Naithani, A.; Feliu-Pérez, J.; Adileh, A.; Eeckhout, L. (2019). Precise Runahead Execution. IEEE Computer Architecture Letters. 18(1):71-74. https://doi.org/10.1109/LCA.2019.2910518

# Precise Runahead Execution

Ajeya Naithani* Josué Feliu† Almutaz Adileh* Lieven Eeckhout*
*Ghent University †Universitat Politécnica de Valéncia

**Abstract**—Runahead execution improves processor performance by accurately prefetching long-latency memory accesses. When a long-latency load causes the instruction window to fill up and halt the pipeline, the processor enters runahead mode and keeps speculatively executing code to trigger accurate prefetches. A recent improvement tracks the chain of instructions that leads to the long-latency load, stores it in a runahead buffer, and executes only this chain during runahead execution, with the purpose of generating more prefetch requests during runahead execution. Unfortunately, all these prior runahead proposals have shortcomings that limit performance and energy efficiency because they discard the full instruction window to enter runahead mode and then flush the pipeline to restart normal operation. This significantly constrains the performance benefits and increases the energy overhead of runahead execution. In addition, runahead buffer limits prefetch coverage by tracking only a single chain of instructions that lead to the same long-latency load. We propose precise runahead execution (PRE) to mitigate the shortcomings of prior work. PRE leverages the renaming unit to track all the dependency chains leading to long-latency loads. PRE uses a novel approach to manage free processor resources to execute the detected instruction chains in runahead mode without flushing the pipeline. Our results show that PRE achieves an additional 21.1% performance improvement over the recent runahead proposals while reducing energy consumption by 6.1%.

**Index Terms**—Microarchitecture, single-core performance, runahead execution

## 1 Introduction

Runahead execution [1], [2], [3] improves processor performance by accurately prefetching long-latency loads. The processor triggers runahead execution when a long-latency load causes the instruction window to fill up and halt the pipeline. Instead of stalling, the processor removes the blocking long-latency load and speculatively executes subsequent instructions to uncover future independent long-latency loads and expose memory-level parallelism (MLP). However, not all instructions executed in runahead mode lead to useful memory prefetches. Instructions that are not part of a dependency chain that generates a long-latency load waste processor resources that could otherwise be used to generate prefetch requests. To improve the energy-efficiency and performance of runahead execution, runahead buffer [4] filters out unnecessary runahead instructions. In runahead mode, this technique identifies the chain of instructions that generates the stalling load, stores it in the runahead buffer, and keeps replaying only this instruction chain in a loop.

The runahead buffer improves energy-efficiency and performance compared to traditional runahead execution. However, it still suffers from significant shortcomings that impact its performance and energy consumption. First, similar to prior runahead techniques, the full instruction window has to be discarded every time runahead execution is invoked. This reduces the potential performance benefits from runahead execution and increases its energy cost. Moreover, runahead buffer limits prefetch coverage to only a single chain of instructions per runahead interval, while several benchmarks access memory through multiple chains. This limited prefetch coverage reduces the potential performance gain from runahead buffer.

In this paper, we propose precise runahead execution (PRE), a technique that remedies the shortcomings of prior runahead proposals. We observe that when runahead execution is triggered, the processor has sufficient unused resources to execute instructions without discarding any instructions from the reorder buffer (ROB)[1]. PRE uses *runahead register reclamation*, a novel mechanism to manage free physical registers in runahead mode while preserving dependencies among instructions. Moreover, PRE stores all instructions of any chain that generates a long-latency load in a dedicated cache, called the stalling slice table (SST). First, it stores the stalling load in the SST, then with every loop iteration it leverages the renaming unit to detect the preceding instructions in the chain, and stores them in the SST. In runahead mode, PRE receives decoded instructions from the front-end but executes only the ones that hit in the SST. Because PRE stores all long-latency load chains in the SST, it does not limit prefetch coverage to a single load chain. As an optimization, PRE can be augmented with an additional buffer to store all the decoded instructions in runahead mode. When normal execution resumes, instructions are then dispatched from this buffer. Therefore, it is not necessary to fetch and decode runahead-mode instructions again. The micro-op queue is used to hold decoded micro-ops in modern-day processors. We propose to extend its size and use it to buffer micro-ops generated during runahead mode. Compared to an out-of-order core, the performance improvements achieved through runahead execution, runahead buffer and PRE amount to 14.5%, 14.4% and 35.5% on average, respectively. While runahead buffer is energy neutral relative to an out-of-order core, PRE reduces energy consumption by 6.1%.

## 2 Background and Motivation

### 2.1 Full-Window Stalls

In an out-of-order core, a load instruction that misses in the last-level cache (LLC) typically takes a couple hundred cycles to bring data from off-chip memory. Soon, the load instruction blocks commit and the core cannot make any progress. Meanwhile, the front-end continues to dispatch new instructions into the back-end. Once the ROB fills up, the front-end can no longer dispatch instructions, leading to a *full-window stall*. We refer to the load instruction that causes a full-window stall as a *stalling*

1. ROB and (instruction) window are used interchangeably.

*load*, and to its backward chain of dependent instructions as a *stalling slice*.

### 2.2 Runahead Execution

Runahead execution [2] pre-executes an application's own code to pre-fetch data closer to the core. A full-window stall marks the 'entry' to runahead mode. The processor checkpoints the Architectural Register File (ARF), the branch history register, and the return address stack. The processor identifies the results of stalling loads and their dependents as invalid. In runahead mode, the processor retires these instructions without affecting the processor architectural state in order to unblock the ROB and keep the pipeline running. Once the stalling load returns, the pipeline is flushed and the checkpointed architecture state is restored. This marks the 'exit' from runahead mode.

### 2.3 Filtered Runahead Execution

The original runahead proposal executes all the instructions coming from the front-end of the processor. However, many instructions are not necessary to calculate the memory addresses used in subsequent long-latency loads. Hashemi et al. [4] propose a technique to track and execute only the chain of instructions that leads to a long-latency load. Upon a full-window stall, they perform an expensive backward data-flow walk in the ROB and the store queue to find a dependency chain that leads to another instance of the same stalling load. This chain is stored in a buffer named the *runahead buffer* that is placed before the rename stage. In runahead mode, the instruction chain stored in the runahead buffer is renamed, dispatched and executed in a loop, instead of generating new instructions in the front-end. Therefore, the front-end can be power-gated to save energy in runahead mode. By executing only the stalling slice, this technique runs further ahead than traditional runahead, exposing more MLP and achieving higher performance.

### 2.4 Shortcomings of Prior Techniques

Both traditional runahead execution and runahead buffer significantly improve single-threaded performance. However, their full potential is limited by the following key factors.

**Flushing and Refilling the Pipeline.** Runahead execution speculatively executes and pseudo-retires instructions. At the exit of runahead execution, the processor flushes the pipeline and starts fetching instructions starting from the stalling load. Flushing and refilling the pipeline for every runahead invocation incurs significant performance and energy overheads that limit the potential performance gain from accurate prefetching. Assuming that the ARF can be saved/restored in zero cycles, we estimate that every runahead invocation incurs a performance penalty of approximately 56 cycles assuming a 192-entry ROB: (1) refilling the front-end (8 cycles), and (2) refilling the ROB by re-dispatching 192 instructions with a dispatch width of 4, starting from the stalling load (48 cycles). These cycles cannot be hidden and thus directly contribute to the total execution time of the application. Our experimental results reveal that compared to an out-of-order core, runahead execution improves performance by 14.5% on average. However, the speedup has the potential to reach up to 20.6% if the instructions that occupy the ROB when the core enters runahead mode are not discarded.

Prior work [5] shows only a minor performance benefit from reusing valid pseudo-retired instructions when returning into normal mode. After reusing these instructions, the program's critical path created by invalid pseudo-retired instructions still dominates the execution. These instructions must be fetched and executed again after a pipeline flush.

**Limited Prefetch Coverage.** Runahead execution has limited prefetch coverage because it executes all future instructions in runahead mode, which limits how deep in the dynamic instruction stream runahead execution can speculate. Runahead buffer filters the most dominant stalling slice per runahead interval and executes only this dominant stalling slice. Although this allows runahead execution to go further down the instruction stream, runahead execution is limited to a single slice. Unfortunately, this does not match the characteristics of applications that access memory through a diverse set of instruction slices and multiple different load instructions.

**Short Runahead Intervals.** Prior runahead proposals avoid initiating runahead mode if they estimate the runahead interval to be short. The overhead of invoking runahead execution outweighs its benefit for short runahead durations [6]. However, a significant fraction of runahead intervals are short. In particular, for memory-intensive workloads, we find that 27% of the runahead intervals take less than 20 cycles on average. Therefore, such a restriction in current runahead techniques wastes a significant opportunity to enhance the degree of memory-level parallelism in runahead execution.

## 3 Precise Runahead Execution

We propose precise runahead execution to alleviate the limitations of prior runahead proposals. We observe that at the entry of runahead mode, the processor has enough free resources to execute stalling slices without tampering with the instructions in the ROB. Therefore, none of the instructions in the ROB are discarded at the entry of runahead mode, and the ROB is not flushed at the exit. PRE leverages the renaming unit to execute all forthcoming stalling slices. We rely on a novel register allocation and reclamation mechanism to execute instructions in runahead mode while preserving instruction dependencies. Figure 1 depicts the main components of PRE and the following subsections describe its operation.

### 3.1 Entering Precise Runahead Execution

As in prior techniques, PRE is invoked on a full-window stall. First, PRE checkpoints the Register Allocation Table (RAT). The instructions filling the ROB can still execute as they would in normal mode. However, no instructions are committed from the ROB in runahead mode. Therefore, no updates are propagated to the ARF and the L1 D-cache. During runahead execution, PRE dynamically identifies the instructions that are part of potential stalling slices as they arrive from the decode unit (as described in the next section), and speculatively executes them.

### 3.2 Identifying Stalling Slices

PRE tracks the individual instructions that form a stalling slice in a new cache that we call the *Stalling Slice Table (SST)*. As Figure 1 shows, the SST is accessed after the decode stage in a typical out-of-order pipeline. The SST is a fully-associative cache that contains only instruction addresses (i.e., PCs). If an instruction address hits in the cache, that instruction is part of a stalling slice. Whenever a stalling load blocks the ROB, we store it in the SST. To facilitate tracking the chain of instructions that leads to that load, we extend each entry in the RAT to hold the PC of the instruction that last produced that register.

We track the stalling slices in an iterative manner. First, the stalling load is stored in the SST. When the stalling load is decoded again, e.g., in the next iteration of a loop, the PC of the stalling load hits in the SST. PRE checks the RAT entry for

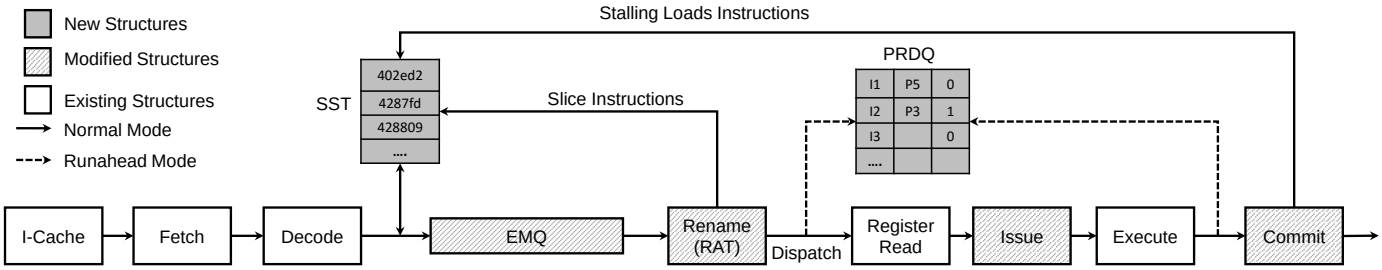


Fig. 1: Core microarchitecture for precise runahead execution.

the load's source registers to find the PCs of the instructions that last produced those registers; these PCs are then stored in the SST. Similarly, whenever an instruction hits in the SST in the following iterations, we track the PC information of its producer instructions and add those to the SST as well. By tracking all stalling slices in the SST, PRE does not limit prefetch coverage to a single slice as in the runahead buffer proposal.

### 3.3 Execution in Runahead Mode

PRE filters and speculatively executes all stalling slices that follow the stalled window using the SST. After instruction decode, PRE executes only the instructions that hit in the SST because they are necessary to generate future loads. PRE properly maintains dependencies among the executed instructions and manages the allocation and reclamation of registers in runahead mode (as described in the next section). PRE executes future stalling slices for the entire length of a runahead interval. The instructions executed in runahead mode are fetched and decoded again for execution in normal mode. However, to avoid wasting the work and energy of the front-end in runahead mode, we propose an Extended Micro-Op Queue (EMQ) as shown in Figure 1. We store all the decoded instructions (including the ones that hit in the SST) in the EMQ. When the processor resumes normal mode execution, it does not need to re-fetch and re-decode all these instructions again. These instructions are directly dispatched and executed in the back-end. Note that with this optimization, the number of speculatively executed instructions in runahead mode is constrained by the size of the EMQ. When the EMQ fills up, the core stalls until the stalling load returns, at which point the processor exits runahead mode.

### 3.4 Recycling Resources in Runahead Mode

PRE requires sufficient issue queue entries and physical registers to run ahead without discarding the instructions in the ROB. Our evaluation reveals that at the time of runahead entry, 37% of the issue queue entries, 51% of the integer registers, 59% of the floating-point registers are free on average. Stalling slices are usually short and therefore issue queue entries are quickly reclaimed and are unlikely to hinder forward progress of runahead execution. In all of our experiments, we did not come across instances of issue queue pressure during runahead.

In an out-of-order core, a physical register can be freed only when the last consumer of the renamed architectural register commits [7]. Since instructions in runahead mode are discarded after execution, we cannot rely on the original renaming policy to free physical registers. Thus, we devise a new mechanism, called *runahead register reclamation*, to free physical registers in runahead mode. This process relies on a new FIFO hardware structure that we name the *Precise Register Deallocation Queue (PRDQ)* in Figure 1. Each entry in the PRDQ has three fields: an instruction identifier, a physical register (tag) to be freed, and an 'execute' bit that marks whether the instruction has completed execution. PRDQ entries are allocated in program order at the PRDQ tail. Register renaming maps a free physical register to the destination architectural register of an instruction in runahead mode. We mark the old physical register mapped to the same (destination) architectural register in the PRDQ entry. A PRDQ entry is deallocated when the instruction is executed (i.e., 'execute' bit is set) and reaches the PRDQ head. PRDQ deallocation is done in program order. The old physical register associated with the instruction is freed upon deallocation. While instructions may execute out-of-order and thus mark the 'execute' bit out-of-order, in-order PRDQ deallocation guarantees that a physical register is freed only when there are no more instructions in-flight that may possibly read that register. The PRDQ is only enabled in runahead mode and its entries are discarded once the processor returns to normal mode.

### 3.5 Exiting Precise Runahead Execution

The core exits runahead mode when the stalling load returns. On exit, the checkpointed RAT is restored and execution resumes in normal mode. As instructions are preserved in the ROB, the core starts committing instructions right away starting from the stalling load.

### 3.6 Hardware Overhead

PRE relies on the SST and PRDQ to implement runahead execution. We find that a 256-entry SST holds stall slices with almost no misses. With 4-byte tags, the SST requires 1 KB storage. We conservatively provision the PRDQ to hold 192 entries, for a total of 768 Bytes. We extend each mapping of the 64-entry RAT by 4 bytes for a total of 256 Bytes. This leads to a total hardware cost of 2 KB. When employing the (optional) EMQ, the overhead changes according to the EMQ size. We show results with a 768-entry EMQ, adding an extra 3 KB. Runahead buffer requires about 1.7 KB and uses expensive CAM lookups in the ROB to find stalling slices. Overall, the hardware cost and complexity of PRE is comparable to the runahead buffer proposal.

## 4 Methodology

We evaluate PRE using the most accurate cycle-level core model in Sniper 6.0 [8]. The configuration for our baseline out-of-order core is provided in Table 1. For a fair comparison, we maintain the same ROB and issue queue sizes as in the runahead buffer proposal [4]. For a 192-entry ROB, we base the number of physical registers on the Haswell core [9], [10]. We select the same set of memory-intensive benchmarks, from SPEC CPU2006, as in runahead buffer [4] and we simulate 1-Billion instruction SimPoints [11] for each benchmark. The SST is modeled as a 256-entry fully-associative cache with 8 read and 2 write ports. We assume the front-end can deliver up to 8 micro-ops per cycle. We also evaluate the case with EMQ optimization using an EMQ of 768 entries or 4× the ROB size. The PRDQ and

TABLE 1: Baseline configuration for our out-of-order core.

| | |
|---|---|
| Core | 2.66 GHz out-of-order, ROB: 192,<br>Issue/Load/Store queue: 92/64/64,<br>Width: 4, Depth (front-end only): 8 stages |
| Register file | 168 int (64 bit), 168 fp (128 bit) |
| SST | 256 entry, fully assoc, LRU |
| PRDQ size | 192 |
| EMQ size | 768 |
| L1 I-cache | 32 KB, assoc 4, 2 cyc |
| L1 D-cache | 32 KB, assoc 8, 4 cyc |
| Private L2 cache | 256 KB, assoc 8, 8 cyc |
| Shared L3 cache | 1 MB, assoc 16, lat 30 cyc |
| Memory | DDR3-1600, 800 MHz<br>ranks: 4, banks: 32, page size: 4 KB<br>bus: 64 bits, `tRP-tCL-tRCD`: 11-11-11 |

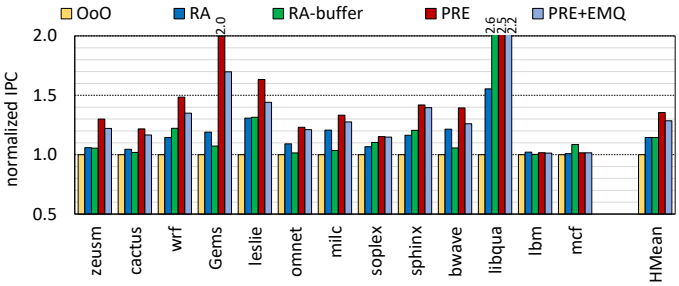


Fig. 2: Performance normalized to OoO.

EMQ are modeled as in-order queues with 4 read and 4 write ports each. We use McPAT [12] to calculate power assuming a 22 nm chip technology. We calculate power for the SST, EMQ and PRDQ using CACTI 6.5 [13] and we add those numbers to the core and DRAM power numbers calculated using McPAT.

## 5 Evaluation

We compare the performance and energy-efficiency of the following four runahead proposals compared to a baseline out-of-order (OoO) core:

- Runahead Execution (RA): Traditional runahead (Section 2.2) with optimizations from Mutlu et al. [6].
- Runahead Buffer (RA-buffer): See Section 2.3.
- Precise Runahead Execution (PRE).
- PRE with EMQ optimization (PRE + EMQ).

### 5.1 Performance

Figure 2 reports performance for all the runahead proposals normalized to the baseline out-of-order core. RA and RA-buffer improve performance by on average 14.5% and 14.4%, respectively. PRE yields a significantly higher performance improvement of 35.5%. PRE+EMQ improves performance by 28.6% — the length of a runahead interval is limited by the EMQ size. PRE achieves this improvement by avoiding the frequent pipeline flushing and refilling overheads incurred by RA and RA-buffer. This has the additional benefit of allowing PRE to invoke runahead execution for slices with relatively short runahead intervals, exposing more MLP. We find that PRE and PRE+EMQ invoke runahead execution 1.62× and 1.95× more frequently than traditional runahead. RA-buffer outperforms the other techniques in cases where only a single slice leads to all full-window stalls (e.g., `libquantum`). However, for the majority of benchmarks where more than one slice stalls the ROB frequently, precise runahead performs better than runahead buffer. This is because precise runahead can execute multiple slices upon every full-window stall.

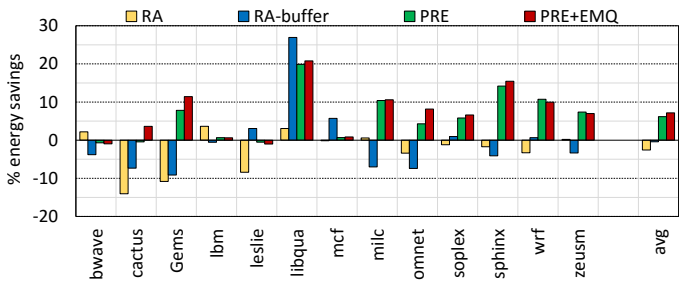


Fig. 3: Energy savings relative to OoO.

### 5.2 Energy Consumption

Figure 3 compares the energy savings (core plus DRAM) accrued by all mechanisms over the baseline. All four runahead proposals increase the dynamic instruction execution within the core, which increases power consumption. However, both RA and RA-buffer fetch, decode and execute an entire window worth of instructions twice due to flushing and refilling the pipeline. RA increases energy consumption of the core by 2.7% on average. On the other hand, PRE does not discard the ROB and PRE+EMQ preserves the work of the front-end in the EMQ. Overall, PRE and PRE+EMQ consume 6.1% and 7.2% less energy, respectively, compared to an out-of-order core. In contrast, RA-buffer does not provide any energy saving.

## 6 Conclusion

Runahead execution improves processor performance by uncovering future independent long-latency instructions upon full window stalls. We show that the performance of prior runahead proposals suffers from mandatory pipeline flush/restore overheads and limited prefetch coverage. We propose Precise Runahead Execution (PRE) to alleviate shortcomings of prior runahead techniques. Relative to an out-of-order core, PRE improves performance by 35.5% on average (compared to 14.4% for recent runahead proposals). In addition, PRE reduces energy consumption by 6.1%.

## References


[1] J. Dundas *et al.*, "Improving data cache performance by pre-executing instructions under a cache miss," in *ICS*, 1997, pp. 68–75.

[2] O. Mutlu *et al.*, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA*, 2003, pp. 129–140.

[3] M. Hashemi *et al.*, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *MICRO*, 2016, pp. 1–12.

[4] M. Hashemi *et al.*, "Filtered runahead execution with a runahead buffer," in *MICRO*, 2015, pp. 358–369.

[5] O. Mutlu *et al.*, "On reusing the results of pre-executed instructions in a runahead execution processor," *CAL*, vol. 4, 2005.

[6] O. Mutlu *et al.*, "Techniques for efficient processing in runahead execution engines," in *ISCA*, 2005, pp. 370–381.

[7] H. Tabani *et al.*, "A novel register renaming technique for out-of-order processors," in *HPCA*, 2018, pp. 259–270.

[8] T. E. Carlson *et al.*, "An evaluation of high-level mechanistic core models," *TACO*, vol. 11, no. 3, p. 28, 2014.

[9] P. Hammarlund *et al.*, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, pp. 6–20, 2014.

[10] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs," https://www.agner.org/optimize/microarchitecture.pdf.

[11] T. Sherwood *et al.*, "Automatically characterizing large scale program behavior," in *ASPLOS*, 2002, pp. 45–57.

[12] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009, pp. 469–480.

[13] S. Li *et al.*, "CACTI-P: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *ICCAD)*, 2011, pp. 694–701.