parents. The register's bitvector is loaded into a special row, the stalled loads bitvector, as shown in Figure 9, which is compared to all rows. Any row with a matching bit (i.e., it is directly or indirectly dependent on a stalled load) has its RCT counter stalled.

V. Evaluation

We model our design in gem5 [17] and run the SPEC CPU2006 benchmark suite with the ARM v7 ISA using system call emulation. We have excluded only *dealII* of the 29 SPEC benchmarks as it is not functional in our simulation infrastructure. For SMT workloads, we generate mixes of 28 different SPEC benchmarks, such that each benchmark appears an equal number of times in each workload, according to the "Balanced Random" mix methodology proposed by Velasquez et al. [18]. We measure performance using system throughput (STP), a metric proposed by Eyerman and Eeckhout [6] that considers both performance improvement and fairness across threads in a multi-threaded mix. STP is the sum of the ratios of each thread's clocks-per-instruction in single-threaded and multi-threaded execution. It reflects the number of programs completed per unit time. We report results for the benchmark mix with the maximum, minimum, and median STP improvement over the baseline, as well as averages across the random mixes. Using the reference input set, we fast forward all threads to the highest-weighted SimPoint [19] within 50 billion instructions of the start of the benchmark. We warm microarchitectural structures for 100 million instructions on each thread prior to the SimPoint location.

Table I details our configuration. We assume a 2GHz clock for all configurations to focus on the microarchitectural effects of our technique. Unless otherwise stated, our evaluation focuses on a 4-thread SMT configuration using the ICOUNT fetch policy [16]. The ROB, load queue (LQ) and store queue (SQ) structures are partitioned across threads, based on [20], as are the front-end pipeline buffers and the shelf to prevent stalled threads from blocking others. Our baseline core has a 64-entry ROB and 32-entry IQ, LQ, and SQ. We augment this core with a 64-entry shelf. We also measure a core where all structures are doubled (128-entry ROB, 64-entry IQ, LQ, SQ), which represents an upper bound for the performance improvement the shelf can provide.

Component	Configuration
Core	4-thread SMT OOO @ 2.0 GHz
	4-wide OOO with 8-wide fetch
	6 cycles fetch-to-dispatch
ROB	64 or 128
IQ, LQ, SQ	32 or 64
Shelf	64
Steering	5-bit RCT entries, 4-load PLT
L1I	32KB, 2-way, 1-cycle
L1D	32KB, 2-way, 2-cycle
L2	2MB, 8-way, 32-cycle
Memory	100ns latency

Table I: System Configuration

For power, energy, and area analysis, we use the McPAT framework [21] to model the power breakdown of a physical register-based OOO design, incorporating changes from [22]. We extend McPAT to model the shelf, RAT/free list, rename logic, expanded issue/scheduling logic, speculation shift registers, dependency tracking, and steering structures/logic. Our additions to McPAT are consistent with its models for baseline scheduling and mapping logic and storage structures. We report on the power consumption of the core including L1 caches.

A. Performance

We first consider the performance impact of our design with practical shelf and steering mechanisms. Figure 10 reports the improvement in system throughput over the baseline 64-entry ROB design. We include results for the workload mixes with the lowest, median, and highest STP improvement over the baseline (the axis labels in Figure 11 report the benchmarks in these three mixes). Finally, we report a geometric mean across all 28 mixes. The rightmost (dark-blue) bar in each group reflects the STP improvement of an out-of-order core where all microarchitecture structures are doubled. This bar represents a theoretical upper bound for the improvement of the shelf.

The shelf-augmented microarchitectures improve performance over the baseline by 8.6% and 11.5% on average and up to 15.1% and 19.2% for the conservative and optimistic microarchitecture assumptions, respectively. Our approach captures almost half of the throughput improvement of the larger OOO core with substantially less hardware. In partic-

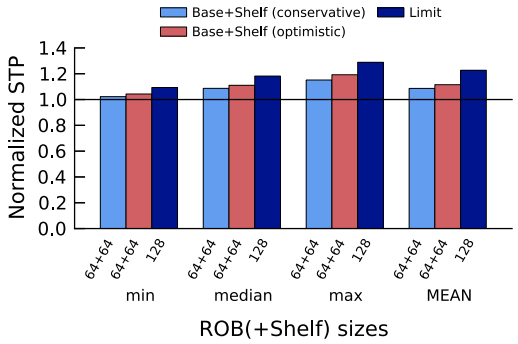

Figure 10: Performance of the shelf with conservative and optimistic microarchitecture assumptions.

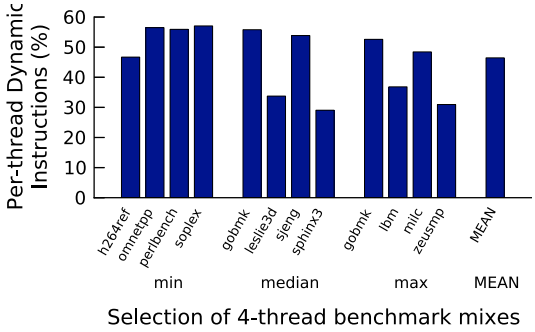

Figure 11: Fraction of in-sequence instructions for a selection of 4-thread benchmark mixes.

ular, the larger design has a 64-entry issue queue, whereas the scheduling logic in our 64+64 entry design considers only 32 reordered instructions and the heads of each shelf. Hence, it is likely the 64+64 design can achieve a higher clock frequency, which is not reflected in this comparison. Nevertheless, the shelf is not as flexible as a larger OOO instruction window. The shelf loses performance when (1) less than half of all in-flight instructions are in-sequence, (2) when window requirements are imbalanced across threads (the shelf is statically partitioned), (3) the steering heuristic mis-steers instructions, or (4) when reordered instructions require more LQ or SQ resources.

Figure 11 shows the fraction of instructions from each thread that are in-sequence for the three selected mixes, as well as the arithmetic mean across all benchmarks. On average, about half of instructions are in-sequence, but some benchmarks have fewer in-sequence instructions. The imbalance in in-sequence instructions across workloads contributes to the gap between the 64+64 entry design and the theoretical upper bound.

The practical steering mechanism makes numerous simplifying assumptions about the future instruction schedule. To gauge the degree to which these approximations lead to mis-steered instructions, we compare oracle and practical steering. Figure 12 shows the resulting performance comparison. Approximately 16% of instructions are steered incorrectly by the practical mechanism relative to the oracle. Nevertheless, the ability of one SMT thread to make progress while another is stalled hides the brief stalls created by incorrect steering decisions, allowing even our simple mechanisms that assumes all memory accesses are L1 cache hits to nonetheless make effective use of the shelf.

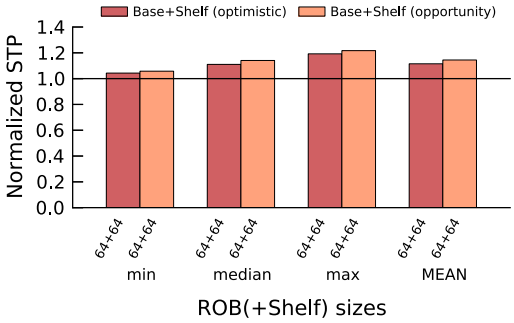

Figure 12: Performance impact of practical steering.

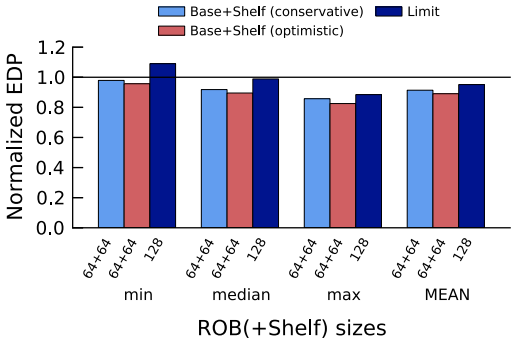

Figure 13: Energy delay.

B. *Energy and Area Efficiency*

We compare the energy efficiency of the 64+64-entry shelf-augmented design to both the baseline 64-entry and doubled 128-entry microarchitectures. Figure 13 shows the energy-delay product (EDP) of each design. Although it consumes more power, a 128-entry design is more energy-efficient on the average than a 64-entry design, improving EDP by 4.9%. However, a 64+64-entry shelf-augmented design is even more energy efficient. The performance advantage of the shelf more than compensates for the slight increase in power consumption, resulting in a net energy-delay win relative to the 64-entry baseline. Adding a shelf improves energy-delay product by 8.6% and 10.9% on average for conservative and optimistic microarchitecture assumptions, respectively.

Table II reports the area increase of the 64+64 and 128-entry designs relative to the 64-entry baseline. Excluding the area of L1 caches, adding a shelf and the associated

L1 caches included	Base+Shelf 64+64	Base 128
no	3.1%	9.7%
yes	2.1%	6.6%

Table II: Area increase over Base 64.

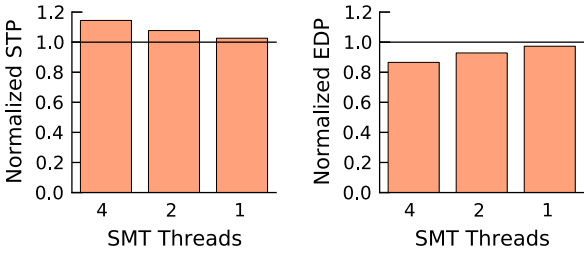

Figure 14: Opportunity with fewer threads.

scheduling, steering, and tracking structures increases the core area by 3.1%. In contrast, doubling the capacity of the IQ, ROB, LQ, SQ, and instruction scheduling logic for the 128-entry design increases area by 9.7%.

C. Fewer Threads

Although in-sequence instructions arise in single-threaded execution, the interleaving of multiple threads in a SMT core spreads the issue of dependent instructions apart, substantially increasing the fraction of in-sequence instructions (see Figure 1). Hence, we do not expect a shelf to improve performance in single-threaded execution. Nevertheless, it is desirable that the shelf not adversely impact performance or energy-efficiency when an SMT core is running only a single thread. Figure 14 compares the STP and energy-delay product of the 64-entry and 64+64-entry designs for single-threaded and two-threaded executions, averaged over 28 benchmarks/mixes. There is no opportunity for a shelf in single-threaded execution. With two threads, the shelf provides a modest improvement in performance and energy delay. Nevertheless, we find that the shelf does not adversely affect performance. Note that the shelf can easily be disabled by steering all instructions to the IQ if it causes pathological behavior in a particular workload.

VI. Related Work

Hily and Seznec [2] show that the performance of an in-order core approaches that of an out-of-order core as the number of SMT threads increases, and argue that OOO cores are not cost-effective for SMT designs with many threads (four in their study). At the two extremes, OOO cores are suited to single-threaded workloads or those with few SMT threads, while workloads with a high number of threads favor in-order cores for efficiency. We reason that middle-range designs, which balance single-threaded performance and throughput, require a new underlying microarchitecture.

We borrow the name and concept for the shelf from the Metaflow architecture [5], which focused on the principle of shelving instructions to defer their execution, thereby enabling the out-of-order execution of other instructions. Ultimately, the Metaflow design was an OOO core centered on the DRIS structure, a combination of the ROB, IQ and renaming logic for all instructions. Our shelf physically separates in-sequence (deferred) instructions into a more efficient structure, while reordered instructions utilize the full capabilities of a modern OOO core.

Khubaib *et al.* rely on the same observations as Hily and Seznec to propose MorphCore [23], a design wherein the core can "morph" from an OOO with a low number of threads (two threads in their work) into an INO core with many (eight) threads. Whereas MorphCore offers a coarse-grain switching mechanism, our design enables the selection of OOO versus INO mechanisms on an instruction-by-instruction basis. MorphCore and our work target different objectives: MorphCore attempts to capture two workloads that do not often coincide, single-threaded and highly threaded, on one core; whereas, our design highlights an area where neither INO nor OOO cores are an efficient design point. Similar works provide a set of configurable cores by morphing, fusing or composing standalone cores [24], [25].

Viewed from another angle, our design attempts to approach the performance of a larger OOO instruction window through the use of in-order hardware. [26], [27] relieve the IQ by redirecting ready-before-dispatch instructions through energy-efficient functional units. Seng, Tune and Tullsen [28] advocate reducing power by utilitizing in-order IQs. Tseng and Patt [29] utilize compiler techniques to achieve a high performing schedule on in-order hardware, which approaches the single-threaded performance of OOO hardware. These designs, however, do not alleviate pressure on the ROB, LSQ and PRF. McFarlin, Tucker and Zilles [30] advocate similar designs by showing that OOO performance can be mostly achieved with static schedules, given the speculation support needed to permit those schedules. One such design is the in-order Continual Flow Pipeline (iCFP) [12], which targets long-latency operations like cache misses that block in-order cores. Miss-dependent instructions drain into a slice buffer, including any "side" inputs, to allow independent younger instructions to execute out-of-order. Drained instructions are re-executed from the slice buffer once the miss returns. In contrast, we steer instructions to OOO/INO up front (one-time execution). To enable correct out-of-order execution on iCFP, speculation is handled via checkpointing, which may be undesirable for SMT where the aggregate architectural state of all threads is much larger. Our mechanism does not require checkpoints. Several other latency-tolerant designs [31], [11], [32], [13] similarly rely on potentially expensive checkpoints; none of these designs leverage in-order hardware.

The shelf effectively reduces instruction occupancy in

OOO structures. Several related works target similar goals without leveraging in-sequence instructions. Whereas there are many ways to reduce pressure on OOO structures, we note here those mechanisms most closely relate to our contributions. Sembrant *et al.* [33] park instructions that are predicted to be non-critical to memory-level parallelism (MLP) prior to renaming, temporarily reducing pressure on the IQ and PRF until those instructions are resumed. Elmoursy and Albonesi [34] reduce pressure on the IQ via predictive SMT fetch policies. Gonzalez *et al.* [35] reduce pressure on the PRF by decoupling tags (virtual registers) from PRIs (physical registers). Some works leverage checkpointing to release OOO resources early [36], [37]. Adaptive cores additionally provide the ability to disable unused structure entries [38], [39], [40].

Clustered microarchitectures divide the monolithic IQ structure across functional unit clusters to improve cycle time and scalability. Palacharla, Jouppi and Smith [41] advocate using FIFO queues in this manner to reduce complexity. Prior work focuses on steering instructions to clusters so as to minimize inter-cluster forwarding penalties and for load balancing [15], [42], [43]. Similar to our practical steering algorithm, these designs make use of dependence chain information for steering. While we do not cluster our execution units in this paper, it is a possible dimension for the shelf and the IQ to belong to different clusters.

Several works examine heterogeneous cores [3] and datapaths [4]. These works fix a set of heterogeneous hardware resources, e.g., an OOO and an INO core, and attempt to schedule threads among them. Note that threads do not simultaneously use two heterogeneous components, but rather switch from one to the other. Scheduling schemes have targeted specific ILP/MLP regions [44], serializing bottlenecks in parallel code [45], and other indicators [46]. A number of these works advocate fine-grained switching at hundred- or thousand-instruction granularity [47], [4] but still fall short of interleaving in-sequence and reordered instructions in the same window. Our shelf and IQ datapaths can be seen as statically provisioned heterogeneous backends, however, a single-thread context is able to utilize both simultaneously, which is a central contribution of our microarchitecture.

VII. Conclusion

Whereas OOO execution can improve performance for moderately threaded SMT designs, the resulting hardware utilization is inefficient, as many instructions are scheduled in-sequence. We have described a new microarchitecture that augments an OOO core with an energy-efficient in-order scheduling mechanism, the shelf, allowing in-sequence instructions to interleave correctly at fine granularity with reordered instructions. Adding a shelf to a baseline 4-thread core with a 64-entry ROB improves performance by 11.5% (up to 19.2% at best) and energy delay by 10.9% (up to 17.5% at best).

Acknowledgment

The authors thank Scott Mahlke, Ronald Dreslinski, Reetuparna Das, Shruti Padmanabha, Andrew Lukefahr, and the anonymous reviewers for their feedback. This work was partially supported by grants from ARM, Ltd.

References

[1] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. 22nd Int'l Symp. on Computer Architecture*, Jun 1995.

[2] S. Hily and A. Seznec, "Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading," in *Proc. 5th Int'l Symp. on High-Performance Computer Architecture*, Jan 1999.

[3] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. 36th Int'l Symp. on Microarchitecture*, Dec 2003.

[4] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Proc. 45th Int'l Symp. on Microarchitecture*, Dec 2012.

[5] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman, "The metaflow architecture," *IEEE Micro*, vol. 11, no. 3, June 1991.

[6] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, May 2008.

[7] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proc. 12th Int'l Symp. on Computer Architecture*, Jun 1985.

[8] I. Park, C. L. Ooi, and T. N. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proc. 36th Int'l Symp. on Microarchitecture*, Dec 2003.

[9] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proc. 25th Int'l Symp. on Computer Architecture*, Jun 1998.

[10] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proc. 20th Int'l Conf. on Parallel Processing*, Aug 1991.

[11] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *Proc. 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct 2004.

[12] A. Hilton, S. Nagarakatte, and A. Roth, "icfp: Tolerating all-level cache misses in in-order processors," in *Proc. 15th Int'l Symp. on High Performance Computer Architecture*, Feb 2009.

[13] A. Hilton and A. Roth, "Bolt: Energy-efficient out-of-order latency-tolerant execution," in *Proc. 16th Int'l Symp. on High Performance Computer Architecture*, Jan 2010.

[14] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *Proc. 28th Int'l Symp. on Computer Architecture*, May 2001.

[15] P. Salverda and C. Zilles, "A criticality analysis of clustering in superscalar processors," in *Proc. 38th Int'l Symp. on Microarchitecture*, Nov 2005.

[16] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proc. 23rd Int'l Symp. on Computer Architecture*, May 1996.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug 2011.

[18] R. Velasquez, P. Michaud, and A. Seznec, "Selecting benchmark combinations for the evaluation of multicore throughput," in *Proc. Int'l Symp. on Performance Analysis of Systems and Software*, Apr 2013.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[20] D. Koufaty and D. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, March 2003.

[21] 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 *Proc. 42nd Int'l Symp. on Microarchitecture*, Dec 2009.

[22] S. Likun Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in mcpat and potential impacts on architectural studies," in *Proc. 21st Int'l Symp. on High Performance Computer Architecture*, Feb 2015.

[23] K. Khubaib, M. Suleman, M. Hashemi, C. Wilkerson, and Y. Patt, "Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp," in *Proc. 45th Int'l Symp. on Microarchitecture*, Dec 2012.

[24] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, "Composable lightweight processors," in *Proc. 40th Int'l Symp. on Microarchitecture*, Dec 2007.

[25] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *Proc. 34th Int'l Symp. on Computer Architecture*, Jun 2007.

[26] H. Vandierendonck, P. Manet, T. Delavallee, I. Loiselle, and J.-D. Legat, "By-passing the out-of-order execution pipeline to increase energy-efficiency," in *Proc. 4th Int'l Conf. on Computing Frontiers*, May 2007.

[27] R. Shioya, M. Goshima, and H. Ando, "A front-end execution architecture for high energy efficiency," in *Proc. 47th Int'l Symp. on Microarchitecture*, Dec 2014.

[28] J. S. Seng, E. S. Tune, and D. M. Tullsen, "Reducing power with dynamic critical path information," in *Proc. 34th Int'l Symp. on Microarchitecture*, Dec 2001.

[29] F. Tseng and Y. Patt, "Achieving out-of-order performance with almost in-order complexity," in *Proc. 35th Int'l Symp. on Computer Architecture*, Jun 2008.

[30] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?" in *Proc. 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar 2013.

[31] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez, "Toward kilo-instruction processors," *ACM Trans. Architecture and Code Optimiztion*, vol. 1, no. 4, Dec 2004.

[32] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance sparc cmt processor," *IEEE Micro*, vol. 29, no. 2, March 2009.

[33] 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 *Proc. 48th Int'l Symp. on Microarchitecture*, Dec 2015.

[34] A. El-Moursy and D. Albonesi, "Front-end policies for improved issue efficiency in smt processors," in *Proc. 9th Int'l Symp. on High-Performance Computer Architecture*, Feb 2003.

[35] A. Gonzalez, J. Gonzalez, and M. Valero, "Virtual-physical registers," in *Proc. 4th Int'l Symp. on High Performance Computer Architecture*, Feb 1998.

[36] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed early resource recycling in out-of-order microprocessors," in *Proc. 35th Int'l Symp. on Microarchitecture*, Dec 2002.

[37] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-order commit processors," in *Proc. 10th Int'l Symp. on High-Performance Computer Architecture*, Feb 2004.

[38] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster, "Dynamically tuning processor resources with adaptive processing," *IEEE Computer*, vol. 36, no. 12, Dec 2003.

[39] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources," in *Proc. 34th Int'l Symp. on Microarchitecture*, Dec 2001.

[40] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook, "A circuit level implementation of an adaptive issue queue for power-aware microprocessors," in *Proc. 11th Great Lakes Symp. on VLSI*, Mar 2001.

[41] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proc. 24th Int'l Symp. on Computer Architecture*, May 1997.

[42] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Dynamically managing the communication-parallelism trade-off in future clustered processors," in *Proc. 30th Int'l Symp. on Computer Architecture*, Jun 2003.

[43] A. Baniasadi and A. Moshovos, "Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors," in *Proc. 33rd Int'l Symp. on Microarchitecture*, Dec 2000.

[44] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Proc. 39th Int'l Symp. on Computer Architecture*, Jun 2012.

[45] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *Proc. 17th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar 2012.

[46] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. 5th European Conf. on Computer Systems*, Apr 2010.

[47] H. Najaf-abadi and E. Rotenberg, "Architectural contesting," in *Proc. 15th Int'l Symp. on High Performance Computer Architecture*, Feb 2009.