RIO: ROB-centric In-order Modeling of Out-of-order Processors

Wim Heirman, Stijn Eyerman, Kristof Du Bois, and Ibrahim Hur

Abstract—Architectural studies of the cache and memory hierarchy need a fast simulation model for the processor core that accurately conveys the impact of memory subsystem changes on application performance. We propose the RIO model (ROB-centric In-order model for Out-of-order cores), a single-pass core performance model based on finding the earliest possible future issue time for out-of-order execution. RIO can natively model second-order effects of overlapping and interacting miss events, significantly improving accuracy over interval simulation. Yet it is no more complex to implement and run, providing a compelling speedup over more detailed models. We implement RIO in Sniper and evaluate it on 2000+ application traces, and find it has an average absolute prediction error of 10.3% over Sniper's most detailed model, while simulating 2.8× faster on average (up to 5× on memory-bound workloads).

Index Terms—processor performance simulation, mechanistic core models

1 Introduction

TRADITIONAL cycle-by-cycle modeling is generally considered too slow to drive memory studies with high core counts and large caches, which need relevant chunks of realistic applications to be simulated in reasonable time. Higher-abstraction models such as interval simulation are much faster, but can lack the accuracy needed to make reliable performance predictions.

We propose a new technique, the ROB-centric In-order model for Out-of-order processors (RIO). It is a single-pass model, which visits instructions only once and in-order while computing their earliest possible execution time. RIO can be viewed as a hybrid between instruction-window centric (IWC) modeling [1], and interval simulation [2]. Like IWC, our technique models the reorder buffer (ROB) as a central structure, but like interval simulation it only requires a single pass over all instructions.

Executing the model is therefore much less computationally expensive than IWC because it does not need to perform bookkeeping on large numbers of pending instructions. It is also more accurate than interval simulation, because it provides mechanistic dependency-based modeling rather than analytical approximations at the boundaries of miss events such as branch mispredictions and long-latency loads. This is important especially when such phases occur close together or overlap, e.g., when branch mispredicts or instruction cache misses are partially hidden under a long-latency load. RIO naturally models these second-order effects, while offering a simulation speed similar to interval simulation. It is therefore well-suited to drive large multi-core memory hierarchy studies that use complex applications with high rates of miss events, such as datacenter workloads.

2 Background

Interval simulation is a high-abstraction level performance model for out-of-order processors [2]. It is based on the observation that the execution profile can be seen as a sequence of intervals of smooth execution flow, separated by miss events (cache or TLB misses, branch mispredictions) during which the core pipeline stalls. Given the processor's window size (number of entries in the reorder buffer, ROB) and width of the dispatch stage,[1] interval simulation inspects the dynamic instruction stream in-order and keeps two windows' worth of instructions: the *new* window of upcoming instructions and the *old* window of past instructions.

As instructions graduate from the new to the old window, the time is estimated at which they would pass the dispatch stage. During an interval of smooth execution, the critical path of dependent instructions through the old window determines at what rate old instructions at the head of the ROB can be committed, and hence how quickly new instructions can be dispatched into the ROB. For instructions that cause miss events, the penalty is computed (access latency for TLB and cache misses, branch resolution plus recovery time for branch mispredicts) and added to the current timestamp. After each miss event, the old window is flushed under the assumption that all independent instructions have completed under the miss. For long-latency data cache misses, in addition, the new window is scanned for independent memory loads that may resolve in the shadow of the initial miss event.

In practice, interval simulation is very fast and works well when miss events are isolated. However, for complex application codes where miss events of different types and lengths occur close together, the model can make large errors as it does not take the second-order effects of interacting miss events into account.

Instruction-window centric (IWC) simulation was proposed as a new mechanistic core model, bridging the gap in complexity and accuracy between interval simulation and cycle-accurate simulation [1]. IWC simulation is closer in concept to a fully detailed simulator, and keeps track on a

• *All authors are with Intel Corporation, {wim.heirman, stijn.eyerman, kristof.du.bois, ibrahim.hur}@intel.com*

1. We use the terminology *dispatch*, *issue*, and *commit* in this paper; these are equivalent to *allocate*, *execute*, and *retire*, respectively.

cycle-by-cycle basis how instructions flow through the processor pipeline. But unlike detailed simulation, IWC simulation only models those microarchitecture components that are relevant to performance, driven by the insights provided by the interval model. For instance, after dispatching a mispredicted branch,[2] nothing is simulated on the wrong path—the dispatch stage is simply stalled until the branch is resolved. This allows IWC simulation to natively model overlapping misses, in fact it does not explicitly consider intervals or miss events, it naturally models the behavior of the processor over subsequent clock cycles. The downside of this approach is that significantly more work is done per instruction: instead of visiting instructions in-order and only once, the IWC core model needs to revisit instructions each time they move into a new pipeline stage, and potentially multiple times when structural hazards (e.g., a limited number of execution units to handle certain complex instructions) cause additional delays.

3 The RIO core performance model

The RIO core model implements the timing model of an out-of-order processor core. Relevant parameters are the width of the dispatch stage (W), size of the reorder buffer (R), and instruction latencies for each type. The model is fed by the dynamic instruction stream provided by a trace or functional simulator, and interfaces with separate models for the instruction and data caches and subsequent memory subsystem, and with a branch predictor model.

3.1 RIO data structures and modeling algorithm

The data structure at the center of RIO is a table representing the reorder buffer (ROB) of the processor. This table contains all instructions that are dispatched, but not yet committed. For each instruction, it stores information related to the instruction itself (program counter, disassembly, input/output registers), as well as *issued* and *completed* times.[3] In addition, the register dependency table (RDT) lists, for every architectural register, a timestamp when its value becomes available. The memory dependency table (MDT) contains, for the last N stores (N being at least the size of the processor's load-store queues), the address, access size and completion time of recent store instructions. Finally, a variable w keeps track of how many instructions were dispatched during the current cycle (up to W, the dispatch width of the machine).

The simulation progresses by modeling, in instruction order, the dispatch and issue stages for up to W instructions that all dispatch in the same cycle, then models the commit stage. The local timestamp is incremented as instructions are dispatched, while skipping over cycles during which no modeling work needs to be done.

Step 1. First, we initialize the dispatch counter $w = 0$.

Step 2. The dispatch and execute stages are modeled by inspecting instructions in program order.

Step 2a. The instruction cache model is interrogated to determine whether the instruction hits in the I-cache. If it does not, w is reset to 0 and the local timestamp is incremented by the I-cache miss latency provided by the memory simulation model.

Step 2b. We model the execution stage immediately, by finding the earliest time at which the instruction can issue. Like the interval model, RIO honors all instruction dependencies but will be optimistic in that it assumes a suitable execution unit is always available. For instructions that read from registers, each input register's available time is read from the RDT. For memory loads, the MDT is scanned for fully or partially overlapping stores and the maximum completion time is recorded. We then set the instruction's issue time to the maximum out of (i) each of its input registers from the RDT, (ii) any overlapping store from the MDT, (iii) the current local timestamp (which represents the instruction's dispatch time). We then add the instruction latency to the issue time to compute the completion time. For memory loads and stores, the execution latency is determined by the data cache simulation model. The completion time is recorded in the ROB. We also look at all output registers that are produced by this instruction, and set the timestamp of each of these registers in the RDT to the completion time. For stores, we insert an entry into the MDT with the memory address, access size, and issue time.[4] The MDT is managed as a FIFO, so adding a new store pushes the oldest entry out.

Step 2c. For branches, we query the branch predictor model to determine whether the branch condition (for conditional branches) or target (for indirect branches) was predicted correctly. If not, we use the branch's completion time (which is when the correct branch outcome or target will be known), add a constant value for modeling the front-end refill time, reset the local timestamp to this new value, and continue to the modeling of the commit stage (step 3).

Step 2d. In the absence of branch mispredictions, we increment w. If this counter reaches W, a full dispatch width has been achieved for that cycle, so we increment the local timestamp by one and proceed to step 3. If the ROB is full, we also jump to step 3. Else, as long as w remains below W, the ROB is not full, and there was no branch misprediction, we restart step 2 for the next instruction in the input stream.

Step 3. We now model the commit stage. We remove all instructions from the ROB head as long as they have a completed time that is less than or equal to the local timestamp.[5] If the ROB has filled up (contains R entries), we look at the instruction at the head of the ROB and ensure it can be committed. This will be possible once the local timestamp has reached the instruction's completed time. Hence, we jump ahead by resetting the local timestamp to the completed time of the instruction at the ROB head. We then again remove any further instructions from the ROB head as long as they complete before or at the new current timestamp. This guarantees that we end step 3 with space for at least one instruction in the ROB, ensuring that step 2 can proceed. We then restart the algorithm at step 1.

2. IWC is functional-first, so the correct branch outcome is known even before the branch reaches the timing model.

3. In fact, only the completed time is used by the RIO model, the other fields are for statistics collection and CPI stack construction only.

4. This assumes perfect store-to-load forwarding of through-memory dependencies, given that forwarding can happen as soon as both the address and data values are known. To model lack of forwarding, the completion time can be used instead.

5. This assumes an infinite commit width, this stage is rarely a bottleneck in reality.

Reorder buffer (ROB)

PC	*Instruction*	*Issued*	*Completed*
0x100	load rax := @rbx	10001	10044
0x101	inc rax	10044	10045
0x102	jnz rbx, $207	10001	10016
0x207	store @rax := rbx	10045	10099
0x208	store @rbx := rbx	10016	10035

Time

10016

Register dependency table (RDT)

Register	*Available*
rax	10045
rbx	10001

Memory dependency table (MDT)

Address	*Size*	*Available*
0x12345	8	10099
0x45678	1	10035

Fig. 1. Data structures used by RIO.

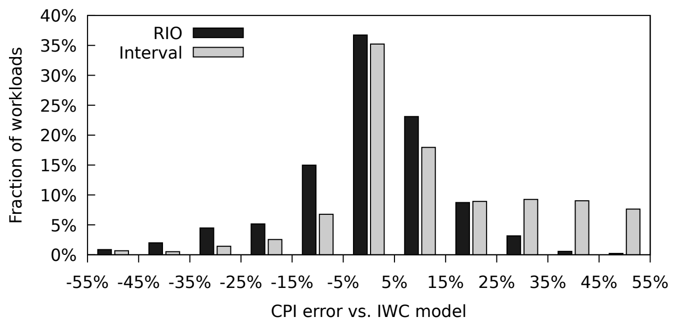

Fig. 2. Histogram of modeling errors over all 2000+ input traces.

3.2 Example

Figure 1 shows all model state and walks through an example code fragment. According to the RDT, register `rbx` was produced by an earlier instruction and became available at timestamp 10001. At this point, the load at PC 0x100 which uses `rbx` as an input register, becomes ready so it executed at timestamp 10001. According to the cache model this was an LLC hit with latency of 43 cycles so the result (register `rax`) becomes available at time 10044. The next instruction is a 1-cycle increment which updates `rax`, this can happen at time 10044 and produces a new value for `rax` at time 10045 which is recorded in the RDT. The next instruction, a conditional jump, is independent of the load but depends on `rbx` as well so it also executes at time 10001. This branch was mispredicted, so the next instruction at PC 0x207 can only happen after the branch resolves and the front-end has refilled, which happen at time 10016. The first instruction at the branch target however also depends on `rax` which isn't ready until time 10045. Hence the store, with latency 54 cycles according to the cache model, completes at cycle 10099 which is recorded in the MDT (first entry). The second store at PC 0x208 also dispatches at time 10016 and only depends on `rbx`, which hasn't changed since time 10001 so this second store can execute immediately, and (with a latency of 19 cycles) completes at time 10035. At this point, we have dispatched 2 instructions during cycle 10016—so if $W = 2$ we would now reset the dispatch counter w and increment the local timestamp to 10017 before continuing. We then model the commit stage. Assuming $R = 5$, the ROB is full at this point so we jump the timestamp to the completion time of the instruction at the ROB head, 10044, and commit all instructions with a completion time $\leq$10044 (in this case, only the load at PC 0x100).

3.3 Discussion

Compared to the interval model, RIO is more accurate in predicting the cost of branch mispredictions and long-latency load events. The interval model splits execution into phases (intervals) separated by miss events, and needs to make several assumptions about what happens during transitions between phases. It uses the new window to determine which independent loads can be overlapped by miss events, but assumes *all* independent instructions in the window will execute—this is overly optimistic when the miss event is relatively short (e.g., LLC hit, rather than a DRAM access), or if the ROB was not full at the time of the miss event (e.g., when I-cache miss and branch mispredictions occur in quick succession). On the other hand, the interval model is often pessimistic when computing branch penalties. It computes the branch resolution time by finding the critical path from the head of the old window to the branch condition (or target). This does not consider the fact that a portion of this path may have already executed by the time the branch dispatches.

In contrast, RIO keeps track of which instructions are in the ROB and when they execute at all times, so the transitional phases around miss events are all handled natively. RIO is therefore more accurate than interval simulation, especially for workloads with high rates of miss events, while its complexity and simulation speed are very similar.

Compared to the instruction window centric (IWC) model, RIO requires much less computation time. Because we determine each instruction's issue and completion times in one pass, we can visit instructions in-order and only once. This is accurate as long as there are sufficient functional units. The IWC model, in contrast, needs to keep track of dependencies and potentially revisit instructions multiple times when they have complex ordering requirements.

4 Results

We implemented RIO in our in-house simulator which is derived from Sniper [3], and already has core models based on interval simulation and IW-centric simulation available. In this experiment we only change the core model, and kept the branch predictor and memory models the same. All experiments are configured to model a single-core Intel Skylake processor, which is a 4-wide out-of-order x86 processor with an ROB of 224 entries. We assume the IWC model as the golden reference and compute the prediction error when running over 2000 input traces from a wide variety of application domains (including SPEC CPU 2017, SPECjbb, high-performance computing, deep learning training and inference, and various datacenter workloads). All traces are 30 million instructions long, and are preceded by a separate cache warming phase of at least 100M instructions.

Overall, the RIO model has an average absolute error of 10.3% when predicting runtime, and a bias (average non-absolute error) of -0.5%. In contrast, interval simulation has an average absolute error of 18.1% and a bias of $+13.1\%$.

Figure 2 shows a histogram of the modeling error of RIO and the interval model, compared to the baseline IWC

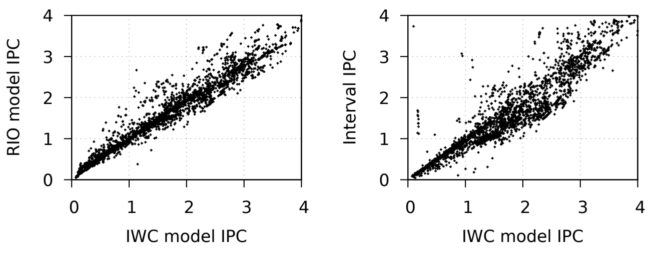

Fig. 3. Modeled vs. reference IPC, for RIO (left) and interval (right).

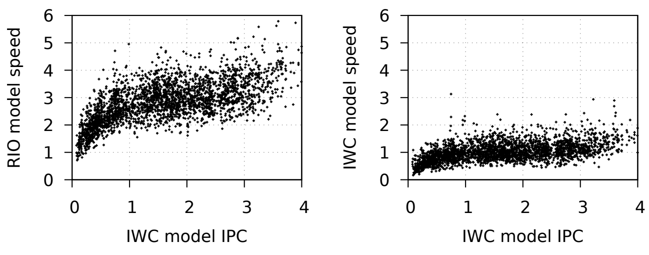

Fig. 4. Simulation speed of RIO (left) and IWC simulation (right) as a function of workload IPC.

results, over all input traces. For 84% of the traces, the application performance predicted by the RIO model is within ±20% of the result computed by the IWC model. In contrast, interval simulation predicts only 65% of the traces to within ±20% while exhibiting larger outliers: 6% of the traces show an error of over 50%. Moreover, a disproportionate fraction (>70%) of traces is predicted too slow, leading to the bias of the model.

In Figure 3 the modeled (Y-axis) vs. reference (X-axis) IPC of all traces is shown, for both RIO (left) and interval simulation (right). Points on the diagonal have accurate prediction, while points above (below) the diagonal indicate the higher-abstraction model predicts performance as too fast (slow). The RIO model has more points clustered close to the diagonal, while interval simulation has a wider variation while also showing several data points very far from the diagonal (very high absolute error).

Detailed analysis of outlier traces points at two important reasons. One is the interval model's already mentioned lack of second-order effects, where real processors can provide more overlap when miss events are close together or can compute part of the branch resolution before the branch dispatches. Another cause for large outliers is in how the interval model fails to handle store-bound workloads: when there are no through-memory dependencies, interval simulation assumes stores are handled fully off the critical path. In reality, this is only true until the store buffer fills up, which will happen in applications that are bound by main memory write bandwidth. In contrast, in the RIO model stores occupy the ROB until they complete, so a store-bound workload naturally fills up the ROB which throttles the request rate to a bandwidth dictated by the memory model.

The errors for the RIO model compared to IWC are mostly caused by the lack of modeling contention for functional units. When determining the (future) issue time of an instruction, in step 2b of the algorithm, we assume a suitable execution unit is always available. In practice, there are only a limited number of functional units of each type. Intel's Skylake can issue up to eight micro-operations in a given cycle, but only one of them may be a store. This causes RIO to be optimistic on those traces where the instruction mix differs significantly from the functional unit mix. On the other hand, RIO can be pessimistic in modeling the front-end of the processor. In reality, part of the I-cache miss latency can be overlapped because the fetch stage typically runs ahead of the dispatch stage. RIO models these stages as one and stalls dispatch for the full I-cache miss latency.

In terms of simulation speed, RIO is as fast as the interval model (within 10%), while being 2.8× faster on average than the IWC model. Figure 4 plots the simulation speed (normalized to IWC's average speed) for both the RIO and IWC models as a function of each workload's IPC. The speedup of RIO over IWC is higher for memory-bound (low-IPC) workloads, where the IWC model spends a lot of time iterating over instructions that are blocked by memory ordering rules or structural hazards. As the ROB and other structures in future cores grow, the number of pending instructions waiting for memory will increase, so we expect IWC's simulation speed to reduce while RIO's complexity remains constant—increasing the speedup of RIO over IWC.

5 Conclusion

The RIO performance model for superscalar out-of-order cores is a hybrid of interval and traditional simulation. We implement RIO in Sniper and compare it to Sniper's most accurate IWC model, where it has an average absolute error of 10.3% over a workload set of 2000+ traces while providing on average 2.8× the simulation speed. When compared to interval simulation, RIO can natively model second-order effects of overlapping and interacting miss events, and fully supports store-bound workloads—leading to fewer and smaller outliers of high prediction error. This makes RIO a useful and reliable performance model for memory subsystem studies on complex applications with large memory working sets, which need fast performance models to cover realistic behavior.

References

[1] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, 2014.

[2] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.