Efficient Instruction Scheduling Using Real-time Load Delay Tracking

ANDREAS DIAVASTOS, Universitat Politècnica de Catalunya, Barcelona, Spain
TREVOR E. CARLSON, National University of Singapore, Singapore

Issue time prediction processors use dataflow dependencies and predefined instruction latencies to predict issue times of repeated instructions. In this work, we make two key observations: (1) memory accesses often take additional time to arrive than the static, predefined access latency that is used to describe these systems. This is due to contention in the memory hierarchy and variability in DRAM access times, and (2) we find that these memory access delays often repeat across iterations of the same code. We propose a new processor microarchitecture that replaces a complex reservation-station-based scheduler with an efficient, scalable alternative. Our scheduling technique tracks real-time delays of loads to accurately predict instruction issue times and uses a reordering mechanism to prioritize instructions based on that prediction. To accomplish this in an energy-efficient manner we introduce (1) an *instruction delay learning mechanism* that monitors repeated load instructions and learns their latest delay, (2) an *issue time predictor* that uses learned delays and dataflow dependencies to predict instruction issue times, and (3) *priority queues* that reorder instructions based on their issue time prediction. Our processor achieves 86.2% of the performance of a traditional out-of-order processor, higher than previous efficient scheduler proposals, while consuming 30% less power.

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

Additional Key Words and Phrases: Instruction scheduling, processor architecture, load instruction delay scheduling, issue time prediction, instruction reordering, microarchitecture

ACM Reference format:
Andreas Diavastos and Trevor E. Carlson. 2022. Efficient Instruction Scheduling Using Real-time Load Delay Tracking. *ACM Trans. Comput. Syst.* 40, 1–4, Article 1 (November 2022), 21 pages.
https://doi.org/10.1145/3548681

1 INTRODUCTION

With each processor generation, architects aim to improve core performance while maintaining energy efficiency. To achieve high levels of performance, a processor must be able to build aggressive schedules that exploit **instruction-level parallelism (ILP)** and memory-level parallelism. One of the main challenges in this process is reordering instructions in a scalable, energy efficient manner. Traditional out-of-order processors schedule ready instructions using complex schedulers that dynamically build dataflow dependencies and implicitly learn instruction delays. They achieve this by monitoring, waking up, and issuing instructions once their operands are produced. However,

Andreas Diavastos was at the National University of Singapore when this work was done.
Authors' addresses: A. Diavastos, Jordi Girona 1-3, Mòdul D6 08034 Universitat Politècnica de Catalunya, Barcelona, Spain; email: andreas.diavastos@upc.edu; T. E. Carlson, National University of Singapore, School of Computing, 13 Computing Drive, Singapore, 117417; email: tcarlson@comp.nus.edu.sg.

0734-2071/2022/11-ART1 $15.00
https://doi.org/10.1145/3548681

as previous studies have shown [29, 30], this technique uses power hungry hardware structures that inefficiently scale processor performance.

To build efficient, scalable hardware that provides both high performance and energy efficiency, previous research has proposed a number of techniques covering both in-order and out-of-order processors. Some examples include parking non-ready instructions to better utilize available resources [20, 21, 34], bypassing stalled instructions or filtering instructions based on their criticality to reduce stalling delays [1, 2, 6, 19], replaying stalled instructions to avoid blocking the instruction queue [12], and instruction prescheduling using dataflow dependencies [26, 31]. Some solutions make the realization that the schedules of general-purpose applications are highly regular and repeat during execution; these propose issue time prediction processors that try to explicitly predict when instructions will be ready to issue, using dataflow dependencies and pre-defined instruction delays [4, 5, 15, 17, 24, 33, 37, 42, 43]. But, unfortunately, without explicit knowledge of real-time instruction delays, the issue time predictions will never be accurate enough to achieve close-to out-of-order core performance in an efficient way. However, some works [29, 30] propose hybrid processors where an out-of-order core produces repeated instruction schedules, taking into account true memory access latency, and offloads them to simple in-order cores. However, these solutions require the implementation of two cores, which increases design cost.

In this work, we aim to overcome the limitation of issue time prediction processors by dynamically building the knowledge of real-time instruction delays with low-cost hardware. In addition, we introduce an instruction reordering technique that uses this knowledge to prioritize instructions based on dataflow and timing information in a highly efficient way. We achieve high performance (and in some cases, outperform cores with expensive on-demand issue structures used by traditional out-of-order cores), with a light-weight structure that understands program dependencies and timing information to prioritize key instructions when necessary. We do this with a delay-based scheduling mechanism that uses latency information as seen by the core itself, instead of pre-defined values that have been used in all previous works up to now.

In this article, we propose a processor microarchitecture that dynamically prioritizes the issuing of instructions, just in time for execution, by recording real-time delays of repeated loads (i.e., in loops) and learning dataflow dependencies of instructions to accurately predict issue times of the same instructions in future appearances. It improves energy efficiency by replacing reservation-station based instruction queues with priority queues that reorder instructions using the predicted issue time as their ordering policy and reduces complexity by issuing only from the head of queues.

In this work, we make the following contributions:

- An efficient issue time prediction processor with prioritization hardware that enables instruction reordering to achieve 86.2% of the performance of the upper-bound (a traditional out-of-order baseline), while consuming 30% less power (Section 4);
- An issue time prediction algorithm that uses real-time load delays to enable accurate prediction of issue times for repeated instructions. A prediction that facilitates the prioritization of key instructions to fill the gaps between stalled instructions, improving the performance of issue time prediction processors by 5.5–25% on average (Section 3);
- A comprehensive evaluation of the proposed microarchitecture with quantitative comparison to state-of-the-art issue time prediction processors (Sections 5 and 6).

2 MOTIVATION AND OVERVIEW

One of the key reasons why out-of-order processors are able to achieve high performance is because of the aggressive scheduling of instructions. Past research suggests that out-of-order

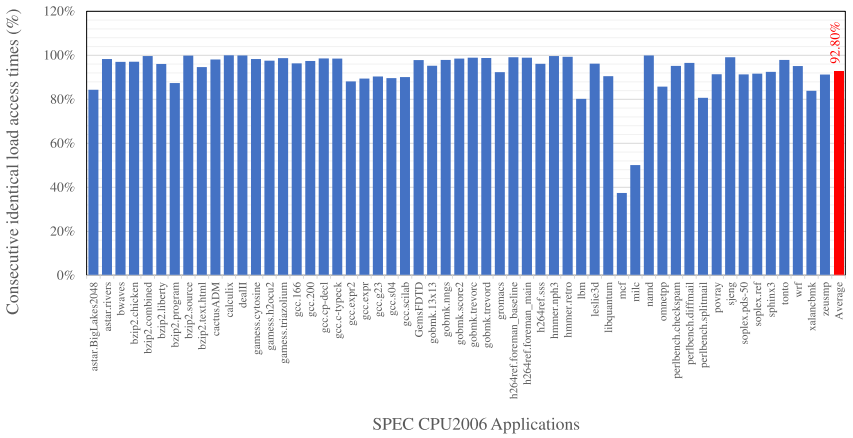


Fig. 1. Percentage of repeated loads with identical access time in consecutive iterations. On average more than 92% of loads in the SPEC CPU2006 benchmark applications have the same access time in consecutive iterations.

schedules are repeated [25] across loop iterations and can be learned [29, 30] or predicted by assuming a pre-defined delay, as specified in the specifications for different types of instructions [4, 5, 26, 31, 43]. But although many instructions execute with static delays, like traditional addition and multiplication, load instructions that miss in the L1 can have variable latency, depending on the level of the memory hierarchy they access.

Our study shows that even for accesses to the same level of the memory hierarchy, different load instructions can have different delays due to bandwidth contention in the memory hierarchy and the variability in DRAM access times. More specifically, we see variations across all PCs of as many as four cycles for L2 accesses and more than 2× the number of cycles for DRAM accesses compared to the specifications defined by the implementation. Therefore, assuming a single minimum predefined delay for memory accesses is not sufficient to accurately predict instructions issue times. However, Figure 1 shows that memory access times of loads in different appearances are repeated over consecutive iterations, on average 92.8% of the time.

The key insight of this article is that *to accurately predict repeated instruction issue times and build a high performance schedule, learning the latest delay of memory accesses is required.* In this work, we build schedules and reorder instructions in an energy efficient way using three core components that can replace the traditional out-of-order scheduler: (1) an instruction delay learning mechanism that tracks delays of load instructions over repeated appearances, (2) an issue time predictor that dynamically predicts when an instruction will be ready to issue, and (3) priority queue reordering that use these predictions to prioritize key instructions, even after dispatch has occurred.

3 ISSUE TIME PREDICTION

To achieve aggressive, high-performing schedules, we need (1) to schedule instructions in a dataflow manner that satisfies producer–consumer relationships and (2) to reorder instructions such that idle cycles between dependent instructions are filled with independent work. Dataflow dependencies provide a scheduling policy where the execution of instructions follows the flow of the data from producers to consumers. The processor dynamically derives dataflow dependencies from the input and output operands of instructions. Because instructions require a certain amount of time to produce their output data, gaps of idle cycles are formed between dependent

instructions. To achieve a high-performing schedule, these gaps must be filled with independent instructions that are ready to execute. In this work, we identify these gaps by learning real-time delays of load instructions that miss in the L1 cache. All other operations have static delays; therefore, learning is not required. Assuming instructions in repeated code (e.g., loops) appear more than once, we combine instruction delays with their dependencies to predict their issue time in future appearances.

3.1 Prediction Algorithm

The predicted issue time of a consuming instruction ($T_{Predicted}$(c)) is estimated as the maximum value of the addition of the predicted issue time ($T_{Predicted}$(p)) and the delay ($T_{Delay}(p)$) of each of its producers (Equation (2)). The delay ($T_{Delay}(p)$) of each producer p is calculated as the difference of its completion time ($T_{Complete}$(p)) and its issue time (T_{Issue}(p)) (Equation (1)). An instruction can be issued only after all its producers have completed, and therefore the algorithm chooses the maximum value. By using the predicted issue time of the producers, the algorithm inherently propagates dataflow dependency chain delays to all instructions, and thus the resulting predicted issue time can directly be used to order instructions.

An instruction's delay is calculated as

$$T_{Delay}(p) = T_{Complete}(p) - T_{Issue}(p). \tag{1}$$

An instruction's predicted issue time is estimated as

$$T_{Predicted}(c) = \max_{p=0}^{P}[T_{Predicted}(p) + T_{Delay}(p)]. \tag{2}$$

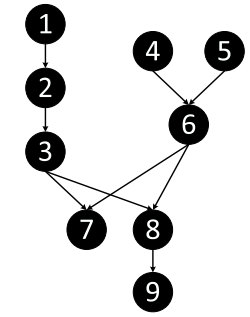
The proposed technique requires the core to observe and store delay information ($T_{Delay}(p)$) of repeated instructions (remember that storing delays is only required for load instructions that miss in the L1 cache as all other instructions have static delays). In the absence of this information (first appearance of an instruction or non-repeated instructions), the algorithm assumes the lowest delay to avoid unnecessary stalls in the execution. Because load instructions access different levels of the memory hierarchy in different iterations in an unpredictable way, constant monitoring and retraining of the predictor is required to keep the delay information up to date. Therefore, stored load delays are updated every iteration.

3.2 Example

To demonstrate how the issue time prediction algorithm works, we annotate a code region, see Table 1, that illustrates two loop iterations of a code snippet of the `hmmer` application from the SPEC CPU2006 benchmark suite. Although this example demonstrates reordering within one basic block, in normal execution, there are no restrictions in reordering instructions between different blocks. Also, for simplicity the example uses instructions; however, the actual implementation uses uops.

For the purpose of this example, we assume a simple in-order core with one issue per cycle, loads/stores take four cycles to execute, and all other instructions execute in one cycle. T_{Issue} corresponds to the issue cycle, T_{Delay} is the instruction's delay in the current iteration, and $T_{Predicted}$ is the predicted issue time for its next appearance in relation to producers. Δ_{this} and Δ_{ooo} are the number of cycles an instruction was issued earlier, compared to a traditional in-order execution, for the proposed solution and a fully out-of-order core, respectively. Note that *now* is a relative time and is different for each instruction. It merely means that an instruction is ready for execution immediately after it is dispatched.

Table 1. Issue Time Prediction Example Code of the hmmer Application

	Repeated Instructions	Prediction Algorithm T_{Issue}	T_{Delay}	$T_{Predicted}$	Δ_{this}	Δ_{ooo}	Dataflow Diagram
Iteration 1	❶ mov (r10,rax,4),ecx	2	4	*now*	0	0	
	❷ add 0x0(r13,rax,4),ecx	6	1	T^1_{pred}+4	0	0	
	❸ mov ecx, 0x4(rdx)	7	4	T^2_{pred}+1	0	0	
	❹ mov 0x18(rsp), rbx	8	4	*now*′	0	-5	
	❺ mov (r9,rax,4), r15d	9	4	*now*″	0	-5	
	❻ add (rbx,rax,4), r15d	13	1	T^5_{pred}+4	0	-5	
	❼ cmp ecx, r15d	14	1	T^6_{pred}+1	0	-3	
	❽ cmovge r15d, ecx	15	1	T^7_{pred}+1	0	-3	
	❾ mov ecx, 0x4(rdx)	16	4	T^8_{pred}+1	0	-3	
	...						
Iteration 2	❶ mov (r10,rax,4),ecx	20	4	*now*	0	0	
	❹ mov 0x18(rsp), rbx	21	4	*now*′	-5	-5	
	❺ mov (r9,rax,4), r15d	22	4	*now*″	-5	-5	
	❷ add 0x0(r13,rax,4),ecx	24	1	T^1_{pred}+4	0	0	
	❸ mov ecx, 0x4(rdx)	25	4	T^2_{pred}+1	0	0	
	❻ add (rbx,rax,4), r15d	26	1	T^5_{pred}+4	-5	-5	
	❼ cmp ecx, r15d	29	1	T^6_{pred}+1	-3	-3	
	❽ cmovge r15d, ecx	30	1	T^7_{pred}+1	-3	-3	
	❾ mov ecx, 0x4(rdx)	31	4	T^8_{pred}+1	-3	-3	
	...						

We assume that instructions are already in the instruction window. Δ is the number of cycles an instruction was issued earlier in the proposed solution and an out-of-order core compared to a traditional in-order core. Rows marked in green show the reordered instructions and in blue instructions that were issued earlier than their previous appearance. While the example uses instructions, the actual implementation uses uops. In the right column, we show the instruction dataflow diagram for one iteration.

As shown in the dataflow diagram in Table 1, load instruction ❶ produces a result for instruction ❷. Instruction ❸ is a store that depends on ❷, while ❹ and ❺ are loads producing the operands for instruction ❻. Instructions ❸ and ❻ are producers of instructions ❼ and ❽, while store instruction ❾ is a consumer of instruction ❽. Based on these dependencies, the loop consists of two major dependency chains: ❶ → ❷ → ❸ and ❹, ❺ → ❻. These chains are independent of one another, and therefore instructions can be reordered between these chains as needed. Instruction ❷ must wait for four cycles before it can issue because of its dependence on load instruction ❶. An in-order core will stall for four cycles between the two instructions. But an out-of-order core will fill these idle cycles by issuing instructions ❹ and ❺ earlier. To emulate this, we keep track of timing information for the relevant instructions. During the first iteration, we collect the issue cycle (T_{Issue}) and the delay (T_{Delay}) of every instruction and associate them with the dataflow dependencies to predict the issue time ($T_{Predicted}$) of the same instructions in future appearances. In the second iteration, instructions ❹, ❺ (marked in green) bypass independent instructions that have a higher predicted issue time.

Observing the Δs in the second iteration allows us to see the benefit of this technique. After one iteration, the prediction algorithm builds a schedule that is the same as the schedule of the out-of-order core as shown by the matching deltas (Δ_{this} and Δ_{ooo}). The Δ_{this} of instructions ❹ and ❺ is −5, because they are issued five cycles earlier compared to execution on an in-order core. Instructions that are part of the same dependency chain will also benefit and will also be able to issue earlier (instructions ❻, ❼, ❽, and ❾ in the example (marked in blue)). The Δ_{ooo} is the same for both iterations, because the out-of-order core can reorder instructions in every iteration.

The issue time predictor requires just one iteration to learn real-time load instruction delays before applying them in the prioritization algorithm that will reorder instructions accordingly. However, in our implementation, instructions are also reordered in the first iteration by assuming L1 hit access time for all load instructions to avoid unnecessary stalls.

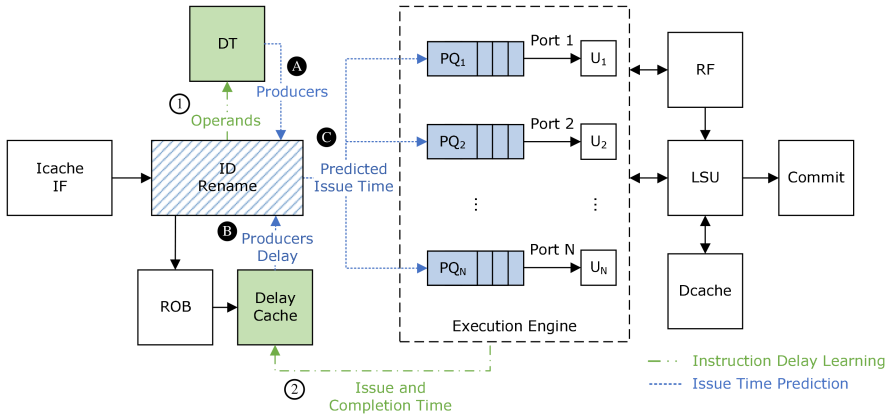


Fig. 2. The proposed microarchitecture design. In green dashed lines we mark the Instruction Delay Learning process and in blue dotted lines the Issue Time Prediction process.

4 PROPOSED MICROARCHITECTURE

By tracking real-time load delays and instruction dependencies, we can more accurately predict instruction issue times and build aggressive schedules that mimic those of an out-of-order core, as shown in the example in Section 3.2. Using priority-based ordering hardware and the issue times predicted as the priority index it can efficiently reorder instructions. Figure 2 shows the schematic representation of the proposed architecture. Green colored components are the structures added to implement the Instruction Delay Learning process, while blue colored structures implement the Issue Time Prediction and Priority Queue Reordering in the execution unit.

In step ① of the Instruction Delay Learning, process instruction dependencies are stored in the **Dependency Table (DT)**, that contains an entry for each physical register, and maps it to the instruction pointer that last wrote to this register. In step ②, the issue and completion time of load instructions that miss in the L1 cache are stored per PC in a direct-mapped memory structure called DelayCache. For every dispatched instruction the Issue Time Prediction algorithm identifies its producers from the DT in step Ⓐ and their delays from the DelayCache in step Ⓑ, to calculate the Predicted Issue Time in step Ⓒ that will be used by the **Priority Queues (PQs)** as a priority index to reorder instructions in the execution engine.

4.1 Instruction Delay Learning

While instructions are being fetched, decoded, and renamed, dependencies are stored and built by the DT. As instructions start executing, delays of instructions that caused upcoming instructions to stall (L1 cache miss) are stored in the DelayCache, initiating the training of the prediction mechanism. In this work, the delay represents the execution time of an instruction with respect to its issue time. An alternative approach not used in our final design stores the delay of an instruction with respect to its dispatch time, but our study shows large slowdowns in such design due to the unpredictability of structural hazards (see Figure 8(a)).

Although most instructions require a static delay before delivering their result, it is loads that cause the majority of the stalls, and their delay can be variable and unknown at dispatch time. In this implementation, we only store delays of loads that miss in the L1 and for all other instructions we use their predefined delay (derived from their type or L1 access time for loads that do not miss). This allows us to minimize the storage overhead and power requirements when implementing the DelayCache.

Due to application characteristics that relate to branch behavior and memory access patterns, load delays are unpredictable in different iterations (see Section 6.5/Figure 9(a)). Therefore, the DelayCache is continuously updated with the latest delay for every stored instruction, and the issue time predictor is trained in every iteration of a repeated code. Our experiments show that, for the majority of the applications tested, training every iteration produces the highest performance. Activity-based power analysis shows that training is not expensive, as only a subset of load instructions have a variable delay that require an update to the DelayCache. Note that in a context-switch scenario flushing the DT and the DelayCache is not required as both units use a least recently used replacement policy that will replace older application's instruction delays with the new application's delays. In the case when not all entries are used by the new application, the data of the older application will remain there when it the core switches back to that application.

Although our implementation trains the issue time predictor using load delay information, the issue prediction mechanism can be applied as-is to other instructions with variable delay, such as floating point division and transcendental functions. In this work, we do not cover their potential performance benefits, as they do not occur often in the applications we evaluate.

4.2 Issue Time Prediction and Dispatch

The delays of instructions stored in the DelayCache combined with the dependencies from the DT provide the necessary inputs for predicting the issue time of instructions as described in Section 3. For every renamed instruction, the DT is queried using the instruction's input operands to find possible producers. In a DT hit, the DelayCache is queried with the producer's addresses, and correspondingly, in a DelayCache hit, the delay will be retrieved to calculate the current instruction's issue time. In case of a miss in the DelayCache, the value of an L1 hit (four cycles in our microarchitecture) is used to avoid unnecessary delays in the absence of misses.

The Execution Engine, which has the primary task of reordering instructions, is built using multiple priority instruction queues, with each functional unit having its own dedicated queue. Although instructions from multiple queues can execute out-of-order, instructions in a single queue can be issued only from the head of the queue and only to the corresponding functional unit. Because each queue corresponds to a specific functional unit, instructions are dispatched to the queues according to their type. If an instruction matches to more than one queue, then dataflow dependencies are used to steer incoming instructions to the first queue that has a producing instruction at its tail; otherwise, it will go to the queue with the least number of instructions. Our studies indicate that round-robin and global dependence steering schemes reduce performance compared to our scheduling methodology (see Figure 10 for more details).

Issue times are predicted for first time appearing or non-repeated instructions even in the absence of delay information by assuming the lowest delay (L1 access hit) to avoid unnecessary execution stalls.

4.3 Priority Queue Reordering and Issue

In the proposed architecture, we remove the traditional reservation-station-based scheduler and instead reorder instructions using light-weight and efficient priority instruction queues in the execution engine. PQ are built using Systolic Priority Queues [22], where instructions are reordered based on a priority index (their predicted issue time in this case). Insertion and removal in a priority queue happen at the head as described in Reference [22], and thus highest-priority inserted instructions are directly available from the head of a PQ on the next cycle; therefore back-to-back instruction execution is achieved. A free list of entries in the queues is also used so that new entries can be inserted at a free position. Because each functional unit has its own instruction queue and only instructions at the head of each queue can be issued, complex selection logic is not required

to decide which instructions to issue every cycle. When an instruction at the head of a queue has unresolved data dependencies the queue blocks. However, instructions in other queues are not affected as only instructions in a blocked queue will stall.

4.4 Register Renaming

Register renaming works in the same way as in traditional out-of-order processors. Renaming replaces destination architectural registers with physical registers to eliminate the name dependencies (output dependencies and anti-dependencies) between instructions, and it automatically recognizes true dependencies. True data dependencies between instructions allow for a more flexible execution of instructions. Maintaining the status for each register, indicating whether or not it has been computed yet, allows the execution of instructions to be performed out-of-order when there are no true data dependencies.

4.5 Memory Dependencies

Memory operations are also reordered to maximize performance. Contrary to register dependencies that can be resolved at decode time, store-to-load memory dependencies with overlapping memory addresses can lead to incorrect execution if loads or stores are executed before older stores that refer to the same address. Memory dependencies are accurately predicted by identifying the stores upon which a load depends (store set) and communicate that information to the issue time predictor [9]. Similarly to a traditional out-of-order processor, using the ROB and the LSU prevents memory violations. The LSU tracks executing memory operations and makes sure that they are committed in program order. Instructions are verified before commit to ensure that no memory violations will be visible to the architecture state.

4.6 Commit

The commit stage checks for exceptions before it releases structures such as store buffer entries and rename registers. Instructions enter in-order into the ROB during dispatch, record their completion out-of-order, and leave the ROB in-order. Interrupts and branch misspeculation events are handled as in other conventional processors. However, retraining of the issue time predictor is not required in this case, and if the core matches a repeated instruction from the DelayCache, they it will be reordered immediately.

4.7 Multi-Core Support

In a multi-core implementation, new connections are added in the memory hierarchy for loads accessing remote memory locations. Issue time prediction in the proposed design is based on a per core memory access latency at any part of the memory hierarchy; therefore, by replicating the new hardware structures (DT and DelayCache) to every core the prediction algorithm will adapt accordingly and learn remote access delays. A two-core setup simulation, with a variety of application pairings, resulted in similar gains as to the ones reported for the single-core scenario in Section 6.1. Specifically, the average performance difference of the multi-core implementation to an out-of-order multi-core baseline is within 0.6% of the the single-core processors performance difference. As the coherence misses or **Simultaneous Multi-Threading (SMT)** scheduling could be less predictable, it would require new studies and, potentially, structure changes to handle these cases. However, this is out of the scope of this work. This core, as implemented, does not change any significant components in the back-end of the processor and, therefore, is compatible with the original coherence and consistency models as described in the core.

Table 2. Power and Area of the New Design Structures

Component	Organization	Ports	Area (μm^2)	Power (mW)
DT	256 entries × 1B	12r4w	14.54 (0.37%)	11.74 (0.37%)
DelayCache	512 entries × 12B	4r1w	103.31 (3.25%)	43.41 (1.87%)
Priority Queues	5 × 2 × 13 entries × 1B	1r1w	0.28 (0.01%)	1.49 (0.05%)

In parenthesis is their overhead over the entire core. The Priority Queues are implemented using 2 × 13 entries per unit to match the 64 entries of the out-of-order baseline.

Table 3. Simulated Microarchitecture Parameters

Component	**Parameters**		
	in-order	This Work	out-of-order
Core	2 GHz, superscalar		
Issue width	4-way	4-way	4-way
Reorder logic	none	128-entry ROB 5×13-entry PQs	128-entry ROB, 64-entry RS
DT	—	256 entries (×1B)	—
DelayCache	—	512 entries (×12B)	—
Branch Predictor	TAGE-SC-L [35]		
Branch Penalty	six cycles	eight cycles	eight cycles
Execution units	2 int, 1 fp, 1 branch, 1 load/store		
L1-I Cache	32 KB, 4-way LRU		
L1-D Cache	32 KB, 8-way, LRU, 4 cycle, 8 outstanding		
L2 cache	512 KB, 8-way, LRU, 8 cycle, 12 outstanding		
Prefetcher	L1, stride-based, 16 independent streams		
Main memory	DDR3-1600, 800 MHz, ranks: 4, banks: 8, page size: 4 KB, bus: 64 bits, `tRP-tCL-tRCD`: 11-11-11		
Technology node	28nm		

5 EXPERIMENTAL SETUP

The performance evaluation of this work was performed on a modified version of the Sniper Multi-Core Simulator [7], version 6.2, that uses the Instruction Window-Centric core model [8]. We use a detailed DRAM model that takes into account DRAM page locality and other low-level details that account for all detailed DRAM delays. Power and energy analysis was conducted with McPAT [23] version 1.3, modified to support our microarchitecture. Applications were compiled with the GCC compiler (-O2 optimization flag) and executed with the reference inputs of the SPEC CPU2006 benchmarks, using a single, representative (SimPoint-based [36]), 750 million instruction trace. Average results are computed by combining output results of common workloads (but different input) into a weighted value before averaging the results across applications. The details of added structures to the core, with area and average power consumption, are listed in Table 2 and the details of the simulated microarchitectures are listed in Table 3. Performance is measured in Instructions per Cycle and energy efficiency in **Million Instructions Per Second per Watt (MIPS/W)** and **Energy Delay Product (EDP)**. Unless explicitly stated, all summary results are weighted average values of all applications, while black bars represent results of the proposed design configuration described in Table 3.

6 RESULTS AND ANALYSIS

6.1 Performance Analysis

The proposed processor achieves 2.7× and 86.2% of the performance of the baseline in-order and out-of-order cores, respectively (Figure 3). Note that simulations using representative regions

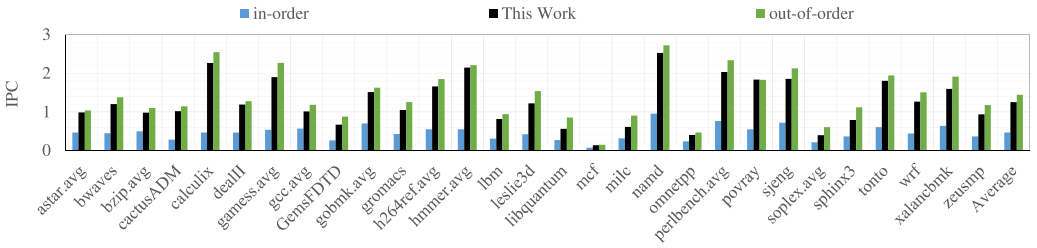


Fig. 3. Performance of the proposed implementation compared to in-order and out-of-order baseline processors. For clarity, we plot average values for applications with multiple inputs (*<application_name>.avg*).

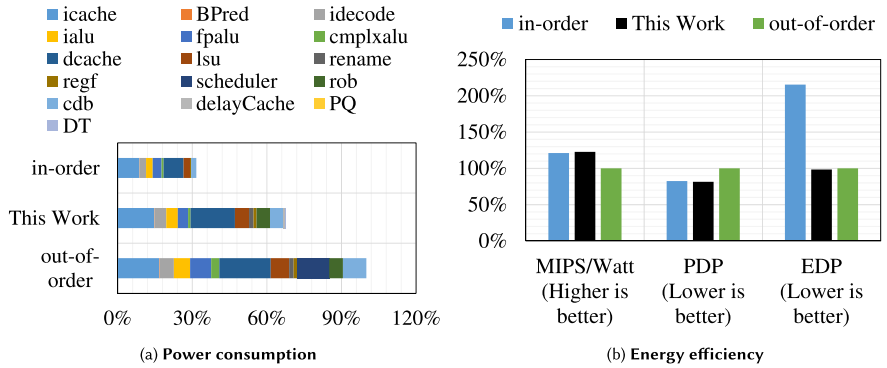
(from Reference [32]) from a set of SPEC CPU 2017 applications were also conducted with a similar performance compared to an out-or-order core (we achieve on average 85.9% of the performance of the baseline out-of-order core). Although instructions issue only from the head of the instruction queues, it achieves near-out-of-order performance by de-prioritizing instructions that were predicted to stall the execution (i.e., consumers of loads that do not hit in the L1 cache). This allows ready instructions to move to the head of the instruction queues. Note that when an instruction at the head is not ready to issue, the queue will block.

Per instruction analysis shows that loads and their address generating instructions are issued earlier in the new design, compared to the out-of-order baseline. This happens because address generating instructions rarely depend on long-latency operations [6], and, therefore, the new processor predicts shorter issue times for them and their consuming loads, even compared to older instructions that are also ready to issue. In an age-based ordering scheduler of an out-of-order core, however, ready instructions are issued based on their fetched order. Therefore, loads that are issued earlier result in shorter data waiting time. This is reflected in applications, like `astar`, `dealII`, and `povray` where this work's performance meets or exceeds the performance of the out-of-order. While, in general, this work performs as well as the out-of-order for compute-intensive applications, there are a few that show lower performance. Applications like `gamess` that are not bound by long-latency memory accesses, stress the multi-queue backend of the new design, where instructions can issue from the head of a queue, to the corresponding functional unit only.

The main reasons the proposed processor is unable to meet the performance of the out-of-order processor are (1) the per functional unit instruction queue design, (2) the prediction algorithm training that requires at least one iteration to learn real-time load delays, and (3) the accuracy of using the previous load delay to predict the next delay. However, the design is a tradeoff made to significantly improve the processor's overall energy efficiency. An alternative single in-order issue queue would severely limit the performance, while adding selection logic over queues that can issue to multiple units or using reservation-station-based queues would greatly increase power consumption (see Figure 8(b)). While the delay prediction training and accuracy is an application dependent overhead that does not have a major impact on overall performance (see Figure 9(a)), even on a larger core. Our analysis shows that on a scaled-up, Skylake-like processor, the additional overhead is only 3%.

6.2 Power and Efficiency Analysis

Figure 4(a) shows power results for the same processors, normalized to the out-of-order baseline. The in-order core consumes 31.6% the power of the out-of-order, while the proposed processor consumes 67.4% its power. One of the main reasons for the power reduction in this work is the removal of the reservation-station-based instruction scheduler that takes 13% of the total power of the out-of-order core (including wake-up and selection logic). The rest of the power gained is



(a) **Power consumption** (b) **Energy efficiency**

Fig. 4. Normalized to the out-of-order: (a) Power consumption and (b) Efficiency (MIPS/W), Power Delay Product (PDP), and EDP.

coming from the difference in runtime compared to the out-of-order. As performance increases, the amount of dynamic power also increases. Dynamic power is data dependent and is closely tied to the number of transistors that change state [27]. The DelayCache, the Priority Queues and the DT contribute only 2% to the total power of the new core. Priority queues are efficiently implemented using simple interconnected FIFO queues, while the small number of delays that need to be stored allows for a small size DelayCache with few accesses, consequently little dynamic power consumed.

Figure 4(b) outlines the energy efficiency normalized to the out-of-order core. Despite its low performance, the simplicity and low-power hardware of the in-order core provide a 21% increase in efficiency over the more complex out-of-order core. The significantly higher performance of this work, in conjunction with the lower total power, achieves an improvement of 22.7% over the out-of-order. On the right side of Figure 4(b), efficiency is outlined as a metric of the Power Delay Product normalized to the out-of-order core. The proposed processor achieves a reduction of 19% and 1% in PDP compared to the out-of-order and in-order, respectively.

6.3 State-of-the-Art Issue Time Predictors

Figure 5 shows performance and energy efficiency results of state-of-the-art issue time predictors implemented on top of our baseline processors and compared to this work. We categorize these processors to those that **eliminate the traditional reservation-station-based scheduler (*without-RS*)** and **those that still use it (*with-RS*)**. To calculate the efficiency for the processors *with-RS* we used the power consumption of the out-of-order core as-is, without adding the overheads of their added structures. We use these results only as a reference and note that in a real implementation, their efficiency would actually be lower.

The *Complexity-Effective* [31] solution steers instructions to in-order queues based on their dependencies alone. Dependent instructions are steered to the same queue, while independent instructions are steered to empty queues. This solution only achieves 61.8% of the out-of-order core performance, because it does not take advantage of the delays between dependent instructions. *Cyclone* [12] uses dependencies to predict instruction issue times and employs a selective replay mechanism for stalled instructions. However, performance is low due to the conflicts arising during instruction flow because of its queue structure and the limitation that only instructions at the head of the queue are candidates for issuing [17]. Using real-time delay information on top of dataflow dependencies to predict instruction issue times in *This Work*, solves these problems and achieves significant performance improvement over other processors *without-RS*.

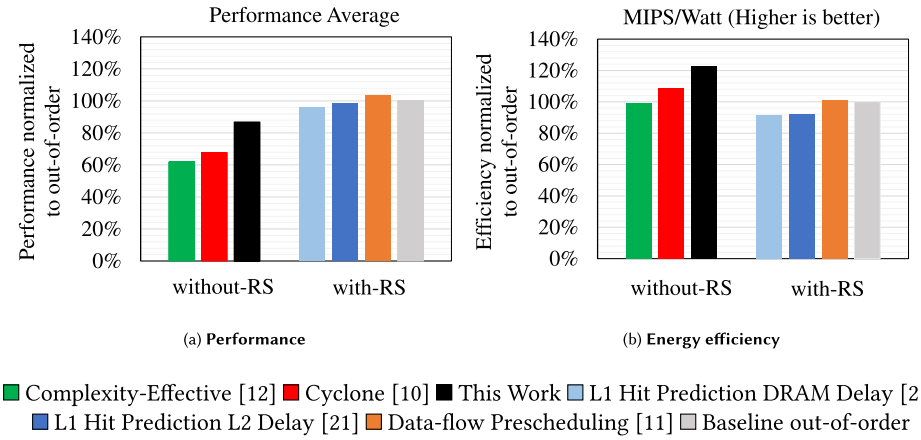


Fig. 5. (a) Performance and (b) energy efficiency of state-of-the-art issue time prediction processors, normalized to the baseline out-of-order processor.

Predicting only L1 hits [43] and assuming L2 (*L1 Hit Prediction L2 delay*) or DRAM (*L1 Hit Prediction DRAM delay*) access delays for all other loads does not improve out-of-order processor performance, because it ignores the actual miss delay that is the key factor for stalling the pipeline. Assuming L2 delay for all misses ignores DRAM accesses and stalls the pipeline for extended periods of time and using DRAM delay makes dependent instructions wait for an unnecessary amount of time, even though they are ready to execute. *Dataflow Prescheduling* [26] reorders instructions before sending them to the instruction window using dataflow dependencies and assuming a L1 cache-hit delay for all loads. This optimistic assumption improves performance over the out-of-order core as it does not delay ready instructions in the issue window. However, as in all solutions *with-RS*, misspredicted instructions will not stall the pipeline, because they will be overlapped using the out-of-order scheduler.

In general, processors *with-RS* produce higher performance (Figure 5(a)), because the reordering is handled by their reservation-station-based instruction queue. However, processors *without-RS* achieve higher energy efficiency (Figure 5(b)) because of the simplicity of their design. These results highlight the importance of using real-time delay information to provide out-of-order performance when predicting instruction issue times, while reordering instructions using priority queues will achieve it in an energy efficient way (as we demonstrate with *This Work*). We note that previous solutions investigated their effectiveness using very large cores. We performed experiments using similar simulation configurations, and this work scales in a similar way.

6.4 State-of-the-Art Issue Time Predictors on the Proposed Hardware

In this section (Figure 6), we implement the same state-of-the-art issue time prediction techniques on-top of our proposed microarchitecture, instead of their original implementation (previous section). With this study we highlight the importance of using real-time delays and their effectiveness in predicting instruction issue time. We categorize these solutions to those that use only dependencies to reorder instructions (Dependence-based) and those that use both dependencies and load delays (Load Delay-based).

Dependence-based solutions achieve low performance, because using only dependencies between instructions does not take into account the idle time between dependent instructions. Load Delay-based solutions outperform Dependence-based solutions, but using a static delay, like *Dataflow Prescheduling* [26], for all types of instructions results in an average performance loss of

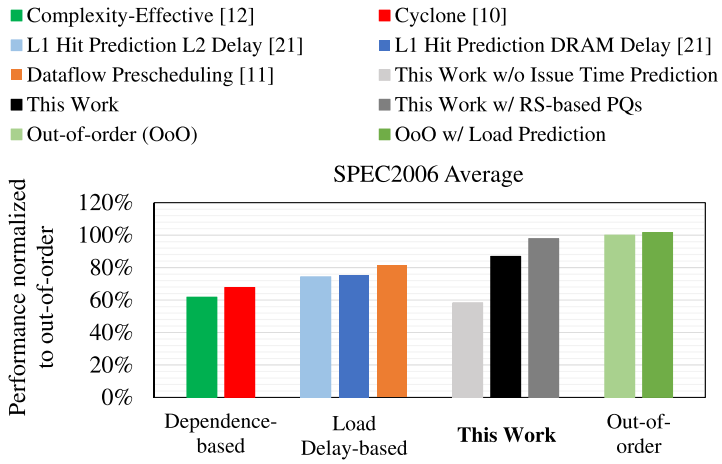


Fig. 6. Performance of state-of-the-art issue time prediction techniques implemented on the proposed microarchitecture, normalized to the out-of-order baseline. Because Complexity-Effective does not use a prediction mechanism, its microarchitecture is implemented precisely as described in Reference [31].

5.5% compared to the *This Work* (detailed analysis shows up to 32% performance loss for memory-intensive applications). Predicting L1 hits [43] and assuming L2 (*L1 Hit Prediction L2 delay*) or DRAM (*L1 Hit Prediction DRAM delay*) delays also incurs significant overhead as they ignore access time to other memory regions and stall instructions at the heads of the in-order multi-queue backend of this design.

Using the *instruction delay learning* mechanism on the baseline out-of-order improves performance only by 1.6%. The ability of reservation-station-based scheduler to monitor and issue instructions based on operand availability is sufficient enough to get optimal performance. However, applying these techniques on top of a simpler in-order-based core offers large performance benefits as results for the *This Work* illustrate. A limit study (*This Work w/ RS-based PQs*) can achieve 97.9% of the out-of-order core performance when associative lookups are performed in the PQs to minimize issue time mispredictions.

6.5 Proposed Design Implementation Analysis

In this work, we take advantage of the high levels of repeatably of the code [25] to learn the delays of instructions up-front and prioritize them on future encounters. Figure 7(a) shows the number of cycles each application spends executing Repeated and Debut instructions. Instructions that appear more than once during execution are called Repeated, while instructions seen for the first time are called Debut (including the first appearance of Repeated instructions). Some applications (like `astar` and `mcf`) see as much as 36% Debut instructions. While some of this is an artifact of application sampling (see Section 5), there will always exist code that is seen only once, either because of large Debut code or because of large number of loops that do not fit in the DelayCache for the entire execution of the application. Overall, a large number of Debut instructions can potentially reduce performance as the issue time predictor will not have real-time information for these instructions and will have to use static delays instead, that can produce issue time mispredictions.

In Figure 7(b), we show the number of stalls at the head of the queues due to unresolved dependencies, normalized to the in-order processor baseline. The out-of-order processor is not represented in this figure as it allows issuing from any position in the instruction queue. Some applications, like `gcc`, have long dependency chains and stall the processor more often, while

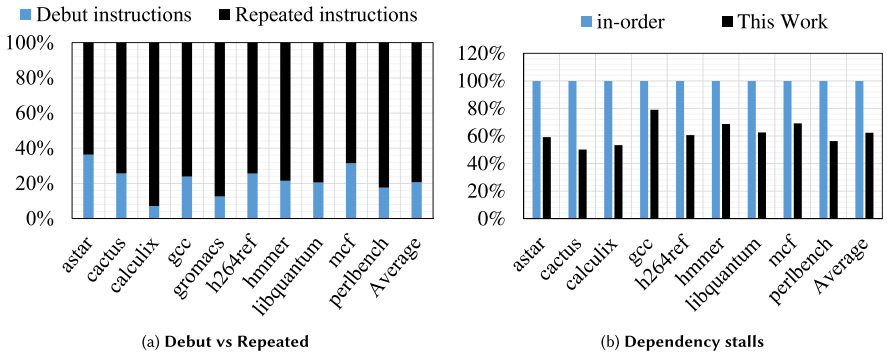


Fig. 7. (a) Cycles spent executing *Debut* instructions (appear for the first time or miss in the DelayCache) and *Repeated* instructions (hit in the DelayCache) and (b) cycles the instruction queues are blocked due to unresolved dependencies.

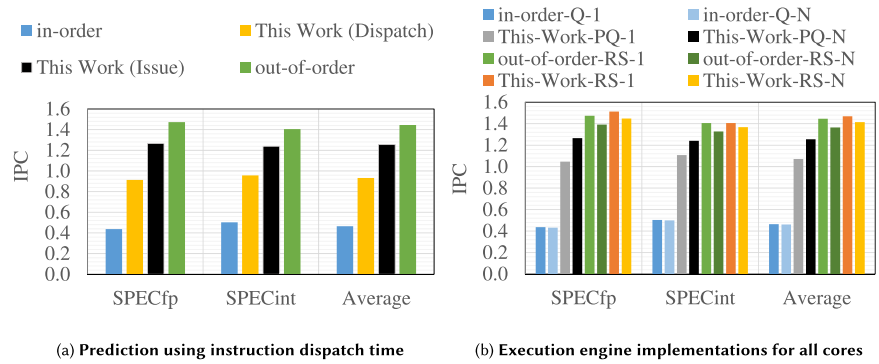


Fig. 8. Implementation options for (a) the prediction algorithm and (b) the execution engine (Q and PQ: issue from the head, RS: issue from any position in the queue, 1: one instruction queue for all units, N: one queue per functional unit).

compute-intensive applications, like `cactus`, can expose more ILP. Overall, the proposed core reduces stalled cycles at the head of instruction queues by an average of 38%.

Structural hazards are another major source of stalls as they can block the instruction queue by increasing resource contention of the functional units. To include the time an instruction waits in the instruction queue when we calculate the delay ($T_{Delay}(p)$) from Equation (1), we replace the issue time ($T_{Issue}(p)$) with the dispatch time ($T_{Dispatch}(p)$) that represents the time an instruction entered the queue. However, instruction waiting times in a dynamic processor environment are unpredictable across iterations. Figure 8(a) shows that using the dispatch time instead of the issue time in the prediction algorithm reduces the performance by an average of 25.7%. The dynamic nature of the proposed core changes the schedules in every iteration and the time an instruction will wait in a queue, making structural hazard delays unpredictable.

To minimize the impact of structural hazards in an energy efficient way, the proposed core implements per functional unit priority queues (PQ-N). Figure 8(b) shows results of implementing a multi-queue back-end for each baseline core. The performance of the in-order core is not improved as instructions still need to be issued in program order (in-order-Q-1 to in-order-Q-N). The proposed core (that uses a PQ-N) achieves a performance improvement of 14.7% over a single priority queue (PQ-1) implementation. Increasing the number of RS in the out-of-order processor and

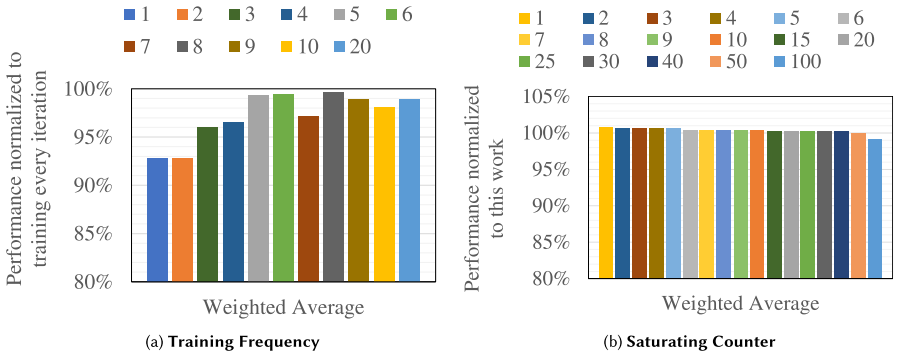


Fig. 9. (a) Train for a number of iterations and use that prediction thereafter (normalized to training every iteration). (b) Use a saturating counter to update the delay every N identical consecutive delays (normalized to This Work). Note that the *y*-axis starts at 80%.

limiting issue of each queue to a single functional unit (RS-N), reduces its performance due to the limited destinations an instruction can issue to. Implementing an RS-1 on this work surpasses out-of-order performance by 1.6% as the RS compensates for issue time mispredictions and removes structural hazards completely. An RS-N solution suffers from performance loss for the same reason as the out-of-order core.

Issue Time Training Frequency. Load latency analysis shows that using the previous load delay (per PC) provides an average of 92.8% accuracy for predicting the next delay value. The training frequency of the issue time predictor depends on the application and the number of times the instruction delay changes throughout the execution. Using different predictor training frequencies (Figure 9(a)) shows that the more often the predictor is trained, the higher the performance that can be achieved. Alternatively, using a saturating counter delay predictor that updates the delay of an instruction in the DelayCache only after the same delay appears for a number of consecutive iterations (Figure 9(b)) shows a marginal average improvement of 0.8% over this work (with `sphinx3` the only exception to achieve a 10% improvement due to the high number of consecutive misses in the cache). Both of these studies show that changes in different iterations of load access delays of repeated instructions are highly unpredictable and do not follow a specific pattern that can be easily learned. But, in-depth analysis of the delays shows that in most consecutive appearances the delay is the same (hence the 92.8% accuracy).

A more sophisticated branch-predictor-like mechanism based on loops could store multiple delays per instruction to train the predictor with higher confidence. However, our study shows that storing as many as five delays per instruction and using the most frequent, the smallest, the largest, or the average in the prediction does not further improve performance.

Instruction Steering Analysis. As described in Section 4.2, the proposed design uses dependencies on instructions at the tails of the queues to dispatch new instructions to the queues (*Tail-Dependencies*). We found this technique to produce higher performance compared to either checking dependencies to all instructions in a queue (*All-Dependencies*) or inserting instructions in a *Round-Robin* scheme. Checking for dependencies only at the tail of each queue achieves on average 2.1% and 0.3% improvement over *Round-Robin* and *All-Dependencies* respectively (see Figure 10).

Core Components Scalability. Figure 11 shows a scalability study of the core structures and how they affect the performance of the proposed design. The size of the priority instruction queues (Figure 11(a)) affects the throughput of the front-end. The number of functional units affects the

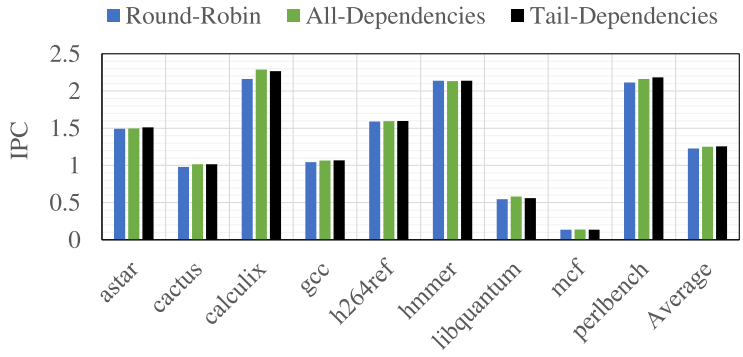


Fig. 10. Steering instructions to priority queues: *Round-Robin*, *All-Dependencies* (follow all producers in a queue) and *Tail-Dependencies* (dependent on an instruction at a queue's tail).

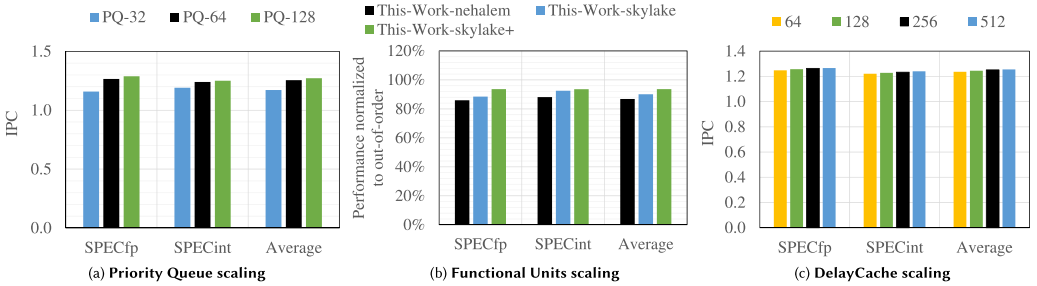


Fig. 11. Scaling the (a) priority instruction queues with a 128-entry ROB (the size refers to the total sum of all queues), (b) number of functional units (each one normalized to its corresponding out-of-order baseline), and (c) DelayCache.

throughput of the backend (Figure 11(b)), while the size of the DelayCache determines the number of delays that can be stored (Figure 11(c)).

Stalling of the front-end can happen when instruction queues are too small. The proposed core can stall even when a single queue is full, and therefore selecting the correct size is important. Figure 11(a) shows that for the same ROB size (128 entries), a total of 64-entries for all queues (13 entries per queue) achieves similar performance to a total size of 128 entries.

The port configuration of this work is based on an Intel Nehalem core that supports three generic, one load, and two store (one for address calculation and one for data) units. Figure 11(b) shows results for a Skylake-based configuration (four generic, two load, and two store units) and Skylake+ (four generic, two load, and four store units). The proposed implementation achieves performance within 10% and within 6.4% of their corresponding out-of-order Skylake and Skylake+ processors, respectively. Figure 11(c) shows that performance improvement for more than 64 entries in the DelayCache is marginal and levels off at 512 entries for all applications tested. The DelayCache can be small as we only store delays of load instructions that miss in the L1 cache.

7 RELATED WORK

There has been extensive work in the past on instruction reordering to reduce runtime delays and improve processor performance. Table 4 presents state-of-the-art hardware solutions in instruction reordering, the delays they try to mitigate, the stage in the processor the reordering takes place, and the type of scheduler used to reorder instructions. High performance comes from

Table 4. State-of-the-Art Instruction Reordering Processors That Try to Mitigate Delays Coming from Dataflow Dependencies (*Static*) and/or Runtime Delays (*Dynamic*)

State of the art hardware instruction reordering	Static	Dynamic	Reorder	Scheduler
Dataflow Prescheduling [26]	✓		Front & Back	RS
Wait Instruction Buffer [21]		✓	Front & Back	RS
Long-Term Parking [34]	✓	✓	Front & Back	RS
Insequence Instructions [37]	✓		Back	RS
WiDGET [42]	✓		Back	RS
Runahead [28]		✓	Back	RS
Continuous Runahead [14]		✓	Back	RS
Load Scheduling [43]		✓	Back	RS
Segmented IQs [33]	✓	✓	Back	RS
Look-ahead Prediction [24]	✓	✓	Back	RS
Dynamos [29]	✓	✓	Back	RS+FIFO
Mirage [30]	✓	✓	Back	RS+FIFO
FIFOrder [2]	✓	✓	Front & Back	RS+FIFO
Dealy and Bypass [1]	✓	✓	Front & Back	RS+FIFO
N-use Issue Logic [5]	✓	✓	Front & Back	AST+FIFO
Deterministic Issue Logic [5]	✓		Back	RS+CQ
Distance Issue Logic [4]	✓		Back	RS+CQ
In-order SMT [16]	✓		Front	FIFO
Load Slice Core [6]	✓		Front	FIFO
Freeway [20]	✓	✓	Front	FIFO
Complexity-Effective [31]	✓		Front	FIFO
iCFP [15]		✓	Front	FIFO
CASINO [19]		✓	Front	FIFO
Wakeup-free [17]	✓	✓	Front & Back	Replay
Cyclone [12]	✓	✓	Front & Back	Replay
This Work	✓	✓	Back	PQ

Reorder designates the stage instructions are reordered at (Back: Back-end and Front: Front-end). *Scheduler* is the reordering mechanism used (RS: Reservation Station; FIFO: First In First Out; AST: Associative Table; Replay: Reschedule stalled instructions; CQ: Circular Queues; and PQ: Priority Queues).

mitigating both *Static* and *Dynamic* delays, while reordering instructions in the backend of the processor provides for higher flexibility. Unfortunately, the majority of past solutions uses an *RS-based* scheduler for reordering instructions that limits energy efficiency improvement. In this work, we argue that smarter solutions are needed to significantly improve energy efficiency, using a simpler and more scalable scheduler (*PQ*) to reorder instructions in the backend, while achieving high performance by addressing *Static* and *Dynamic* delays. In this section, we discuss different categories of solutions that address runtime delays in instruction scheduling.

RS-based Schedulers. Many solutions use dataflow dependencies to preschedule or prioritize instructions to improve the performance or the efficiency of an out-of-order processor. Dataflow Prescheduling [26] fetches and reorders instructions in a prescheduling buffer using dataflow dependencies. This provides for a larger effective window size while keeping the issue buffer small. However, it does not take into account variable delay instructions and assumes static delays for all instructions (all loads are presumed to hit in L1). Segmented Instruction Queues [33] divide large instruction queues into smaller segments that can be clocked at higher frequencies. They use dynamic dependence-based scheduling to promote instructions from segment to segment until they reach a small issue buffer. Data Cache Hit-Miss Prediction [43] tries to predict L1 hits and reschedule load dependent instructions based on that information. But predicting only L1 hits does not take into account off-chip memory delays that have the most impact on the the performance of a processor. In a more complex implementation, Look-ahead Prediction [24] tries to predict load delays using a value predictor. Dynamic solutions like in References [21, 34]

predict and prioritize critical or independent instructions. In Reference [21], instructions that depend on long-latency operations are moved from the issue queue to a much larger waiting instruction buffer until their long-latency producer completes. **Long Term Parking (LTP)** [34] analyzes instructions and parks non-critical instructions from the main instruction stream to prioritize critical ones (address-generating instructions and loads). Similarly, N-Use [5], uses an associative table to park non-ready instructions, Distance Issue Logic [4] assumes unknown load delays and parks all their consumers in an RS IQ until their operands are produced, while Deterministic Issue Logic [5] assumes a static delay for all loads and only parks stalled consumers to the RS IQ. FIFOrder [2] and Delay and Bypass [1] use the knowledge that an OoO-core instruction scheduler offers (availability of the instructions operands) to dispatch ready instructions to FIFO queues to reduce the size and power consumption of complex instruction queues. They differ by the type of instructions to be send to the FIFO queues based on their criticality and readiness. All these solutions require additional hardware to implement and still employ a traditional out-of-order scheduler to handle the reordering and compensate for timing mispredictions of their techniques.

FIFO-based Schedulers. Due to their low power consumption, in-order processors are highly energy efficient. However, they achieve significantly lower performance compared to an out-of-order processor. Complexity-Effective [31] reorders instructions based on their dependencies. Instructions that belong to the same dataflow dependency chain are directed to dedicated in-order queues, while selection logic is used to issue instructions from the head of the queues. The Load Slice Core [6] extends an in-order, stall-on-use core with a second in-order pipeline that allows memory accesses and address-generating instructions to bypass stalled instructions in the main pipeline. Unfortunately, these solutions do not take into account dynamic delays. This creates large gaps between load-dependent instructions in a real execution that stall until the producing load returns from memory, thus limiting their performance improvement.

Cyclone [12] uses a store set dependence predictor to monitor memory dependencies, while mispredicted instructions are replayed from the tail of the queue. But this implementation potentially scrambles the ordering of other instructions in the instruction window, creating a performance bottleneck. Wakeup-free scheduling [17] improves this structural constraints by using a collapsing scheme that does not allow instructions to move while their latency counters are decreasing. But their evaluation is done using a perfect L1 Hit predictor for load latency delays, which does not take into account off-chip memory delays that have the most impact on the the performance of a processor. iCFP [15] uses a Continual Flow Pipeline that switches to an advance execution mode when it encounters a L1 or L2 cache miss. Miss-dependent instructions are diverted into a slice buffer, un-blocking the pipeline for miss-independent instructions to execute. Although it achieves low power consumption, its performance is limited to 68% of the performance of an out-of-order processor [15]. Freeway [20] is an orthogonal solution that implements a technique similar to LTP [34] on top of an in-order core and manages to improve its performance by 80%, while we achieve 180% increase in performance over our in-order baseline core (on the same applications). CASINO [19] uses two in-order queues to filter instructions that block the issuing queue. However, their solution takes no real-time information into consideration when doing the filtering that can potentially lead to even more excessive delays when one of the queues is filled, depending on the applications executed.

Heterogeneous Processors. Mirage Cores [30] and its predecessor Dynamos [29] employ a full out-of-order core to produce fast out-of-order schedules that are stored in a local cache structure and executed by a number of in-order cores on the same processor. WiDGET [42] enables dynamic customization of different combinations of small and/or powerful cores as a way to increase performance and reduce power consumption depending on the executing workload. The

design complexity and cost of these solutions, however, makes them inefficient, as they still require the implementation of an out-of-order core to learn aggressive instruction schedules.

Software Implementations. Compile-time application analysis is also used to categorize and prioritize instructions by predicting the critical path of the execution [13, 41]. Solutions with good balance between performance and energy efficiency use modified hardware equipped with the appropriate compile-time support to statically reorder instructions in advance [3, 10, 18, 38–40, 44]. But, unlike our work, these solutions require modification to the application itself and do not provide backward compatibility for deployed applications.

SMT. In a multi-threaded architecture, independent instructions from different threads can be used to overcome dependency stalls from a single thread [16, 37]. This boosts performance of multi-threaded applications as it increases processor throughput in throughput-sensitive parallel applications. However, these techniques do not address single-thread performance.

Prefetching. Prefetching attempts to minimize cache misses by executing additional instructions [11, 14, 28]. Runahead [28] allows the execution to continue past stalling to pre-execute instructions and generate new cache misses that fetch data earlier for future instructions. Continuous runahead [14] extends previous solutions by dynamically filtering the instruction stream to identify the chains of operations that cause a pipeline to stall. Unfortunately, prefetching techniques alone are not enough as they only try to hide memory latency. All solutions referenced here still use a complex out-of-order scheduler to handle instructions reordering.

8 CONCLUSION

In this work, we propose a novel scheduling scheme that tracks real-time delays of load instructions to accurately predict instruction issue times and a priority-based instruction reordering mechanism that achieves near out-of-order performance in an energy efficient way. To this end, we design a new microarchitecture that builds aggressive schedules and produces near out-of-order performance in an energy efficient way. The proposed design replaces the complex instruction scheduler of an out-of-order processor with a *instruction delay learning mechanism* that monitors load instructions and learns their latest real-time delays, an *issue time predictor* that predicts their issue times, and *priority queue reordering* that efficiently reorder instructions. Together, these three techniques allow the new core to achieve 86.2% of the performance of the baseline out-of-order, while reducing the power consumption for instruction scheduling hardware by 88%.

REFERENCES

[1] 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 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. 424–434. https://doi.org/10.1109/HPCA47549.2020.00042

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

[3] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman. 2015. Denver: Nvidia's first 64-bit ARM processor. *IEEE Micro* 35, 2 (March 2015), 46–55. https://doi.org/10.1109/MM.2015.12

[4] Ramon Canal and Antonio González. 2000. A low-complexity issue logic. In *Proceedings of the 14th International Conference on Supercomputing (ICS'00)*. Association for Computing Machinery, New York, NY, 327–335. https://doi.org/10.1145/335231.335263

[5] Ramon Canal and Antonio González. 2001. Reducing the complexity of the issue logic. In *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*. Association for Computing Machinery, New York, NY, 312–320. https://doi.org/10.1145/377792.377854

[6] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. 2015. The load slice core microarchitecture. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 272–284. https://doi.org/10.1145/2749469.2750407

[7] T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*. 1–12. https://doi.org/10.1145/2063384.2063454

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

[9] George Z. Chrysos and Joel S. Emer. 1998. Memory dependence prediction using store sets. *SIGARCH Comput. Archit. News* 26, 3 (April 1998), 142–153. https://doi.org/10.1145/279361.279378

[10] Neal Clayton Crago and Sanjay Jeram Patel. 2011. OUTRIDER: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 117–128. https://doi.org/10.1145/2000064.2000079

[11] James Dundas and Trevor Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing (ICS'97)*. ACM, New York, NY, 68–75. https://doi.org/10.1145/263580.263597

[12] Dan Ernst, Andrew Hamel, and Todd Austin. 2003. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM, New York, NY, 253–263. https://doi.org/10.1145/859618.859647

[13] Brian Fields, Shai Rubin, and Rastislav Bodík. 2001. Focusing processor policies via critical-path prediction. *SIGARCH Comput. Archit. News* 29, 2 (May 2001), 74–85. https://doi.org/10.1145/384285.379253

[14] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, Article 61, 12 pages. http://dl.acm.org/citation.cfm?id=3195638.3195712.

[15] A. Hilton, S. Nagarakatte, and A. Roth. 2009. iCFP: Tolerating all-level cache misses in in-order processors. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture*. 431–442. https://doi.org/10.1109/HPCA.2009.4798281

[16] S. Hily and A. Seznec. 1999. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*. 64–67. https://doi.org/10.1109/HPCA.1999.744331

[17] J. S. Hu, N. Vijaykrishnan, and M. J. Irwin. 2004. Exploring wakeup-free instruction scheduling. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA'04)*. 232–232. https://doi.org/10.1109/HPCA.2004.10014

[18] Ziqiang Huang, Andrew D. Hilton, and Benjamin C. Lee. 2016. Decoupling loads for nano-instruction set computers. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, Piscataway, NJ, 406–417. https://doi.org/10.1109/ISCA.2016.43

[19] 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 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. 383–396. https://doi.org/10.1109/HPCA47549.2020.00039

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

[21] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*. IEEE Computer Society, 59–70.

[22] Charles E. Leiserson. 1979. *Systolic Priority Queues*. Technical Report. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

[23] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO42)*. ACM, New York, NY, 469–480. https://doi.org/10.1145/1669112.1669172

[24] Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. 2004. Scaling the issue window with look-ahead latency prediction. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04)*. ACM, New York, NY, 217–226. https://doi.org/10.1145/1006209.1006240

[25] Daniel S. McFarlin, Charles Tucker, and Craig Zilles. 2013. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 241–252. https://doi.org/10.1145/2451116.2451143

[26] P. Michaud and A. Seznec. 2001. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings HPCA 7th International Symposium on High-Performance Computer Architecture*. 27–36. https://doi.org/10.1109/HPCA.2001.903249

[27] Sanjeeb Mishra, Neeraj Singh, and Vijayakrishnan Rousseau. 2016. *Understanding Power Consumption Fundamentals*. 13–27. https://doi.org/10.1016/B978-0-12-801630-5.00002-5

[28] 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'03)*. 129–140. https://doi.org/10.1109/HPCA.2003.1183532

[29] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2015. DynaMOS: Dynamic schedule migration for heterogeneous cores. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO48)*. ACM, New York, NY, 322–333. https://doi.org/10.1145/2830772.2830791

[30] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2017. Mirage cores: The illusion of many out-of-order cores using in-order hardware. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO50)*. ACM, New York, NY, 745–758. https://doi.org/10.1145/3123939.3123969

[31] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. 1997. Complexity-effective superscalar processors. *SIGARCH Comput. Archit. News* 25, 2 (May 1997), 206–218. https://doi.org/10.1145/384286.264201

[32] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. 2018. Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon? In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 271–282. https://doi.org/10.1109/HPCA.2018.00032

[33] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. 2002. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. 318–329. https://doi.org/10.1109/ISCA.2002.1003589

[34] 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'15)*. 334–346. https://doi.org/10.1145/2830772.2830815

[35] André Seznec. 2014. TAGE-SC-L branch predictors. In JILP Championship Branch Prediction. Minneapolis, MN.

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

[37] F. M. Sleiman and T. F. Wenisch. 2016. Efficiently scaling out-of-order cores for simultaneous multithreading. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 431–443. https://doi.org/10.1109/ISCA.2016.45

[38] K. 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 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'17)*. 171–184. https://doi.org/10.1109/CGO.2017.7863738

[39] Kim-Anh Tran, Alexandra Jimborean, Trevor E. Carlson, Konstantinos Koukos, Magnus Själander, and Stefanos 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'18)*. ACM, New York, NY, 328–343. https://doi.org/10.1145/3192366.3192393

[40] Francis Tseng and Yale 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'08)*. IEEE Computer Society, 3–12. https://doi.org/10.1109/ISCA.2008.23

[41] E. Tune, Dongning Liang, D. M. Tullsen, and B. Calder. 2001. Dynamic prediction of critical path instructions. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA'01)*. 185–195. https://doi.org/10.1109/HPCA.2001.903262

[42] Yasuko Watanabe, John D. Davis, and David A. Wood. 2010. WiDGET: Wisconsin decoupled grid execution tiles. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 2–13. https://doi.org/10.1145/1816038.1815965

[43] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. 1999. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. IEEE Computer Society, 42–53. https://doi.org/10.1145/300979.300983

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

Received 24 September 2021; revised 9 May 2022; accepted 13 June 2022