Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Markus Wang Halvorsen

Delayed runahead exit policies

Master's thesis in Computer Science (MTDT)
Supervisor: Rakesh Kumar
February 2024

NTNU
Norwegian University of
Science and Technology

Markus Wang Halvorsen

Delayed runahead exit policies

Master's thesis in Computer Science (MTDT)
Supervisor: Rakesh Kumar
February 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The memory wall, a growing gap between processor and memory clock speeds, represents a major bottleneck for processor performance in memory-intensive programs. Long latency loads (LLLs), loads that miss in the last level of cache, have latencies up to hundreds of cycles, causing the processor to stall. Modern out-of-order (OoO) cores can tolerate some memory latency owing to their large re-order buffer (ROB) sizes but are still too small to handle LLLs. Hardware prefetchers cannot always hide these latencies as they are, by design, unable to perfectly predict the future demand accesses of processors. The result is that many loads still cause the ROB to fill up and subsequently the processor to stall.

Runahead execution is a modern prefetching technique implemented in processor microarchitecture that utilizes cache miss stall cycles to execute the future instruction stream. Runahead prefetches critical loads with near-perfect accuracy, thus turning future cache misses into cache hits. However, previous works have found that the prefetching effect of runahead has limited coverage. This is partly because even though these stalls degrade performance in total, each individual stall period is too short for runahead to prefetch enough loads to achieve adequate coverage.

In this thesis, I study the effect that delaying exit from runahead mode has on processor performance by implementing a traditional runahead scheme in the gem5 computer architecture simulator. By simulating three different exit policies I find that delaying exit from runahead can improve overall processor performance by 2.3% compared to a runahead processor that eagerly exits runahead.

Sammendrag

Spriket mellom klokkehastigheten til prosessorer og hovedminne utgjør en stor flaskehals i prosessorytelse for minneintensive programmer. Denne ytelsesbegrensningen skyldes langtidsinnlastere, innlastingsinstrukser som utfører minneaksesser som ikke treffer i siste nivå av prosessorens hurtigbufferhierarki. Disse instruksene har ofte ventetider på flere hundre sykluser. Moderne prosessorkjerner kan tolerere noe av ventetiden til innlastingsinstrukser ved å eksekvere instrukser utenfor rekkefølge (OoO) og spore rekkefølgen deres i store omordningsbuffere, men disse er fremdeles for små til å håndtere langtidsinnlastere. Maskinvarebasert forhåndsinnhenting av minneverdier kan ikke alltid skjule ventetiden for aksesser til hovedminne fordi slike teknikker ikke kan forutsi fremtidige minneaksesser med perfekt nøyaktighet. Sluttresultatet er at omordningsbufferen ofte fylles opp mens prosessoren venter på minne, og dermed at prosessoren blokkeres.

Forutløpende eksekvering er en moderne teknikk for forhåndsinnhenting av minneverdier. Teknikken er implementert i prosessorens mikroarkitektur og tillater prosessorkjernen å utnytte sykluser hvor prosessoren venter på minne. Syklusene utnyttes for å eksekvere fremtidige instrukser som henter inn minneverdier på forhånd med svært høy nøyaktighet. På denne måten blir bom i hurtigbufferen omgjort til treff. Tidligere arbeider har funnet at minneinnhentingseffekten til forutløpende eksekvering dekker et begrenset antall innlastingsinstrukser. Dette skyldes delvis at selv om ventetidene for minne totalt sett skader prosessorytelsen, vil hvert individuelle minneaksess ta så kort til at forutløpende eksekvering ikke har nok tid til å hente inn mange nok minneverdier.

I denne masteroppgaven undersøker jeg effekten av å forsinke byttet fra den forutløpende eksekveringsmodusen tilbake til normal modus. Dette gjøres ved å innføre tradisjonell forutløpende eksekvering i en OoO-prosessormodell i datamaskinarkitektursimulatoren gem5. Ved å simulere tre ulike modeller for forsinket modusbytte viser jeg at en prosessor med støtte for forutløpende eksekvering kan forbedre ytelsen med 2.3% hvis den forsinker byttet ut av forutløpende eksekveringsmodus.

Preface

This thesis builds on the work done for TDT4501, my specialization project, during the spring semester of 2023. I feel lucky to have been able to work on such an interesting and modern development in processor microarchitecture that goes to the heart of one its largest challenges.

I want to thank my supervisor, Rakesh, for his patience, understanding and assistance in finding relevant literature, devising a plan, and asking critical questions of my work. Runahead has been difficult to work with, and many meetings have ended in a-ha moments and fresh insights without which I do not think I could have seen this project through.

I would also like to thank Amund, a PhD student at the institute, for his assistance with configuring gem5 to run the SPEC2017 benchmarks and profile them for regions of interest. Working on setting these up entirely on my own would have taken a significant amount of time.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

OoO	Out-of-order execution
ROB	Reorder buffer
SPEC2017	SPEC CPU2017 benchmarks
HWP	Hardware Prefetcher
LLL	Long latency load
RAS	Return address stack
R-cache	Runahead cache
PRF	Physical register file
RRT	Register renaming table
DC	Dependency chain
SRSL	Source register search list
PRE	Precise runahead
SST	Stalling slice table
PRDQ	Precise register deallocation queue
CRE	Continuous runahead
VRE	Vector runahead
ISA	Instruction set architecture
PC	Program counter
FTDQ	Fetch-to-decode queue
IEW	Issue, execute, writeback
IQ	Issue queue
LSQ	Load-store queue
Opclass	Operation class
FU	Functional unit
AOL	Age order list
L*n(-I/D)* cache	Level *n* (instruction/data) cache
RCL	Runahead-causing load
NLLB	No Load Left Behind
IFT	In-flight threshold
BTB	Branch target buffer
ROI	Region of interest
IPC	Instructions per cycle
NIPC	Normal mode instructions per cycle
L2U	Load to use

1 Introduction

1.1 The memory wall

The clock speed of modern processors has increased at a rapid pace. Meanwhile, the speeds of memory systems have increased at a lesser pace, leading to the CPU cycle latency between the processor and main memory growing ever larger. This problem, known as the memory wall, significantly impacts the performance of most programs because memory accesses are frequent and can take hundreds of cycles to complete if they must be serviced by main memory. For example, approximately 50% of all dynamic instructions in the popular SPEC CPU2017 benchmarks[1] perform operations that either use or write to values located in memory[2].

Cache hierarchies, illustrated in Figure 1.1, allow processors to significantly reduce the latency of many memory accesses. These low-latency, on-chip data stores can serve memory accesses in a fraction of the cycles of a main memory access if the data is present in the caches, in other words if there is a cache hit. Hardware prefetchers (HWP) augment cache hierarchies with the ability to find patterns in memory accesses in an autonomous and transparent fashion. These access patterns can be used to prefetch data into cache ahead of their use. Combined, caches and prefetching exploit temporal and spatial locality, wherein code frequently accesses the same memory locations, or locations close together in memory. However, cache misses still occur frequently due to cold caches, cache mapping conflicts and insufficient capacity. Additionally, HWPs often struggle to prefetch complex or unpredictable memory access patterns such as indirections.

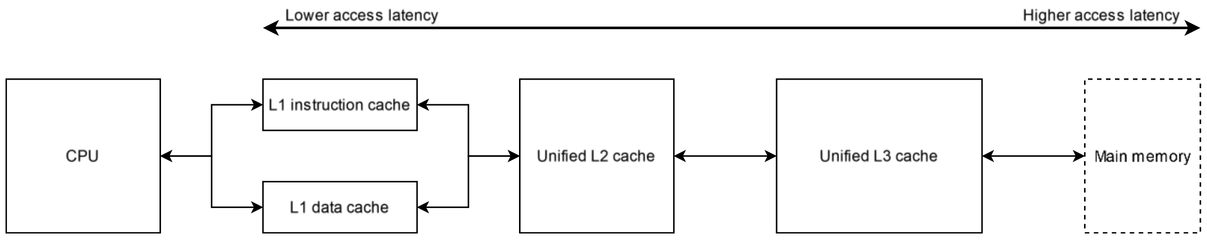


Figure 1.1: A three-level cache hierarchy with a split L1 cache and unified L2 and L3 caches. Memory access latency increases the further down the memory hierarchy a data packet must travel.

Out-of-order (OoO) processor microarchitecture can hide some latency by continuing to queue and execute instructions as long as there is space in the re-order buffer (ROB). The ROB enables instructions to execute out of program order while committing their changes to architectural state in-order to support precise exceptions[3], but the structure is limited in size due to cost and space considerations. If the ROB fills up, the processor cannot track any new instructions, causing a stall. The processor is unblocked only when the oldest instruction is retired. This architecture is helpful in hiding execution latencies, including the lower latencies of memory accesses that hit in cache, but instructions that miss in the last level of cache have such long latencies that they frequently cause the ROB to fill completely[4], and therefore the processor to stall.

1.2 Runahead execution

Runahead is a speculative execution technique which executes future loads, typically when the processor would otherwise be stalled on a last level cache miss. By executing the full dependency chain of load instructions, runahead prefetches future memory accesses with high accuracy. The function and effect of runahead is illustrated in Figure 1.2. In a normal processor, loads that miss in the last level of cache typically cause the ROB to fill up, stalling the processor until the load returns. Instead of stalling, a runahead processor can decide to enter *runahead mode*, in which subsequent loads leading to cache misses are speculatively pre-executed, turning them into cache hits.

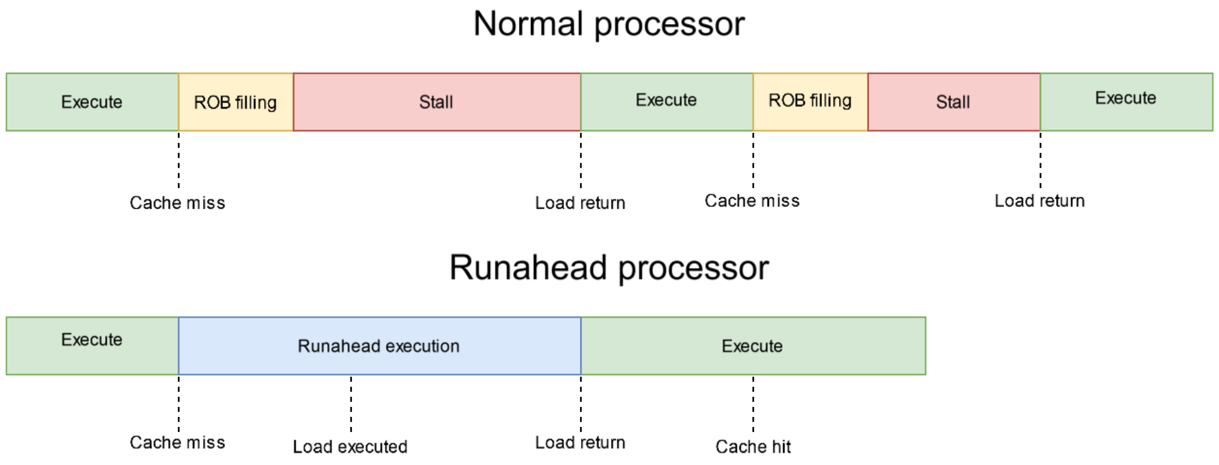


Figure 1.2: Example of how runahead execution prefetches cache misses by pre-executing future loads during cache misses. The pre-execution of the load prefetches the memory value, preventing a future cache miss that would have led to a stall.

The term runahead was coined in 1996[5], but the technology did not see the beginning of its modern development until 2003[4]. Research in the early 2000s was mostly focused on efficiency improvements[6-9], but runahead has since seen major performance improvements[10-12] and even works that leverage hardware acceleration to perform runahead execution[13, 14].

Most existing runahead schemes attempt to minimize time spent in runahead while attempting to utilize the available time as efficiently as possible. Many efficiency improvements have focused on limiting entry into runahead where it is not predicted to be useful[6-8], and nearly all schemes will eagerly exit runahead as soon as the cache miss that caused runahead has returned[4, 9, 10, 12]. Intuitively, this makes sense. Runahead performs useful work by prefetching memory accesses, but it does not perform real work. However, eagerly exiting runahead comes at the cost of reduced prefetch coverage because fewer loads are executed. Recent works has discovered that runahead has poor coverage[14], and that delaying the exit of runahead execution can actually improve overall performance in vectorized runahead[11].

1.3 Delayed exit runahead

Inspired by previous findings about runahead coverage[14] and performance improvements from delayed exits[11], this thesis studies if delayed exit policies from runahead may improve either the efficiency or performance of runahead schemes. A traditional runahead execution CPU model was implemented in the gem5 simulator and used to simulate the SPEC CPU2017 benchmarks[1] (SPEC2017). The implementation of the runahead model used in this thesis is open source and freely available on GitHub[15].

Figure 1.3 illustrates how a delayed exit may prefetch additional long latency loads, thus preventing repeated re-entry into runahead later. Loads executed in the delayed runahead period may increase runahead's coverage, leading to improved processor performance. Additionally, if the additional prefetches prevent a runahead period and are performed by executing fewer instructions than said runahead period, efficiency is increased. Depending on how costly the overhead of entering and exiting runahead is, the saved overhead from reduced runahead may also provide performance benefits.

In this thesis, I develop three delayed exit policies for runahead with the aim of improving a runahead processor's performance. I find that delaying exit from runahead mode can improve performance by 2.3% over traditional runahead with an eager exit policy if the additional time is spent effectively to issue additional loads. This suggests that delayed exit policies have the potential to increase the performance of traditional runahead schemes.

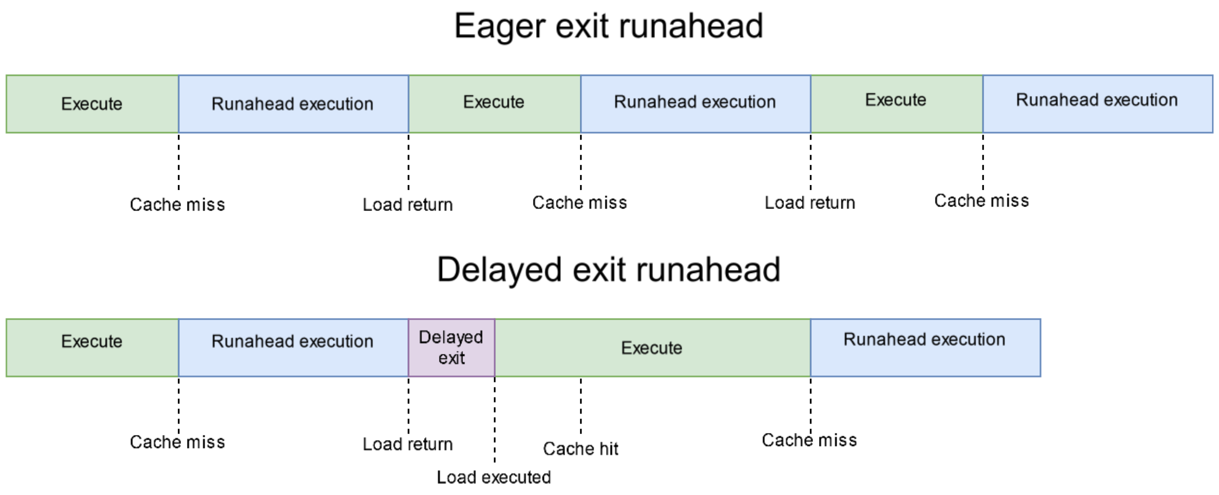


Figure 1.3: Example of how delayed exit from runahead may prefetch additional loads. The extra prefetch during the delayed exit prevents a re-entry into runahead.

1.4 Structure of the thesis

This thesis is divided into chapters roughly relating to the project background, motivation, simulator details, implementation of delayed exit runahead, evaluation methodology, experiment results, a discussion and finally recommendations for future work and a conclusion.

First, chapter 2 explains the background for the project, beginning with the traditional measures taken against long latency loads. Runahead execution in its various forms is then explained. Select runahead schemes from literature are presented and explained. Chapter 3 discusses the motivation for the work done in this thesis, including evidence supporting delayed exit policies' potential to improve performance. Chapter 4 briefly touches on the choice of simulator for the study. I then describe the gem5 simulator and its stock OoO CPU model, O3CPU. Each stage of the processor model's pipeline is described in detail. Chapter 5 details how gem5's O3CPU model was modified to support traditional runahead execution. It also describes some of the improvements from literature that were implemented with the model. The implementation of my delayed exit policies are also detailed here. Finally, it explains how the model was tested for functionality and correctness. Chapter 6 discusses the SPEC2017 benchmarks used to evaluate the CPU models in this project. It explains how gem5 was configured to use the benchmarks as well as the base system configuration used. The process of tuning runahead parameters for the runahead baseline is then discussed. The method for evaluating system performance is also explained. Chapter 7 presents the simulation results of the runahead baseline model compared to the stock OoO CPU model. It also presents the simulation results for each delayed exit policy compared to an eager exit policy. Chapter 8 discusses the performance degradation seen in the runahead model. The results for each delayed exit policy are then discussed. Chapter 9 provides suggestions for improving the processor model and a potential avenue for future work on delayed exit policies. Finally, chapter 10 concludes the thesis by summarizing the work done, the experimental results and explanations for these.

2 Background

2.1 Cache hierarchies and prefetching

The main measure taken against long memory latencies is to structure memory hierarchically, with caches close to the CPU. Caches are small, on-chip memory units that have exceptionally low access latencies compared to main memory. Any memory operation executed by the CPU which hits in cache sees a dramatic speedup compared to a main memory access. Cache misses have become a bottleneck for modern processor performance as main memory accesses can take hundreds of cycles to resolve.

Typically, caches are made using SRAM technology, which is costly in terms of both money and physical space, both of which are coveted resources in microarchitectural design. As such, designers must strike a balance between the performance benefit and overall cost of incorporating the caches. Much effort has also been put into hardware prefetching schemes as a method to increase cache hit rates and hide access latencies.

The matrix multiplication algorithm shown in Figure 2.1 is a good example of how caches can improve performance while highlighting their limitations. Each matrix row and column are accessed repeatedly. The first accesses are not cached, and therefore end up being cold cache misses, incurring lengthy main memory accesses. Subsequent accesses to the same matrix rows and columns will hit in cache, speeding up execution. If the matrices are sufficiently large, the cache might not have sufficient capacity to store the full matrices, leading to capacity misses as older cache entries must be evicted to fit the data accessed in more recent accesses. Depending on how the cache blocks are indexed and how the matrix is laid out in memory, some matrix accesses may also map to the same cache blocks, leading to conflict misses.

```
for (int rA = 0; rA < numARows; rA++)
    for (int cB = 0; cB < numBCols; cB++)
        for (int cA = 0; cA < numACols; cA++)
            C[rA][cB] = A[rA][cA] * B[cA][cB]
```

Figure 2.1: A simple 2D matrix multiplication algorithm for computing the matrix C = AB.

In the example algorithm, cache exploits what is known as temporal locality, in which the algorithm accesses the same addresses repeatedly. The code (and indeed a very large amount of code in general) also accesses data in a predictable pattern. This phenomenon is known as spatial locality and can be exploited by hardware prefetchers to further increase performance. Figure 2.2 shows the intuition behind how a hardware prefetcher works. Because the processor is performing accesses to different addresses in a predictable pattern, a hardware prefetcher can issue prefetch requests down the memory hierarchy to begin loading data into cache before it is demanded by the CPU, thus hiding parts of or even the full latency of the access.

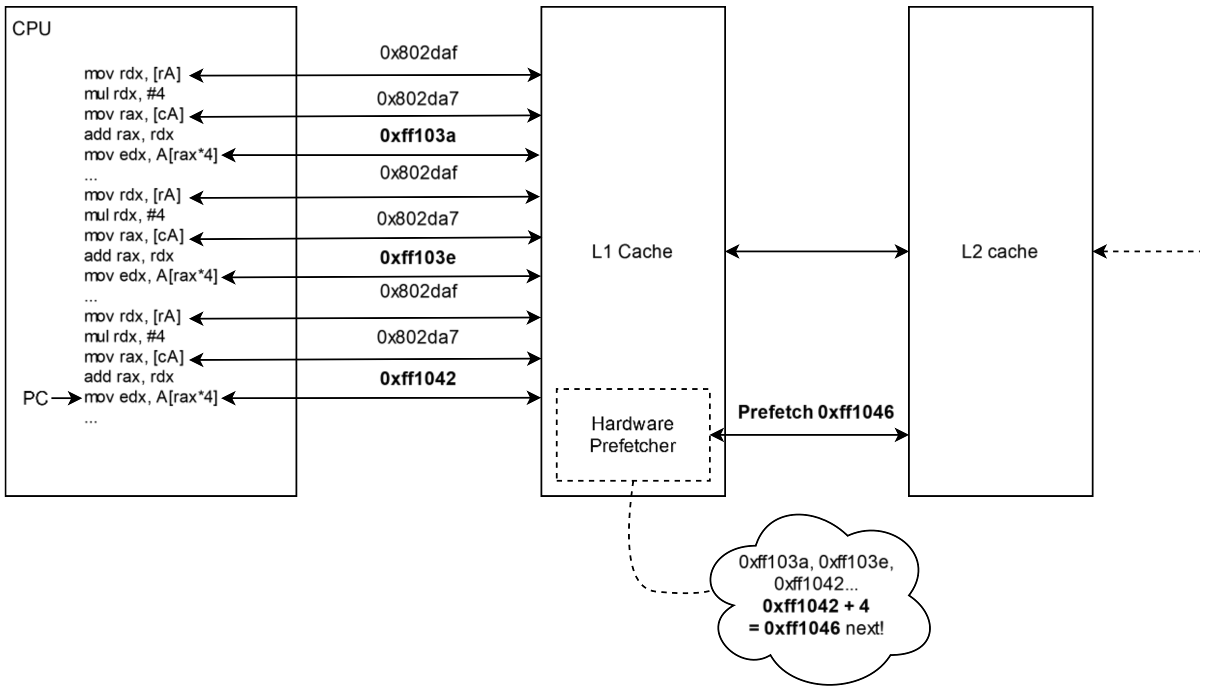


Figure 2.2: Simplified example of how a hardware prefetcher might predict access patterns and move data up the memory hierarchy ahead of their use.

Hardware prefetchers can dramatically increase processor performance. The winner of the 3rd data prefetching championship, the Instruction Pointer Classified based Prefetcher, improved single-core IPC by 43.75% over a CPU model without prefetching[16]. However, all prefetchers have an inherent limitation in that they can only base their prefetches on past and present data. This means there is an upper bound to their accuracy and coverage.

2.2 Runahead execution schemes

Runahead is a broad term that covers many different microarchitectural designs. Despite originally and typically being used to improve cache performance[4-12, 14, 17-20], runahead has been found to have more use cases[13, 21]. As such, this section begins with a broad description of runahead as a framework for generating side-effects. Select runahead schemes from published literature which are particularly relevant to the thesis are then presented and explained. Finally, the performance improvements of these schemes are summarized.

2.2.1 The runahead framework

A normal out-of-order processor cannot process the instruction stream any farther than the current committed instruction, plus the size of the ROB. When the ROB is full, the program order of new instructions cannot be tracked, and the processor must stall. Runahead processors can execute arbitrarily far into the instruction stream by circumventing the usual limitations that ensure architectural state is observably correct at any time. Runahead execution does not necessarily fundamentally differ from normal execution, but any changes made to architectural state by runahead execution will inevitably be undone such that the processor's architectural state returns to the last point at which it was executing normally. Therefore, runahead processors do not necessarily have to stall when the ROB is full. Instead, blocking instructions can be removed to resume execution. Free resources in the processor may also be used to continue execution without utilizing the ROB. It is also possible to dedicate specialized hardware to runahead while not using any of the CPU's resources.

Because all the architectural effects of runahead are eventually discarded, runahead mode can be thought of as a sandbox in which the processor's main goal is to generate as many beneficial side-effects as possible. In most work thus far, runahead has been used to improve cache performance[4-12, 14, 17-20], but it can also be used to pre-compute branch outcomes[13] and even improve soft error rates[21] to increase processor reliability. Figure 2.3 shows a taxonomy of the goals of some runahead schemes that have been published in literature.

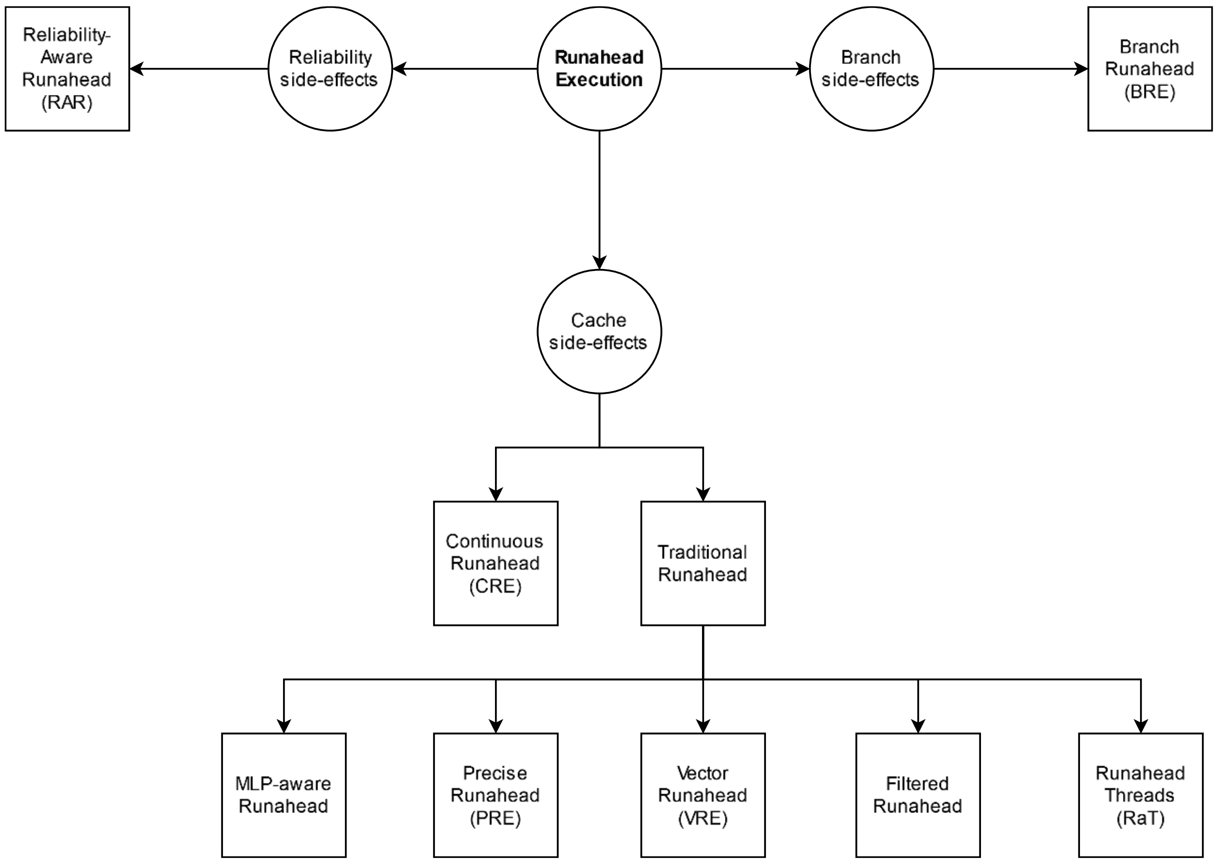


Figure 2.3: A taxonomy of different published runahead schemes. The majority of runahead schemes target cache, with most of those being based on the traditional runahead scheme introduced by Mutlu et al.[4]. MLP-aware[8], RaT[9, 17, 18], reliability-aware[21] and branch runahead[13] are not explained in this thesis.

In schemes that runahead in-processor, it is normal for runahead execution to begin when a memory access misses in last level cache. Such *long latency loads* (LLLs) are known to take a long time to resolve, and therefore typically cause the ROB to fill up and the processor to stall. Since the processor is doing no useful work otherwise, these stalls are an ideal time to use runahead mode. In truth, runahead can be entered at any time, including all the time in specialized hardware acceleration units[13, 14]. To my knowledge, however, when used in-processor runahead mode has only been used when there is a LLL blocking retirement at the head of the ROB.

2.2.2 Traditional runahead execution

In this thesis, traditional runahead refers to the scheme proposed by Mutlu et al.[4], optionally with the efficiency improvements later proposed in Mutlu's dissertation[6, 7]. Figure 2.4 shows a simplified traditional runahead CPU pipeline. This is largely the runahead scheme on which the runahead CPU model in this thesis is based, although there are certain aspects to modelling a functional runahead CPU which are not discussed in Mutlu et al.'s paper[22]. These nuances are discussed further in chapter 5.

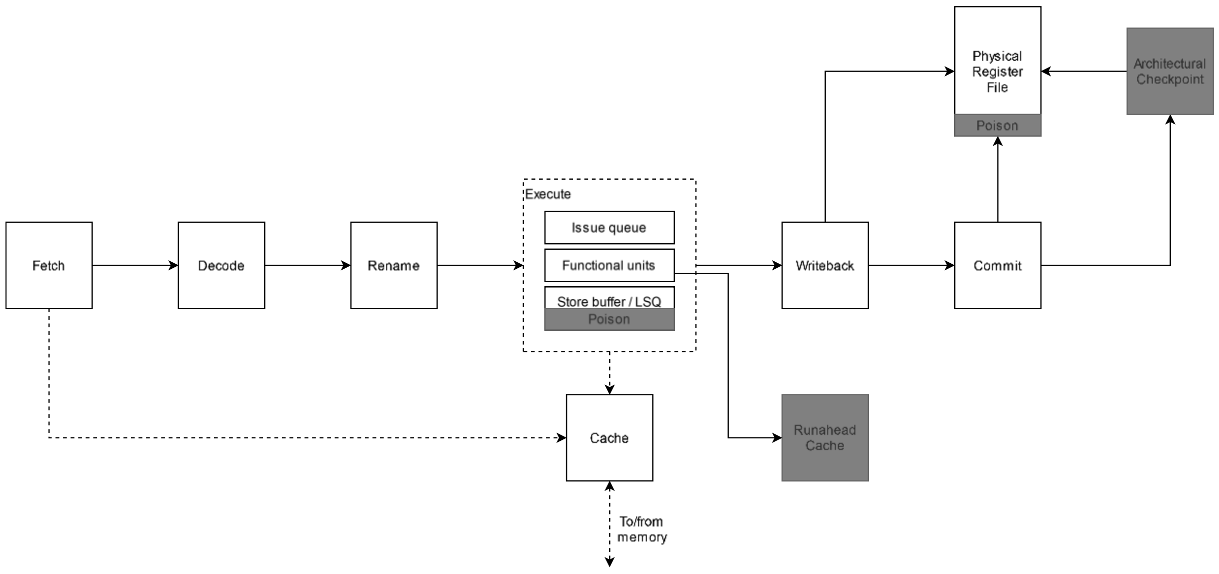


Figure 2.4: Diagram of a runahead CPU pipeline. Modifications to allow for traditional runahead execution are marked in gray.

Traditional runahead begins when a load misses in last level cache[1], at which point the processor stores a checkpoint of the architectural state. The checkpoint contains all architectural register values as well as the branch predictor's branch history and return address stack (RAS). Of these, only the register values are critical to restoring the architectural state and ensure program correctness. Once the checkpoint is taken, the instruction at the head of the ROB (a LLL) is issued a forged result and its destination register(s) are marked as *poisoned* before the instruction is *pseudoretired*, marking the start of a runahead period.

Once in runahead, the CPU works as usual, but must track poisoned data along the way. In the context of runahead, poison is equivalent to invalidity. An instruction using poisoned data is operating on invalid operands, i.e., the source operands do not contain data produced by valid program execution. In hardware, poison can be tracked by a bit for each physical register. Poison bits are propagated when instruction writeback. An instruction that sources poisoned registers will poison all its destination registers. Poison can be cleared if an instruction writes to a register or memory location without sourcing any poisoned operands, for example when writing or storing immediate values.

To handle memory instructions, the store buffer or LSQ (whichever is used for store-to-load forwarding) is augmented with a poison bit for each entry, allowing forwarded loads to become poisoned by stores in the buffer. Committed stores, however, are trickier. Because runahead is speculative, stores cannot be allowed to reach cache since memory

[1] The original paper specifies a miss in L2 cache, but L2 is last level cache in their system configuration.

is part of the committed architectural state of the system. Previous works had simply discarded these stores[5, 23], but one of Mutlu et al.'s major contributions is the addition of a *runahead cache* (R-cache)[4]. R-cache is a very small, on-chip cache that acts as a replacement for real cache during runahead. It exclusively stores the results of runahead stores. This way committed runahead stores can communicate their data and, if needed, propagate poison to dependent loads after they have exited the instruction window. When a load executes, it accesses R-cache in parallel with real cache, prioritizing the use of data returned by R-cache.

Any loads executed during runahead are prefetched if their address is valid. If the source operands are poisoned, however, the address will be invalid, and the instruction is guaranteed to produce a bogus result. This is true for any instruction that sources poisoned operands, and so they can safely skip execution and be sent directly to commit for pseudoretirement. The only exception is store instructions, which will write a poison bit into R-cache if their address is valid. Control instructions pose an additional challenge, as poisoned source operands cause their outcome to be impossible to determine. In these cases, the processor must trust the branch predictor. If the branch predictor is incorrect, it causes a *divergence point*[4] after which the processor is on the wrong control path until runahead exits.

When the LLL that originally caused the processor to enter runahead gets a response from cache, the processor begins to exit runahead. Mutlu et al. note that this can be handled in the exact same way as a branch misprediction squash. Because the runahead-causing LLL was the oldest instruction in the window at the time of entry into runahead, squashing until that LLL will flush the entire pipeline. After the flush, the architectural checkpoint is restored by copying every saved architectural register value into a specific portion of the physical register file (PRF). The register renaming tables (RRTs) are then repaired by overwriting both the front- and backend RRTs with a fixed rename table that maps to the correct registers in the PRF. R-cache is invalidated entirely, and the checkpointed branch history and RAS are restored. Fetch is then redirected to the LLL that caused runahead, and the processor resumes execution in normal mode.

Some simple efficiency improvements to the scheme were proposed by Mutlu in his dissertation[6, 7], and were motivated by the observation that the increase in processor performance came at the cost of a large increase in executed instructions. Three of these improvements are illustrated in Figure 2.5. To reduce the number of executed instructions, the processor may eliminate short, overlapping, and useless runahead periods. Short periods are eliminated by determining a cycle threshold after which in-flight loads can no longer trigger runahead because they are expected to return soon. Overlapping periods are eliminated by preventing re-entry into runahead until the processor has fetched at least as many instructions as were pseudoretired in the last runahead period. Eliminating useless runahead periods is a more open problem for which Mutlu trained a usefulness predictor that determined whether a runahead period was likely to generate useful prefetches, and preventing runahead when it was predicted not to.

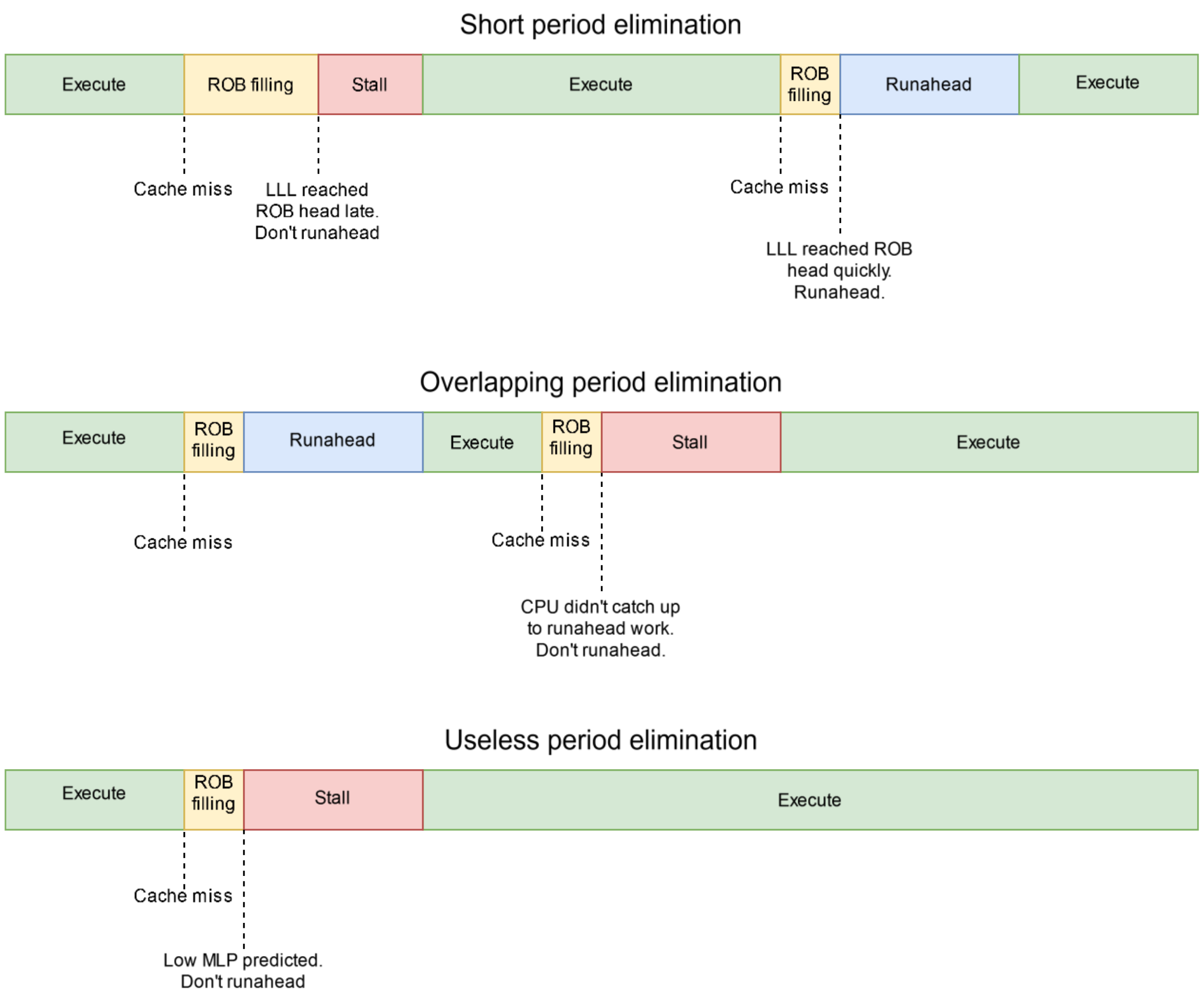

Short period elimination
Execute
ROB filling
Stall
Execute
ROB filling
Runahead
Execute
Cache miss
LLL reached ROB head late. Don't runahead
Cache miss
LLL reached ROB head quickly. Runahead.
Overlapping period elimination
Execute
ROB filling
Runahead
Execute
ROB filling
Stall
Execute
Cache miss
Cache miss
CPU didn't catch up to runahead work. Don't runahead.
Useless period elimination
Execute
ROB filling
Stall
Execute
Cache miss
Low MLP predicted. Don't runahead

Figure 2.5: Examples of how Mutlu's efficiency improvements[6, 7] can prevent entry into runahead, reducing the overall amount of instructions executed by the processor, improving efficiency and reducing overhead.

Mutlu et al. already mentioned in their paper on traditional runahead[4] that delayed exits were being evaluated. To my knowledge, these results were never published. However, Mutlu later evaluated and discussed briefly the feasibility of delaying exit from runahead in his dissertation[6] and PhD defense[7]. He found that delaying exit after the RCL returns may improve performance if the processor prefetches additional LLLs, but that failing to do so degrades performance. Overall, they found it to degrade performance[6].

2.2.3 Filtered runahead

Filtered runahead was proposed in 2015 by Hashemi and Patt[12]. They find that many static LLLs are not distinct from one another, i.e., they share the same PC. Motivated by this, they filter the runahead instruction stream to only the chain of instructions which generates addresses for the LLL blocking the ROB. This *dependency chain* (DC) is then stored in a *runahead buffer* which replaces the front-end entirely during runahead mode. A small DC cache is added to skip chain generation for future instances of the same static load instruction. Figure 2.6 shows how a traditional runahead CPU pipeline is altered further to enable filtered runahead.

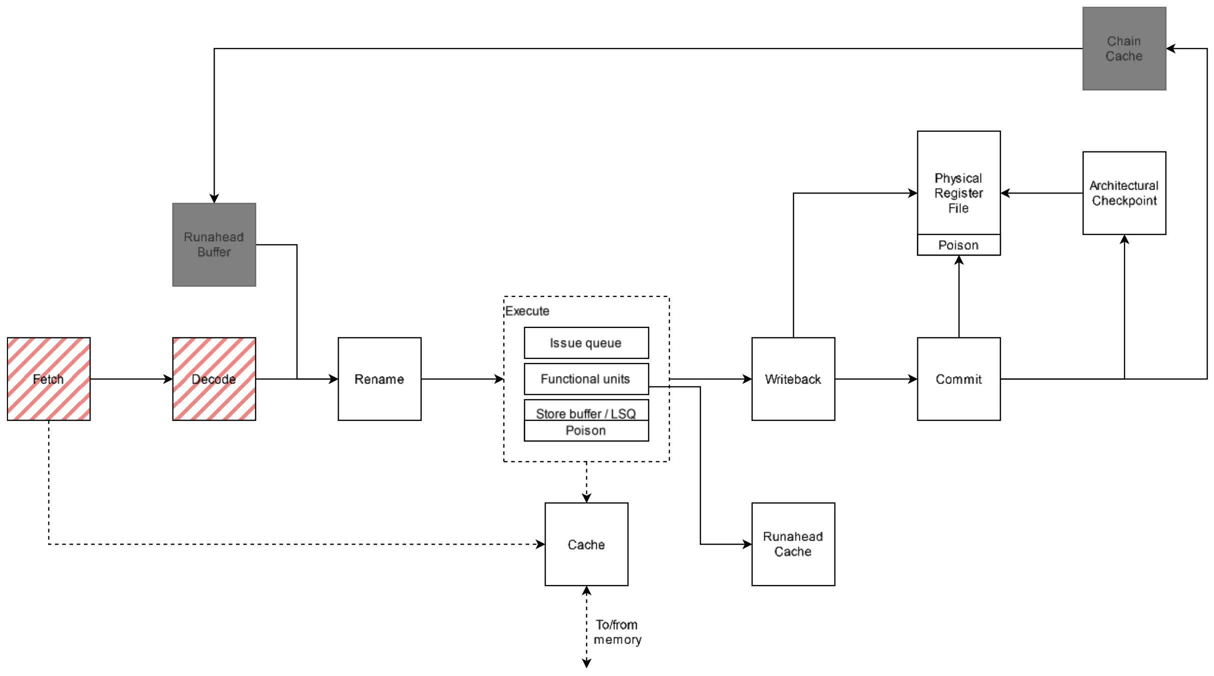


Figure 2.6: Diagram of a filtered runahead CPU pipeline. Modifications compared to traditional runahead are marked in gray. The runahead buffer replaces the fetch-decode frontend entirely during runahead, indicated by the hatching.

The core of filtered runahead are the dependence chains executed during runahead. These chains are generated on entry into runahead, and generation begins by looking for a younger instance of the LLL that is blocking the ROB. If one is found, the entire DC of the LLL must be present in the ROB. The processor then enqueues the younger LLL’s physical source registers in a source register search list (SRSL)[12]. It then iteratively searches the ROB for producers of each register in the SRSL, enqueuing new source registers for each producer found and adding them to the DC. Chain generation ends when the SRSL is empty, or the chain exceeds a certain maximum length (32 instructions in Hashemi and Patt’s paper[12]). Chains that exceed the maximum length are discarded.

Generated DCs are stored in the DC cache, which contains the decoded microoperations (uops) of the DC. The chain cache is indexed by the PC of the LLLs and uses a LRU eviction policy. When the processor enters runahead, the chain cache is checked for a matching DC. If found, DC generation is skipped, and the stored chain is moved into the runahead buffer.

While in runahead, the runahead buffer replaces the front-end entirely if it contains a dependence chain, meaning fetch and decode can safely be disabled through clock or power gating. The runahead buffer feeds decoded uops from the current DC directly into the rename stage, after which the uops flow through the pipeline as usual. Once the chain is exhausted, the buffer loops back to start and begins supplying uops from the beginning of the chain again. If the runahead buffer does not contain a chain at the start of runahead, the processor falls back to traditional runahead.

2.2.4 Precise runahead

Precise runahead (PRE)[10] is a state-of-the-art runahead scheme for processors without vector hardware. The scheme was published in 2020 by Naithani et al. and greatly improves performance and efficiency through eliminating much of the entry and exit overhead of runahead. It does this with a novel instruction filtering and resource reclamation mechanism, and by using only free resources during runahead. Figure 2.7 shows the pipeline modifications necessary to support precise runahead.

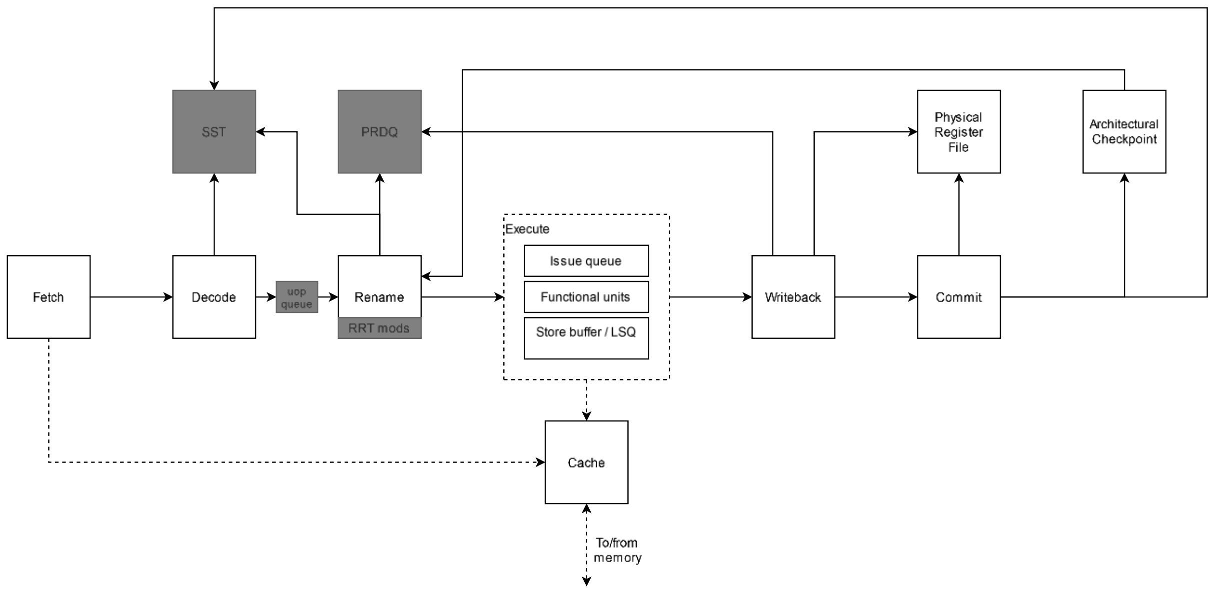


Figure 2.7: Diagram of a precise runahead pipeline. Modifications compared to a traditional runahead pipeline are marked in gray. The processor does not use the commit stage for runahead work. Note that the runahead cache and poison bits are removed.

PRE enters runahead once the ROB is full, and executes only the dependence chains of LLLs causing full-ROB stalls. These LLLs are called *stalling loads*, and PRE names their dependence chains *stalling slices*. Compared to filtered runahead the chains are not cached, and the frontend is not replaced during runahead. Instead, PRE introduces a *stalling slice table* (SST) between the decode and rename stages. The SST contains the PCs of instructions in stalling loads' dependence chains. Slices are iteratively generated and placed in the SST through the decode and rename stage, whose RRT is modified to contain the PC of each register's previous producer. Stalling loads are placed in the SST, and whenever an instruction is decoded, it checks the SST. If there is a hit, the frontend RRT is looked up to find the producer PC of each of the instruction's source registers. This PC is added to the SST. The SST is limited in size and uses a LRU eviction policy, allowing new slices to populate the SST as the program progresses. SST entries do not associate with any instructions or slices, and multiple slices can fit in the SST at once.

While in runahead, the processor uses the SST to filter incoming instructions after the decode stage. If the decoded instruction does not hit in the SST, it is not allowed to progress through the pipeline. Slice instructions that make it past decode only use unused registers for rename. Runahead instructions in PRE are discarded immediately after execution. PRE introduces a ROB-like structure, the *precise register deallocation queue* (PRDQ), to free registers after execution. The PRDQ stores information about old physical destination registers to be freed. Instructions are inserted into the PRDQ in-program-order and removed in-order when they have been executed and reach the head of the PRDQ. Any PRDQ entries are removed when the processor exits runahead.

One notable feature of PRE is that this scheme allows the processor to continue executing normal instructions while the processor is running ahead (if able). The ability for the SST to store multiple slices also means prefetch coverage is increased over the somewhat similar filtered runahead scheme. PRE also stores decoded runahead instructions in an extended uop queue between decode and rename, allowing for instruction reuse when the processor eventually exits runahead.

As with traditional runahead, PRE exits runahead when the stalling load returns. Because resources were never deallocated for runahead execution, the frontend is simply redirected to the checkpointed PC, which is the instruction immediately past the full ROB. The RRT and RAS must still be restored from the checkpoint, however. Note that PRE does not touch the ROB, so the processor can simply retire the front of the ROB as usual to continue execution while the front-end catches up.

2.2.5 Continuous runahead

Continuous runahead (CRE)[14] will only be briefly explained as it is a hardware acceleration scheme, and its relevance to this thesis is limited to its findings about runahead's coverage. The scheme was published by Hashemi et al. in 2016. The paper's main contribution is moving the filtered runahead scheme to a dedicated hardware acceleration unit called the continuous runahead engine. The processor communicates dependency chains of LLLs to the runahead engine, which continuously executes the chains in a loop to prefetch loads.

The development of CRE was motivated by the finding that traditional runahead has poor prefetch coverage, only reaching about 13% of *runahead-reachable* cache misses[14]. Hashemi et al. define runahead-reachable misses as those whose source data is available on-chip at issue. In other words, a runahead-reachable miss could have been prefetched if it had been executed in runahead. When the runahead engine is installed in the memory controller, prefetch coverage increases to 70% of reachable misses. They find that the low coverage of these misses is largely due to each traditional runahead interval lasting for a short time. CRE's coverage improvement comes from its ability to run ahead constantly, something which is not reasonable to do in the processor core.

2.2.6 Vector runahead

Vector runahead (VRE)[11, 19, 20] is the current state-of-the-art runahead scheme for processors with vector hardware. Naithani et al. find that many indirections occur in loops with predictable outcomes for each iteration. They capitalize on this by unrolling loops into vector instructions (vectorizing) to simultaneously issue large amounts of loads during runahead. Doing this enables runahead to resolve indirections effectively, something previous schemes struggled with.

The vectorization mechanism of VRE depends on stride detection to both determine when LLLs' dependency chains can be vectorized, and what offset to use when vectorizing. To find this offset, a prediction table[24, 25] is used to track the delta between memory addresses of the same load PC. This detects strides and tracks the confidence that the stride is accurate. A prediction table entry with high confidence indicates a striding load, which allows VRE to vectorize the load chain in runahead. Lastly, a terminator PC[11] is stored to determine the end of the load chain.

VRE enters runahead when the ROB is blocked by a load, and the ROB is full or the IQ is 80% full. If the blocking load is not vectorizable, PRE[10] is used. Vectorization is done by injecting 512-bit vector loads into the pipeline using the LLL's memory address and the detected stride. Any arithmetic operations or producing loads in the LLL's dependency chain are also vectorized. To handle poison, a taint vector[11] is used to track it and mask vector lanes during execution. Control flow instructions are vectorized once per iteration and actioned in the form of a lane mask where all lanes that take the same direction as the first lane are executed. Register resources are handled by a deallocation queue similar to the one used in PRE[10].

Unlike previous schemes, VRE does not exit runahead when the original LLL returns. It instead relies on four separate conditions which can terminate runahead. However, in some circumstances VRE can continue past these. VRE dynamically computes the amount of loop iterations to unroll and execute in a process called *vector unrolling*[11] which works like this: One iteration of the loop is executed first. Then, a vector load is issued for the next N=8 stride values in the sequence[2]. This process is then repeated a variable number of times, U. Ultimately, the loop is unrolled U times, each of which contains N values to load. This means that runahead issues N times U iterations worth of loads in one runahead period before the core is allowed to resume normal execution. If N=8 and U=8, VRE will unroll the loop into 8 vector lanes 8 times, totaling 64 loads in a runahead period. This exit policy is important for VRE's performance. Compared to an eager exit policy, it allows VRE to achieve a speedup of x1.79 compared to x1.69[11]. They also note that a similar exit policy for PRE increased performance by a further 3.5%. The speedup that VRE gains from a delayed exit is therefore mostly due to the sheer scale of its increase in MLP.

2.2.7 Performance of previous runahead schemes

Traditional runahead execution achieved a 22% IPC improvement over a no-prefetching baseline, a similar performance to a machine with triple the ROB size[4]. A hybrid filtered runahead + traditional runahead policy managed a 21% IPC improvement over a more modern no-prefetching baseline[12]. PRE is simulated on a model based on the modern Intel Skylake processors and improves performance by 38.2% over a no-prefetching baseline[10]. VRE speeds up execution by 79% over an OoO core with a stride prefetcher and 49% over PRE[11]. Vector runahead is so effective that it sometimes closes in on full MSHR saturation, with x2.3 MSHR utilization over OoO and x1.2 over PRE.

[2] Assuming AVX-512. In general, N is the number of words that can fit in a vector.

3 Motivation for the thesis

Although Mutlu originally found that delayed exits degrade performance[6], the work done in this thesis is primarily inspired by later findings by Hashemi et al. that runahead achieves poor coverage due to its short time spent in runahead mode[14], and VRE's performance gains from preventing exit until a predetermined amount of work is performed. These suggest that runahead, in general, may gain additional performance benefits from delaying exit when there is additional MLP to be extracted.

Based on this, a traditional runahead scheme was implemented in the gem5 simulator[15]. Multiple metrics were added to investigate whether a delayed exit policy could potentially improve performance. Initial experiments with the traditional runahead model revealed evidence of *runahead stutter*, in which the processor frequently switches in and out of runahead mode because it failed to prefetch critical loads in the previous runahead period. Figure 3.1 shows the distribution of lengths of *interim periods* in terms of retired instructions. An interim period is a normal mode period that begins after an exit from runahead and lasts until runahead is re-entered. The figure shows that in a traditional runahead processor, less than 150 instructions are retired in 34.2% of all interim periods, and less than 50 instructions are retired in 12.8% of all interim periods. The discovery of runahead stutter was the main motivation for the commitment to study runahead exit policies, with the goal to improve performance through developing a delayed exit policy capable of eliminating runahead stutter.

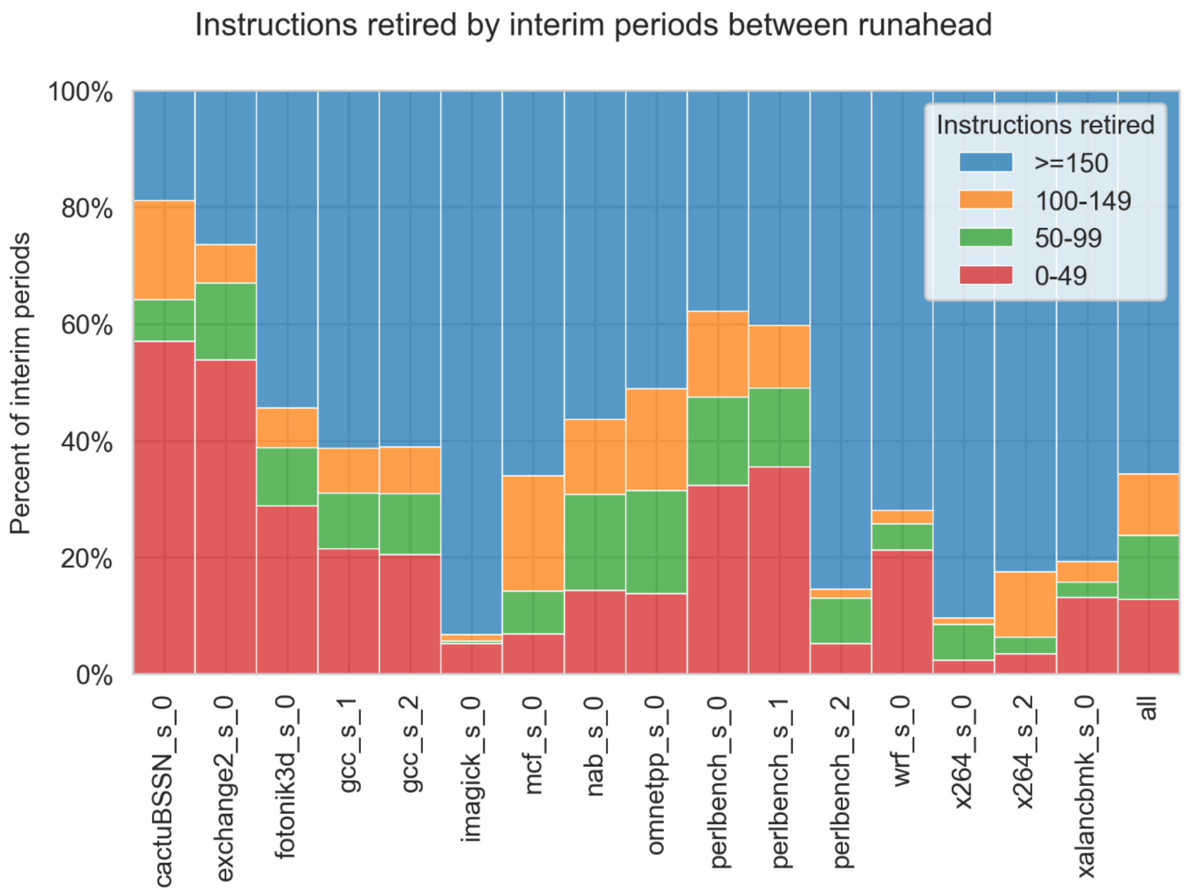


Figure 3.1: Instructions retired in the interim period between two runahead periods. Across all benchmarks, 34.2% of all interim periods retire less than 150 instructions before re-entering runahead, 23.7% retire less than 100 and 12.8% less than 50.

```
for (int i = 0; i < N; i++)
      y += buffer[hash(i)];
```

Figure 3.2: Example code of a loop accessing a buffer at a hashed index.

Figure 3.2 shows example code that may experience runahead stutter. In this case, the same static load instruction produces multiple dynamic instances that miss in cache because the hash is unpredictable. Such situations have already been shown to be a leading cause of full-ROB stalls[10, 12, 14]. Hardware prefetchers struggle with such loops due to the unpredictability of the hash. Runahead, however, is well suited for these situations for two reasons. For one, no iteration depends on another, meaning runahead does not get stuck on dependency-related issues like poison or other cache misses to prefetch further into the future. Second, and most importantly, it has access to processor resources, allowing it to compute the hash and accurately prefetch the correct indices of the buffer. However, runahead's usefulness is limited because it exits as soon as the first cache miss returns. The processor then catches up to the work done in runahead and promptly misses in cache again, re-entering runahead. This cycle repeats multiple times until the loop completes, as shown in Figure 3.3. The problem is exaggerated if the hash computation is expensive because runahead cannot quickly compute the indices.

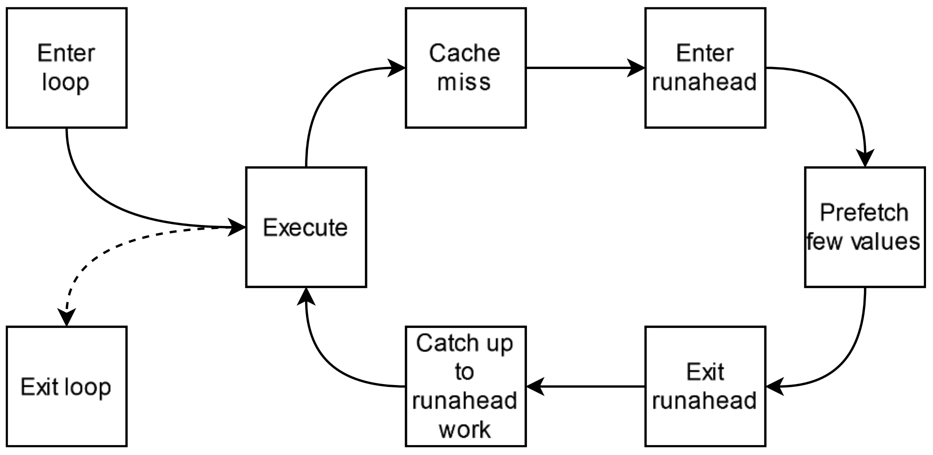


Figure 3.3: Illustration of processor behavior when it experiences runahead stutter.

Naithani et al.[11] have, in essence, implemented a *minimum work* policy in which the processor is not allowed to exit runahead until a specific amount of work has been done. The question I would like to answer is if similar policy would have a positive performance impact on traditional runahead schemes, and if alternative delayed exit policies have the potential to increase processor performance. If so, similar ideas could be applied to other runahead schemes to improve their performance.

4 Simulation infrastructure

4.1 Choice of simulator

The choice of simulator is justified in greater depth in the project report[22] preceding this thesis and submitted for TDT4501, but is summarized in short here. SMTSIM[26], Multi2Sim[27], Scarab[28], ChampSim[29], Sniper[30] and gem5[31] were evaluated as potential simulators for the project based on previous use in literature[8, 10-14, 21] or recommendations[32].

Of these, SMTSIM, Multi2Sim and Scarab were discarded as they were either outdated, had their development ceased, or both[22]. ChampSim is purpose built for championship prefetcher simulations, and therefore discarded as too simple for a core simulation. The quality of documentation and apparent ease of development was heavily weighed, and ultimately gem5 was chosen over Sniper due to its higher simulation fidelity and better documentation.

4.2 The gem5 simulator and the O3CPU model

The gem5 simulator[31] is a computer architecture simulator written in C++. The simulator has a modular design that allows mixing and matching various simulation components as defined by simulation configuration scripts written in Python. The simulator also has a powerful statistics engine that allows developers to rapidly implement new metrics in their models, including automatically computed histograms, averages, distributions, vector metrics and more. Powerful debugging tools are available, ranging from print traces to full debugger support for both the simulator and simulated programs.

Everything in gem5 is executed in an event loop, which allows it to decouple simulated time from real time, maintain different clock domains, deschedule idle systems to speed up simulation, and easily model latencies. By default, the event loop runs at a frequency of 1THz, giving the simulation a picosecond resolution. For example, a 2GHz core schedules its cycles for execution once every 500 simulation ticks.

Out of the box, gem5 features a variety of simple and complex core models that fit within various simulation paradigms. In this project, I used the O3CPU model[33] for baseline simulations, as well as a basis for the implementation of runahead. The following description of the O3CPU model is a direct quote from this thesis's preceding specialization project report[22] (with fixed citations):

> "The out-of-order core model provided by gem5, the O3CPU, provides an execution-driven simulation "loosely based on the Alpha 21264"[33] that uses 5 pipeline stages - fetch, decode, rename, issue/execute/writeback (IEW) and commit. In contrast to many other simulators[27-30], gem5's O3CPU model performs execute in the execute stage. The model is ISA agnostic and faithfully models many microarchitectural intricacies, such as pipeline stage bandwidth, functional unit contention and branch/memory order misspeculation." [22]

I now describe how the O3CPU model functions stage-by-stage. Although gem5 supports multiple hardware threads, only single-threaded workloads were used in this project. Therefore, mechanisms to handle multiple threads are not explained. Some minute details are skipped if not crucial to how the stage functions.

4.2.1 Fetch and time buffers

Figure 4.1 shows a simplified timeline of the work fetch performs in a single cycle. The fetch stage is responsible for fetching static instructions from the instruction cache (I-cache) according to the PC. Although not accurate to real systems, fetch also decodes instructions and builds the dynamic instructions that are used in the remainder of the pipeline[33]. It does this using a decoder which is fed bytes from a fetch buffer containing un-decoded data from I-cache.

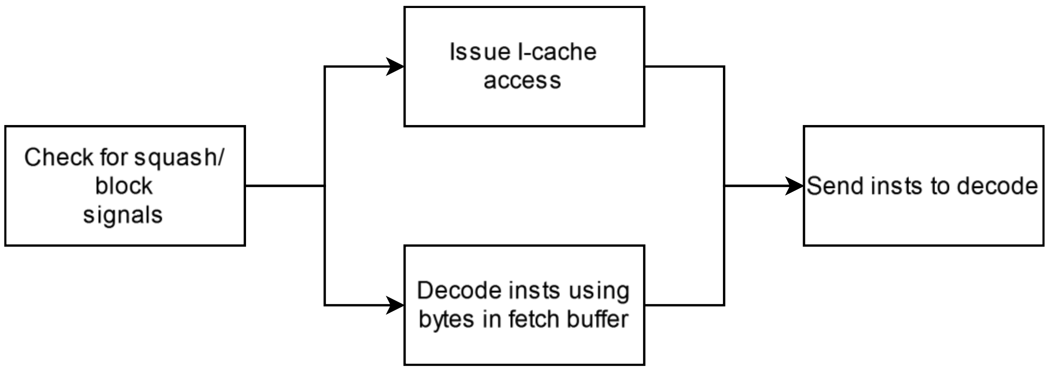

Check for squash/ block signals
Issue I-cache access
Decode insts using bytes in fetch buffer
Send insts to decode

Figure 4.1: Work done by fetch in a typical cycle. I-cache accesses and instruction decoding are mutually exclusive, only one can be performed in a single cycle.

At the beginning of a cycle, fetch begins by reading any signals from later stages. If decode signals that fetch must block, it switches state and blocks until it reads an unblock signal later. Both commit and decode may also send a squash signal including information about the PC to reset to. If asked to squash, fetch clears a fetch-to-decode queue (FTDQ) containing instructions to be sent to decode and resumes fetching at the signaled PC on the next cycle. Any outstanding I-cache requests are dropped. At the end of the signal checks, fetch checks if there are any outstanding I-cache misses, and blocks if there are.

After the signal check, if fetch is running, the stage will check if the fetch buffer is invalid or if the current fetch address has crossed an instruction boundary. If so, it issues a fetch to I-cache for the PC currently being processed. As with all gem5 memory requests, this requires address translation before the access can take place, so fetch will wait until translation is finished. Once finished, fetch sends the access packet to I-cache and waits for a response, which it copies into the fetch buffer. If the address translation faulted, fetch constructs and injects a no-op to carry the fault to commit where it will be handled.

On cycles where the fetch buffer contains valid data, it attempts to build instructions while there is remaining fetch bandwidth and space in the FTDQ. This is done by feeding bytes from the fetch buffer into a static instruction decoder. These microoperations are then wrapped into dynamic instructions containing additional information used by the processor in the remainder of the pipeline such as a global, monotonically increasing sequence number, the associated macrooperation, instruction state, register information and memory request state. The branch predictor is then consulted to update the next PC of fetch. A predicted branch here may halt construction of additional instructions to allow fetch to begin fetching from the new predicted PC. Note that because fetch decodes instructions before passing them over, the branch predictor has access to information it normally would not, such as instruction type. Certain branch predictor models utilize this information[34], although it is not clear to me exactly how this affects overall processor model performance.

When instructions are built, they are placed in the FTDQ. At the end of the cycle, instructions are read out of the FTDQ queue with bandwidth equal to the decode width and placed in a *time buffer*[34] to decode. In gem5, time buffers are circular queues that are typically *advanced* at the start of each cycle. Placing an element at index 0 is equivalent to inserting data in the "present". When the buffer advances, the data at each index moves back by one in practice. Accessing an element at index -N is therefore equivalent to reading data that was inserted N cycles "in the past". Thus, time buffers enable latency-bound communication between pipeline stages. They are heavily used in the O3CPU model and are the core of all inter-stage communication.

4.2.2 Decode

Decode is a rather simple stage because instruction decoding is already done at fetch. Its function in the O3CPU is to perform simple preprocessing of certain instructions and to act as a bandwidth-restricted choke for instructions passing through the pipeline. Figure 4.2 shows the work done by decode.

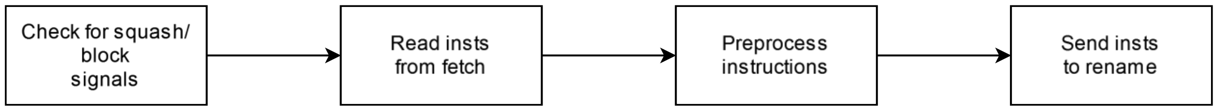


Figure 4.2: Work done by decode in a typical cycle.

Decode starts each cycle by reading instructions from the time buffer from fetch and placing them into instruction buffers. A signal read and state update is then performed. Rename may signal a block, and commit may signal a squash, which simply empties the instruction buffers.

If decode was not blocked or set to squash, it reads instructions from its buffers up to the decode bandwidth. It then does some simple preprocessing. Instructions with no source operands are immediately flagged as issuable. PC-relative branches are also resolved at this stage and trigger a squash in fetch and decode on a mispredict. If a non-branch instruction was predicted as a branch, this is detected here, triggering a squash. Any processed instructions are then placed in a time buffer to the rename stage.

4.2.3 Rename, the RRT and free lists

Rename utilizes a frontend RRT to translate architectural source registers to their corresponding physical register, and, for each architectural destination register, renames it to an available physical register on a free list. A scoreboard[34] is used to mark registers that are known to be ready as such. Figure 4.3 shows the work done by rename.

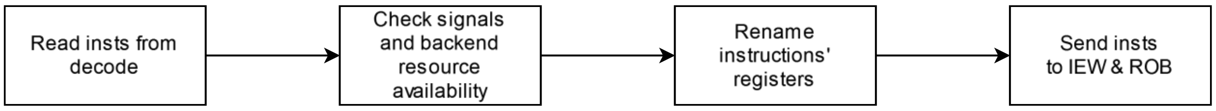

Read insts from decode
Check signals and backend resource availability
Rename instructions' registers
Send insts to IEW & ROB

Figure 4.3: Work done by rename in a typical cycle.

Each rename cycle begins by reading instructions from decode. Signals are then read from IEW and commit to determine if rename should block or squash. Squashes empty the instruction buffers and consults a rename history buffer to undo any changes to the RRT up to and possibly including the squashed instruction's sequence number, depending on the nature of the squash. In addition, rename is signaled information about available issue queue (IQ), load-store queue (LSQ) and ROB slots. If the IQ or ROB are full, rename blocks as every instruction will be inserted into these.

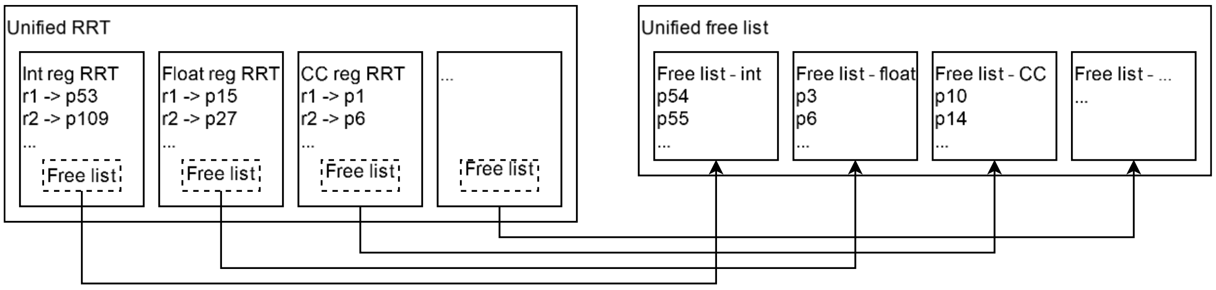


Figure 4.4: The structure of the O3CPU's register rename tables. Each RRT contains one individual rename map for every type of register, which in turn contains a pointer to a free list.

While there is available rename bandwidth and instructions to rename, instructions are read out of the incoming instruction buffers to rename their registers. Load and store instructions must additionally check if there is free space in the load/store queues before they are renamed. Architectural source registers are then renamed by a RRT lookup. The RRTs used by the O3CPU are structured as a unified map containing multiple rename maps for each register type, as shown in Figure 4.4. At the same time, rename performs a lookup in the register scoreboard, which stores information about register readiness. Registers are *set* in the scoreboard on writeback and *unset* when used in a renaming. Any source registers set in the scoreboard are immediately marked as ready in the dynamic instruction information. Destination registers are then renamed by asking the RRT for renames on the architectural destination registers. The RRT dequeues a free physical register, updates the corresponding rename map and returns the renamed register with which rename updates the dynamic instruction information and unsets the register on the scoreboard. Finally, a history entry containing the instruction's sequence number, architectural register, its previous mapping, and new mapping are stored in the history buffer for use in squashes.

At the end of the cycle, all renamed instructions are sent to IEW through a time buffer. Rename then reads a signal from commit containing the sequence number of the last committed instruction's sequence number. Any history buffer entries with a younger sequence number correspond to committed state and are therefore removed from the rename history.

4.2.4 Issue, Execute, Writeback

Issue, Execute, Writeback (IEW) is the most complex stage of the O3CPU and is responsible for instruction scheduling, execution and writeback handling. As seen in Figure 4.5, the order of work is actually execute, writeback, issue. For the sake of clarity, I describe the stage in the same order as in the stage's name. An in-depth overview of IEW's structure is shown in Figure 4.6.

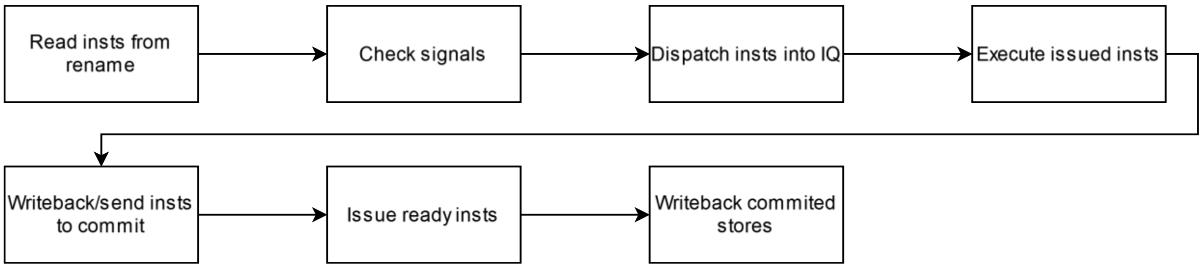


Figure 4.5: Work done by IEW in a typical cycle.

Before explaining the instruction flow through IEW, it's helpful to know how gem5 performs instruction execution. Every instruction in gem5 is responsible for its own execution, and has an associated *operation class*[34] (opclass). Each functional unit (FU) has a set of capabilities that describe which opclasses the FU can execute. FUs are grouped together into *FU pools*, which act as an interface through which the processor can utilize the FUs. The pool provides information on opclasses that can be executed, FU availability per opclass and their execution latencies. When an instruction is issued in the O3CPU, the processor grabs an available FU from the FU pool and schedules a FU completion event on the event loop. The event is processed after a time corresponding to the execution latency of the FU, in which the instruction is moved from the issue stage to the execute stage. Execute then invokes the execution routine of the instruction, including writing any results to the physical destination registers.

As with previous stages, IEW begins its cycles by reading incoming instructions from rename. Signals are then read from commit. If a squash was signaled, all instruction buffers are emptied, and the signal is propagated to the IQ and LSQ. After checking signals, IEW dispatches any instructions in its incoming buffers to the IQ while there is remaining dispatch bandwidth and space in the IQ. If the IQ becomes full, IEW blocks and sends a block signal to rename. If the instruction being dispatched accesses memory, it is also added to the LSQ. When instructions enter the IQ, they update the *dependency graph*[34], which maps physical registers to their producing instruction and a list of dependents. The graph is used for wakeup.

Issue is wholly handled by the IQ. IEW invokes the IQ to issue any ready instructions to the appropriate functional units. The IQ contains multiple ready lists, one for each of the operation classes that an instruction can have. An *age order list*[34] (AOL) is used to sort the individual ready lists according to the oldest instruction in each list. The head of the AOL contains the opclass for which the corresponding ready list has the oldest ready instruction in the entire IQ. When the IQ attempts to issue, it will grab an opclass off the AOL, then pop an instruction off the corresponding ready list and attempt to issue it to an available FU. In other words, the IQ always attempts to schedule the oldest instruction.

Once the FU completion event scheduled by the IQ is processed, the instruction is sent to execute through a time buffer with latency equal to the issue-to-execute latency.

Instructions are read out of the time buffer from issue to execute and executed. Most instructions immediately complete execution here, but memory instructions are handed to the LSQ as they require additional handling compared to other instructions. The LSQ initiates the memory access for the instruction and returns any faults. If the translation is delayed due to a page table walk, or the instruction couldn't execute due to a lack of available cache ports, the instruction is placed in a special deferred or blocked instruction queue and added back to the ready list later for re-execution. The LSQ may report a memory order violation, which also triggers a squash. Control instructions are resolved at this point and may trigger a squash if mispredicted. Because instructions are in the ROB at this point, they require extra handling. These squashes are therefore signaled to commit, which handles the squash. Executed non-memory instructions are immediately placed in a time buffer to commit but are inserted in "the future" if writeback bandwidth would be exceeded. This way, writeback handling is guaranteed to be performed before commit reads the instructions.

The amount of time needed to complete a memory access depends on the simulated system's memory configuration, which schedules relevant events as data packets travel through the memory hierarchy. Because of this, memory instructions can take a variable number of cycles to execute and are held in the LSQ until their memory accesses return. Once the LSQ is notified of a memory access completion, it finally invokes an access completion routine on the associated memory instruction using the data packet received from memory to writeback any results. The instruction is then placed in the time buffer to commit in the same way any other instruction would be.

Stores are an exception to this instruction flow because stores cannot be allowed to write their data to memory before they are committed state. The LSQ will only initiate address translation when a store first goes through IEW. To write data, they must first propagate to the head of the ROB and become committed. Once they are, commit signals this information back to IEW, which prompts the LSQ to send the write packets to cache.

Writeback to the register file happens on instruction execution, so the only function of IEW's writeback mechanism is to update the register readiness state and to wake any dependent instructions. Writeback is performed after execute and reads outgoing instructions in the time buffer to commit. The IQ is notified of the instruction completion and wakes any dependents using the dependency graph, adding them to the ready list if all source registers are ready. All destination registers are then set on the register scoreboard, including the IQ's internal scoreboard.

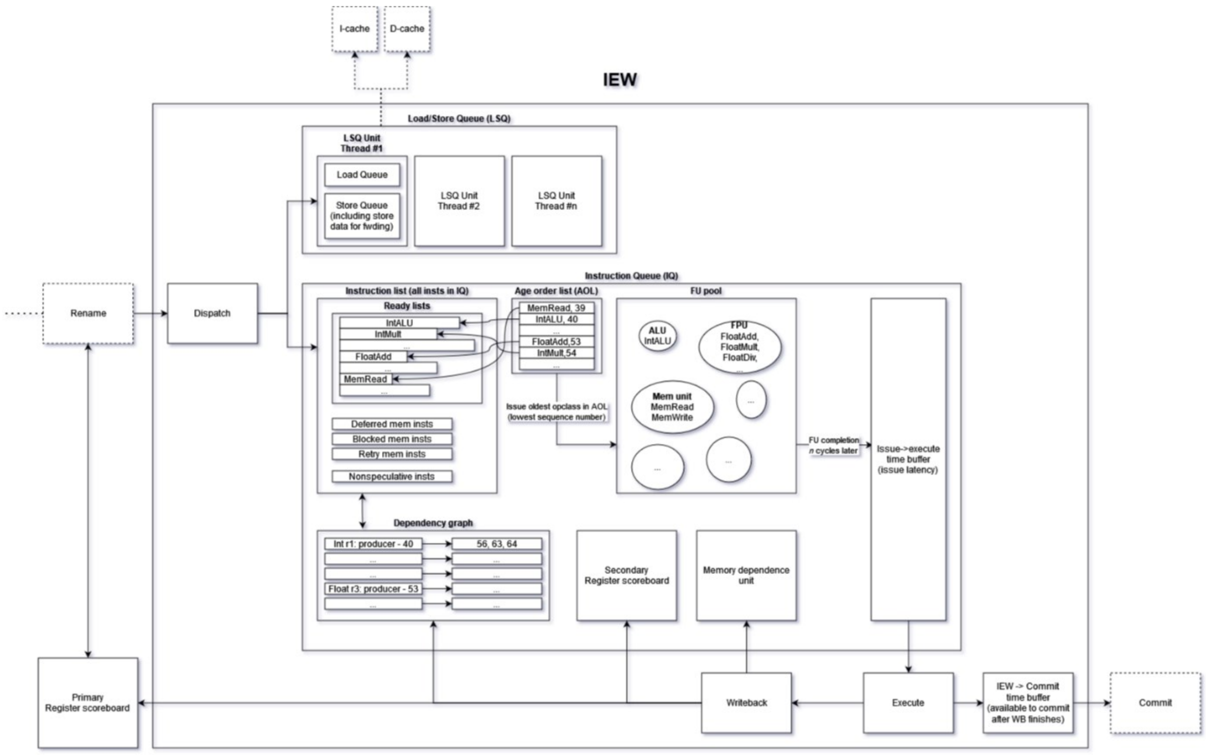


Figure 4.6: Structural diagram of the O3CPU model's IEW stage. The IQ contains all instructions due for execution and maintains multiple ready lists that use a dependency graph and internal register scoreboard to determine instruction readiness. FU bandwidth is modeled by a pool of available FUs for each opclass. IEW also features a memory dependence unit[35] for predicting memory dependencies.

4.2.5 Commit

Commit is, in addition to instruction retirement, responsible for handling faults, interrupts and squashes that are detected after insertion into the ROB. Figure 4.7 shows the work done by commit.

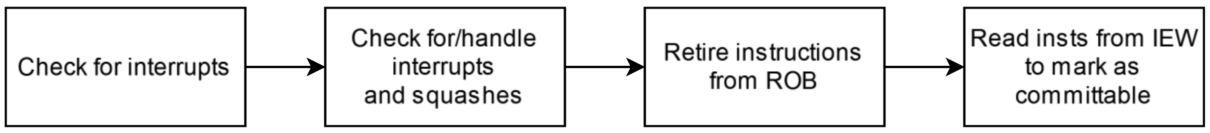

Check for interrupts
Check for/handle interrupts and squashes
Retire instructions from ROB
Read insts from IEW to mark as committable

Figure 4.7: Work done by commit in a typical cycle.

Before beginning to commit, the stage checks for any interrupts. If one is detected, it is processed at the end of the cycle while a trap squash is scheduled after a configurable latency. After the interrupt check, any squashes are handled, including scheduled trap squashes and branch mispredict or memory order violation squash signals from IEW. A squash causes commit to initiate a bandwidth-restricted squash in the ROB. At the same time, commit sends a squash signal to all previous stages through a time buffer containing information about how far back to squash.

If the processor isn't squashing, commit then considers the head of the ROB for retirement. If the instruction is flagged as committable, the dynamic instruction data is used to update the backend RRT containing architectural register mappings. The committed PC state is also updated, then the instruction is retired from the ROB, marking its completion. Instructions with a fault are handled here by invoking the fault, then scheduling a trap event for later which will drain the pipeline through a trap squash. This process repeats until commit's bandwidth is exhausted, the ROB is empty, or the head of the ROB is not ready to commit.

At the end of the cycle, commit reads incoming instructions from IEW and marks them as committable. This includes stores, which are later committed but sent back for writeback.

5 Implementing delayed exit runahead

The runahead execution model used in this thesis is a traditional runahead execution model based on gem5's O3CPU model. The source code is structured as an extension to gem5 version 20.0.0.2[34]. The code is open source and freely available on GitHub[15].

A major design decision was to keep the runahead CPU model decoupled from the base simulator code of gem5. This was done because in theory, the modifications required to support runahead schemes are typically confined to the CPU. I also hoped this would simplify any further work based on the model by confining all runahead-related code in one place. In the end, it turned out to be limiting as it became difficult to work with certain swappable components like the branch predictor without compromising the structure of the project. Suggestions to solve this issue are discussed in chapter 9.

In general, attempts have been made to support hardware threads in the runahead code where possible. However, this implementation has not been tested with multiple hardware threads because all test programs and benchmarks utilize a single thread. To test the implementation before using it on the SPEC2017 benchmarks, a test benchmark performing matrix multiplication was written. The source code of this test program is attached in appendix A.

This chapter details the modifications made to the O3CPU to support runahead execution, as well as the runahead optimizations that were implemented. I also explain the implementation of the four delayed exit policies used in this study.

5.1 Detecting LLLs and entering runahead

In gem5, every memory request associated with a dynamic load instruction tracks *access depth*[34]. This depth starts at zero and is incremented by one every time the request misses in cache. Access depth is used to determine when a load is considered to be a LLL. A configurable parameter called the *LLL depth* is added to the CPU model. If the access depth of a load instruction's memory request equals or exceeds the LLL depth, it is considered a LLL. For example, a request with access depth 0 on completion is a hit in L1 cache. A request with access depth 2 has missed in L1 and L2 cache. By default, the LLL depth is set to 3 such that a miss in L3 cache qualifies as a LLL. Because the time taken by caches to process a request is modeled, access depth changes dynamically as the data packet travels down the memory hierarchy. Therefore, some time passes before the CPU can confirm that a load is a LLL.

When commit inspects the ROB head and finds that it is not ready to commit, it checks if the instruction is a load with an active memory request. If so, the access depths of any associated memory requests are checked. When they exceed the LLL depth, the load is considered a LLL, and commit will attempt to enter runahead. Entry may be prevented by the CPU depending on certain conditions, as described later in section 5.6.

If entry into runahead is allowed, the processor instantly checkpoints all architectural registers and the last committed PC. As noted by Mutlu et al.[4], performance loss due to checkpointing can, in theory, be avoided by incrementally updating the checkpoint when instructions are retired. For this reason, no checkpointing latency is modelled as it simplifies the implementation. The processor then flags itself as "in runahead" and marks every uncommitted instruction in the instruction window as runahead. Every line in the runahead cache is then invalidated, and the runahead-causing load (RCL) is marked as poisoned. The LSQ then schedules a fake writeback event typically used for store-to-load forwarding, which allows the RCL to drain from the pipeline and unblock the ROB. Once this is done, the processor is in runahead.

5.2 Poison propagation

As mentioned in section 5.1, the RCL is poisoned on entry into runahead. In my runahead model, poison is tracked as a flag on dynamic instructions as well as on every physical register. If a runahead instruction is flagged as poisoned during writeback, every physical destination register will be marked as poisoned. On the other hand, if it was not poisoned, the poison flag is cleared from the register. Instructions are marked as poisoned when they issue to a FU and any source register is poisoned. Waiting until issue to check for poison guarantees that any poison has been propagated to source operands by their producers.

Poisoned instructions are skipped when encountered at execute. Stores are an exception to this rule because they need to propagate poison through the runahead cache. Therefore, they are allowed to initiate address translation. If this produces a fault, they are skipped like any other instruction. Otherwise, they are allowed to execute as detailed in section 5.3.

5.3 Runahead cache and memory instructions

Memory instructions are the only instruction type to receive special treatment in runahead. Load execution must be amended to perform R-cache accesses in parallel with real cache accesses, and stores cannot be allowed to write speculative data to real memory. Changes to instruction execution are largely confined to the LSQ and LSQ unit code, while the runahead cache is implemented as a separate structure.

Runahead cache is implemented as a simplified direct mapped cache. The size of the full R-cache is configurable and specified in terms of usable storage. Cache blocks are stored in a C++ standard library vector. Each cache block contains a tag, valid bit, poison bit and a data field. Block size, index masks and tag extraction bit shifts are computed dynamically from the R-cache size.

The way R-cache handles memory accesses is greatly simplified compared to gem5's typical caches. Any lookups and accesses are processed immediately upon receiving the packet, leaving the LSQ to schedule any packet handling latency. When a packet is passed to R-cache, it constructs a copy of the incoming packet to update and send back, then uses it to perform either a read or write operation, depending on the packet type. On success, R-cache returns the copied packet, and on failure it returns a null pointer. To process packets, the index and tag are extracted from the packet's address field and used to find the associated cache block.

If the packet is a read, R-cache performs a lookup to check if the cache block's tag matches, and if the valid bit is set. If either of these checks fail, the access immediately fails. If the packet reads a poisoned cache block, the memory request state associated with the packet is updated to indicate that the packet is poisoned. On success, the data is copied from the cache block into the data packet.

If the packet is a write, no lookup is required since R-cache is an isolated memory system. In addition to writing data, write packets always update the cache block tag and set the valid bit. The memory request state is then checked for a poison flag to determine if the cache block should be poisoned or if the poison bit should be cleared. Any write packets sent to R-cache are always successful.

Load instructions are modified to access R-cache through the LSQ only when a packet was successfully sent to real cache. The success or failure of R-cache lookup resolves instantly and updates the load instruction's memory request state to track any expected R-cache packets separately from real cache packets. If the access was successful, the LSQ schedules a cache response for one cycle later. If the memory request is tracking at least one R-cache packet, the instruction will only accept packets from the list of tracked R-cache packets. This causes loads to only use R-cache responses and discard any replies from real cache. If commit encounters a valid load at the head of the ROB, it will unblock the ROB the same way as with the RCL. Store instructions are never allowed to send data to real cache. Instead, they only send packets to R-cache, which always succeed.

5.4 Architectural state checkpointing

The architectural state checkpoint is maintained as a separate structure in code, and only stores checkpointed register values. On entry into runahead, every architectural register and any miscellaneous registers defined by the ISA are read and saved in the checkpoint.

To repair the RRTs after runahead, both the front- and backend RRTs are restored to their default state after runahead exits. After reparation, the architectural checkpoint is used to copy checkpointed values into the register file using the repaired RRTs. This is functionally equivalent to the checkpoint restoration mechanism in Mutlu et al.'s traditional runahead scheme[4]. The free list and register scoreboard are also reset to make every physical register available and ready.

A technical limitation of how the source code is structured turned out to be that accessing the branch predictor state is difficult. For these reasons, the branch predictor history and RAS are not checkpointed.

5.5 Exiting runahead and exit policies

Any time a data packet is received from real cache, the processor checks if the packet is intended for the RCL. If so, commit is immediately notified that runahead is safe to exit. Commit may then set a *runahead exit flag* depending on which *exit policy* the CPU model is configured to use. An alternative trigger for runahead exit is if a page fault occurs in fetch. Runahead mode does not handle faults, so this is an unrecoverable situation from which the CPU model simply exits runahead.

Exit from runahead is triggered on the first cycle after commit's runahead exit flag is set. A runahead exit is for the most part handled in the same way as a memory order violation squash, which squashes all instructions up to and including the RCL. Because the RCL was the oldest instruction in the instruction window when runahead was entered, this squash causes the entire pipeline to be drained and fetch to reset to the PC of the RCL. At the same time as the squash is started and signaled to preceding stages, the CPU flags itself as not in runahead. Commit then sends a signal to itself through a time buffer which will cause it to restore state from the architectural state on the next cycle. Architectural state is restored by resetting and repairing the RRTs, then restoring register values, all as described in section 5.4. Finally, all poison is then cleared from the register file.

In my runahead model, I have implemented four different exit policies, three of which are *delayed exit policies*:

1. **Eager exit** – The traditional exit policy in which runahead exits immediately after the RCL receives a cache response.
2. **Minimum work** – The processor is forced to pseudoretire a minimum number of instructions before it is allowed to exit runahead.
3. **No Load Left Behind (NLLB)** – Runahead exit is delayed until all valid loads in the ROB are sent.
4. **Dynamic delayed exit** – An exit policy that dynamically decides whether to delay exit based on the presence of nearby loads.

Regardless of the runahead exit policy, commit checks if the runahead exit flag was set. If not, a *runahead deadline* event is scheduled for a configurable number of cycles later. When the deadline event is processed, it forces the processor to exit runahead, thereby acting as a dead man's switch for delayed runahead periods which fail to exit within a reasonable amount of time.

I now describe the implementation of each of the delayed exit policies.

5.5.1 Minimum work

The minimum work exit policy forces the processor to pseudoretire a minimum number of instructions before allowing the processor to exit runahead. This is implemented using a pseudoretired instruction counter which is compared to a minimum work parameter. Whenever commit pseudoretires an instruction in runahead mode, it increments the counter. Commit checks the number of pseudoretired instructions every cycle while in delayed runahead. If enough instructions have been pseudoretired, the runahead exit flag is set, causing runahead to exit on the next cycle. It is possible for the processor to have pseudoretired the minimum number of instructions before the RCL returns, in which case minimum work behaves the same as an eager exit.

5.5.2 No Load Left Behind

In the No Load Left Behind policy, the processor inspects the ROB for any unsent, valid loads. If one or more are found, the processor model stores the sequence number of the youngest such load and continues runahead execution until it detects that the load has been executed. If no such load is found, runahead is exited eagerly. Currently, one oversight in this policy is that if the ROB contains loads that have not yet been issued, NLLB will delay exit for these loads, even if they become poisoned once they issue.

5.5.3 Dynamic delayed

The dynamic delayed policy decides whether to delay exit from runahead based on whether there are unsent loads near the head of the ROB. When runahead is safe to exit, the dynamic exit policy inspects the ROB for any unsent, valid loads within a maximum number of instructions. This number is configurable through the same parameter used for the minimum work policy. If any loads are found, the processor continues runahead until the runahead deadline is met. If the CPU is configured to enable instruction stream filtering (explained in section 5.6), it will exit immediately if the processor couldn't find a dependency chain for the runahead period. If the RCL returns and the processor has not yet worked through the instructions that were in the ROB on entry into runahead, dynamic delayed also exits runahead eagerly.

5.6 Efficiency and performance improvements

Entry into runahead can be disallowed based on the configuration of the CPU model. Some schemes eagerly enter runahead[4, 6, 8, 9, 12, 17], while others wait for the ROB to fill[10, 11, 21]. In my runahead model, this is a configurable parameter that can block entry if the ROB is not full. Additionally, Mutlu’s short and overlapping period elimination[6, 7] are implemented with configurable parameters. The CPU model compares the cycles elapsed since the LLL was first issued against an *in-flight threshold* (IFT) to deny entry into potentially short runahead periods. Overlapping periods are prevented by tracking how many instructions are pseudoretired each runahead period, P, and how many instructions have been retired since the last runahead period, R. If $R < P$, the processor does not enter runahead. Note that this is a stricter requirement than Mutlu’s implementation, which compared the number of instructions fetched since the last runahead period. This change was made because retirement guarantees that the instruction has been executed.

Filtered runahead[12] is partially implemented in the runahead CPU model. When runahead is entered, the ROB reads the head instruction and uses it to attempt chain generation as described by Hashemi and Patt. Due to time constraints, no runahead buffer is implemented. Therefore, the use of any discovered dependency chains is to filter the instruction stream at fetch. If the processor found a chain at the start of runahead, instructions not in the chain are simply discarded. If no dependency chain could be constructed on entry into runahead, traditional runahead is used. When fetch reaches the tail of the dependence chain, it resets its PC and begins fetching at the head of the chain again to execute the chain as a loop.

5.7 Miscellaneous modifications

Branch divergence[4] is detected at IEW when poisoned instructions are skipped by checking if the instruction was a control instruction. Since it is impossible to resolve the branch, and therefore determine if the branch was correctly predicted, the CPU simply flags that execution is possibly diverging. This flag is not currently used for any purpose other than statistics.

Because branch predictor state is not checkpointed before runahead, fetch’s use of the branch predictor is modified in runahead. In normal mode, the branch predictor updates the next PC of fetch, including any state updates performed by the predictor. In runahead, fetch asks the branch predictor for a prediction, then performs a manual BTB lookup if the instruction is a predicted taken branch. The next PC is then updated with the address in the BTB, or the next PC if the instruction was not predicted as a taken branch. The aim of this is to use the branch predictor without updating its state.

One issue that has, to my knowledge, been poorly documented by literature so far is how runahead should handle faults and interrupts. In my model, runahead faults are simply ignored as they are not architecturally real faults. Runahead page faults, if produced by valid instructions and occurring while the processor is on the correct path, could technically hide some latency if the processor had initiated paging early, but the latency of a paging operation is so large that even runahead would fail to make a notable impact. Interrupts are handled by postponing them until runahead exits. One alternative to handle them earlier would have been to immediately exit runahead upon detecting an interrupt, but I found they happen so rarely that the additional technical complexity of doing this was not worth addressing.

Various statistics were added to the runahead model both globally as well as to the individual stages of the processor. A list of all new statistics computed by the runahead model is attached in appendix B.

5.8 Testing the implementation

During early development, the runahead CPU model was tested with a simplified system configuration running a test matrix multiplication program in syscall emulation mode[33]. The test program source code is attached in appendix A. The program generates two square matrices and multiplies them together, storing the result in a new matrix. Before multiplication, the program outputs the matrices, and the product matrix is printed at the end of the program. Progress through the multiplication is printed regularly. The size of the matrix can be scaled at runtime, and the order in which the matrices are multiplied can be randomized. Both are controllable as input parameters to the program.

Once the test program successfully ran with the runahead CPU model, the correctness of the model was tested by running the program with the stock O3CPU model and my runahead CPU model. The output of each run was compared and confirmed to match 1:1 for a variety of different matrix sizes, both with and without multiplication order randomization.

After validating the correctness of the CPU model with the test program, the model was promptly used with the final benchmarking configurations described in chapter 6. This revealed many new bugs which were dealt with as they were discovered. These include bugs that caused the simulator itself to crash, but also bugs that caused the program to behave incorrectly inside the simulated system, leading to in-system crashes, most often segmentation faults. Due to the complexity of the CPU model and the simulated system, it is difficult to judge which bugs, if any, remain in the model. However, the final runahead model used for evaluation has been confirmed to run crash-free with all benchmarks for the entire length of their respective regions of interest (ROIs).

6 Evaluation of delayed exit runahead

I used a subset of the SPEC CPU 2017[1] (SPEC2017) benchmarks to evaluate and compare the performance of the baseline OoO model, the runahead baseline and the runahead models using the delayed exit policies. The x86 ISA is used with all the simulation models. This chapter details some characteristics of the benchmarks and how I found representative regions to simulate. The process for determining the parameters of the runahead model is also explained, along with how performance is measured in this thesis.

6.1 Characteristics of the SPEC2017 benchmarks

The SPEC2017 benchmarks[1] constitute multiple programs representing both compute and memory intensive real life workloads. While most of the benchmarks are unique programs, some are simply input variations on the same program. The benchmarks are split into two suites, SPECspeed and SPECrate. In this thesis, the SPECspeed suite is used. The benchmarks are designed to be a mix of both compute and memory intensive programs and are carefully chosen and configured to have predictable execution.

An independent study focused on the memory characteristics of the SPEC2017 suite[2] found that it is incredibly memory intensive, with nearly 50% of all dynamic instructions referencing memory values through either source or destination operands. Certain benchmarks also have considerable memory footprints, using up to 16GB of main memory. The working set size of each benchmark varies greatly, but Singh and Awasthi find that the cache performance, measured in MPKI, of *xalancbmk, nab, fotonik3d* and *lbm* have high MPKI while being particularly invariant to increasing cache sizes. On average, the main memory footprint of the SPEC2017 benchmarks is only 1.82GB, but *bwaves_s*, *roms_s*, *fotonik3d_s*, *cactuBSSN_s* and *xz_s* were found to have very large memory footprints, ranging from roughly 7-16GB. Many of these benchmarks are also found to have high bandwidth traffic to off-chip memory.

On average, the SPEC2017 SPECspeed suite of benchmarks runs for 22.19 trillion dynamic instructions[2], with most of these belonging to floating point workloads. Most benchmarks have a roughly 40-40-20% split of ALU-only, memory read and memory write instructions, respectively. Singh and Awasthi find that *exchange2* and *pop2* are an exception to this, where 79.6% and 73.5% of all dynamic instructions only use the ALU, respectively. Floating point benchmarks are notably more compute intensive than the integer ones, consisting of roughly 60% ALU-only instructions.

6.2 Configuring gem5 for the SPEC2017 benchmarks

Full system simulation is required to run the SPEC2017 benchmarks in gem5[33, 36]. When gem5 is configured for full system mode, the processor model boots an operating system and loads a disk image into the system. As in my preceding specialization project[22], I used Ubuntu 18.04 with a Linux 5.4.49 kernel and a disk image containing the SPEC2017 benchmarks, a boot shell script and a gem5 binary capable of emitting special pseudo-instructions recognized by the simulator. The kernel and disk image were provided by a PhD student at the department of computer science (IDI)[36].

To run the benchmarks on the disk image, the configuration script must attach a *runscript* to the simulated system[33]. Once the simulator finishes booting the operating system, the runscript is read to determine commands to be run in a shell. In this project, each benchmark has a unique runscript which begins by invoking the gem5 binary to exit the simulation. The binary does this by emitting a special pseudo-instruction recognized by gem5. This allows the configuration script to discard metrics produced by boot and swap the simulation core model. The simulation is then restarted, and the runscript proceeds to launch the benchmark. If left to run on its own, the benchmark completes and the runscript emits another exit pseudo-instruction to end the simulation. As noted in section 6.1, the benchmarks contain on the order of trillions of dynamic instructions[2], and would take days or weeks to complete even on simplified core models. Because of this, the maximum number of executed instructions is restricted by the configuration script.

One major problem with running the SPEC2017 benchmarks under the x86 ISA in gem5 is their memory footprint. x86 systems in gem5 can only use 3GB of main memory[33], although the reason is poorly documented**.** Benchmarks with a larger footprint than this risk being OOM-killed by the simulated operating system. Many benchmarks did not work with my simulation configuration for exactly this reason, and certain benchmarks only appear to function because they do not run for long enough to allocate too much memory. *cactuBSSN_s_0* is one such example, although there are more among the benchmarks used in this study. Some benchmarks also failed due to runtime errors such as segmentation faults in the simulated system. Table 6.1 lists all benchmarks and if they were functional for the purposes of runahead simulation. In total, 16 of the 28 benchmarks on the SPEC2017 disk image were usable.

Table 6.1: A list of all SPEC2017 benchmarks present on the simulation disk image and whether they were functional for the purpose of the project or not. Functional benchmarks are marked in bold text. Benchmarks which were found to get OOM-killed or crash when allowed to run for longer than 50B instructions are marked with *.

Benchmark	**Functional?**
bwaves_s_0	No
bwaves_s_1	No
cactuBSSN_s_0	**Yes***
cam4_s_0	No
deepsjeng_s_0	No
exchange2_s_0	**Yes**
fotonik3d_s_0	**Yes***
gcc_s_0	No
gcc_s_1	**Yes***
gcc_s_2	**Yes***
imagick_s_0	**Yes***
lbm_s_0	No
leela_s_0	No
mcf_s_0	**Yes***
nab_s_0	**Yes**
omnetpp_s_0	**Yes**
perlbench_s_0	**Yes**
perlbench_s_1	**Yes**
perlbench_s_2	**Yes**
pop2_s_0	No
roms_s_0	No
wrf_s_0	**Yes***
x264_s_0	**Yes**
x264_s_1	No
x264_s_2	**Yes**
xalancbmk_s_0	**Yes**
xz_s_0	No
xz_s_1	No

6.3 Finding representative regions

To find representative simulation intervals of the benchmarks, I used SimPoint analysis[37] on the first 50 billion instructions of each benchmark with a SimPoint interval of 100 million instructions. A SimPoint probe is attached to a simplified CPU model in a profiling run of each benchmark. The SimPoint probe listens for instruction retirement to identify and count occurrences of unique basic blocks in the workload. At each SimPoint interval boundary, the collected information is stored as a basic block vector (BBV). The results of each profiling run are BBVs for each SimPoint interval. The BBVs are then fed into the SimPoint software to find intervals that are representative of the benchmark. For each interval, I generated SimPoint weightings which indicate how much time the program spends executing code similar to the interval. The most representative SimPoint intervals are then chosen as a ROIs.

For example, SimPoint might find that interval #432 is a good representative of a given benchmark and assign it a weighting of 63%. This indicates that the benchmark spent roughly 63% of its time executing code similar to the interval between 43.2B-43.3B instructions. Note that since SimPoint profiling was only performed for the first 50 billion instructions, this weighting is only true for the first 50 billion instructions of the benchmark.

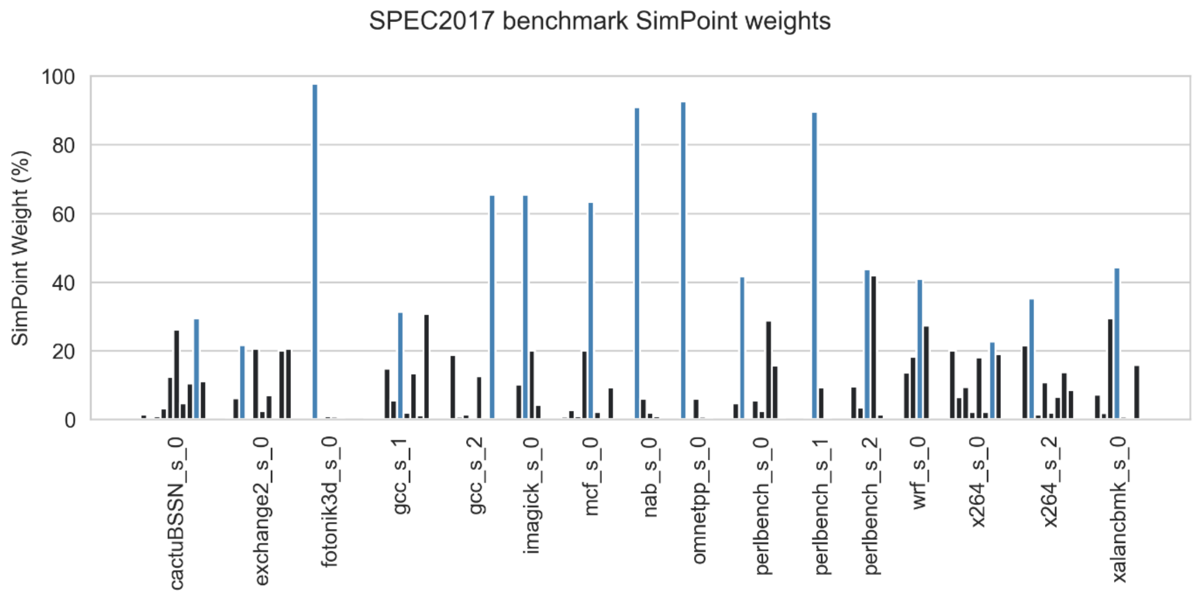


Figure 6.1: Weightings of all SimPoints extracted from the first 50 billion instructions of each SPEC2017 benchmark used in the CPU model evaluation. For each benchmark, the SimPoint with the largest weighting is colored blue. All other SimPoints are colored black.

Once representative ROIs were found, a second run of the benchmarks was performed to create checkpoints 1 million instructions before each ROI. The additional 1 million instructions are included to facilitate warmup of caches and processor buffers. The weighting of each ROI is shown in Figure 6.1. Seven benchmarks have overwhelmingly representative ROIs, although the remainder have multiple ROIs with similar weightings. For these benchmarks, it would have been ideal to simulate multiple intervals and aggregate their statistics, but the varied nature of gem5's hundreds of statistics complicated the development of a statistics compilation tool to the point that it was judged as too time consuming. For this reason, only the single most representative ROI is used for simulation.

In a real simulation run, the simulation configuration script restores the checkpoint using a simplified CPU model. The CPU is then swapped out for the detailed core model which is supposed to be simulated. The simulation then runs for 1 million instructions, plus 100 million instructions in the ROI. As such, each benchmark runs for 101 million instructions in the ROI which is most representative of that benchmark's first 50 billion instructions. One important thing to note is that runahead instructions do not contribute to the executed instruction count. This ensures that the simulation always measures the actual time and work taken to execute 101 million real instructions.

6.4 Measuring performance

As will be shown in section 7.1, my runahead experiments show that when compared to the stock OoO CPU of the gem5 simulator, runahead performs worse in terms of IPC. This is despite main target metrics such as load-to-use (L2U) cycles and overall cache hit rates improving. Possible explanations for this are discussed further in section 8.1, but there is overwhelming evidence that runahead improves processor performance[4-12, 17-21], so this is considered a performance bug/anomaly.

The problem with this anomaly is that configuring the system based on IPC will encourage configuring the system in a manner that minimizes the amount of runahead performed. In fact, judging by IPC, the optimal system configuration is to not use runahead whatsoever. This is obviously not well suited for a study of runahead. To mitigate this problem, I use an additional throughput metric in this study which I call *normal IPC* (NIPC). In the context of this study, normal can be thought of as "not runahead". Therefore, NIPC is defined as the number of normal instructions executed by the processor divided by the number of cycles the processor spent in normal mode. One intuition for NIPC is to imagine that every load that misses in cache triggers a "magical" pipeline flush. The flush instantly prefetches the load and possibly some additional future loads (i.e., any loads prefetched by runahead). NIPC therefore mostly describes the performance effect runahead has on normal execution. Any overhead related to filling buffers and structures after runahead has exited is still represented in NIPC.

It is extremely important to note that NIPC is not a perfect solution to system performance analysis in the face of the performance anomaly. While IPC encourages minimizing runahead, NIPC can encourage maximizing runahead, particularly where there are many independent loads, because the time cost of runahead execution is nullified. For this reason, I use a mix of IPC, NIPC and other system metrics like L2U cycles, pseudoretirement counts and cycle counts to analyze system performance. Additionally, for the delayed exit models, I only compare their performance to the runahead baseline with the understanding that while performance is degraded compared to the stock CPU, it may still improve compared to the runahead baseline. If performance improves relative to the runahead baseline, it should hopefully also improve performance in a runahead processor which does not exhibit the performance anomaly.

To attempt to explain the performance anomaly, overhead metrics were added to attempt to measure the number of cycles spent by the processor when entering and exiting runahead mode before resuming work as usual. Entry overhead is defined as the number of cycles from the processor flagging as being in runahead until the RCL is pseudoretired from the ROB. Exit overhead is more complicated to measure because snapshotting the full state of the processor for comparison is difficult. My implementation of the metric snapshots the number of instructions present in the IQ on entry into runahead. Exit overhead is then defined as the number of cycles from the processor flagging as being back in normal mode until an equal number of instructions has been inserted back into the IQ. The intent behind this is to capture a point in time at which the processor has begun to process at least as many instructions as before runahead was entered. This is still not entirely accurate because instruction readiness and the contention state of various buffers are not entirely the same.

6.5 Base system parameters

The baseline system configuration is given in Table 6.2. The core and cache configuration is largely based on the system configurations used in Naithani et al.'s paper on vector runahead[11], whose core configuration is in turn based on the Intel Skylake architecture[38].

Table 6.2: Base system configuration for stock, runahead baseline and delayed exit CPUs.

Parameter	**Value**
Clock frequency ROB size IQ size LQ size SQ size	3.2GHz 224 96 64 60
Branch predictor	8KB TAGE-SC-L
Physical register file	180 integer registers 180 floating point registers
Pipeline widths	4 insts/cycle – fetch, decode, rename, issue 8 insts/cycle – writeback, commit, squash
Functional units *Pipelined operations are marked in bold*	3 int ALU (1 cycle) 1 int M/D (**3 cycle mult**, 20 cycle div) 1 FP ALU (**2 cycles**) 1 FP M/D (**4 cycle mult**, 12 cycle div) 2 mem R/W
L1-I cache L1-D cache L2 cache L3 cache	32kB, 4-way assoc, 4-cycle lookup & access latency, parallel lookup/access 32kB, 8-way assoc, 4-cycle lookup & access, parallel lookup/access 256kB, 8-way assoc, 8-cycle lookup & access, parallel lookup/access 6MB, 12-way assoc, 30 cycle lookup & access, parallel lookup/access
Hardware Prefetcher	Stride-based in all cache levels
Memory *Based on Micron MT41J512M8*[39]	3GB, DDR3-1600, 2 channels tRP-tCL-tRCD = 13.75-13.75-13.75ns

6.6 Determining runahead baseline parameters

To determine the parameters of the baseline runahead model, a series of sensitivity analyses were performed on certain parameters that influence the CPU's runahead behavior. The parameter value with the best NIPC was chosen to encourage an aggressive runahead configuration. For parameters where NIPC did not vary greatly, the number of pseudoretired instructions also motivated the decision. The initial runahead configuration of the system is given in Table 6.3.

Table 6.3: Initial runahead parameters for baseline sensitivity analysis. Parameters marked in bold were tested in the sensitivity analysis.

Parameter	**Value**
Runahead cache size	2kB
In-flight limit	250 cycles
Overlapping runahead	Disallowed
Eager entry	Yes
Fetch stream filtering	No

The initial analysis investigated whether it is beneficial to immediately enter runahead upon encountering a LLL at the head of the ROB (eager entry). The alternative is to wait until the ROB fills (lazy entry), in which case the CPU would typically stall. The result of this analysis is shown in Figure 6.2. Both models have a slight increase in relative NIPC, with the eager entry model achieving a 4.8% higher NIPC compared to the baseline model. In comparison, the lazy entry model improves NIPC by less than 1%. The eager entry model not only enters runahead earlier, but more often, causing more instructions to be executed and pseudoretired. Based on these results, an eager entry policy was chosen for the runahead CPU model.

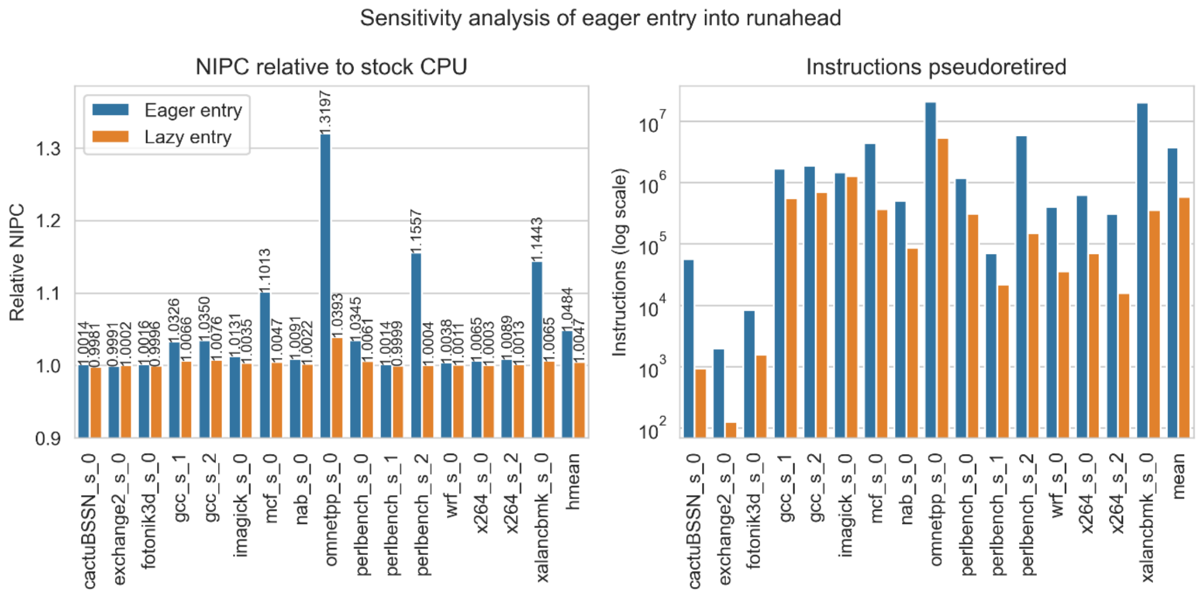


Figure 6.2: Comparison of relative NIPC and pseudoretired instructions between an eager and lazy entry runahead CPU model. (Harmonic) means are across all benchmarks.

Next, the effect of allowing or disallowing overlapping runahead periods was analyzed. Figure 6.3 Shows that NIPC is mostly invariant to overlapping periods, but disallowing overlap slightly reduces the executed instruction count. Based on this, overlapping periods are disallowed in the runahead model.

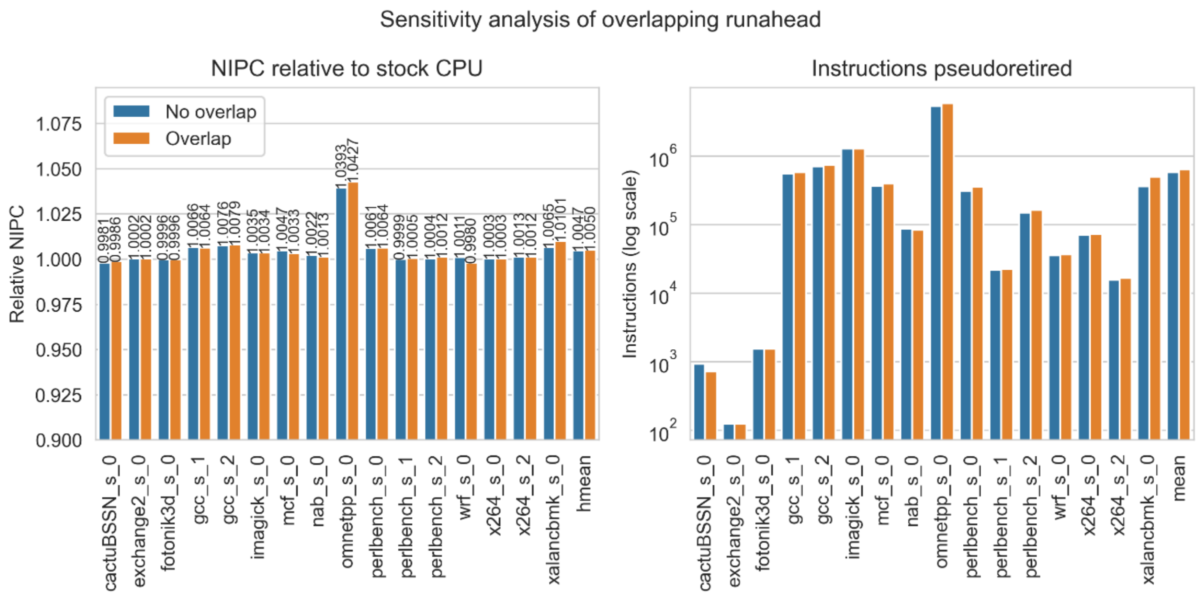


Figure 6.3: Comparison of relative NIPC and pseudoretired instructions by a runahead CPU model that allows overlapping runahead periods, and one that does not.

The effect of fetch stream filtering is shown in Figure 6.4. Even without the runahead buffer[12], filtering the instruction stream to LLL dependence chains seems to increase both performance and efficiency. The difference in performance is negligible, but fewer dynamic instructions are executed. Based on this, filtering is enabled.

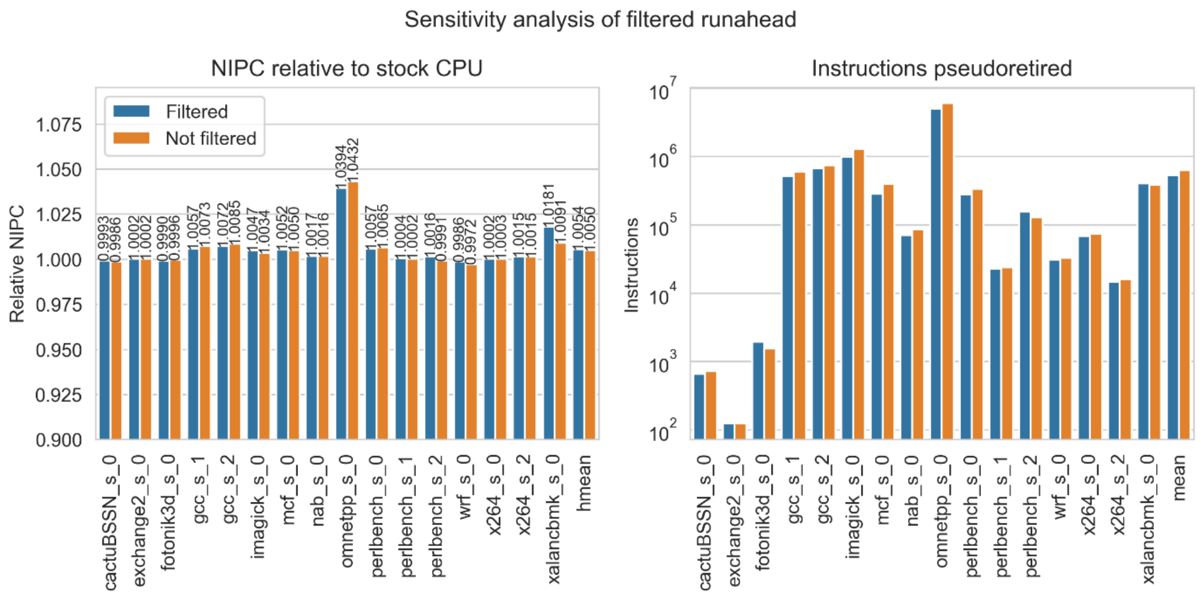


Figure 6.4: Comparison of relative NIPC and number of pseudoretired instructions when filtering the instruction stream to dependence chains.

Finally, the in-flight threshold (IFT) was determined by simulating with six different values. The NIPC and pseudoretired instruction count of each model is shown in Figure 6.5. Across all benchmarks, the CPU model isn't particularly sensitive to IFT. Higher IFTs do lead to a slight increase in the amount of runahead work done, however, and so increases NIPC and pseudoretired instruction count. An IFT of 350 provides the best relative NIPC of 1.066, but an IFT of 300 provides a relative NIPC improvement of 1.065 while pseudoretiring about 110K fewer instructions. Because of this, an IFT of 300 is chosen.

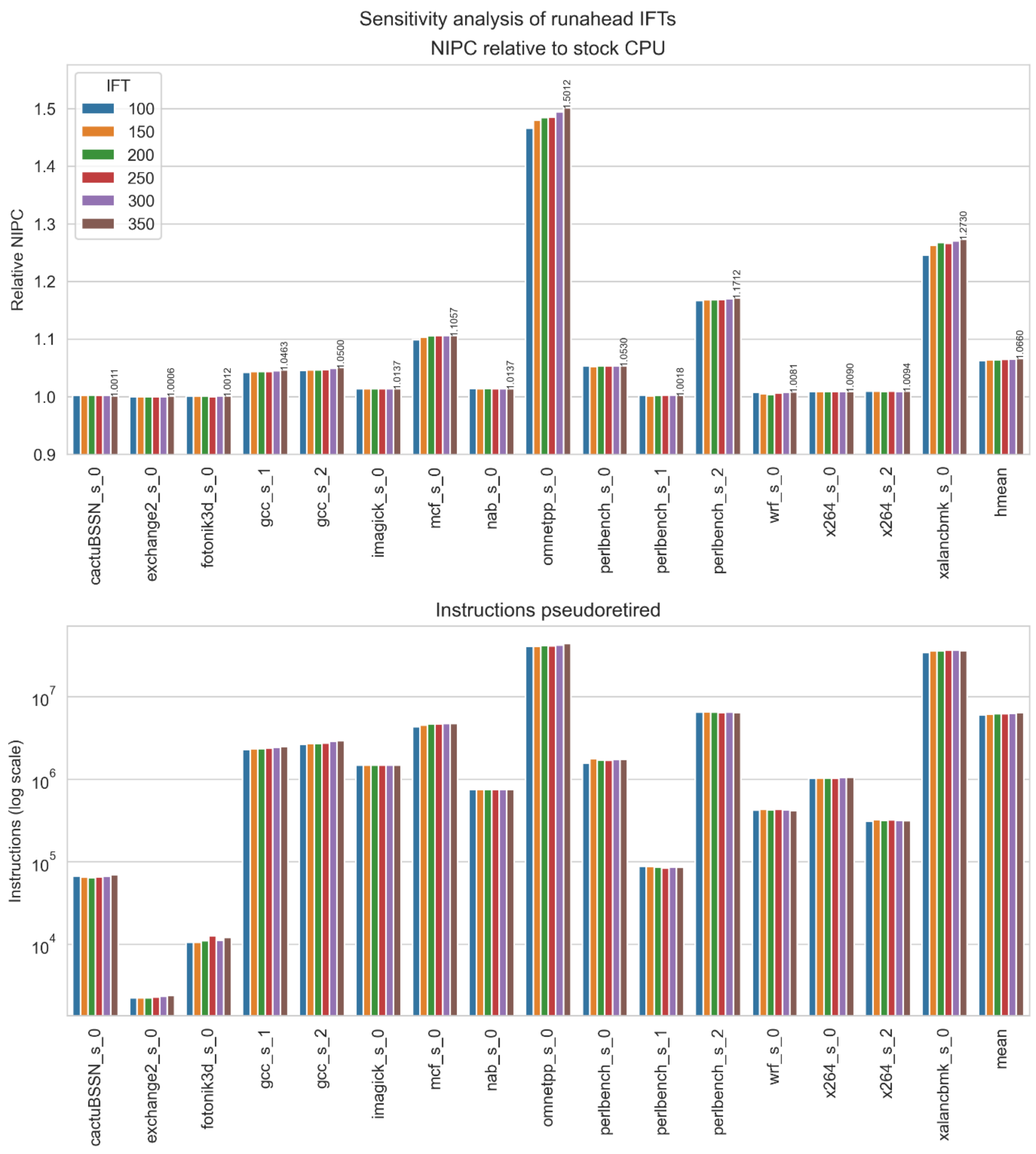


Figure 6.5: Sensitivity analysis of various in-flight thresholds (IFT) with regard to their impact on NIPC and pseudoretired instructions. Exact NIPC is displayed above the bars for IFT=350.

7 Results

This chapter presents the system statistics for the runahead baseline compared to the stock out-of-order CPU that comes with gem5. It also presents the results for the various delayed runahead exit policies. The different models presented in this chapter are:

- **O3 –** The stock OoO CPU model provided by gem5. The runahead baseline is compared to this.
- **Runahead *or* Eager exit** – The runahead baseline. All delayed exit models are compared to this.
- **Minimum Work** – Minimum work runahead models with various exit deadlines and minimum work values, as described in section 5.5.1.
- **NLLB** – The No Load Left Behind runahead model, as described in section 5.5.2. Uses an exit deadline of 100 cycles.
- **Dynamic Exit** – The dynamic exit runahead model, as described in section 5.5.3. Inspects a maximum of 25 ROB instructions and uses a 100-cycle exit deadline.

The performance of various runahead exit deadlines is also analyzed and presented in this chapter in the context of a forced delayed exit with no clauses on what work is done during these cycles. This is presented alongside the minimum work results.

7.1 The runahead baseline

Figure 7.1 shows the IPC of the runahead baseline model. Surprisingly, runahead performs worse than the stock CPU in nearly every benchmark. *exchange2_s_0* and *perlbench_s_2* have particularly striking performance degradations of 21.9% and 14.1%, respectively. The only benchmark to gain performance from runahead is *mcf_s_0*, which sees a 1.6% speedup. Combined across all benchmarks, IPC decreases by 3.8% because runahead spends nearly 100 million additional cycles executing all the benchmarks. Discarding all runahead cycles, NIPC increases by 4%.

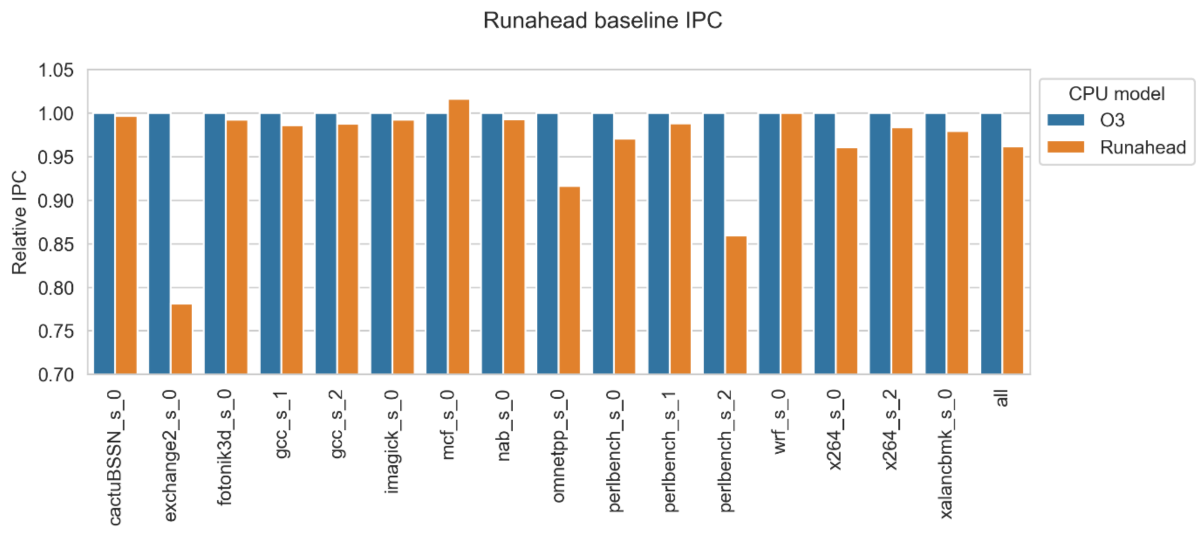


Figure 7.1: IPC of the runahead baseline model relative to the stock OoO CPU model. "all" is the sum number of instructions divided by the sum number of cycles across all benchmarks.

Despite IPC decreasing, the model improves load-to-use times significantly. Figure 7.2 shows that mean L2U for normal load instructions is decreased in all benchmarks. *cactuBSSN_s_0*, *exchange2_s_0* and *fotonik3d_s_0* have particularly low L2U times to begin with, indicating they should not be particularly affected by load-induced full-window stalls. *mcf_s_0*, *omnetpp_s_0* and *xalancbmk_s_0* do not have excessive mean L2U times, but their variance is large, meaning there are more loads which miss in cache. Runahead's most important contribution here is that L2U variance is greatly reduced. The number of loads that exceed 300 cycles load-to-use is decreased by 73.8% across all benchmarks, an entire order of magnitude less when compared to the stock OoO model. This indicates that the runahead implementation is very effective at prefetching LLLs.

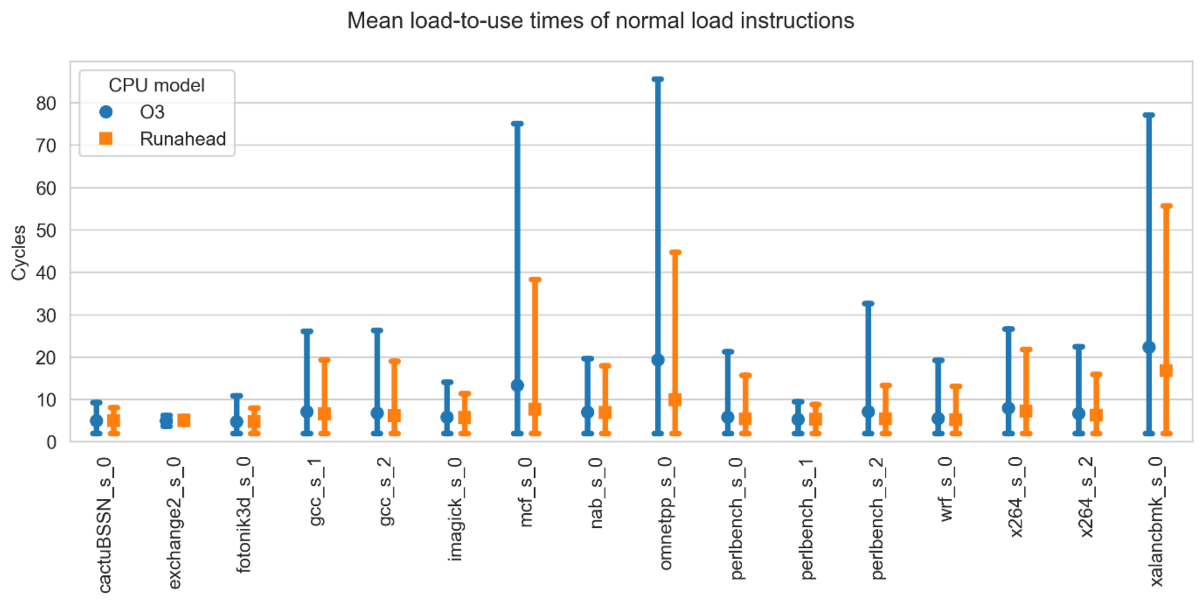
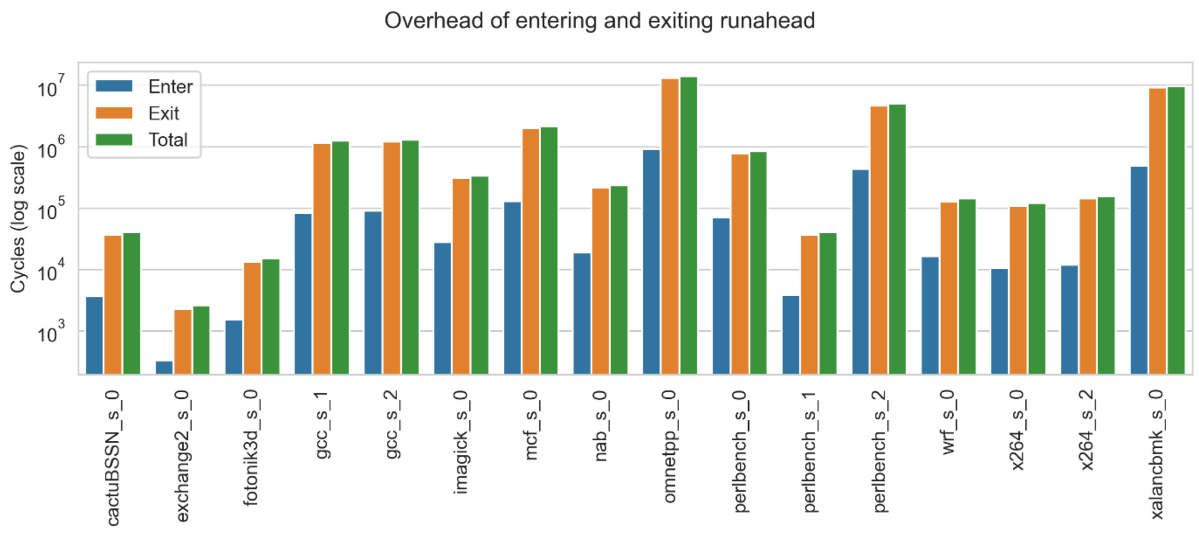


Figure 7.2: Mean load-to-use cycles for normal loads with the stock and runahead CPU models. The error bars show one standard deviation from the mean.

The overhead of entering and exiting runahead does not explain the performance degradation. Figure 7.3 shows that even when removing all overhead cycles from the IPC calculation, runahead causes a performance degradation. This could indicate that:

- The processor is entering runahead on loads that would not have stalled the ROB, incurring unnecessary overhead penalties,
- The processor may be staying in runahead for longer than intended,
- Or runahead causes side-effects whose negative impact on performance is greater than the positive impact of reduced load latencies.

Despite not checkpointing branch predictor state, branch mispredictions in normal mode are reduced by 0.7% across all benchmarks, and infinite L1-I cache experiments still showed a relative performance degradation.



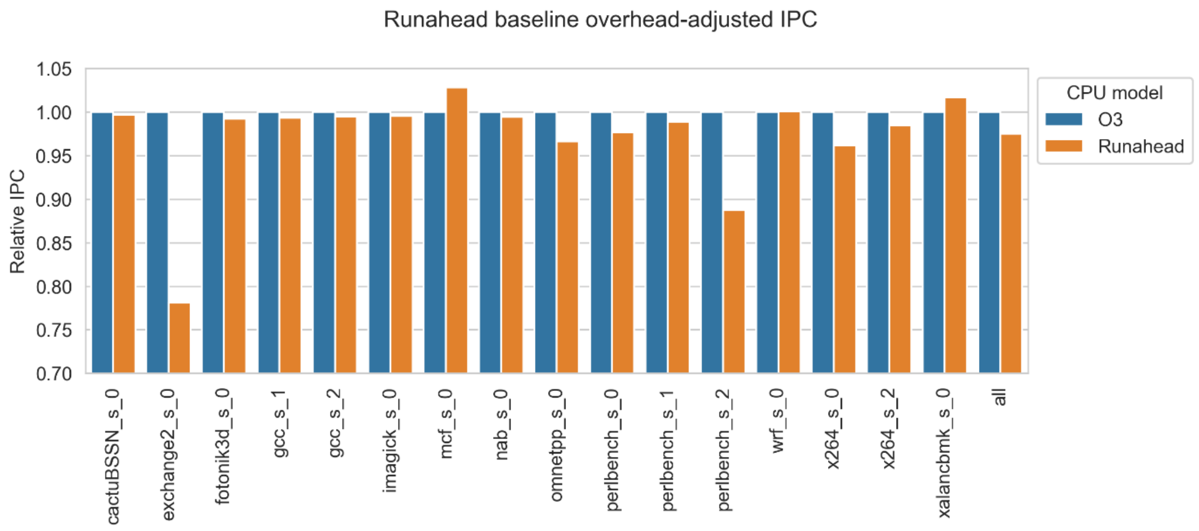


Figure 7.3: Cycle overhead of entering and exiting runahead mode (top) and the runahead model's adjusted IPC with overhead cycles removed (bottom).

It turns out that the chosen ROIs were not particularly memory intensive, at least not to the point that the processor spends a significant amount of time stalled due to the ROB being full, as evidenced by Figure 7.4. In total, the stock CPU model spends only 1.4% of its time with a full ROB, drastically lower than the >50% stall times seen in previous publications[4, 10-12]. Some benchmarks enter runahead very few times, as shown in Figure 7.5. The median number of times runahead is entered across all benchmarks is only 7742, with *exchange2_s_0* only entering runahead 106 times in 101M dynamic instructions. Regardless, runahead does improve the fraction of normal cycles in which the processor stalls on a full ROB. The fraction of runahead cycles with a full ROB is not shown due to a bug in how these cycles are tracked while the processor is idling.

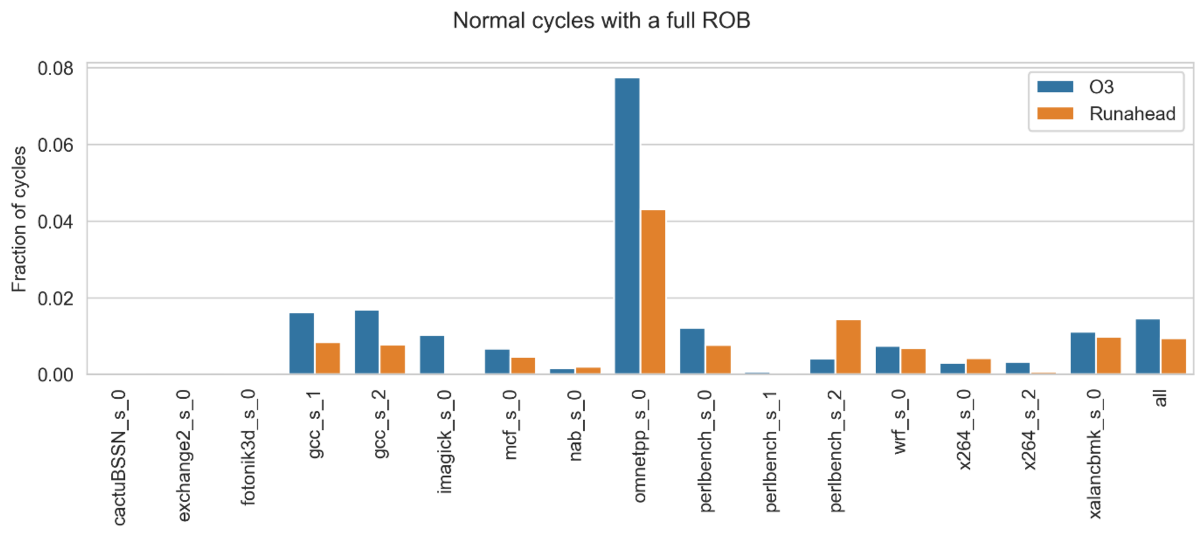


Figure 7.4: Fraction of all normal cycles in which the ROB is full.

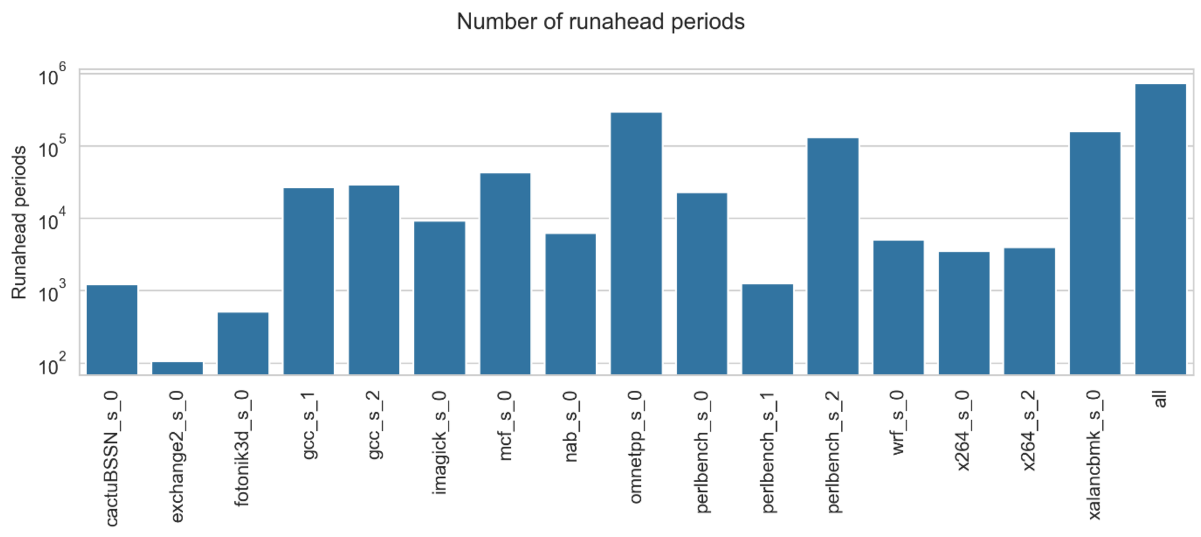


Figure 7.5: Number of times runahead was entered for each of the SPEC2017 benchmarks. Note the logarithmic scale.

As shown in Figure 7.6, the runahead processor still spends many cycles with detected LLLs at the head of the ROB without entering runahead. Across all benchmarks, roughly 67.9 million cycles are spent in this state, which is a 75.1% decrease compared to the stock CPU (~272M cycles), although unlike the runahead CPU, the OoO CPU cannot do anything to move these loads out of the ROB until they are completed. While a few of these cycles can be attributed to entry overhead, the vast majority are caused by the processor refusing to enter runahead to prevent short or overlapping periods, so these do not necessarily represent cycles that the processor could have utilized effectively.

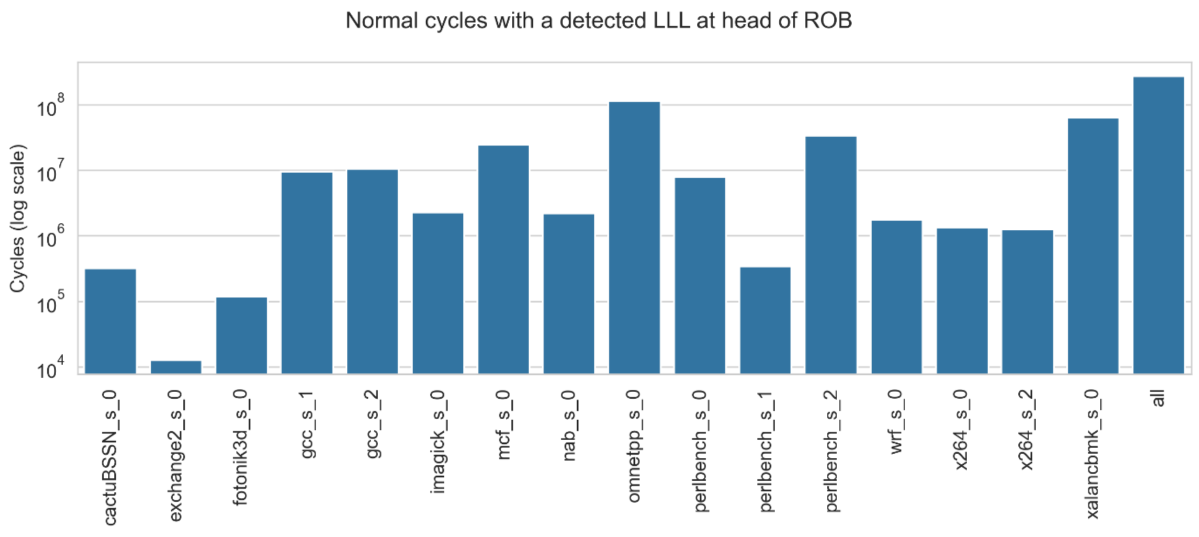


Figure 7.6: Number of normal cycles in which the runahead processor had a confirmed LLL at the head of the ROB.

7.2 Minimum work and exit deadlines

When compared to the runahead baseline, the minimum work exit policy achieves a very slight speedup in some configurations, as seen in Figure 7.7. In general, a small amount of minimum work and a short exit deadline provide the best speedups, with performance degradation occurring shortly after unrestricted minimum work exceeds 50 instructions or the unrestricted work deadline exceeds 150 cycles. The best models were the one which unconditionally delayed runahead by 25 cycles and the one which executed a minimum of 25 instructions without a deadline. These achieved a relative IPC increase of 2% and 1.2%, respectively. Combining a minimum work of 25 instructions with a 25-cycle deadline gives a 2.3% speedup compared to the runahead baseline. For NIPC, a deadline of 25 or minimum work of 200 perform the best, with a 2.4% and 2.9% NIPC increase, respectively. *exchange2_s_0* achieves a surprisingly large improvement of roughly 27.9% over the runahead baseline, but still represents a slight performance degradation when compared to the stock OoO CPU model.

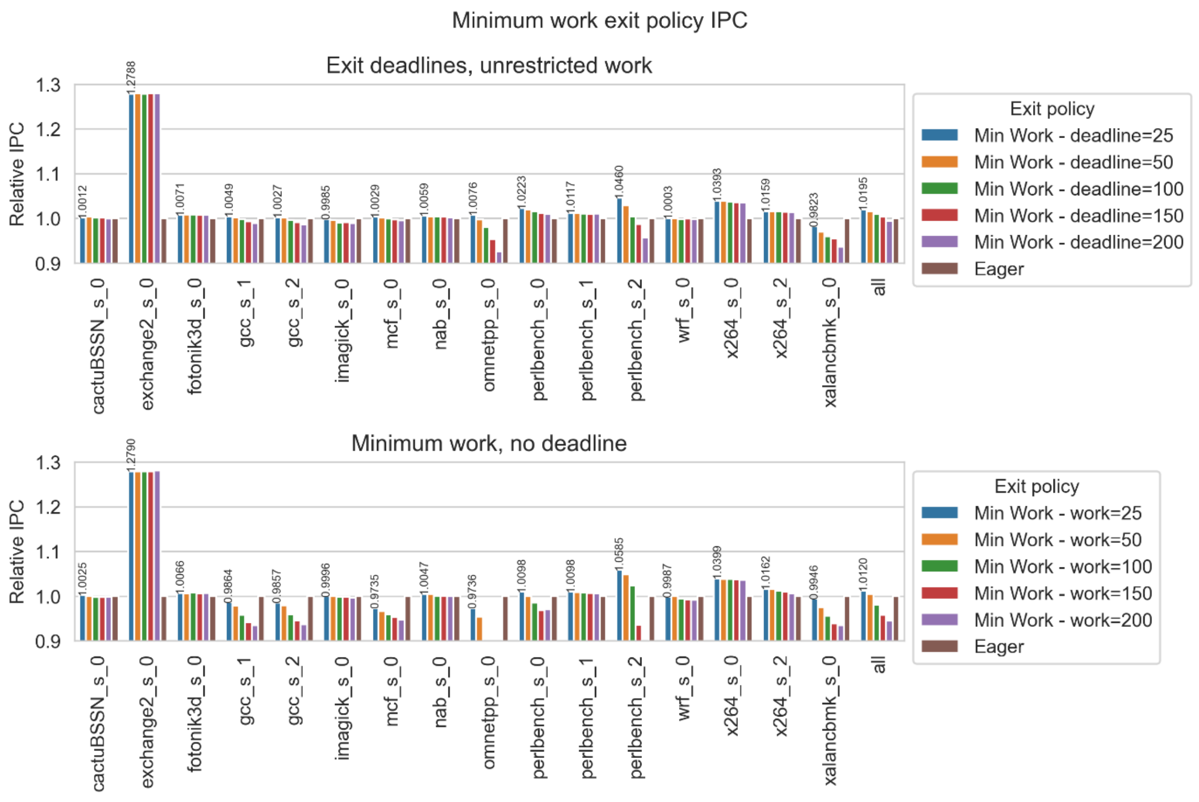


Figure 7.7: Sensitivity analysis of exit deadlines (top) and minimum work (bottom) with regard to relative IPC in a runahead model with a minimum work exit policy. The number above the best configurations show the relative IPC to the runahead baseline.

The remainder of the minimum work model results use a deadline of 25 cycles and a minimum work of 25 instructions unless otherwise is stated. Figure 7.8 shows that minimum work retires <0.1% additional instructions overall, which is expected but not a noteworthy amount. *imagick_s_0*, *omnetpp_s_0* and *wrf_s_0* end up retiring fewer instructions. The reduction is only on the order of 1000s of instructions but is still interesting as it indicates that the model can reduce the number of times the processor enters runahead. Indeed, the minimum work model enters runahead 2.5%, 3.5% and 4.3% fewer times, respectively, for these benchmarks than the runahead baseline. Overall, minimum work causes runahead to trigger 3% fewer times.

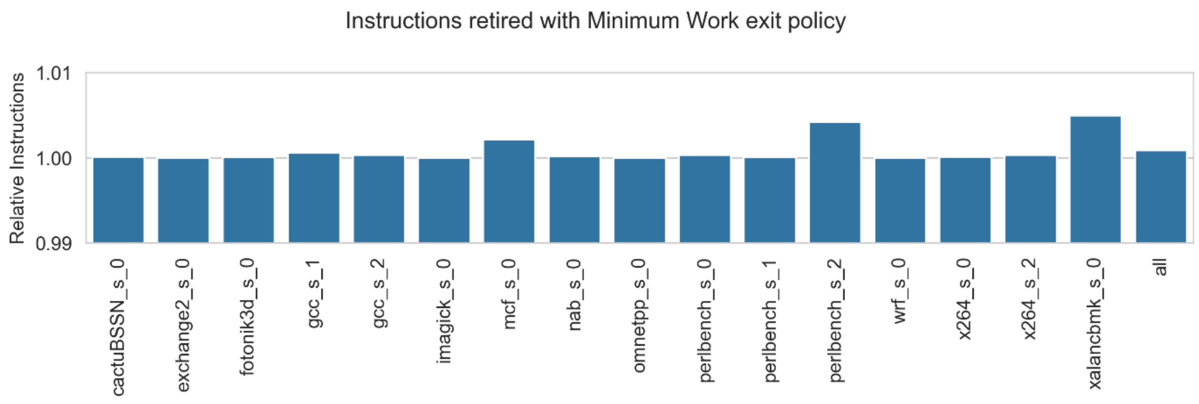


Figure 7.8: Number of instructions retired by the minimum work model relative to the runahead baseline.

The minimum work model also delivers on the target of reducing runahead stutter. The runahead baseline has interim periods shorter than 150 instructions 33.6% of the time. In the final minimum work model, this percentage is reduced to 31.7%. The most aggressive 200-instruction minimum work model reduced it to only 9.7% of all interim periods, showing that delayed exits are highly effective at reducing runahead stutter. The interim period length breakdown for the aggressive minimum work model is shown in Figure 7.9.

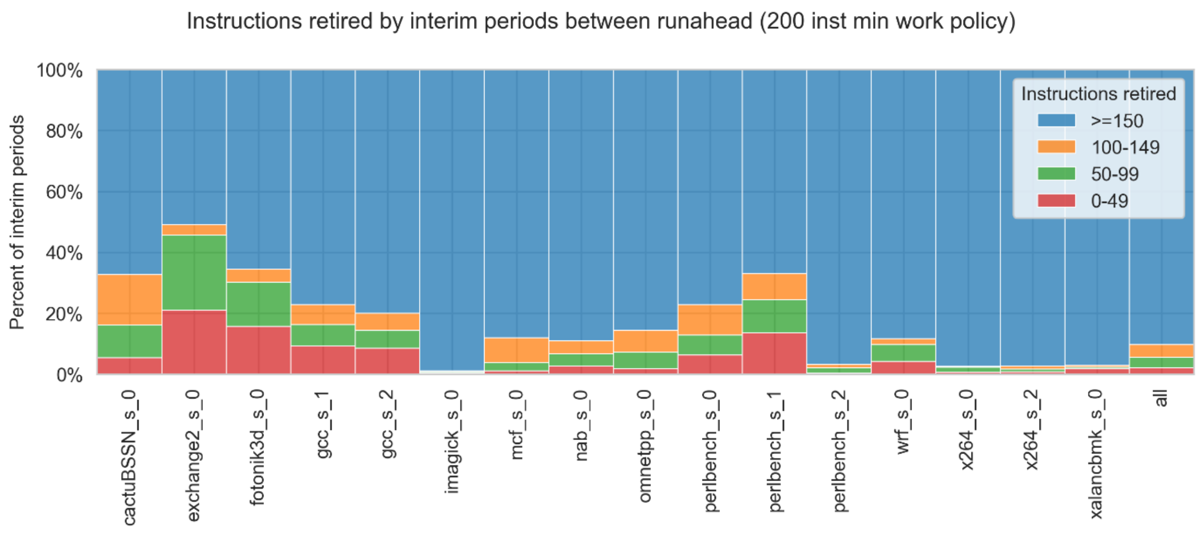


Figure 7.9: Interim period length breakdown for a runahead processor with a 200-instruction minimum work exit policy.

7.3 No Load Left Behind

The relative IPC of the NLLB model with an exit deadline of 100 cycles is shown in Figure 7.10. Compared to the runahead baseline, NLLB improves IPC by 1.9%, which is 0.4% less than the minimum work exit policy. NIPC is up by 2.2%, also less than minimum work. In other words, NLLB decidedly performs worse than the minimum work policy. *exchange2_s_0* still exhibits a striking performance improvement.

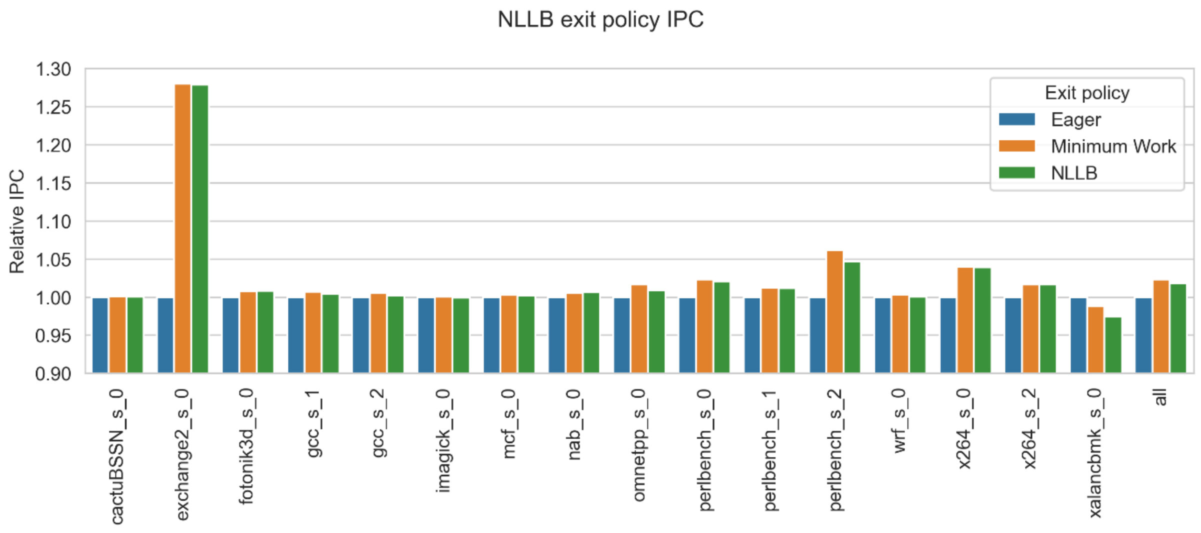


Figure 7.10: Relative IPCs of the NLLB runahead model to the runahead baseline, compared with the relative IPCs of the minimum work model.

NLLB also retires more instructions, as shown in Figure 7.11. Overall, NLLB increases the retired instruction count by 0.8%. *perlbench_s_2* sees the largest increase in retired instructions, possibly because loads are frequent, but with large dependency chains, meaning the distance between loads is large. No benchmarks retire fewer instructions when using the NLLB exit policy.

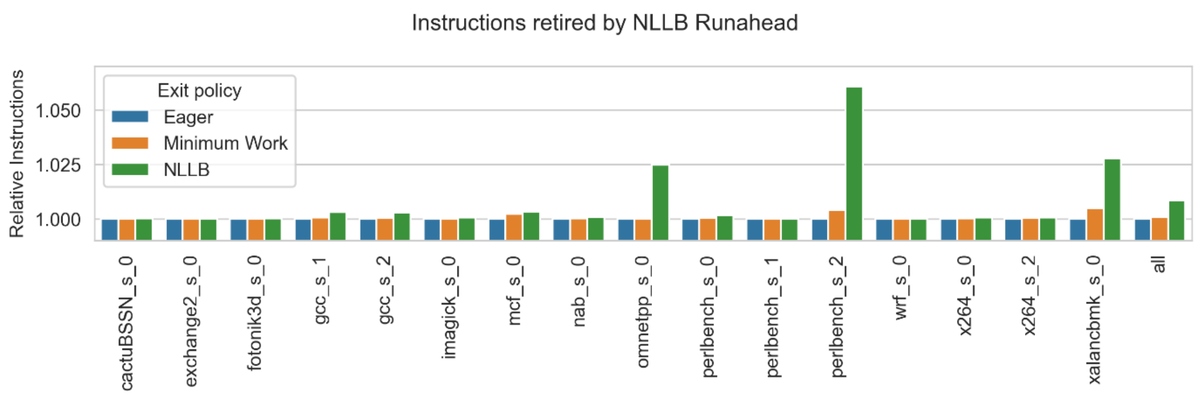


Figure 7.11: Instructions retired by the NLLB exit model, relative to the minimum work and eager exit policies.

NLLB is somewhat better at reducing runahead stutter than minimum work. Figure 7.12 shows the interim period breakdown for the NLLB exit policy. With NLLB, 25% of all interim periods retire fewer than 150 instructions, and across all benchmarks NLLB enters runahead 10.7% fewer times. In comparison, an unrestricted work policy with a 100-cycle deadline, the same as NLLB, entered runahead 19.1% times less, but this configuration does not provide the best performance. The best performing minimum work configuration only enters runahead 3% times less. Overall, NLLB is the best policy for reducing the total number of times runahead is entered.

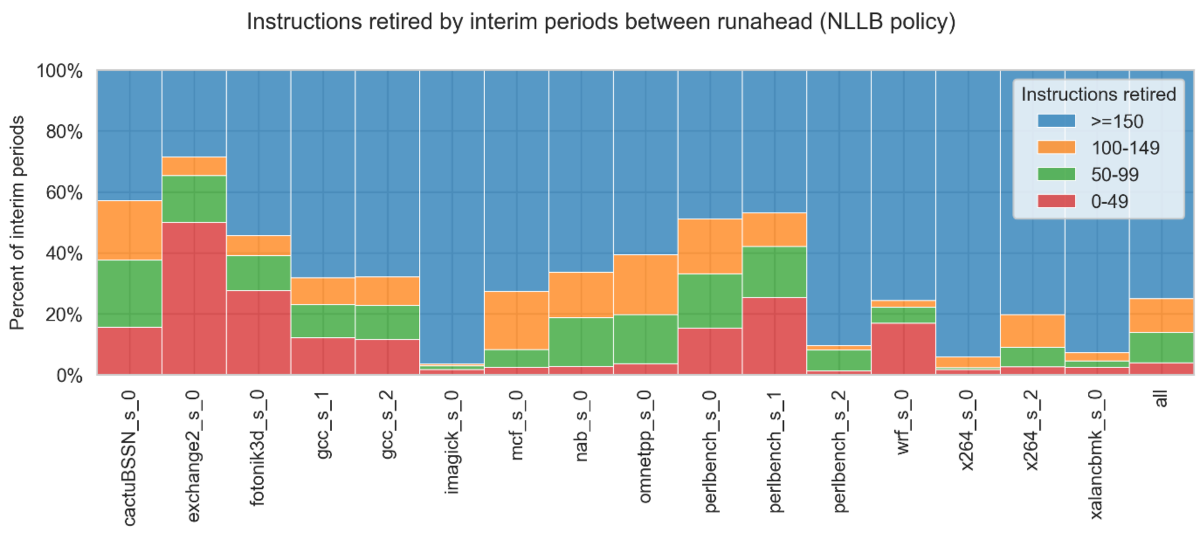


Figure 7.12: Interim period length breakdown for a runahead processor with a NLLB exit policy.

7.4 Dynamic delayed exit

The dynamic exit policy has the best IPC of all exit policies, shown in Figure 7.13. Compared to the eager exit policy, dynamic exit provides a 2.3% speedup that is within 0.01% of minimum work. The per-benchmark IPC trends are roughly the same as with the other exit policies. In terms of NIPC, a dynamic exit policy gives the best result with a 2.5% improvement over the baseline, 0.1% more than the minimum work policy.

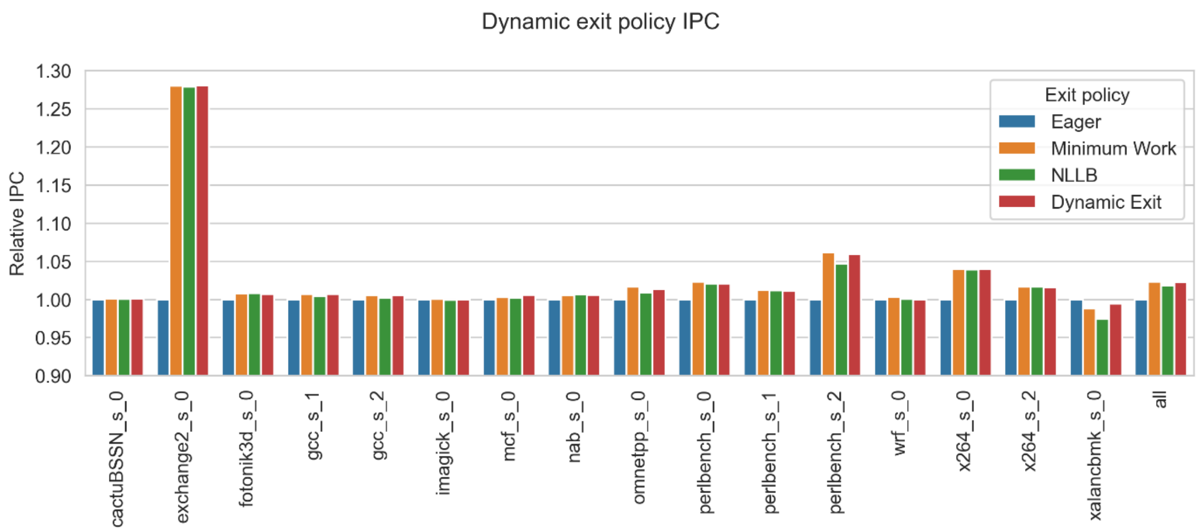


Figure 7.13: Relative IPCs of the dynamic exit runahead model to the runahead baseline, compared to all other exit policies.

Dynamic exit is also more efficient in terms of retired instruction counts, performing only 0.03% more retirements than an eager exit policy. These extra instructions number roughly 494K and bring the total number of runahead instructions to 58.1M.

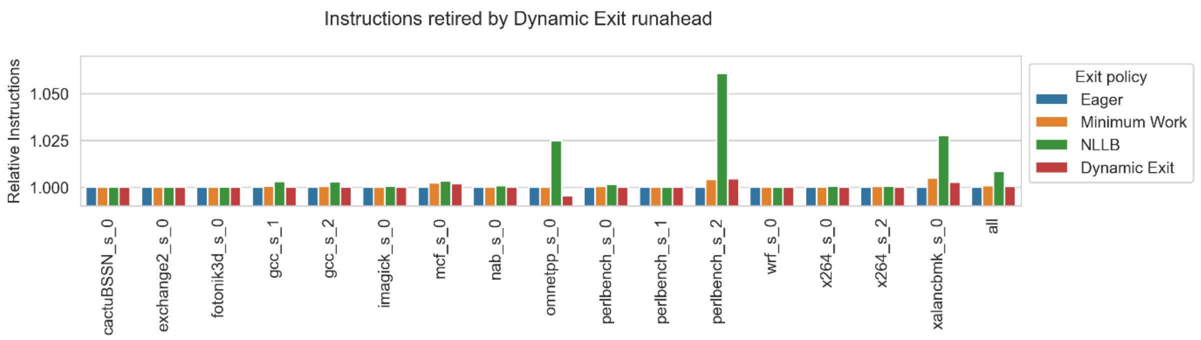


Figure 7.14: Instructions retired by every exit policy, relative to the eager exit policy.

34.3% of all interim periods retire fewer than 150 instructions with a dynamic exit policy, which is worse when compared to the runahead baseline. Figure 7.15 shows the interim period length breakdown for the dynamic exit policy. Compared to the runahead baseline, dynamic exit enters runahead 0.2% more often. Many of these extra runahead periods come from *x264_s_2*, which enters runahead 9.7% more often.

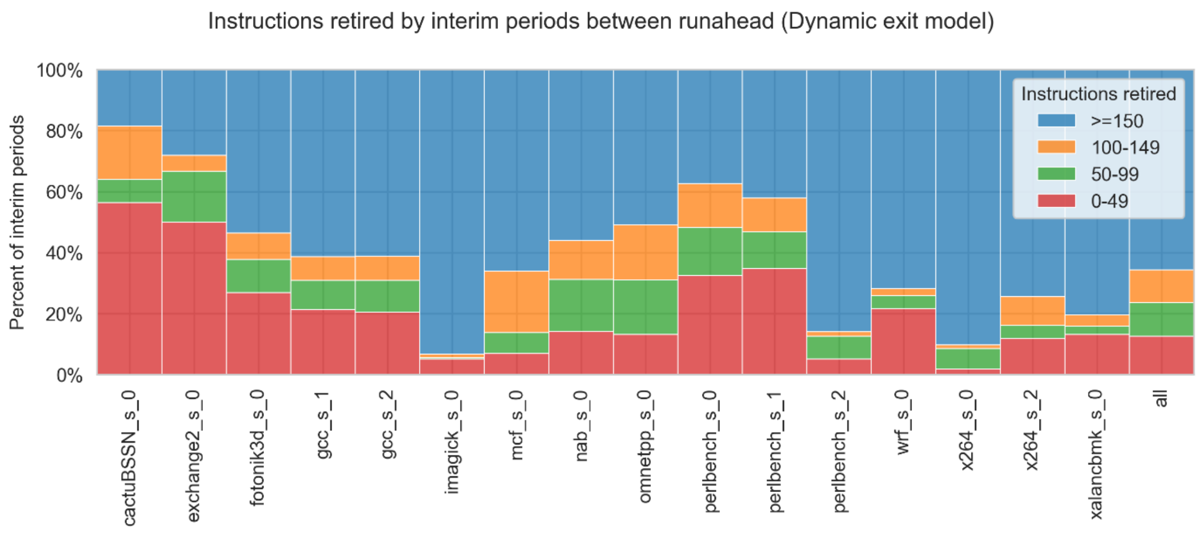


Figure 7.15: Interim period length breakdown for a runahead model with a dynamic exit policy.

7.5 Delayed runahead period metrics

This chapter presents a few metrics exclusive to delayed runahead periods. First, Figure 7.16 shows the additional cycles spent in runahead mode due to the delayed exit. NLLB clearly spends the most time, owing to its longer exit deadline and lack of restriction on maximum work. Dynamic exit spends orders of magnitude fewer cycles in delayed runahead than both minimum work and NLLB. *cactuBSSN_s_0*, *fotonik3d_s_0* and *imagick_s_0* never delay exit under the dynamic policy. The amount of pseudoretired instructions in delayed runahead follows the same trend, as shown in Figure 7.17, with dynamic exit pseudoretiring orders of magnitude fewer instructions than the alternatives.

Figure 7.18 shows that compared to the minimum work and NLLB policies, the dynamic exit policy significantly increases the relative number of loads executed in delayed runahead. Across all benchmarks, the percentage of pseudoretired instructions that are loads are 18.5%, 15.6% and 57.7% for minimum work, NLLB and dynamic exit, respectively.

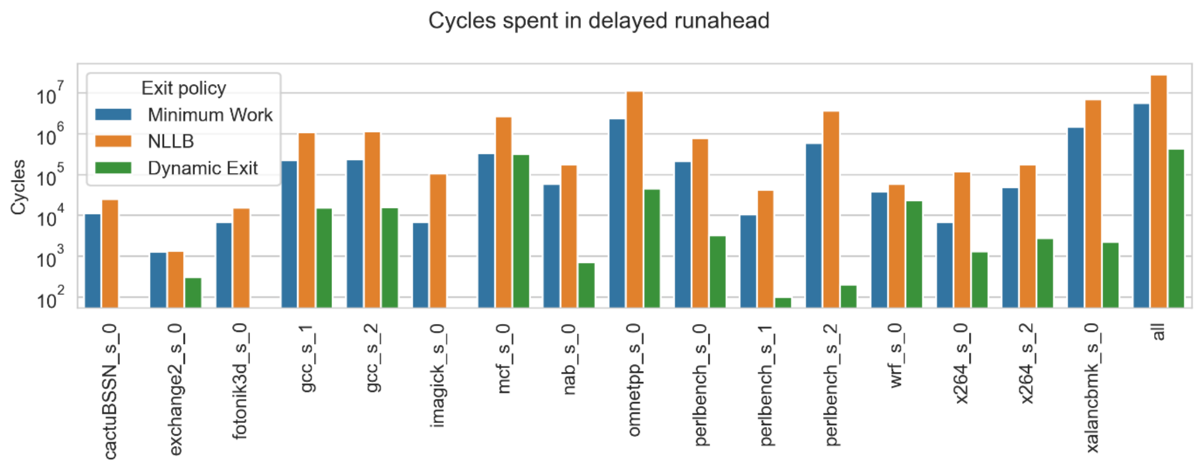


Figure 7.16: Cycles spent in delayed runahead for each exit policy. Note the log scale.

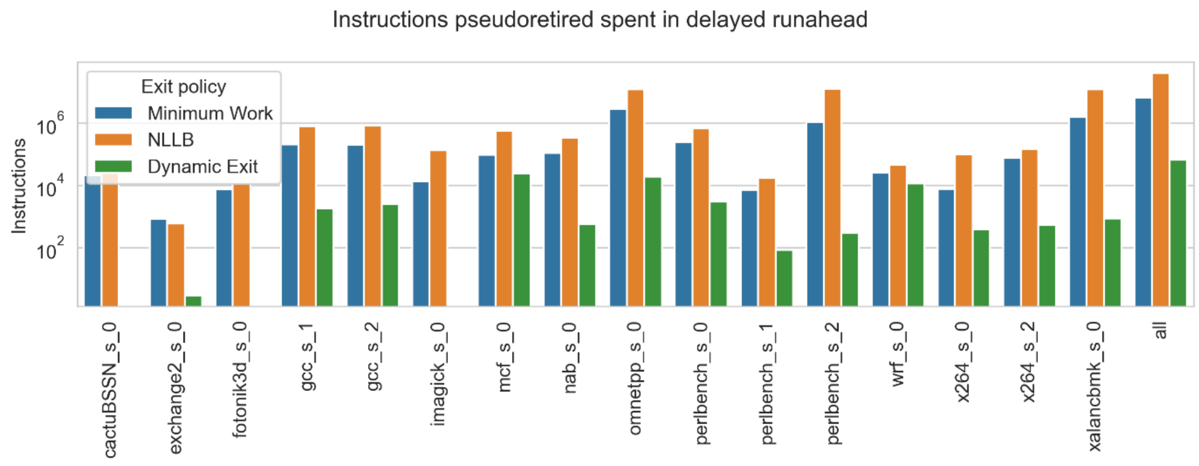


Figure 7.17: Instructions pseudoretired in delayed runahead by each exit policy. Note the log scale.

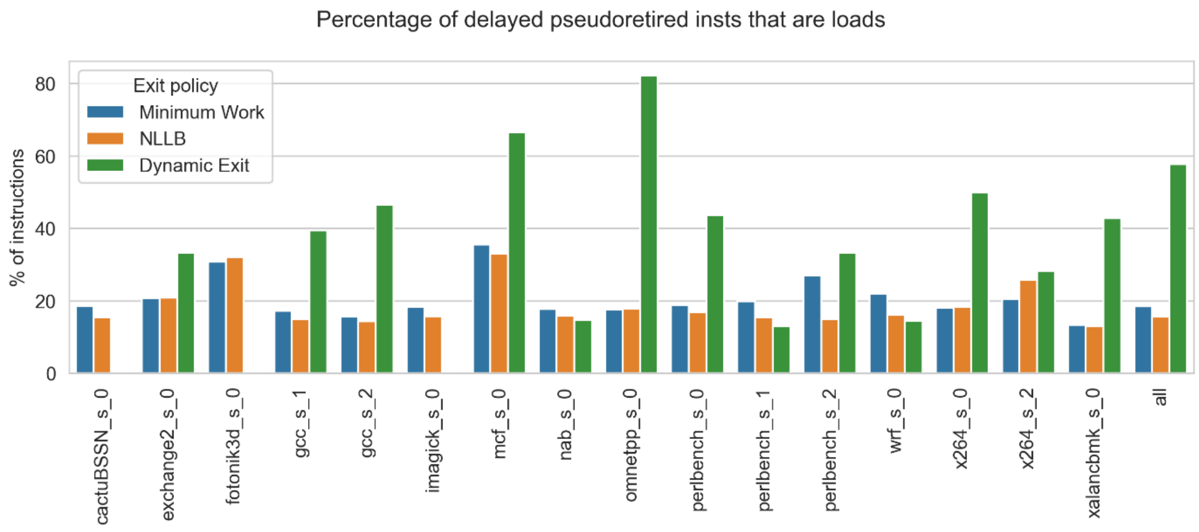


Figure 7.18: Percentage of pseudoretired instructions that were loads in delayed runahead for each exit policy.

8 Discussion of runahead & delayed exit

8.1 Runahead performance degradation

As shown in section 7.1, my runahead model degrades performance compared to an out-of-order processor. There is overwhelming evidence in previous publications that runahead has major performance benefits[4-12, 17-21], which lends confidence to the assertion that there must be a flaw in the runahead model. This chapter discusses flaws in the ROIs, possible explanations for the performance degradation and what this means for the results of this study.

A fundamental error in this study is that while many of the ROIs are highly representative of their respective benchmarks, they are not representative of the problem which runahead execution attempts to solve. To begin with, the ROIs were extracted from the first 50B dynamic instructions of each benchmark even though they last, on average, for over 22 trillion instructions[2]. Figure 7.4 showed that some benchmarks practically do not struggle with full-ROB stalls whatsoever. With the baseline runahead model, *cactuBSSN_s_0*, a benchmark found to be memory intensive by Sing and Awasthi[2], only entered runahead 1206 times in the full ROI. In these cases, it can be argued that runahead simply does not occur enough to reasonably conclude anything about its actual impact on the performance of the benchmark.

However, there are benchmarks which do enter runahead an appreciable number of times. *omnetpp_s_0* entered runahead mode nearly 300K times, *perlbench_s_2* about 130K times and *xalancbmk_s_0* around 159K times. Arguably, this should be enough to showcase the effect runahead can have, yet *omnetpp_s_0* and *perlbench_s_2* experienced some of the worst performance degradations out of all the benchmarks, even when adjusting for overhead. Meanwhile, *xalancbmk_s_0* has a lesser true performance degradation, yet a performance improvement when adjusting for overhead. These benchmarks also show the largest improvements in load-to-use cycles for normal mode loads and reduce the number of cycles the processor is stalled when using runahead.

If the overhead metrics were perfect, they would, in theory, capture all cycles lost to entering and exiting runahead. Adjusting IPC by removing these cycles should then show a performance gain. If the processor were not capable of runahead, any remaining cycles would be either productive normal cycles (including those in which there is a blocking LLL, but the ROB is filling) or cycles which the processor would have been stalled on. Assuming the overhead metrics are accurate, this shows that runahead may be misusing productive normal mode cycles. The overhead is also likely overestimated because the first instruction to go back through the pipeline after runahead is the RCL, which has now been prefetched. Thus, when the exit overhead period is considered to end, the readiness state of instructions in the IQ is improved compared to before runahead.

I also found that when using a dynamic delayed exit policy, simulation statistics differed from the runahead baselines in the benchmarks that never delayed exit. It's difficult to tell if this is due to the benchmarks being non-deterministic, but it may be evidence of a correctness bug in the runahead implementation. As mentioned in section 5.8, the implementation was tested with a comparatively simple program in syscall emulation mode. While the full system SPEC2017 benchmarks run until the end of their ROIs without crashing, this is not a guarantee of correctness.

Side-effects to the frontend is likely not the cause for the performance degradation. As was mentioned in section 7.1, the processor experiences slightly fewer branch mispredictions outside of runahead. Additionally, infinite instruction cache experiments still show a performance degradation, so the instructions fetched during runahead should not be causing significant L1-I cache pollution. Runahead mode does not action any faults, and so it should not incur any costly paging operations either.

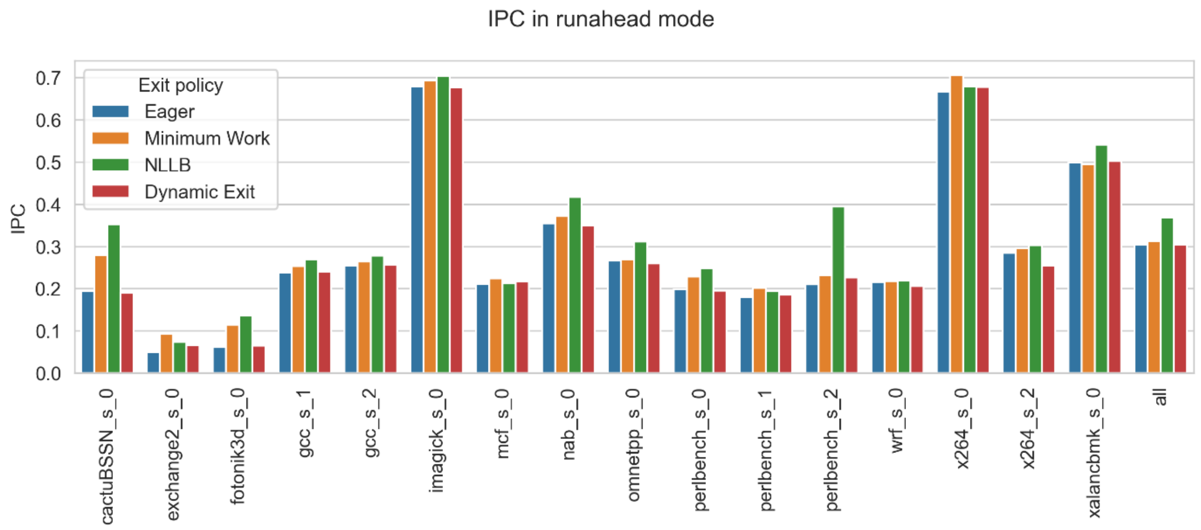


Figure 8.1: IPC of the runahead model while it is in runahead mode for each of the exit policies.

One aspect of the runahead model that has not been thoroughly inspected is its behavior while in runahead mode. In all benchmarks, the CPU model exhibits uncharacteristically low IPCs while in runahead mode. It's rare for the model to breach 0.4 IPC in runahead and it hovers around 0.3 across all benchmarks, which is a little less than half of the overall IPC of the model. In other words, the processor tends to slow down drastically when it enters runahead. The reason for this is not entirely clear, but it could be that runahead instructions are not properly handled, causing them to fill pipeline structures and block execution until runahead is exited. If this is the case, runahead performs little to no useful work while incurring overhead penalties. This would not necessarily be visible in the load-to-use results presented in chapter 7 because many LLLs are replaced by their prefetched instance once runahead exits.

I believe that the performance degradation can mostly be explained by the fact that the processor often enters runahead when it normally would not have stalled on a full ROB. In fact, some benchmarks never entered runahead when the processor was configured to use a lazy entry policy that waits for the ROB to fill completely before entering runahead. In cases where the processor enters runahead, but would not have stalled otherwise, runahead incurs an overhead penalty consisting of cycles which a normal OoO CPU would simply have used for normal execution. In other words, such runahead periods are speculating purely at the cost of normal execution cycles, not stall cycles. This increases the number of cycles taken to execute the program and the number of instructions processed, leading to both a performance and efficiency degradation.

The performance bug makes it difficult to compare runahead to the stock OoO CPU, but since all delayed exit models use the same base mechanism, it should be appropriate to compare results among them. The results presented in this thesis are not well suited to conclude anything about runahead or any of the delayed exit policies when compared to non-runahead out-of-order cores. However, the results should still indicate whether delaying runahead can be a performance-improving modification to runahead schemes.

8.2 Minimum work delayed exit

The minimum work policy is successful in increasing the performance of the runahead model, although this is not necessarily due to any conscious efforts to do so in the design of the policy. The performance benefit comes from an increase in prefetched loads, but since the policy does not account for the types of instructions being executed in delayed runahead, the policy is wholly reliant on a high density of loads in the instruction stream. If delayed runahead takes place during a compute intensive portion of the program, the minimum work exit policy only wastes cycles doing compute work that will be discarded. For each such isolated case, delayed runahead through minimum work causes a performance degradation when compared to an eager exit policy.

A large minimum work was also shown to severely reduce runahead stutter, although performance is also degraded for those configurations. At the same time, NIPC increases the most for these configurations. This means that while the additional time spent in runahead does prefetch additional loads, the prefetches do not adequately make up for the time spent issuing these prefetches. This is supported by Figure 7.17 and Figure 7.18, confirming that extra loads are indeed issued in delayed runahead, but at a low rate.

8.3 NLLB delayed exit

The "No Load Left Behind" policy was the second iteration on delayed exit policies and was motivated by the fact that the minimum work policy does not guarantee execution of loads. By design, NLLB guarantees that, if exit from runahead is delayed, at least one load is pseudoretired unless the exit deadline expires.

This approach is still flawed, because NLLB does not consider how many instructions it must execute to reach the youngest load in the ROB. In the system configuration used in this study, NLLB could be asked to execute over 200 instructions before reaching the youngest load if the ROB is near full. With tight exit deadlines, the processor is not even likely to make it to the target load. Thus, NLLB struggles with the same issue as minimum work - there is no guarantee that it performs useful work.

In NLLB's case, however, the issue is exaggerated because the exit policy is more ambitious. In my simulations, it was given a longer exit deadline, meaning NLLB potentially does even more useless work than the minimum work policy was allowed to. Additionally, because NLLB does not care about the instructions before the youngest load in the ROB, it risks executing many long-latency instructions (of any type) in delayed runahead. In these cases, NLLB exhausts the deadline while issuing few to no useful prefetches, effectively wasting cycles. In the worst case, the processor could be on the wrong control path and execute a halt instruction. This is exceedingly rare, but it does happen in the benchmarks and was the cause of a soft-lock bug during development.

NLLB also does not address one of the core issues of minimum work, namely being overly reliant on a high load density in the instruction stream. As shown in section 7.5, NLLB pseudoretires a smaller fraction of loads when compared to minimum work. This is because it typically attempts to execute more instructions than minimum work while having a longer deadline near-indiscriminately.

8.4 Dynamic delayed exit

Taking lessons from the failings of the minimum work and NLLB policies, the dynamic delayed exit policy is designed to guarantee that delayed time is spent more effectively. This is done through two measures. First, the dynamic exit policy only enters runahead if the instruction stream is filtered to a load chain. Second, runahead is only delayed if there are loads close by. Combined, these conditions often ensure that the processor will execute a prefetch soon after delayed runahead begins, and that the upcoming instruction stream has a reasonably high load density.

While the dynamic exit policy improves performance by a nearly unnoticeable amount over minimum work, it achieves this speedup in orders of magnitude fewer cycles and with fewer instructions retired. This can be explained by the 57.7% load rate in delayed runahead. The dynamic exit policy’s performance improvement comes from guaranteeing that the load density in the delayed instruction stream is high.

Despite the initial assumption that runahead stutter is a cause of poor runahead performance, the dynamic exit policy increased stutter by a very slight amount. This indicates that while the dynamic policy prefetches additional loads, it doesn’t necessarily prefetch those loads which are critical to prevent stutter. It also shows that delaying exit from runahead can be positive for performance even without improving runahead stutter. The exact reason why stutter increases is not clear to me but may be because critical cache blocks are evicted by the delayed runahead prefetches.

9 Future work

9.1 Improving the simulation model

As discussed in section 8.1, the performance degradation of the runahead model brings the correctness of the model into question. The gem5 source code includes a checker CPU[34] which, while mostly undocumented, seems to execute instructions independently to verify the results of the simulation model. It is unclear to me whether it is currently functional as related source code lines have been commented out of the O3CPU model's parameter configurations. If functional, however, this could be used to verify the correctness of the runahead model. An alternative is to continue debugging the model by inspecting its behavior. For instance, it would be good to verify that the runahead model does not stay in runahead for much longer than the corresponding RCL would have stalled the processor for if it did not use runahead. Debugging the processor's behavior in runahead mode could also be insightful as it currently has uncharacteristically poor performance in runahead.

Certain parts of the runahead implementation are currently unfinished. Particularly, vector operations are not supported by the runahead model because vector registers are not checkpointed. This is due to gem5 implementing variably sized vector registers, complicating their checkpointing when compared to other register types. It should be noted that this was not a problem in this study because none of the benchmarks used vector instructions. While gem5 supports multiple hardware threads, support for them was not a priority during development of the runahead model and it is likely that it will misbehave or crash if multiple hardware threads are used. Again, the lack of hardware thread support is because none of the benchmarks use more than one hardware thread.

The current instruction stream filtering is somewhat limited compared to the one proposed by Hashemi and Patt[12], who additionally introduced a runahead buffer to replace the fetch-decode frontend during runahead. Implementing such a buffer should speed up the runahead frontend and amplify its prefetching effect, also increasing the effect of delayed runahead. Alternatively, introducing something like a stalling slice table[10] could increase runahead's coverage even further.

As mentioned in chapter 5, the current structure of the source code makes it difficult to make changes to core gem5 features. This is because any such changes would impact the traceability offered by source control. Currently, to track and apply any changes to gem5, patch files need to be generated and applied to the gem5 source code. To fix this, I would suggest moving the runahead CPU model directly inside a fork of the gem5 source code[34].

To facilitate further development of the model, the source code is published openly on GitHub[15]. The code repository includes instructions on how to setup the project and build gem5 with the runahead model extension. There are also instructions and guidelines for working with the source code, particularly how to setup debugging.

9.2 Delayed exit policies

The current dynamic delayed exit policy shows that delaying exit from runahead has the potential to improve performance for traditional runahead schemes. However, the policy can be improved. Currently, loads close to the head of the ROB are used as evidence of high load density, but this is not a guarantee. Long dependence chains for which delaying runahead is not worth it can still trigger a delayed exit if the load happens to be close to the head of the ROB by pure chance. One way to fix this is to simply check the length of the dependence chain, although an appropriate cutoff length would have to be determined.

Alternative load-density based approaches to delayed exits could also be explored. Such policies would determine if and how long to delay runahead for based on the density of loads in the instruction stream. For example, the processor can compute the fraction of loads present in the ROB to estimate load density once the RCL returns. If load density is high, it might be a good idea to delay runahead for a long time, a short time if load density is medium with nearby loads and to eagerly exit if the load density is low. MLP distance prediction[8] could also be used to determine when and how long to delay runahead for.

10 Conclusion

This thesis has found that traditional runahead execution schemes experience runahead stutter, in which the processor rapidly re-enters runahead after exiting it. This motivated a study of the effect of delaying exit from runahead execution in a traditional runahead execution scheme. The hope was that delayed exits from runahead would prefetch additional loads that allow the processor to continue execution for longer before encountering new runahead-triggering cache misses. An out-of-order CPU model in the gem5 simulator was modified to support traditional runahead and three different delayed exit policies were implemented. When used to simulate 16 of the SPEC CPU2017 benchmarks, the runahead baseline was found to degrade performance compared to the stock out-of-order CPU. This is assumed to be due to an implementation bug or flawed selection of benchmark ROIs. Benchmarking the delayed exit policies against a runahead baseline showed that they improve performance, making up for the delayed cycles with additional prefetches. The experiments show that delayed runahead can be effective at reducing runahead stutter, but that this is not critical to performance. Instead, ensuring that the instruction stream has a high density of loads is found to be important as it allows the processor to make effective use of the delayed cycles.

References

[1] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-Generation Compute Benchmark," presented at the Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, Berlin, Germany, 2018. [Online]. Available: https://doi.org/10.1145/3185768.3185771.

[2] S. Singh and M. Awasthi, "Memory Centric Characterization and Analysis of SPEC CPU2017 Suite," p. arXiv:1910.00651doi: 10.48550/arXiv.1910.00651.

[3] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers,* vol. 37, no. 5, pp. 562-573, May 1988.

[4] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 12-12 Feb. 2003 2003, pp. 129-140, doi: 10.1109/HPCA.2003.1183532.

[5] J. D. Dundas and T. N. Mudge, "Using Stall Cycles to Improve Microprocessor Performance," Advanced Computer Architecture Laboratory, Ann Arbor, Michigan, USA, Technical Report CSE-TR-301-96, September 1996. Accessed: 02.06.2023. [Online]. Available: https://tnm.engin.umich.edu/wp-content/uploads/sites/353/2019/04/1996-Using-Stall-Cycles-to-Improve-Microprocessor-Performance.pdf

[6] O. Mutlu, "Efficient runahead execution processors," PhD Dissertation, Electrical and Computer Engineering, The University of Texas at Austin, Austin, Texas, 2006. [Online]. Available: https://repositories.lib.utexas.edu/handle/2152/2778

[7] O. Mutlu, "Efficient runahead execution processors: A power-efficient processing paradigm for tolerating long main memory latencies," ed: ETH Zurich, 2006.

[8] K. Van Craeynest, S. Eyerman, and L. Eeckhout, "MLP-Aware Runahead Threads in a Simultaneous Multithreading Processor," Berlin, Heidelberg, 2009: Springer Berlin Heidelberg, in High Performance Embedded Architectures and Compilers, pp. 110-124.

[9] T. Ramírez, A. Pajuelo, O. J. Santana, O. Mutlu, and M. Valero, "Efficient Runahead Threads," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 11-15 Sept. 2010 2010, pp. 443-452.

[10] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise Runahead Execution," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 22-26 Feb. 2020 2020, pp. 397-410, doi: 10.1109/HPCA47549.2020.00040.

[11] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector Runahead," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 14-18 June 2021 2021, pp. 195-208, doi: 10.1109/ISCA52012.2021.00024.

[12] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 5-9 Dec. 2015 2015, pp. 358-369, doi: 10.1145/2830772.2830812.

[13] S. Pruett and Y. Patt, "Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches," presented at the MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, 2021. [Online]. Available: https://doi.org/10.1145/3466752.3480053.

[14] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 15-19 Oct. 2016 2016, pp. 1-12, doi: 10.1109/MICRO.2016.7783764.

[15] *gem5-runahead*. (2023). GitHub. [Online]. Available: https://github.com/halworsen/gem5-runahead

[16] S. Pakalapati and B. Panda, "Bouquet of Instruction Pointers: Instruction Pointer Classifier based Hardware Prefetching," presented at the ISCA 2019: The 46th International Symposium on Computer Architecture, June 2019, 2019. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/pdfs/Bouquet.pdf.

[17] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero, "Runahead Threads to improve SMT performance," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 16-20 Feb. 2008, pp. 149-158, doi: 10.1109/HPCA.2008.4658635.

[18] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero, "Code Semantic-Aware Runahead Threads," in *2009 International Conference on Parallel Processing*, 22-25 Sept. 2009 2009, pp. 437-444, doi: 10.1109/ICPP.2009.17.

[19] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector Runahead for Indirect Memory Accesses," *IEEE Micro,* vol. 42, no. 4, pp. 116-123, 2022, doi: 10.1109/MM.2022.3163132.

[20] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled Vector Runahead," presented at the Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, Toronto, ON, Canada, 2023. [Online]. Available: https://doi.org/10.1145/3613424.3614255.

[21] A. Naithani and L. Eeckhout, "Reliability-Aware Runahead," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2-6 April 2022 2022, pp. 772-785, doi: 10.1109/HPCA53966.2022.00062.

[22] M. W. Halvorsen, "Exploring Runahead Execution in gem5," Norwegian University of Science and Technology, June 2023.

[23] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," presented at the Proceedings of the 11th international conference on Supercomputing, Vienna, Austria, 1997. [Online]. Available: https://doi.org/10.1145/263580.263597.

[24] O. Mutlu, K. Hyesoon, and Y. N. Patt, "Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses," *IEEE Transactions on Computers,* vol. 55, no. 12, pp. 1491-1508, 2006, doi: 10.1109/TC.2006.191.

[25] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," presented at the Proceedings of the fifth international conference on Architectural support for programming languages and operating systems, Boston, Massachusetts, USA, 1992. [Online]. Available: https://doi.org/10.1145/143365.143486.

[26] D. M. Tullsen, "Simulation and Modeling of a Simultaneous Multithreading Processor," presented at the 22nd Annual Computer Measurement Group Conference, December, 1996.

[27] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: a simulation framework for CPU-GPU computing," presented at the Proceedings of the 21st international conference on Parallel architectures and compilation techniques, Minneapolis, Minnesota, USA, 2012. [Online]. Available: https://doi.org/10.1145/2370816.2370865.

[28] hpsresearchgroup. "Scarab: Joint HPS and ETH Repository to work towards open sourcing Scarab and Ramulator." https://github.com/hpsresearchgroup/scarab (accessed June 1, 2023.

[29] N. Gober *et al.*, "The Championship Simulator: Architectural Simulation for Education and Competition," 2022, doi: 10.48550/arXiv.2210.14324.

[30] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations," presented

at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), November, 2011.

[31] J. Lowe-Power *et al.*, "The gem5 Simulator: Version 20.0+," p. arXiv:2007.03152doi: 10.48550/arXiv.2007.03152.

[32] R. Kumar, "Simulator recommendations," private communication. February, 2023.

[33] J. Lowe-Power. "gem5 documentation." https://www.gem5.org/documentation/ (accessed 06.06, 2023).

[34] *gem5 v22.0.0.2*. (2022). GitHub. [Online]. Available: https://github.com/gem5/gem5/tree/v22.0.0.2

[35] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," presented at the Proceedings of the 25th annual international symposium on Computer architecture, Barcelona, Spain, 1998. [Online]. Available: https://doi.org/10.1145/279358.279378.

[36] A. B. Kvalsvik, "Assistance with setting up SPEC2017 benchmarks in gem5," private communication. March, 2023.

[37] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and More Flexible Program Analysis," *Journal of Instruction Level Parallelism,* vol. 7, September 2005.

[38] J. Doweck *et al.*, "Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake," *IEEE Micro,* vol. 37, no. 2, pp. 52-62, 2017, doi: 10.1109/MM.2017.38.

[39] Micron, "Micron MT41J512M8," MT41J512M8 datasheet. 2009.

Appendix A

Matrix multiplication test program

This source code is also available on GitHub: https://github.com/halworsen/gem5-runahead/blob/d01b53017ad37bc140e6597fde2483d9964696ae/gem5-extensions/configs/test/matmul.cc

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <cassert>
#include <vector>
#include <random>
#include <algorithm>

struct Matrix {
private:
    size_t rows;
    size_t columns;
    bool dataAllocated = false;
    long long int *data;
public:
    Matrix(size_t rows, size_t cols) : rows(rows), columns(cols) {
        data = (long long int*) calloc(rows * cols, sizeof(long long int));
        dataAllocated = true;
    }

    ~Matrix() {
        if (dataAllocated)
            free(data);
    }

    size_t getRows() { return rows; }
    size_t getCols() { return columns; }
    long long int get(size_t x, size_t y) { return data[rows * x + y]; }
    void set(size_t x, size_t y, long long int element) { data[rows * x + y]
= element; }

    void print() {
        for (int r = 0; r < rows; r++) {
            printf("[ ");
            for (int c = 0; c < columns; c++) {
                printf("%li ", get(r, c));
            }
            printf("]\n");
        }
    }
};
```

```
// Populate a matrix with random values
void populateMatrix(Matrix *matrix) {
    for (int r = 0; r < matrix->getRows(); r++) {
        for (int c = 0; c < matrix->getCols(); c++) {
            matrix->set(r, c, (std::rand() % 1000000) - 500000);
        }
    }
}

void multiplyMatrices(Matrix *a, Matrix *b, Matrix *out) {
    assert(a->getCols() == b->getRows());

    for (int rA = 0; rA < a->getRows(); rA++) {
        float progress = ((float)rA / (float)a->getRows()) * 100.0f;
        printf("Progress: %f%\n", progress);

        for (int cB = 0; cB < b->getCols(); cB++) {
            for (int cA = 0; cA < a->getCols(); cA++) {
                long long int cell = out->get(rA, cB);
                cell += a->get(rA, cA) * b->get(cA, cB);
                out->set(rA, cB, cell);
            }
        }
    }
}
```

```
void multiplyMatricesRandom(Matrix *a, Matrix *b, Matrix *out, unsigned int
seed) {
    assert(a->getCols() == b->getRows());

    // make a vector of indices for matrix A's rows/columns
    std::vector<int> aRowIdxs;
    std::vector<int> aColIdxs;
    std::vector<int> bColIdxs;

    for (int rA = 0; rA < a->getRows(); rA++)
        aRowIdxs.push_back(rA);
    for (int cA = 0; cA < b->getCols(); cA++)
        aColIdxs.push_back(cA);
    for (int cB = 0; cB < b->getCols(); cB++)
        bColIdxs.push_back(cB);

    // shuffle
    auto rng = std::default_random_engine(seed);
    std::shuffle(aRowIdxs.begin(), aRowIdxs.end(), rng);
    std::shuffle(aColIdxs.begin(), aColIdxs.end(), rng);
    std::shuffle(bColIdxs.begin(), bColIdxs.end(), rng);

    int prog = 0;
    for (auto i = aRowIdxs.begin(); i != aRowIdxs.end(); i++) {
        int rA = *i;

        float progress = ((float)(prog++) / (float)a->getRows()) * 100.0f;
        printf("Progress: %f%\n", progress);

        for (auto j = bColIdxs.begin(); j != bColIdxs.end(); j++) {
            int cB = *j;
            for (auto k = aColIdxs.begin(); k != aColIdxs.end(); k++) {
                int cA = *k;
                long long int cell = out->get(rA, cB);
                cell += a->get(rA, cA) * b->get(cA, cB);
                out->set(rA, cB, cell);
            }
        }
    }
}
```

```cpp
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: matmul MATRIX_SIZE RANDOM\n");
        return 1;
    }

    int matrixSize = std::stoi(std::string(argv[1]));
    printf("Matrix size: %ix%i\n", matrixSize, matrixSize);

    bool random = (bool) std::stoi(std::string(argv[2]));
    printf("Random: %s\n", random ? "yes" : "no");

    unsigned int seed = 85354712;
    std::srand(seed);

    Matrix matrixA = Matrix(matrixSize, matrixSize);
    Matrix matrixB = Matrix(matrixSize, matrixSize);
    populateMatrix(&matrixA);
    populateMatrix(&matrixB);

    printf("Matrix A:\n");
    matrixA.print();
    printf("Matrix B:\n");
    matrixB.print();

    Matrix matrixC = Matrix(matrixA.getRows(), matrixB.getCols());
    if (random)
        multiplyMatricesRandom(&matrixA, &matrixB, &matrixC, seed);
    else
        multiplyMatrices(&matrixA, &matrixB, &matrixC);

    printf("Result:\n");
    matrixC.print();

    return 0;
}
```

Appendix B

List of additional statistics in the runahead CPU model

CPU statistics

runaheadCycles	Total cycles spent in runahead
realCycles	Total cycles spent in normal mode
numROBFullCycles	Number of cycles starting with a full ROB
numRealROBFullCycles	Number of normal mode cycles starting with a full ROB
pseudoRetiredInsts	Number of pseudoretired instructions
runaheadCpi	CPI in runahead mode
runaheadIpc	IPC in runahead mode
realCpi	CPI in normal mode
realIpc	IPC in normal mode
runaheadPeriods	Total number of runahead periods
runaheadCycleDist	Distribution of cycles spent in runahead periods
refusedRunaheadEntries	Number of times the CPU refused to enter runahead, by cause
instsPseudoRetiredPerPeriod	Histogram of instructions retired in runahead
instsFetchedBetweenRunahead	Distribution of instructions fetched in the interim period between two runahead periods
instsRetiredBetweenRunahead	Distribution of instructions retired in the interim period between two runahead periods
triggerLLLinFlightCycles	Histogram of number of cycles a load has been in-flight when it triggered runahead
dependenceChainLength	Distribution of identified dependence chain lengths
intRegPoisoned	Number of times an integer reg was poisoned
intRegCured	Number of times an integer reg had poison cleared
floatRegPoisoned	Number of times a float reg was poisoned
floatRegCured	Number of times a float reg had poison cleared
vecRegPoisoned	Number of times a vector reg was poisoned
vecRegCured	Number of times a vector reg had poison cleared
vecPredRegPoisoned	Number of times a predicate reg was poisoned
vecPredRegCured	Number of times a predicate reg had poison cleared
ccRegPoisoned	Number of times a CC reg was poisoned
ccRegCured	Number of times a CC reg had poison cleared
miscRegPoisoned	Number of times a miscellaneous reg was poisoned
miscRegCured	Number of times a miscellaneous reg had poison cleared

Fetch statistics

icacheStallRealCycles	Total number of normal mode cycles in which fetch stalled due to an I-cache miss
tlbRealCycles	Total number of normal mode cycles in which fetch was waiting for address translation
pendingTrapRealStallCycles	Total number of normal mode cycles in which fetch stalled due to waiting for a pending trap
runaheadInsts	Total number of instructions fetched in runahead
discardedRunaheadInsts	Total number of instructions that were discarded in runahead because they did not belong to the active runahead chain
runaheadInstsToDecode	Total number of instructions that were sent to decode in runahead
runaheadChainLoops	Total number of times fetch reset to the start of the active runahead chain

Decode statistics

realBranchMispred	Total number of times decode detected a branch misprediction in normal mode

IEW statistics

divergentFaults	Total number of times a memory uOp caused a fault after a divergence point in runahead mode
numPoisonedInsts	Total number of poisoned instructions skipped by IEW
numNonSpecRunaheadInsts	Total number of non-speculative instructions encountered in runahead mode
numPoisonedBranches	Total number of poisoned branches skipped by IEW

Commit statistics

realBranchMispredicts	Total number of times a branch misprediction occurred in normal mode
runaheadBranchMispredicts	Total number of times a branch misprediction occurred in runahead mode
loadsAtROBHead	Total number of cycles with loads at the head of the ROB
lllAtROBHead	Total number of cycles with a LLL at the head of the ROB
normalLLLAtROBHead	Total number of normal cycles with a LLL at the head of the ROB
instsPseudoretired	Total number of pseudoretired instructions
loadsPseudoretired	Total number of loads pseudoretired
validLoadsPseudoretired	Total number of non-poisoned loads pseudoretired
commitPoisonedInsts	Total number of pseudoretired instructions that were poisoned
runaheadEnterOverhead	Histogram of cycles spent from runahead entry until the RCL is pseudoretired, per period
runaheadExitOverhead	Histogram of cycles spent from runahead exit until the youngest instruction in the IQ at the start of runahead re-enters the IQ, per period
totalRunaheadEnterOverhead	Total number of cycles spent entering runahead
totalRunaheadExitOverhead	Total number of cycles spent exiting runahead
totalRunaheadOverhead	Total number of cycles spent entering and exiting runahead
runaheadDelayedCycles	Total number of runahead cycles in which it was safe to exit runahead (delayed runahead)
runaheadDelayedInsts	Total number of instructions pseudoretired in delayed runahead
runaheadDelayedLoads	Total number of loads pseudoretired in delayed runahead
fullROBLoads	Total number of times a load caused a full ROB stall
runaheadExitCause	Number of times runahead exited, by exit cause