Marta Walentynowicz

Running ahead of memory latency — processor runahead

Computer Science Tripos — Part II

Queens' College

2022

UNIVERSITY OF CAMBRIDGE

Declaration

I, Marta Walentynowicz of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed

Marta Walentynowicz

Date

May 12, 2022

Proforma

Candidate number:	**2323G**
Project Title:	**Running ahead of memory latency - processor runahead.**
Examination:	**Computer Science Tripos — Part II, 2022**
Word Count:	**11737**[1]
Code line count:	**3,551,274**[2]
Project Originator:	Dr Timothy Jones
Project Supervisors:	Dr Timothy Jones

Original Aims of the Project

The aims of this project were to implement two methods of speculative execution and evaluate their performance on a computer system simulator tool — gem5. The first method, runahead, enhances an out-of-order CPU by utilizing unused computation resources when the processor becomes stalled due to a long-latency operation. The second, Precise Runahead Execution (PRE), aims to improve the runahead method by reducing its overheads and exploiting further optimisations. The implementation goals also included setting up a baseline CPU design, configuring system parameters and choosing appropriate benchmarks to evaluate the two techniques.

Work Completed

The project met all its success criteria. I have implemented fully functional runahead and PRE methods, evaluated these together comparing their performance and energy consumption across a number of metrics. I have explored which benchmark suites lend themselves well to runahead optimisations and how system parameters affect the performance gains. Last but not least, a significant part of my work consisted of scrutinizing the simulator tool, understanding its inner mechanisms and analysing implementation choices that are feasible within the simulator's constraints.

[1] Counted with Overleaf's built-in feature using TeXCount.

[2] In the entire repository. Counted with `git ls-files | xargs wc -l`. Gem5 subfolder contains 2,423,436 out of these lines (for comparison the original gem5 source code contained 2,395,385 lines), McPAT source code contains 419,922 lines, the benchmarks contain 674,666 lines.

Special Difficulties

I have faced quite a few health challenges throughout the project's duration. These included getting infected with Covid and experiencing its symptoms. Overall this has contributed to delays in my project timeline leaving me with too little time to begin work on my most ambitious extension – Vector Runahead.

Acknowledgements

I would like to express my sincere gratitude to:

- Dr Timothy Jones - for proposing a project personalised to my interests, for agreeing to supervise me and, most importantly, for sharing his passion, insightful knowledge and always offering his advice.
- My family, who supported me throughout the years at university, and without whom I would not be able to achieve any of this.
- My Directors of Studies, who have been reassuring me throughout the degree, providing guidance and fostering an interest in computer science in me.
- All my friends, who were always eager to listen and provide help during these intense university years.

Contents

Chapter 1

Introduction

It has only been 40 years since the home computer revolution of the 1980s, and yet these machines have changed our lives irreversibly. One contributing factor was the astounding improvements in performance that the field of computer architecture has witnessed.

The first home computers used processors with such low clock frequencies that accessing memory was considered rapid. One of the most common microprocessors in that era was the MOS Technology 6502. Its CPU was clocked at 2MHz, while the RAM was twice as fast. This allowed the crafty engineers to implement bus sharing — a technique to essentially hide the memory latency by interleaving it with computation [1].

Fast-forward 40 years and the world has turned upside down. Nowadays, a single access to memory takes hundreds of CPU clock cycles. As a consequence, modern architectures do their utmost to prefetch data, i.e. to fetch it from slower storage to a fast-access local memory before it is needed for computation.

In my project I explore methods of speculative data prefetching. These allow possible enhancements to the performance of modern processors by increasing the overall throughput of the system.

1.1 Processor-memory performance gap

Ever since Gordon Moore made his famous prediction in 1975[1] we have witnessed a rapid growth in the transistor budget and the performance of CPUs. Processor-design techniques went all the way from exploiting bit-level parallelism, through instruction- and thread-level parallelism, to highly specialised accelerators. As a result, CPU performance kept improving at a stunning rate of 52% per year [2]. This growth continued until the 2000s when the 'free lunch' for computer architects slowed down significantly. On the other hand, memory access time was improving by around 7% each year. The difference between these improvement rates is visible in figure 1.1.

Today, the CPU-memory gap is so big that accesses to memory take hundreds of CPU clock cycles (as opposed to half a cycle back in the 1980s). The von Neumann bottleneck, presented in figure 1.2, has never been more of an issue.

[1]The initial observation was made in 1965 and predicted doubling the number of transistors every year. The refined version anticipates the transistor budget to double every two years: `https://en.wikipedia.org/wiki/Moore%27s_law`.

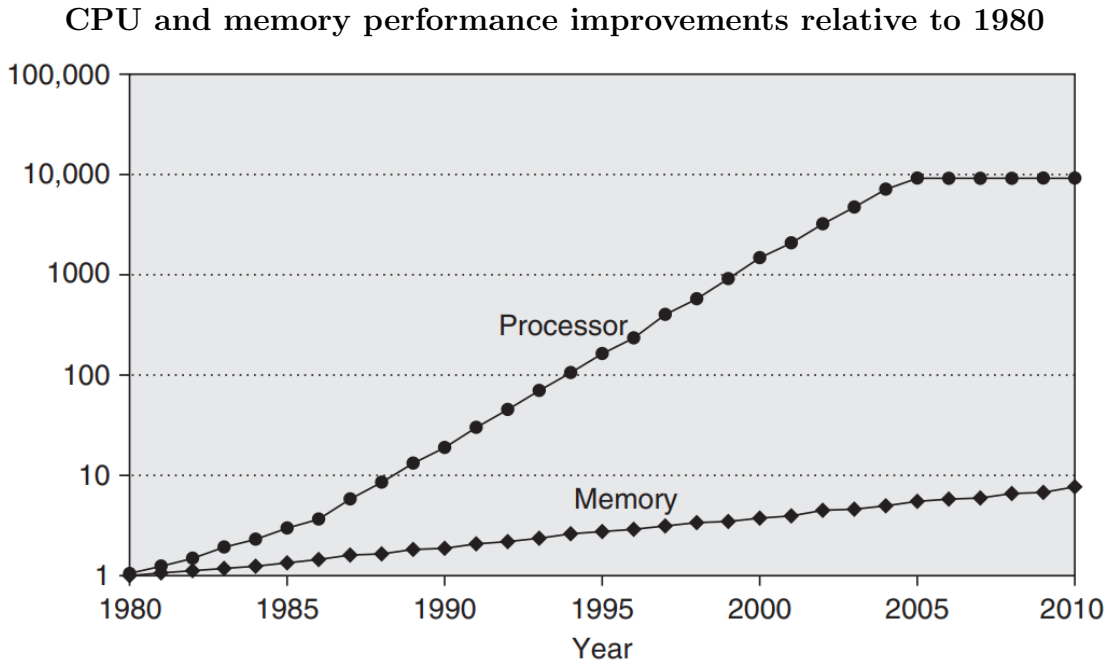


Figure (1.1) The processor line shows advancements in the number of memory requests made per second on a single-core CPU. This is contrasted with the memory line — the improvements in the number of DRAM accesses per second. The y-axis is in a logarithmic scale.
It is worth noting that this comparison is relative to the state in 1980. If we were to measure the performance of the CPU relative to the performance of the main memory, then the memory line on the plot would have to start above the processor line in the early 1980s. Picture credit: J. Hennessy, D. Patterson [3].

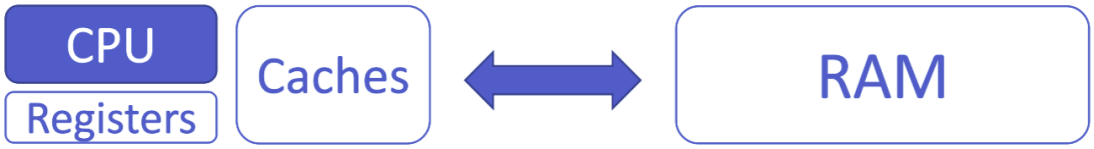


Figure (1.2) The Von Neumann bottleneck — an inevitable latency cost arising from the computer architecture model.

In an ideal world, to keep up with the speed of computation, modern processors would require bus bandwidth of up to thousands of GB/sec. Unfortunately, the physical bandwidth to memory constitutes only about 5% of that.[2] Numerous methods have been developed to mitigate the costs of main memory access. Notable improvements to the DRAM technology include: fabricating faster hardware (e.g. SDRAM), parallelizing accesses to blocks of memory or passing the critical word first. Moreover, the growing transistor budget allowed implementation of better hardware prefetching mechanisms, bigger cache hierarchies and introduced out-of-order (OoO) execution.

My work will be concerned with the last of these mechanisms, present in most modern chips. OoO processors use a re-order buffer (ROB). Instructions enter the ROB, specula-

[2]Intel Core 7, 8th generation, has 12 (virtual) cores, each with a base frequency of 3.2 GHz. Assuming 2 data memory references per second, this multiples to a data bandwidth of 819.2 GB/sec and instruction bandwidth can even double this requirement! The maximum memory bandwidth of Intel Core i7 is only 41.6 GB/sec [4].

tively execute out-of-order, and they retire from the ROB in-order to preserve the program's correctness. In case of a cache miss, all non-stalled instructions in the ROB can execute while the CPU is waiting for data to come back from main memory. Thus, the size of the ROB determines how much latency the processor can tolerate before experiencing a performance decrease. This has led to designing ever-growing ROBs, significantly increasing power consumption and complexity. The research community has shown interest in exploring alternative approaches.

1.2 Runahead techniques

Runahead is a speculative technique aiming to increase the tolerance of memory latency in an out-of-order CPU. This technique was introduced by Mutlu et al. in 2003 [5]. When a long-latency instruction is encountered (e.g. last-level cache miss), the CPU will enter runahead mode, speculatively execute the following instructions and then, when the long-latency instruction has returned its result, discard all results of the computation, reverting to the normal mode of execution. The data and instructions that were fetched into the caches during runahead constitute very accurate prefetches of the future work of the processor. Even though the results are discarded, the CPU is now able to use the prefetched data (which resides in the caches). This helps to avoid making subsequent costly requests to the memory.

A further improvement to this technique has been proposed — Precise Runahead Execution (PRE). It builds on top of runahead and benefits from speculatively executing only those instructions that lead to loads from memory in the future instruction stream. Both techniques will be discussed comprehensively in the preparation chapter.

1.3 Objectives

In my project, I aim to meet the following criteria:

- Implement the *runahead* and *precise runahead execution* methods, building on top of an out-of-order CPU model using the gem5 simulator. This will, as far as I am aware, provide the first open-source implementation of these techniques.
- Evaluate the implementations together using 8 benchmark suites. Compare the results to a baseline out-of-order CPU, being particularly attentive to the number of clock cycles executed and L2 misses performed.
- Measure energy consumption of the new design by integrating the gem5 simulation with McPAT — a power, area, and timing modelling framework [6].
- Investigate the potential impact on the performance of some microarchitectural features such as cache and re-order buffer sizes.

Chapter 2

Preparation

Before diving deeper into possible improvements to OoO execution, I provide an introduction to the CPU simulator tool I used (§2.1) and revisit the concept of instruction pipelining (§2.2). After describing limitations of out-of-order execution (§2.3) I present an exhaustive explanation of the runahead techniques that I implement in this dissertation (§2.4) and (§2.5). Finally, I describe the requirements analysis together with the engineering practices employed in §2.6 and §2.7.

Throughout the following chapters, I am using the terms "cache miss" and "main memory access" interchangeably. By this, unless specified otherwise, I mean a miss happening in *the last-level cache (LLC)*, which results in fetching the data from main memory.

2.1 Gem5 Simulator

I implemented both runahead techniques in a computer system simulator — gem5 [7] — which is a platform widely used in computer architecture research. It is written mainly in C++ and Python, and provides two modes of simulation:

- Full system mode (FS) — simulating the whole operating system and devices.
- Syscall emulation mode (SE) — simulates only programs in user space, while the system's functionalities are provided by the simulator.

Runahead does not modify the operating system's behaviour, but only modifies the underlying architectural design. Therefore, for the purpose of my project, SE mode was the appropriate one to use.

My project extends the O3CPU model — one of the 4 CPU models provided by gem5. This is an out-of-order processor, the design of which was inspired by the Alpha 21264 [8]. The detailed description of its code structure will follow in §3.3.1.

Gem5 is a huge software tool. The version I worked on — 21.1.0.2 [9] — initially contained over **2 million** lines of code! This meant that, before the project could come up to speed, I had to spend many hours trying to understand the underlying implementation and some undocumented behaviours. Throughout the project I have gained a profound understanding of the O3CPU model, however, I could never say that I understand gem5 fully.

Figure (2.1) Pipeline stages of an out-of-order CPU that is used throughout the project. The pipeline's design is based on how the O3CPU model is implemented in the gem5 simulator.

2.2 Instruction pipeline

To describe how a microprocessor executes instructions out-of-order, I present an overview of a simple out-of-order pipeline. Figure 2.1 depicts a pipeline that is intentionally not representative of a real processor pipeline, but it is based on how gem5 implements it. This description will lay the grounds for the discussion of my implementation in later chapters. The five stages are:

- **Fetch** — the address (or PC) of a program instruction is sent to the instruction cache and the corresponding instruction is returned. Fetch increments the PC to the next sequential address as well as supporting branch prediction.
- **Decode** — in gem5 this stage does not do much apart from handling PC-relative unconditional branches. In a real processor instructions would be decoded into micro-operations, i.e. operations targeted for the given microarchitecture.
- **Rename** — this stage performs register renaming, i.e. assigns a physical register to each virtual register used by the instruction (the destination register and all source registers). When there are no free physical registers that could be used, then the pipeline has to be stalled until sufficient resources are available.
- **Issue/Execute/Writeback (IEW)** — a combination of 3 stages. First, instructions are dispatched into an instruction queue (IQ), from which they are issued for execution. Later, the execution and write back are handled simultaneously. Instructions leave the IQ once they are executed.
- **Commit** — this stage maintains a FIFO queue of instructions that can be executed outside the program order, called a re-order buffer (ROB). Each clock cycle the oldest instructions from the ROB are removed if they are executed. Commit also handles any faults that the instructions might have caused. If the instruction was mispredicted then a signal is sent to all previous stages of the pipeline, effectively redirecting the path of execution.

2.3 Out-of-order CPUs

In-order processors can only execute instructions sequentially. An out-of-order processor improves over that concept by keeping a window of instructions that can be executed outside the program order — a re-order buffer (ROB). Instructions enter the ROB in program order and execute out-of-order. This means that a younger instruction (i.e. one that entered the pipeline more recently) can perform its operation before an older one.

(a) The re-order buffer is a FIFO structure. Instructions I_0 to I_4 enter the ROB in program order, later instructions I_1 and I_2 become executed at some point.

(b) Instruction I_0 is at the head and it was executed. Hence, the first 3 instructions at the head of the ROB can now be **committed**.

(c) More instructions enter the ROB. If the head instruction involves a long latency operation, then no instruction in the ROB can be committed until the head instruction executes. However, the CPU can still execute *out-of-order* instructions that reside in the ROB and are ready to execute (i.e. have all the required resources to carry out their computation).

(d) A **full instruction window stall** — all instruction slots are occupied, no more instructions can be dispatched into the ROB.

(e) All instructions in the ROB are either blocked or executed. No useful work can be done until the head instruction is committed.

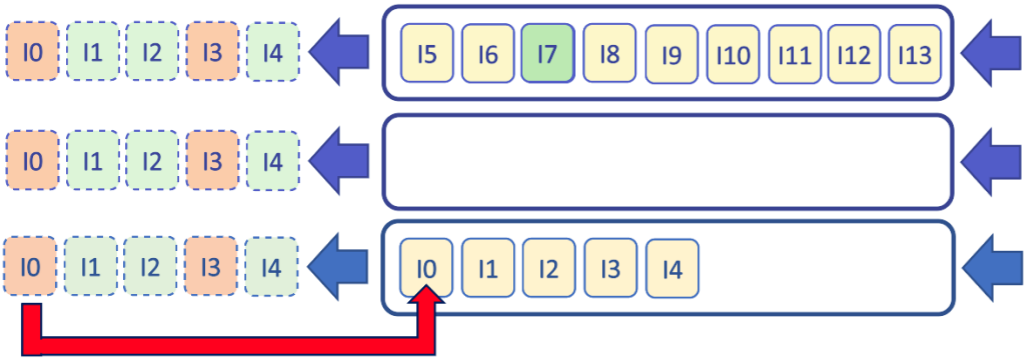
Figure (2.2) A visualisation of instructions' flow through the re-order buffer in an out-of-order CPU. Points (c)–(e) highlight how the size of the ROB limits the performance of the CPU.

When an executed instruction reaches the head of the ROB (i.e. becomes the oldest instruction), it is ready to be *committed*: it updates the architectural state (i.e. writes back its results and frees the occupied resources) and is *retired* out of the ROB. The retirement must happen in the order specified by the program, providing an illusion of sequential code execution to the programmer.

In the event of a cache miss, instructions in the ROB can continue to execute while the CPU is waiting for data to come back from main memory, provided they are not dependent on the instruction that caused the caches miss. During this time, the cache-missing instruction will move towards the head of the ROB, and will likely end up at the head, blocking commit, as depicted in figure 2.2. Therefore, the size of the ROB effectively determines how much latency the processor can tolerate before experiencing a performance decrease. This has led to designing ever-growing instruction windows, hence significantly increasing power consumption and complexity.

(a) Instructions enter the ROB and execute out-of-order. A runahead CPU detects if the head instruction has missed in the last-level cache, potentially causing a stall of the whole ROB. The CPU's state is checkpointed (along with the PC address of the triggering instruction) and it enters runahead mode.

(b) The stalling instruction I_0 is pseudo-retired from the ROB. I_1 is now head of the ROB, but it has already been executed so it can pseudo-retire as well. The same happens with I_2. Any following stalling instructions (e.g. I_3) will be treated just like the one which triggered runahead.



(c) Instructions continue to pseudo-retire until the stalling load returns (i.e. the data is retrieved from memory and the instruction that triggered runahead is ready to execute). When this happens, the CPU switches to normal mode of execution. It first needs to flush all instructions from the ROB and then begin fetching instructions again from the checkpointed address.

Figure (2.3) A visualisation of the re-order buffer in an out-of-order CPU with runahead. A standard OoO CPU would become stalled with instruction I_0, therefore not being able to fetch and execute instructions beyond I_8. The runahead method provides the opportunity to speculate further in the instruction stream and fetch data used by future instructions into the caches.

2.4 Runahead

Runhead execution was proposed as an alternative to unreasonably large instruction windows [5]. As discussed in the previous section, the ROB can become blocked as a result of a *long-latency (stalling)* instruction reaching the head of the ROB. Scrutinization of this concept is left until §3.2.1. In a runahead CPU, the stalling instruction is tossed out of the instruction window, effectively unblocking it. The registers' state needs to be checkpointed, and the CPU enters runahead mode. In runahead the instructions are executed as normal, but their results are not committed. Instead, they *pseudo-retire* from the ROB, which means that they do not update the architectural state. Once the stalling operation is completed, all results from runahead mode are thrown away, checkpointed values are restored and the CPU resumes normal execution. Figure 2.3 presents how the runahead CPU benefits from continuing to fetch instructions while the standard OoO CPU's execution would be blocked. The premise is that during runahead mode instructions will load data into the caches far in advance before it is needed for execution.

2.4.1 Dependent instructions

Instructions in the pipeline are often dependent on the computation result of one of the preceding instructions. This causes problems in runahead mode since the pseudo-retired instructions do not always produce valid results, thus instructions dependent on the bogus result will not execute properly. One possible solution is to introduce invalid bits for each register, in order to communicate to successive instructions that the value stored in this register is not correct.

Let us look at a very short sequence of instructions, where the second instruction is a load, and it causes a transition into runahead mode. Instructions are presented using a standard 3-address code: $< operation > < dest\_reg > < src\_reg1 > < src\_reg2 >$.

```
A:      MOV r1, $1
B:      LD  r2, 0($4)        // stalling load instruction
C:      ADD r3, r1, r2
D:      ADD r4, r1, r3
```

The following will happen once the CPU enters runahead mode:

1. The load instruction (B) will be pseudo-retired and no meaningful result will be written into its destination register *r2*. The invalid bit for *r2* must be set.

2. Instruction C cannot be executed since it sources an invalid register *r2*. Hence, its destination register *r3* becomes invalid as well and C can be pseudo-retired.

3. Instruction D reaches the head of the ROB, it depends on the value stored in register *r3*, which is invalid and so, the destination register of D, i.e. *r4*, is also invalidated which allows the instruction to pseudo-retire.

In the example above, instructions I3 and I4 are deemed invalid as they depend on yet unavailable data. The mechanism described allows detecting invalid instructions so that they can quickly leave the ROB instead of causing further stalls.

2.5 Precise Runahead Execution (PRE)

PRE is an improvement to runahead which was suggested in 2020 by Naithani et al. [10], addressing the following limitations:

- **Limited prefetch coverage** (i.e. the number of useful instructions fetched in runahead). Any instruction that does not lead to a long latency load can be considered a waste of execution cycles and CPU resources. If, for instance, the majority of CPU instructions perform a complex arithmetic computation and only a few instructions load data from memory, then in runahead we should only be concerned with these few load instructions. The computation results will be thrown away after exiting runahead anyway. Hence, it would be most efficient to only speculatively execute the instructions that cause data to be moved into the caches. However, the original runahead proposal executes all encountered instructions, leaving lots of space for improvement.

- **Significant overhead of switching between normal and runahead modes**. In runahead instructions pseudo-retire from the ROB. Hence, when normal mode is resumed, the processor needs to flush and refill the pipeline, re-fetch and re-process all instructions that were in the ROB before entering runahead. The incurred cost may be as high as 56 cycles[1]! This means that any runahead interval shorter than 56 cycles is simply not worth the hassle.

PRE manages to minimise the switching overhead upon exit from runahead mode by clever use of available CPU resources (§2.5.1). Moreover, it improves prefetch coverage by using a Stalling Slice Table (§2.5.2) and by introducing a novel method of register reclamation (§2.5.3).

2.5.1 Execution in PRE mode

Being an idea akin to runahead, PRE differs significantly in implementation from its predecessor. The first distinctive factor is the moment when the CPU enters runahead mode. To avoid confusion in the terminology, I will use "runahead mode" only when describing the first technique, while the term "PRE mode" will be used to indicate the "execution in runahead mode in a CPU implementing PRE". That said, PRE mode is entered when a stalling instruction reaches the head of a **full** ROB (i.e. upon a *full-window stall*). On the contrary, runahead would be triggered whenever a stalling instruction is at the head of the ROB, regardless of whether the ROB is full or not.

The second key modification concerns instruction flow through the pipeline. In PRE mode no instruction is retired from the ROB, nor will any instruction be dispatched (inserted) into the ROB. In other words, the ROB is effectively frozen and the stalling instruction remains at its head. However, PRE still speculatively executes future instructions in the previous stages of the pipeline i.e. fetch, decode, rename and IEW stages. The speculative micro-operations never reach the commit stage.

This is a crucial change because non-speculative instructions residing in the ROB are immediately ready to be committed when the CPU exits PRE (i.e. when the data for the head instruction becomes available). On the other hand, runahead requires all speculative instructions to be drained out of the ROB and re-fetched. This enhancement significantly reduces the overhead arising from entering runahead mode, hence making it profitable to enter PRE even for shorter time intervals.

Thirdly, PRE makes an observation that there are sufficiently many CPU resources available to continue execution when runahead is entered. This means that the processor can continue to use the unoccupied registers, functional units and instruction queue entries as normal during PRE mode. Crucially, this allows the non-speculative instructions in the ROB to continue occupying their resources throughout the whole runahead interval. This way there is no need for checkpointing of the architectural state as the instructions in the ROB never release their resources.

[1] According to the PRE paper: for a 192-entry ROB the performance penalty is 56 clock cycles, assuming we ignore any overhead from saving and restoring the architectural register file: 8 cycles for refilling the pipeline, 48 cycles for refilling the ROB and re-dispatching 192 instructions (with dispatch width of 4 instructions).

2.5.2 Stalling Slice Table (SST)

PRE achieves an increase in prefetch coverage by executing only the chains of instructions that lead to a stalling load. A fully associative cache — Stalling Slice Table (SST) — is introduced to store PC addresses of instructions that will be executed. The premise here is that most performance improvement comes from prefetching data for instructions that occur inside loops. Their PC will be fetched on every loop iteration and they will be allowed to execute in PRE. The idea of the SST is similar to a breadth-first search algorithm:

1. Upon entry to PRE the stalling instruction's PC is added to the SST.
2. When the same PC is decoded again, it hits in the SST. The decode stage looks up the producers of this instruction's source registers. Their PCs are added to the SST.
3. Instructions that hit in the SST are allowed to continue to the rename stage of the pipeline. Other instructions are discarded at the decode stage.
4. Steps 2–3 are repeated until the CPU exits PRE.

Here, a 'register's producer' is an instruction that was the most recent one to write into this register. To give an illustration of what I mean, let's consider a simple example where instructions A, B, C are executed inside a loop:

```
Format: [operation] [destination register] [source register]
A:      OP1 r2, r1
B:      OP2 r3, r2
C:      OP3 r4, r3
```

- Instruction C triggers transition into PRE mode and is added to the SST.
- On the next loop iteration instruction C will be decoded again and this time its address hits in the SST. Its source register $r3$ was last produced by instruction B, therefore, B's PC is added into SST.
- The next time instruction B is decoded its address will hit in the SST. Its source register was produced by instruction A, so A's PC is added into the SST.
- When instruction A is decoded its address hits in the SST and the producer of register $r1$ will be added to the SST.

It is clear from this example that the algorithm results in tracking all instructions in a **backward slice** of any long latency operation. PRE benefits from only processing instructions that result in data being brought into the caches, as the other ones are discarded quickly. This allows CPU cycles to be used more efficiently and the method is able to speculate deeper in the stream of instructions.

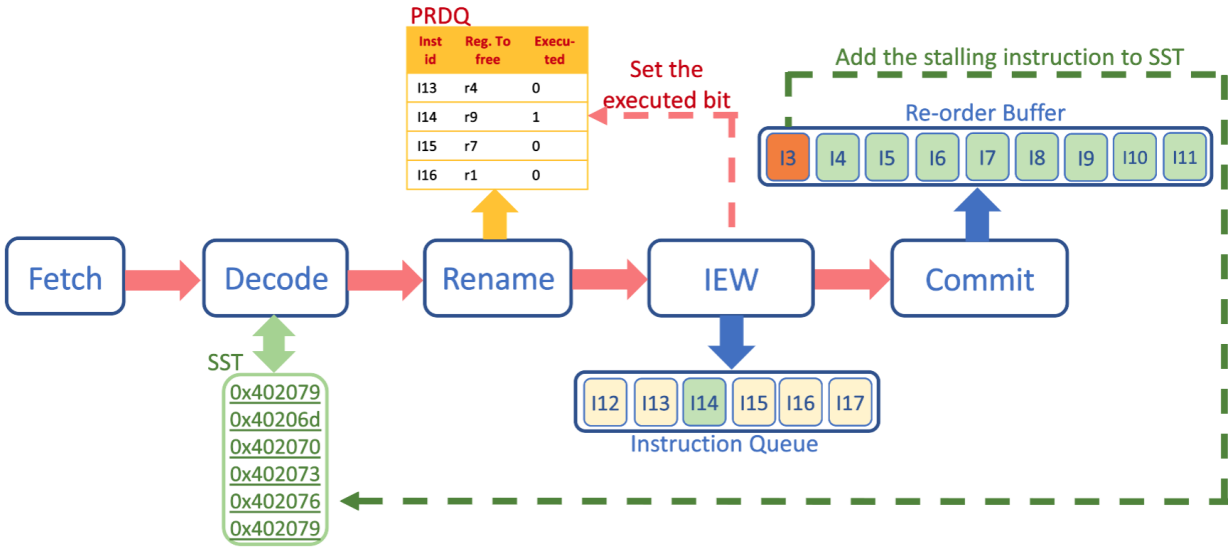


Figure (2.4) Illustration of execution in PRE mode. The PC of the stalling instruction I_3 is stored in the SST, which is accessed by the decode stage. The rename stage creates entries in the PRDQ as well as attempts to retire the oldest entries if the corresponding instruction was executed. The IEW stage, on the other hand, sets the executed bits in the PRDQ entries.

2.5.3 Runahead Register Reclamation (RRR)

Lastly, PRE introduces a new method of register reclamation. In a conventional OoO CPU registers are freed when the last consumer of a register is committed. However, since instructions in PRE mode never reach the commit stage, their registers must be handled differently. An additional hardware structure — the precise register deallocation queue (PRDQ) — is used to track the registers that are used by instructions in PRE mode. It operates as follows:

- When a runahead instruction reaches the rename stage it creates its entry in the PRDQ table. The entry contains an "Inst. ID" field.
- A free physical register is chosen and renamed to the destination architectural (virtual) register of this instruction. At the time of the renaming, this architectural register is mapped to a different physical register, this mapping will be overwritten and the old physical register will no longer be used. Hence, the PRDQ entry includes a pointer to this previous physical register in a field called "Register to free".
- The "Executed" field is initially false and will be set to true once the instruction is executed in the IEW stage.
- The head entry in PRDQ can be deallocated only when the executed bit is set. This is when the old physical register becomes freed.

Figure 2.4 illustrates how the pipeline stages interact with the PRDQ and the SST.

2.6 Requirements Analysis

The functional requirements of my project were already listed in §1.3. These correlate to the core success criteria described in my proposal (Appendix A), namely:

1. Setting up the simulation environment with a baseline configuration.
2. Implementing runahead.
3. Implementing precise runahead execution.
4. Evaluating the baseline, runahead and PRE together.

All the above-mentioned criteria will be used for the evaluation of my work along with the non-functional requirements, which include but are not limited to:

- **Effectiveness** — runahead methods should contribute towards a performance improvement over the baseline for a set of memory-intensive benchmarks. This will be primarily measured using the *cycles per instruction (CPI)* count. Moreover, gem5 allows adding customised statistics to the simulation, which I will exploit at various stages of my project, including the evaluation and testing of my design.
- **Efficiency** — the design introduces new hardware components that should introduce a negligible overhead in the performance of the system. To evaluate this requirement, I integrate my simulation with a power modelling tool — McPAT (described in §4.4).
- **Readability and modifiability** — from the outset my simulation was intended to be open-sourced, hence I invested significant time in documenting the runahead functionalities. I made every effort to structure my code in such a way that it is straightforward to extend (as far as the simulator structure permitted).
- **Volume testing** — a number of bugs can exhibit in very rarely seen configurations. In order to safeguard against these errors, it is crucial to test runahead techniques not only on numerous small benchmarks but also on a few programs of vast size.

2.6.1 Development model

The development was carried out using a mixture of agile strategies and the spiral methodology. Throughout the project, I have been scheduling my tasks into 2-week long sprints (following a SCRUM methodology). The initial plan had to be revised a few times due to some unforeseen delays (mainly sicknesses). Nevertheless, this planning strategy helped me to keep progressing even in the face of unexpected events.

Furthermore, the first 3 key components (as listed in §2.6) follow similar development cycles. Moreover, at each of the iterations, my project required a thorough investigation of the existing simulator source code. This aligns well with the spiral model's focus on risk assessment and evaluating alternative solutions. My risk corresponds to attempting to implement a solution that is unfeasible under the constraints imposed by gem5. This is why I decided to obey a spiral development methodology [11]. Following this work cycle ensured that my design choices were always thought-through before proceeding to the implementation stage. Figure 2.5 summarises how I divided each work stage into subtasks, this made keeping track of the progress undeniably easier.

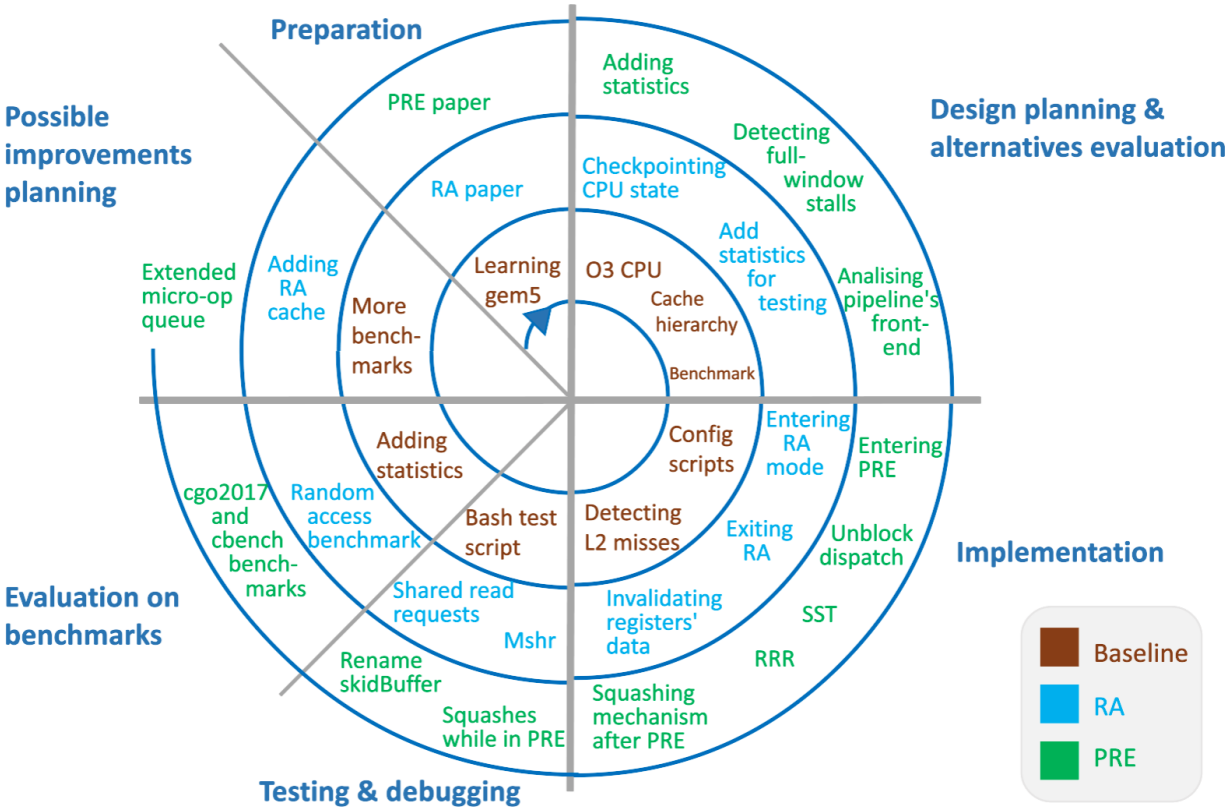


Figure (2.5) Diagram of my spiral development methodology. Each of my work iterations focused on one of the key components: the baseline and project setup, runahead, and PRE.

2.6.2 Testing strategy

Looking ahead to the project evaluation, I decided to leverage gem5's statistics generating system. My implementation was structured in a very iterative manner. This way, having added a new feature, I could easily compare the statistics between different versions and ensure the anticipated performance improvement is visible (further explained in §4.2.1).

2.6.3 Software engineering techniques

Managing code version control and keeping a backup were carried out using Git, GitHub and hard-drive storage. In order to ensure that my changes were saved automatically even before I committed to the remote repository, I also used a cloud storage service called *pCloud* [12]. For the dissertation itself, I used Overleaf — a free online LaTeX editor, which allowed seamless facilitation of the feedback from my proofreaders. Linking Overleaf with a GitHub repository provided robustness to possible failures of the system.

Throughout my project, I implemented several bash and Python scripts to automate the process of running the simulation, gathering statistics and displaying data clearly in a command-line interface. I have benefited from these solid foundations for experiments when I began parallelising my simulations.

The development process further benefited from regular meetings with my supervisor. I logged my notes from these meetings and recorded any arising problems in an old-fashioned paper notebook. It helped me to trace my daily progress and quickly get on track after a break from work. In total, I gathered 56 pages (size A4) of handwritten notes.

2.6.4 High-performance computing resources

As initially predicted (and stated in the proposal — Appendix A), my personal laptop was unable to carry out the simulation in a reasonable time-bound. A month into the project it began imposing significant slow-downs on the work schedule — a single compilation of gem5 could take up to 30 minutes and the execution of a benchmark program could easily double that time. Therefore, I moved my working environment to Sofia — one of the high-performance machines in the Computer Lab. With the plenitude of 104 CPUs to use, my development could continue much smoother.

One downside of working on a remote machine was that I experienced many issues with the SSH plugin for *Visual Studio Code* [13]. As a result, I switched to using *SSHFS* [14] — a tool that allows mounting a remote filesystem over SSH). Although there is a notable delay (around 10–30 seconds) before SSHFS manages to copy the local changes to the remote machine, it still proved to be a more reliable tool to work with.

2.6.5 Licensing

Gem5 is released under a Berkeley-style open source license (BSD) [15]. This is a permissive license allowing redistribution of the source code under the condition of including the copyright notice. In order to maintain compatibility with it, I decided to distribute my code under the *BSD 2-Clause "Simplified" License* [16]. This is a short, permissive license that allows distribution, modification, private and commercial use under the above-mentioned condition. My code is available as open-source software on GitHub.

2.7 Starting point

I have no previous experience with hardware architectures outside the IB Computer Design course. Many of the concepts I worked on were also covered in the Advanced Computer Architecture course in Part II, but nevertheless, that is a knowledge I had to learn myself prior to attending this course.

Throughout the dissertation, I will be explaining numerous implementation details of the gem5 simulator. This is the knowledge that I had to acquire during the past 6 months. Although gem5 is a fairly documented tool itself [17], there is very little explanation of the O3CPU's implementation [18] on which I based my work. Thus, I had to seek information around the topic in various sources, often inferring the underlying system behaviour from the existing code, or debug traces of the simulation. This means that my personal code contribution is not extensive in the number of code lines added, but it did require a significant cognitive effort to integrate inconspicuous changes into a gigantic software tool with densely coupled components.

Chapter 3

Implementation

This chapter explains the work carried out during the project. I begin with a description of the baseline processor, configurations of gem5 and repository overview (§3.1). I proceed with describing my implementation of runahead and PRE methods in §3.2 and §3.3. This structure closely follows my work timeline, as each of these sections corresponds to one iteration of the spiral development model described in §2.6.1.

3.1 Project setup

My experiments are based on the out-of-order (OoO) processor model — O3CPU — provided by gem5. This CPU constitutes a baseline to compare with runahead methods. Before the implementation of runahead could begin, I configured the simulation that is used throughout the project (§3.1.1), considered tradeoffs relating to the structure of my code repository (§3.1.2) and implemented a mechanism that detects data misses in the L2 cache (§3.1.3). Scripts to automate testing, which aided my development, are presented towards the end of this chapter (§3.1.4).

3.1.1 Configuration of architectural features

Gem5 provides a simple way to manipulate architectural parameters using Python scripts. Amendable features include: the size of the ROB, cache hierarchy structure, cache sizes, the delays resulting from accessing the caches, delays in communication between pipeline stages, the width of the pipeline (i.e. how many instructions can execute in each stage during one clock cycle) and many more. In total there are around 60 parameters to configure, and I have added more for my runahead implementations (§4.2.1).

The configuration script I used throughout the project sets up the cache hierarchy to include L1 instruction and data caches and a single L2 cache (figure 3.1).

3.1.2 Repository overview

In table 3.1 I present a high-level view of my repository with a description of the most essential subfolders. Gem5 is structured in a very modular way. The source code for different subsystems (e.g. memory, kernel, CPU) is spilt into directories. Subfolders that contain different CPU models are also implemented in an object-oriented way. The CPU class

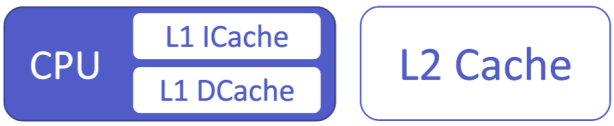


Figure (3.1) Hierarchy of caches used in my project.

Top	Subfolders		Description
gem5/	src/cpu/	o3/	The baseline CPU implementation.
		ra/	The runahead CPU that extends O3CPU.
		pre/	The PRE CPU that extends O3CPU.
	src/mem/		The memory system, contains caches and MSHRs.
	src/kern/		Implementation of system-call interfaces.
	configs/		Python scripts configuring architecture features.
	m5out/		Simulation statistics (used by McPAT).
	stats/		Scripts for parsing and displaying chosen statistics.
	build/		The build directory contains gem5's binary files.
bench-marks/	cbench/		Banchmarks based on standard type files e.g. txt, png, mp3.
	cgo2017/		Memory-latency bound benchmarks.
energy/	mcpat/		Source code of the McPAT power consumption tool [6].
	Gem5McPat		Script converting gem5 statistics to a format used by McPAT.
	parse.py		Script that runs parsing and McPAT, saves the output.
	mcpat_in/		Output of Gem5McPat and input to McPAT.
	mcpat_out/		Energy output from McPAT.
plots/			Parsing and plotting statistics and McPAT output.

Table (3.1) Table presenting an overview of the project repository.

has member variables representing pipeline stages, these are instances of classes: Fetch, Decode, Rename, IEW (Issue, Execute, Writeback), Commit. Moreover, processor models are placed into separate folders which allows using their implementations interchangeably. For instance, the configuration script can specify whether an in-order or out-of-order CPU is used.

Creating the repository structure was the first design decision I made. The most OOP-principled approach would be to create RunaheadCPU and PreCPU classes and have them inherit from the O3CPU class, overriding only the functions that needed modification. Unfortunately, a CPU is a densely-coupled mechanism with occasionally cumbersome interactions in gem5's implementation. I realised that isolating the right parts would be considerably more difficult than copying the entire O3CPU folder and implementing the modifications in-place. As a newcomer to the huge tool, I also found that onboarding onto the simulator was quicker when I could see the code directly without having to jump around in the codebase too much.

3.1.3 Detecting misses in the L2 cache

From the outset, I wished to know what the room for improvement in the baseline design was. This can be measured by counting misses in the last-level cache — the L2 cache. The bigger this count is, the better performance we can expect from runahead. There is a subtle challenge in evaluating this statistic. While detecting cache misses is simple, some misses may affect performance more than others. For example, instructions that were squashed or executed should not be taken into account at all because they already left the pipeline.

Moreover, detecting misses in L2 is crucial for entering runahead mode. However, in gem5 the cache miss event is not linked back to the instruction requesting the data. This knowledge is essential to determine which instruction triggered runahead. Therefore, I decided to add a mechanism tracking this by adding pointers to the instruction in the *Request* objects as well as augmenting the instructions with a boolean flag *missedInL2*.
Let's begin the description of my implementation by describing a few classes used in gem5:

- **BaseCache** — The base class for all cache objects; L1 and L2 caches inherit from it.
- **DynInst** — An instruction object; a *Request* to access memory is generated by it.
- **Request** — A *Request* object represents a data transfer between a load instruction and the memory. It is created when the instruction begins its execution. It contains the physical address in memory that is accessed, the size of the request and flags indicating its current state. The lifetime of a Request object spans all the way from the requestor, to the ultimate destination in memory (or cache) and back.
- **Packet** — A *Request* can be conveyed by several Packet objects. These encapsulate the transfer of data between two objects (e.g. between the L1 and L2 caches). Hence, its lifetime is shorter than the lifetime of a Request object.
- **MSHR** — Miss Status Holding Register is a buffer that keeps track of the outstanding cache miss. This is needed for non-blocking caches [19], which gem5 implements, i.e. caches that can serve multiple misses simultaneously. The MSHR holds a 'busy' bit, the block address and destination registers where the data should be written.

Adding pointers to the instruction object

Upon a miss in the L2 cache, pointers to the *Packet* and *Request* objects are available, but we need to access the *DynInst* object of the instruction triggering it. Unfortunately, the instruction class uses the *Request* object, hence using a simple pointer from Request to *DynInst* would result in a circular dependency. Furthermore, the instruction class cannot be forward declared as I need to access its members in the *Cache* object (to set the *missedInL2* flag). Therefore, I decided to create a parent class for the *DynInst* class, which contains some basic functionalities that are needed for runahead instructions. The Request object can reference this class, avoiding any circular dependencies.

Another intricacy is that an instruction may create multiple *Request* objects when accessing data that is split across cache lines. Thus, my implementation needs to maintain a vector of all outstanding requests to memory, rather than a single reference. Additionally, it is important to reset the *missedInL2* flag only when the last of the requests returns.

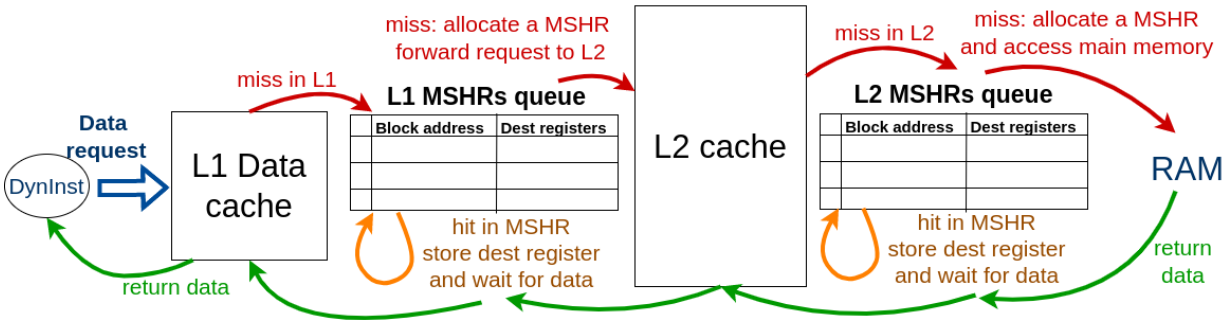


Figure (3.2) The flow of a memory request in the memory system. It displays misses and accesses to a lower level of cache i.e. closer to the main memory (red), the returned data path (green), the hits in MSHRs and waiting for a previous request to the same memory block to return (orange).

To illustrate this, let's consider an instruction initiating *Requests* A and B. A misses in L2 and sets the *missedInL2* flag in the associated instruction. When B hits in the L2 cache it will reset this flag even though the instruction still has one outstanding data *Request*. This situation would leave the instruction in the ROB without the flag set, but it is effectively stalled waiting for data and cannot retire.

Handling cache accesses

Figure 3.2 summarises how data requests are handled in the cache hierarchy:

1. The cache receives a Packet specifying the address to be accessed.
   (a) If the *Packet*'s request can be satisfied by this cache (i.e. there is a cache hit) then the data is returned.
   (b) If the *Packet* misses in the cache, then all entries in the MSHR queue are checked in parallel. If there is a hit in an MSHR, i.e. there is an outstanding cache miss for the same block, then the destination register is added onto this MSHR to avoid propagating duplicate *Request* objects;
   (c) Otherwise, the *Request* is passed on to the next level of the memory hierarchy (either to another cache or to the memory itself).
2. When the data is returned, all MSHRs are searched again in parallel to find the one that requested the data. The value is forwarded to the specified destination registers.

Request flow through the cache hierarchy

Requests can travel through L1 and L2 MSHRs. Let's assume that requests A and B try to access the same block in memory, where A is made earlier. The two possible orderings are:

- An entry for A is created in the L1 MSHR, A misses in the L2 and the associated instruction is marked as *missedInL2*. While A is being served, B hits in the MSHR for L1 and hence the instruction associated with B should also be marked as *missedInL2*.
- An entry for A is created in the L1 MSHR, B hits and is added to the same MSHR. Later A misses in the L2 and so both requests should now mark their associated instructions as *missedInL2*.

Algorithm 1 Detecting L2 cache misses

```
1: procedure recvTimingReq(PacketPtr pkt)        ▷ The cache receives a Packet
2:     if pkt hits in cache then
3:         Return the data
4:     else pkt misses in cache
5:         if pkt hits in MSHR then                ▷ Adding pkt to an existing MSHR
6:             add pkt to the MSHR
7:             add the MSHR to req→traversedMshrs
8:             if MSHR is already marked as missed in L2 then
9:                 mark the associated instruction missed in L2
10:        else pkt misses in MSHR
11:            create a new MSHR entry
12:            add the MSHR to req→traversedMshrs
13:                                              ▷ Instruction miss in L2
14:        if cache is an L2 cache and req→inst is not squashed or committed then
15:            for mshr in req→traversedMshrs do
16:                mark mshr missed in L2
17:                for inst in mshr→requestingInsts do
18:                    mark inst missed in L2
```

Figure (3.3) The algorithm presents my modifications to function *recvTimingReq*. This is a method implemented in a base class of all caches (L1 and L2 caches inherit from it). The function is called when a cache receives a request to access specified data. If the data is present in the cache then it is returned straight away. Otherwise, the Miss Status Holding Register is checked and if there is no MSHR entry for the same cache line, then the request is forwarded to the next level in the memory hierarchy.

To deal with both these scenarios I modified the Request object to hold a set of MSHRs, which were traversed by the request. Considering the above, my implementation of detecting misses in L2 caches works as depicted in figure 3.3.

3.1.4 Automating simulation runs

I implemented, and repeatedly enhanced, a script for automating compilation and parallel simulation runs (*gem5/run.sh* in the main repository). The script runs gem5 with relevant configuration and mode (baseline, runahead or PRE), defines debug flags, gathers statistics, summarises them and prints to the standard output. In the end, this turned out to be a script of almost 200 lines. I chose to implement it in bash because I am mostly using shell commands. Python would be an equally good implementation choice, but I wanted to brush up my bash programming skills as well. Automation early on in the project sped up the testing phase considerably.

3.2 Runahead

Having laid the groundwork, we can now move on to the description of my runahead implementation. Firstly, I describe implementation choices concerning when to trigger runahead (§3.2.1). Then I proceed to describe how execution in runahead mode is handled (§3.2.2) and how the CPU reverts to the normal mode of execution (§3.2.3).

3.2.1 Entering runahead mode

Runahead mode could be entered at any point of execution. Ideally, the CPU would detect a long latency instruction that causes a stall of the pipeline and begin the transition then. In reality, however, it is not so simple to estimate how long a load will take to complete.

In my implementation, runahead mode is triggered when the head instruction of the instruction window is marked as missed in L2 (as described in §3.1.3), according to the original paper [5]. This can only be detected at the commit stage of the pipeline. It then signals the event to the CPU class which will trigger runahead mode.

Obviously, this is just a heuristic, and it may not always turn out to be an optimal choice. For instance, assuming that average access to main memory takes around 100 cycles and an instruction reaches the head of the ROB when it generated its request to memory 100 cycles previously, then it is quite likely that the request will soon return. If this instruction triggers runahead, then the CPU may need to revert to normal mode soon, resulting in a very short **runahead interval**. Possible improvements to this scheme may include triggering runahead only when the stalling instruction has spent less than a specified threshold of cycles in the ROB. This would require tuning of the threshold parameter and that does not lie within the aims of my project.

Checkpointing the architectural state

As explained in the preparation chapter, runahead instructions are executed in a conventional manner. However, they *pseudo-retire*, i.e. they do not update the contents of the *register alias table (RAT)* — a structure that maintains mappings between the architectural and physical registers. I could avoid updating the RAT by either adding a second RAT (pseudo-RAT), which would only be used in runahead mode, or by checkpointing the state of the RAT upon entry to runahead and later restoring this checkpoint when leaving runahead. Using a pseudo-RAT would require adding checks on whether the CPU is in runahead mode in all places where the RAT is accessed. Hence, the second solution seemed more appealing for me to implement. Therefore, I make copies of the following objects when entering runahead mode:

- The PC of the instruction that triggered runahead.
- The register rename map at the Commit stage (the history of committed renames). Copying this object involved implementing deep copy constructors for several classes used in the rename map.

3.2.2 Execution in runahead

In runahead, all but the last pipeline stages behave exactly as they do in normal mode. The commit stage differs because it handles the pseudo-retirement of instructions, i.e. leaving the ROB without updating the architectural state of the CPU. Of course, the first instruction to pseudo-retire will be the one that triggered runahead in the first place.

Pseudo-retirement

In my implementation the head instruction can be pseudo-retired in 4 different scenarios, i.e. if it is: executed, squashed, marked as missed in L2, or marked as invalid. The first two scenarios are straightforward as they constitute a logical extension of the normal mode. The third one corresponds to an event when runahead mode would be triggered, since the CPU is in runahead already then there is no need to wait for this instruction to access its data. The last case is a bit more intricate, which arises from the dependencies between instructions.

Invalidating instructions

In order to track invalid instructions in the pipeline as described in §2.4.1, I have introduced invalid bits to the physical register and the instruction classes. I follow these principles when invalidating registers and instructions:

- All instructions that source an invalid register are marked as invalid.
- Each instruction that misses in L2 and pseudo-retires does not write valid data to its destination registers, hence it is also invalid.
- Destination registers of any invalid instruction are also invalidated.
- The invalid bit of a register is reset when it is renamed again.

3.2.3 Exiting runahead

The CPU exits runahead mode when the stalling instruction receives the requested data. To achieve this, I mark the instruction that triggered runahead mode. When the request to memory returns with its data, the Load-Store queue class receives a response *Packet*. If the instruction associated with this response is the triggering instruction then a function handling exit from runahead is called. This function needs to:

- Squash instructions that are present in the ROB. In my implementation, this is achieved by exploiting an already existing mechanism for squashing instructions after a branch misprediction. The squashing mechanism will be described in greater detail in §3.3.7. For now, it suffices to say that when an instruction is marked as squashed it will free all occupied resources and leave the pipeline without having any effects.
- Restore the checkpointed state of the CPU — copy over the checkpointed rename map and overwrite all changes made in the rename map during runahead.

3.2.4 Testing

The biggest hurdle of my runahead implementation was getting the detection of L2 misses right. This is a crucial mechanism for the stream of instructions to keep progressing while in runahead mode. Furthermore, to ensure that my mechanisms of invalidating dependent instructions is correct, I have implemented a check that no instruction gets stuck at the head of the ROB for more cycles than it would in the baseline. Using the gem5 statistics I could determine what is the maximum number of cycles one instruction can be at the head and use that count as a threshold. Whenever the simulation would report that an instruction is stuck longer at the head, I would use the debugging output to trace its exact origin. Sometimes, to understand why the instruction is broken into microoperations, I would dig deeper into the benchmark's source code and analyse its objdump [20]. This allowed me to establish that the implementation is behaving as predicted and meets my second success criteria.

3.3 Precise Runahead Execution

As already indicated in §2.5, runahead comes with its own downsides. Moreover, in §4 we will see that it does not always aid the performance of the evaluated benchmarks. Implementing Precise Runahead Execution (PRE) required significant code changes as well as a more in-depth understanding of the simulation sources. This is why I begin with a description of the O3CPU's pipeline mechanism (§3.3.1), which is crucial for understanding my further code modifications. Later, I describe the details of my implementation.

3.3.1 O3CPU code structure

I begin by describing how instructions flow through the pipeline in the O3CPU. Each of the pipeline stages is represented by a class object that is owned by the CPU class. Figure 3.4 summarises interactions between the stages highlighting function calls and queue accesses.

1. Instructions are fetched from the instruction cache. A pointer to the DynInst object is created, it represents an instruction passing through the pipeline stages. This instruction object is put onto a *Fetch Queue* that is later read by the *Decode* stage.

2. *Decode* receives instructions from the rename stage; function *decodeInsts()* pushes decoded instructions onto the *Decode Queue*. If during this process *Decode* has to block (e.g. due to a squash signal caused by branch misprediction, or due to not sufficiently many resources being available), then:

   (a) All instructions coming from Fetch in this cycle are put into a *skidBuffer*.

   (b) Once *Decode* is unblocked it first reads instructions from the *skidBuffer* and then proceeds with instructions coming from the *Fetch Queue*.

3. Registers are renamed and put onto the *Rename Queue*. This structure is later accessed by both IEW and Commit stages. The rename stage uses the same *skidBuffer* mechanism as described in the previous point. It is quite common for rename to block if there are not enough CPU resources available, e.g.:

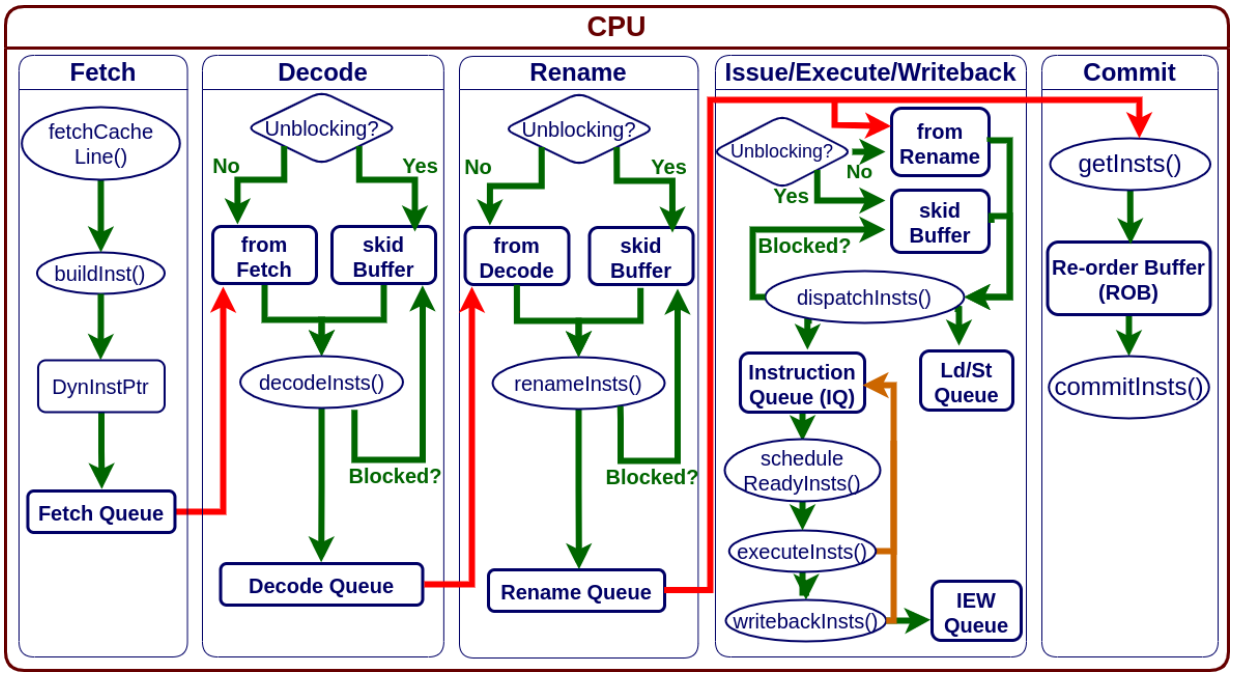


Figure (3.4) The flow of instructions in the O3CPU. The CPU class holds references to the classes representing each pipeline stage. Functions in the diagram are represented using oval shapes, data structures (rectangular) and conditions (diamond shapes). Communication between stages takes place using queues (interstage accesses are marked in red). The 'Unblocking' condition in *Decode*, *Rename* and *IEW* indicates checking what is the current state of this stage and determines where should instructions be read from in the given cycle.

(a) the *Instruction Queue* or the ROB has become full and there are no free slots to insert the instruction;

(b) lack of free physical registers to rename the instruction's architectural registers.

4. The IEW stage is a bit more involved as it consists of 3 smaller stages:

   (a) *Issue*: Reads instructions coming from Rename (or from IEW's skidBuffer). Function *dispatchInsts*() inserts them into an *Instruction Queue (IQ)* where they may need to wait until all required CPU resources are available.

   (b) *Execute*: Picks ready instructions (i.e. those that are not waiting on any previous dependencies or functional units) from the IQ and executes them. Results are written to the destination registers. The operations can involve simple ALU computation, memory loads or stores which are all handled differently at this stage. When the operation is finished, the instruction is marked as executed and the instruction is removed from the IQ.

   (c) *Writeback*: All dependent instructions of the currently executed one are woken up (i.e. if there are no further dependencies then they are marked as ready to execute within the IQ) and can now use the results of the executed instruction.

5. The Commit stage also reads instructions from the *Rename Queue* and inserts them into its *re-order buffer (ROB)*. This means that instructions can be simultaneously present in the IQ and the ROB. Instructions leave the ROB when they reach its head and are marked as executed, which happens in function *commitInsts*().

3.3.2 Entering PRE mode

Having explained how the O3CPU executes instructions, we can now move to the implementation details of PRE. The first significant difference from runahead is that PRE mode is entered upon a *full-window stall* as described in §2.5.1. Since no instruction will be pseudo-retired in PRE mode, there is no need to checkpoint the state of the rename map at this point. However, I still need to mark the instruction that triggered PRE in order to later detect when its memory request returns.

3.3.3 Execution in PRE mode

Even though during PRE no instructions reach the commit stage, the CPU still speculatively executes future instructions in the previous stages of the pipeline i.e. fetch, decode, rename and IEW stages. Instructions that are in the commit stage (i.e. in the ROB) before PRE is triggered continue their execution as if they were normal-mode instructions. They still occupy their CPU resources and only release them once executed. On the other hand, instructions that are in any of the pipeline stages prior to commit or are fetched in PRE mode are *PRE-mode instructions*. The underlying assumption is that there are sufficiently many IQ and register resources when PRE mode is entered to continue processing instructions in the earlier pipeline stages without the need to free any of the resources used by instructions that reside in the ROB.
The pipeline logic that I have implemented differs from the baseline in the following stages:

1. **Fetch**: this stage is identical to the baseline, with a small exception — all instructions that are fetched in PRE mode are marked as *PRE instructions*.

2. **Decode**: during PRE mode it inserts an instruction into the *Decode Queue* only if the instruction's PC hits in the *Stalling Slice Table (SST)*, otherwise the instruction is discarded. My implementation of the SST mechanism is described below in §3.3.4.

3. **Rename**: this stage would normally stall when there are no free ROB entries available. In PRE mode this requires amendments — instructions in the earlier pipeline stages need to progress even though there is no space in the ROB. This required updates in several functions that check for possible stalls at this stage. It is also worth reiterating that instructions in the ROB still hold their registers. This means that the performance of PRE is limited by the number of physical registers and the number of entries in the IQ (this will be discussed in §4.3.4).

4. **IEW**: runahead instructions are put into the IQ as normal. Once they are executed and have woken up any dependent instructions they simply leave the IQ. As opposed to normal-mode instructions, there is no need for communicating to the commit stage that this instruction has now executed since there is no space in the ROB for it.

5. **Commit**: this stage does not do anything different during PRE mode. However, some complications arise when the CPU has just exited PRE and the pipeline has not yet been flushed completely. There are still PRE-mode instructions present in the earlier pipeline stages. Since the CPU is not in PRE mode anymore, the head of the ROB can finally retire and hence the ROB is no longer full. At that point, commit

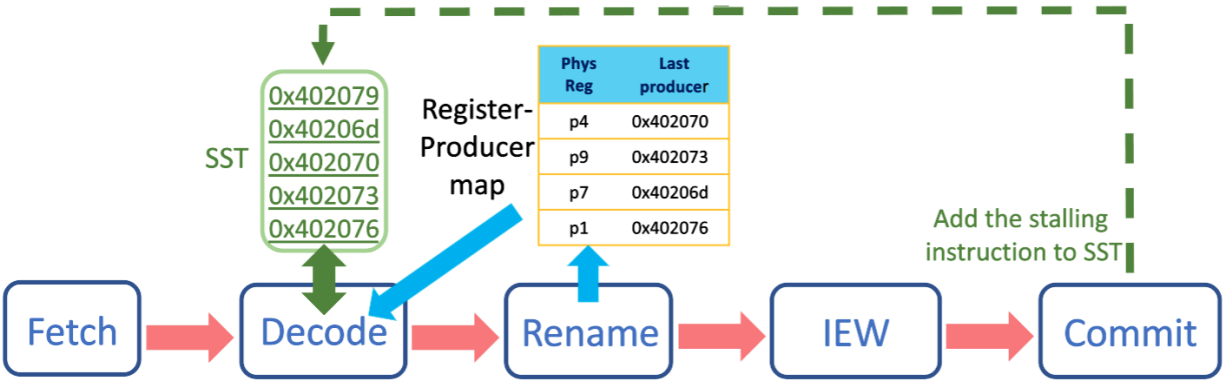


Figure (3.5) Pipeline interaction with the Stalling Slice Table and Register-Producer map. *Commit* accesses the SST upon entry into PRE mode. In PRE mode, *Decode* forwards only instructions whose addresses hit in the SST. *Rename* inserts new entries into the Register-Producer map.

will automatically attempt to insert instructions into the ROB from the *Rename Queue*. This causes a threat that a PRE-mode instruction will be inserted into the ROB before it is flushed from the pipeline. Hence, the function *getInsts()* needs to first check whether it is inserting a PRE-mode instruction, if that is the case then it must discard the instruction and stop inserting in this clock cycle.

3.3.4 Stalling Slice Table (SST)

As described earlier in §2.5.2, PRE uses the SST to detect and only execute those instructions that lead to full-window stalls. In order to implement this, I have modified the CPU object to own two additional structures: the SST itself and a map between physical registers and addresses of their most recent producers (Register-Producer map). The appropriate data structures to implement these functionalities were a set and map from the C++ standard library. Figure 3.5 presents interactions between the pipeline stages and these two structures. My implementation of SST follows these steps:

1. First, when a stalling instruction causes entry into PRE mode, its PC is stored in the SST, which is handled in the Commit class.

2. Later, when a PRE instruction is decoded, the Decode class checks whether the instruction's address hits in the SST:

   (a) If not, then the instruction is discarded and not executed in PRE.

   (b) On the other hand, if the PC hits in the SST, then the instruction is put into the *Decode Queue*; all source registers of this instruction are checked to determine the PCs of the instructions that produced these values most recently; these PCs are added to the SST.

The last point is where the Register-Producer map comes in handy. Its entries are updated whenever a destination register is renamed. The map entry associated with the given physical register stores the address of the instruction that produces the register's value. When a later instruction uses the same physical register, the entry in the Register-Producer map will be overwritten with the PC of the most recent producer.

Algorithm 2 Decode stage — interaction with the SST.

```
1: procedure DECODE::DECODEINSTS(THREADID TID)
2:     while instructions available and decodeWidth was not exceeded do
3:         determine_inst_source()          ▷ Instructions come from Fetch or skidBuffer.
4:         DynInstPtr inst = insts_to_decode.front()
5:         verify_that_inst_not_squashed(inst)
6:         if CPU in PRE mode then
7:             if CPU→ isInSST(inst→ instAddr()) then
8:                 for reg_idx in inst→ sources do
9:                     if reg_idx is in CPU→ reg_to_last_producer then
10:                        CPU→ addToSST(CPU→ reg_to_last_producer[reg_idx])
11:            else continue                                  ▷ instruction missed in SST
12:        ⋮                          ▷ Process instruction and put onto the Decode Queue
```

Figure (3.6) This code snippet presents the inner mechanism of the function responsible for decoding instructions. *Decode* can process at most *decodeWidth* of instructions per clock cycle (in my implementation the width is set to 8 instructions). The instructions can come from either the *Fetch queue* or *skidBuffer*.

Figure 3.6 presents how the *decodeInsts()* function interacts with the SST. The reader may wish to refer back to figure 3.4 to see how this function is placed in the pipeline.

3.3.5 Runahead Register Reclamation (RRR)

The SST is one of the enhancements introduced by PRE, the second one is a novel method of register reclamation which was described in the preparation chapter in §2.5.3.

The main issue with executing runahead instructions is that their registers never get freed in PRE mode and hence the CPU runs out of physical registers quickly. This happens because registers are freed only when an instruction is retired from the ROB, and PRE-mode instructions never even reach the commit stage. This problem is solved by *Runahead Register Reclamation*.

RRR uses a *Precise Register Deallocation Queue (PRDQ)*, whose entries are allocated at the back and are removed from the front (in FIFO order). My PRDQ's elements are structures, each containing three fields: *physRegToFree* (pointer to a physical register object), *instId* (a unique instruction sequence number) and *executed* (a bit field indicating if the associated instruction was executed). In order to implement RRR I added the following features to the Rename class:

1. A PRDQ entry is created during renaming of the instruction's destination registers.
2. Whenever a runahead instruction is marked as executed, the PRDQ is searched for the entry associated with this instruction. The *executed* field is set to true.
3. Every clock cycle the Rename stage attempts to retire entries from the front of the PRDQ. The head entry can be removed only if it is marked as executed. The associated register is then freed, as described in the following section.

Freeing registers

The trickiest part of implementing RRR was discovering the right way to free a physical register in gem5. The Rename class holds a *freeList* of registers used for renaming. However, to free a register is not as simple as adding it to the list because there are various special-purpose registers that cannot be freed (e.g. miscellaneous [21] or zero registers). In order to determine what registers can be freed, I make use of another rename structure — a list called *historyBuffer*. It holds the history of all register renames consisting of:

- the architectural register that was renamed,
- the new physical register,
- the old physical register that the architectural register mapped to
- the sequence number of the instruction that renamed.

This list is used to undo the mappings if an instruction is squashed, or to free the old physical register when an instruction is committed. In order to free a physical register associated with a PRDQ entry, I need to iterate through the *historyBuffer* looking for the instruction ID specified in the PRDQ entry. Given the entry, I can now determine whether the old physical register is a special purpose one — if the old and new physical registers are the same then it is special-purpose and should not be freed. Otherwise, I free the register by adding it to the *freeList*. However, because the instructions are renamed in the program order, their entries in the PRDQ appear consecutively. Hence, I can optimize the naive approach by iterating the *historyBuffer* only once per instruction ID.

Importantly, the *historyBuffer* entry is not removed at this stage, but later when the CPU exits PRE mode and all instructions in the pipeline are squashed. While squashing, the *historyBuffer* will be iterated through again and all renames will be reverted.

Lastly, special attention must be paid to not free any register that is used by an instruction residing in the ROB. This can be achieved by not creating PRDQ entries for any normal-mode instructions.

3.3.6 Exiting PRE

The last considerable difference from runahead implementation is the squashing mechanism. Like in runahead (described in §3.2.3) I exploit the existing squashing mechanism in gem5. This time, however, I wish to only squash PRE-mode instructions (i.e. these in stages prior to Commit) and avoid modifying anything in the ROB. Instructions are squashed in reverse order i.e. beginning with the youngest one and ending with the oldest runahead instruction. Once an instruction is squashed, it leaves the pipeline without having any effect, its register renames are undone and all occupied resources are freed.

Nonetheless, the implementation differs significantly from runahead. My runahead implementation squashes all instructions until and including the one that triggered runahead. Whereas the PRE implementation should ideally squash all instructions until but excluding the last instruction in the ROB. However, a couple of implementation-specific complications arose, and I decided to modify this condition to: squash all instructions until and including the youngest valid instruction in the ROB (one that is not marked as squashed). I describe the exact reasons for this choice in the following section.

3.3.7 Debugging challenges

Throughout my work with gem5, especially PRE implementation, I have encountered several insidious bugs. In this section, I focus on shedding the light on the mysteries and peculiarities I managed to resolve.

Squashing mechanism

An instruction squashing mechanism is extremely convoluted, and implementing it myself would be beyond the scope of my project. Hence, I decided to use the existing squashing signals that were used for branch mispredictions. This, however, added some complexity:

1. The squash signal needs to travel from IEW to Commit, and the latter propagates the squash signal to all other stages. This means that there is a delay (precisely 3 CPU clock cycles) between when the CPU exits PRE and when all pipeline stages have squashed PRE-mode instructions. Hence, care must be taken to avoid treating PRE-mode instructions as normal ones at all pipeline stages. This became especially prominent when the Commit stage attempted to insert runahead instructions into the ROB, one or two cycles after PRE was exited.

2. Commit does not propagate the squash signal if the *squash sequence number* (identifier of the instruction until which the CPU is squashing) is younger than the youngest unsquashed instruction in the ROB (indicated by an instruction sequence number *youngestSeqNum*). This prevents squashes from younger instructions from overriding squashes from older instructions. For my implementation, this means that:

   (a) I need to squash at least one instruction in the ROB because otherwise Commit would ignore this squash signal and not send it to the earlier stages. Hence, instead of squashing until one instruction past the full ROB, I should squash until the last instruction in the ROB.

   (b) However, the tail instruction of the ROB may be younger than the *youngestSeqNum* in the ROB. In this scenario, I should squash all instructions until one preceding the *youngestSeqNum*. But finding the preceding instruction turns out not to be straightforward either.

   The name "the youngest valid instruction in the ROB" as documented in gem5 was quite misleading because the instruction with *youngestSeqNum* is not always present in the ROB. Let's imagine that the ROB contains a sequence of instructions: 1, 2, 100, 101, ... Such a gap in the numbering may occur e.g. if instructions 3–99 were runahead instructions and were squashed. Now, if a branch misprediction causes all instructions not older than 100 to be squashed, then the new *youngestSeqNum* according to the existing code is: $100 - 1 = 99$, but the corresponding instruction is not present in the ROB.

   This unstated assumption does not affect the baseline, however, it did confuse me for a couple of weeks. The solution is quite simple — iterate through the ROB and find the actual youngest valid instruction in it, squash up to the corresponding sequence number and including it.

These issues are particularly painful to debug as it is difficult to observe PRE-mode instructions. There is no single structure that keeps track of all the in-flight instructions like the ROB does for the uncommitted microoperations. They are scattered across multiple stages and queues simultaneously.

IEW Skid buffer

Another poorly documented structure that cost me lots of debugging was the IEW dispatch *skidBuffer* whose intended behaviour I described in §3.3.1. The O3CPU model silently assumes that upon a squash all instructions in this *skidBuffer* can be safely flushed. However, this assumption is problematic for my PRE implementation when the following happens:

1. The CPU enters PRE; an instruction I is in the ROB and in the IEW's *skidBuffer* simultaneously.

2. The instruction queue (IQ) becomes full and as a result, the instruction I is stuck in the *skidBuffer* for the entire duration of the PRE interval.

3. The CPU exits PRE; signals a squash of all instructions until an instruction younger than I. The current implementation would simply empty the *skidBuffer* and the aforementioned instruction I would never be scheduled for execution. This leads to the instruction becoming stuck at the head of the ROB forever and the simulation not terminating. To remedy this, I needed to change the mechanism to **not** remove older instructions than the squash sequence number from the *skidBuffer*.

4. Additionally, if after exiting from PRE the *skidBuffer* is non-empty, then the IEW dispatch stage needs to change status to *Blocked*. Otherwise, the dispatch function will not start reading instructions from the *skidBuffer* but from the *Rename Queue* instead, effectively ignoring those instructions waiting for the empty IQ slot forever.

Those issues posed a tremendous challenge to resolve during my project. The behaviours were undocumented, and I had to dig deep into the gem5 code in order to grasp some understanding. Finally, both of my implementations were running correctly on every benchmark tested, and I could proceed to the rewarding part of evaluating my project.

Chapter 4

Evaluation

In this chapter, I describe how my project exceeds all its evaluation success criteria. In §2.4, I set out that runahead execution is an alternative to unreasonably large instruction windows. I prove this is consistent with my results in the following analysis. First, I present my simulation setup (§4.1) and the benchmark programs used (§4.2). Later, I move on to analyse the performance of runahead and PRE methods (§4.3), as well as their impact on power consumption (§4.4).

Throughout this chapter my plots omit error bars as the single-core simulator runs are deterministic. Simulations start with empty caches and there is no perturbation introduced during execution. It is also not uncommon for computer architecture papers to omit error bars altogether ([5], [10], [22]).

4.1 Configuration

Evaluation of multiple benchmarks demanded a versatile simulation setup. I wrote a script which allowed me to parallelise simulation runs with different parameters:

- Path to the benchmark program run on the simulated CPU;
- The mode of execution, i.e. baseline, runahead or PRE;
- Sizes of the L1 and L2 caches [kB];
- Size of the ROB [number of instructions].

I have explored 15 configurations for each benchmark and execution mode, running more than 300 processes in parallel. Crucial system parameters are presented in table 4.1.

4.2 Testing and benchmark suites

I used a total of 7 benchmark programs commonly used in computer architecture. These are of two types:

- Memory-latency bound programs used for testing prefetching of indirect memory accesses [23] that include: Conjugate Gradient (CG), RandomAccess (RA). The latter benchmark accepts an argument indicating the size of an array which is accessed. I evaluate two sizes and hence call my programs randacc500k and randacc600k.

Frequency	2.66 GHz
Instruction Queue	92
Load Queue	64
Store Queue	64
Pipeline width	8
Branch predictor	Tournament predictor
Register file	256 integer registers 256 floating point registers 256 vector registers 32 predicate registers
L1 I-cache	32 KB, 4-way associative, 2 cycles access latency
L1 D-cache	32 KB, 8-way associative, 4 cycles access latency
L2 cache	varying size, 8-way associative, 8 cycles access latency

Table (4.1) Configuration of the simulated CPU. These parameters were chosen to resemble the simulation setup described in the PRE paper [10]. Throughout this chapter I analyse various sizes of the ROB.

- A collection of benchmarks based on standard file types (i.e. text, audio, image) [24]: qsort, susan, bzip2d, dijkstra, consumer_lame.

4.2.1 Statistics

The O3CPU (§3.1) comes with some built-in statistics, such as IPC, number of cycles, number of full physical registers events etc. These constitute a reliable source of information about how the system is performing. I have extended these statistics to include more parameters that are crucial for testing my implementations. The essential statistics evaluated in this chapter are:

- *totalCyclesRA* — The number of cycles spent in runahead mode in total. This will constitute an indication of how many opportunities for improvement during runahead a given benchmark exhibits.
- *l1Miss*, *l1MissRA* and *l2Miss*, *l2MissRA* — The numbers of misses in L1 and L2 data caches taking place respectively in normal mode and in runahead modes.
- *pctRobEmptyRA* — The percentage of cycles spent in runahead mode when the ROB is empty. This considers scenarios when the CPU is still waiting on data to return, but the pipeline cannot keep up with dispatching new runahead instructions (e.g. due to an Icache miss) and effectively no useful work is being done.
- *freeRegsAvg* — The average number of free registers when PRE mode is entered. Tracking this statistic allows me to test the assumption of PRE that this number is big enough to continue execution.

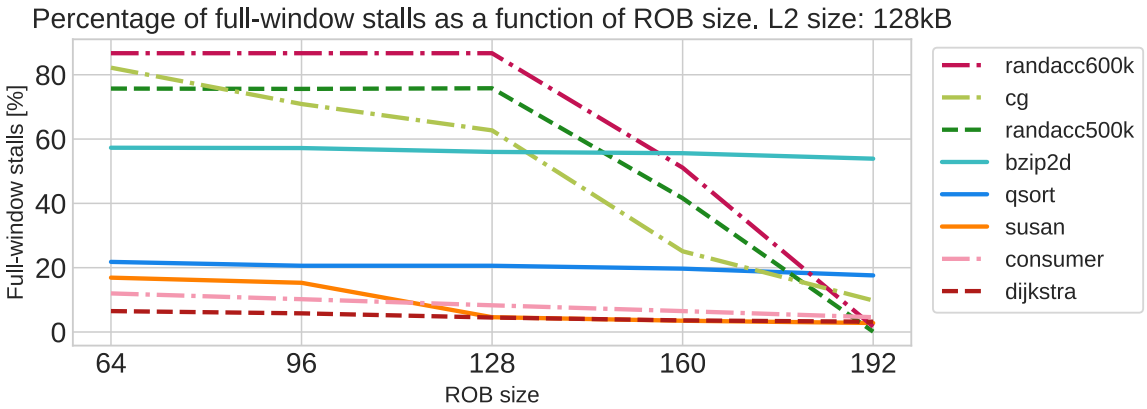


Figure (4.1) The percentage of full-window stalls. As the ROB size increases full-window stalls occur less often. However, some workloads e.g. bzip2d, qsort, consumer and dijkstra experience roughly the same amount of full-window stalls regardless of the ROB size.

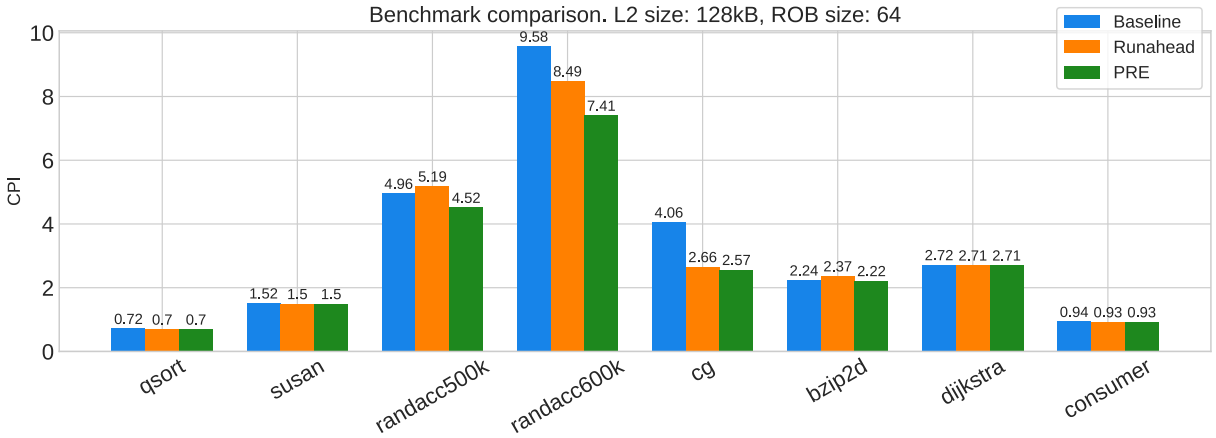


Figure (4.2) Cycles per instruction (CPI) comparison. Runahead methods visibly improve memory-intensive workloads which exhibit high CPI count. Performance improvement on the remaining programs is still observable but less manifesting since the CPI is already low.

4.3 Performance analysis

According to the paper, which proposed runahead [5], modern computers can spend even 70% of all cycles in full-window stalls, cycles when the ROB is full and its head instruction is not ready to commit. In figure 4.1 I demonstrate that this observation significantly depends on the ROB size. The bigger the ROB the less often full ROB stalls occur and, consequently, runahead methods will get fewer opportunities to enhance performance. Therefore, benchmarks exhibiting a higher percentage of full-window stalls lend themselves better to runahead optimisations. Figure 4.2 investigates the cycles per instruction (CPI) count for a ROB size of 64 entries. It is apparent that PRE always improves on the baseline's CPI, while runahead may impair the overall performance for some benchmarks.

To verify the relation between the ROB size and possible runahead improvements I inspect the number of cycles spent in runahead and PRE modes as a function of the ROB size. Figure 4.3 highlights the quick decrease in the number of PRE-mode cycles as the ROB size increases. This implies that PRE has few opportunities to enter the speculative mode when ROB sizes exceed 128. Hence, the most significant performance gains are likely going to arise on smaller ROB sizes. Furthermore, this observation confirms the premise

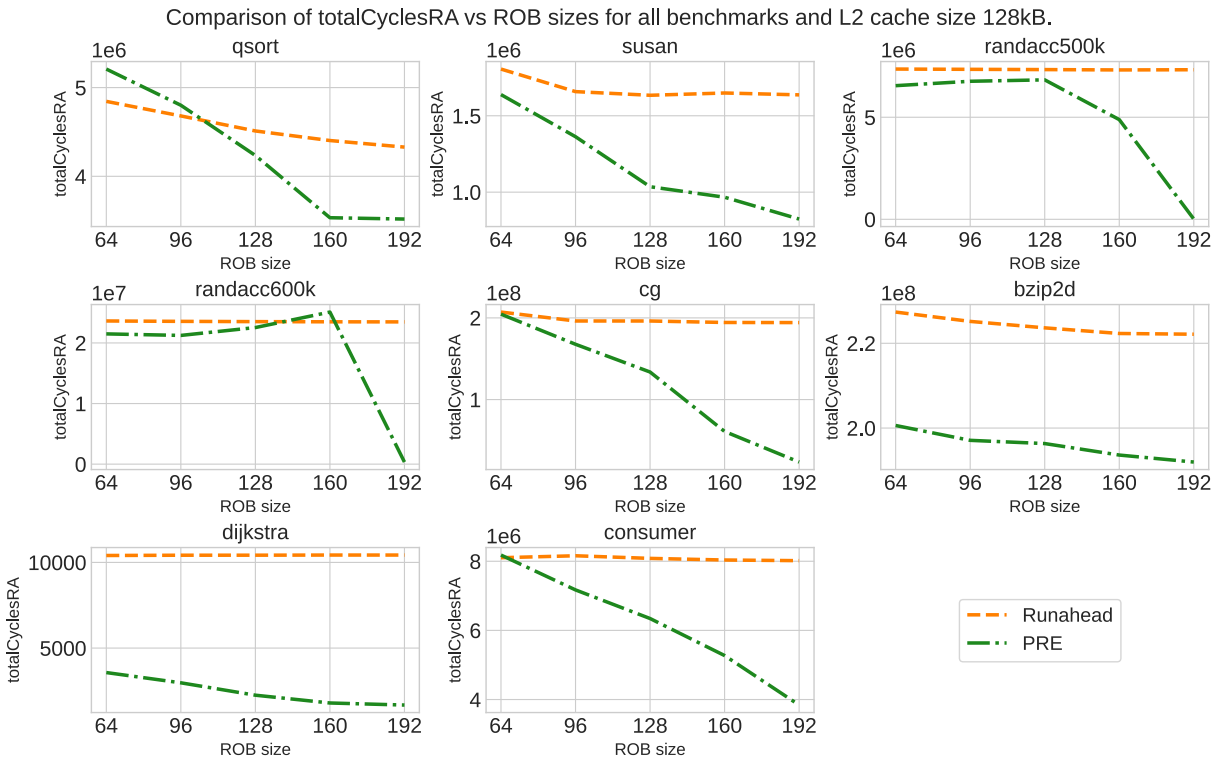


Figure (4.3) The total number of cycles spent in runahead modes. In runahead, triggering runahead mode is independent of the ROB size. Nevertheless, bigger ROB sizes allow instructions more clock cycles to execute before reaching the head. Hence, benchmarks qsort, susan and bzip2d exhibit a visible decrease in the number of cycles spent in runahead mode as the ROB increases. On the other hand, PRE usually gets fewer opportunities to enter PRE mode when the ROB is bigger. This results in a significantly lower number of clock cycles in PRE mode when the ROB has e.g. 192 entries.

that runahead methods constitute an alternative for large instruction window sizes.

In this section I further evaluate what impact runahead methods have on the overall runtime of a program (§4.3.1), the throughput of the system (§4.3.2), and how they influence the caching behaviour (§4.3.3). Lastly, I discuss some performance limitations (§4.3.4).

4.3.1 Program runtime

It is crucial to compare the time a benchmark takes to complete with and without runahead. A good indication of how quickly the program runs is the total count of CPU clock cycles executed which is summarised in figure 4.4. Evaluating this metric for varying ROB sizes I found that PRE always takes fewer cycles to complete than the baseline. Moreover, it provides a speed-up in all test runs with ROB sizes smaller than 160 entries. On the other hand, runahead often worsens the performance by slowing down the simulation and requiring more clock cycles. This is expected, the performance of runahead depends heavily on the benchmark's nature and may sometimes degrade performance. PRE, however, combines the best of both worlds providing improvement where possible and not impairing performance otherwise.

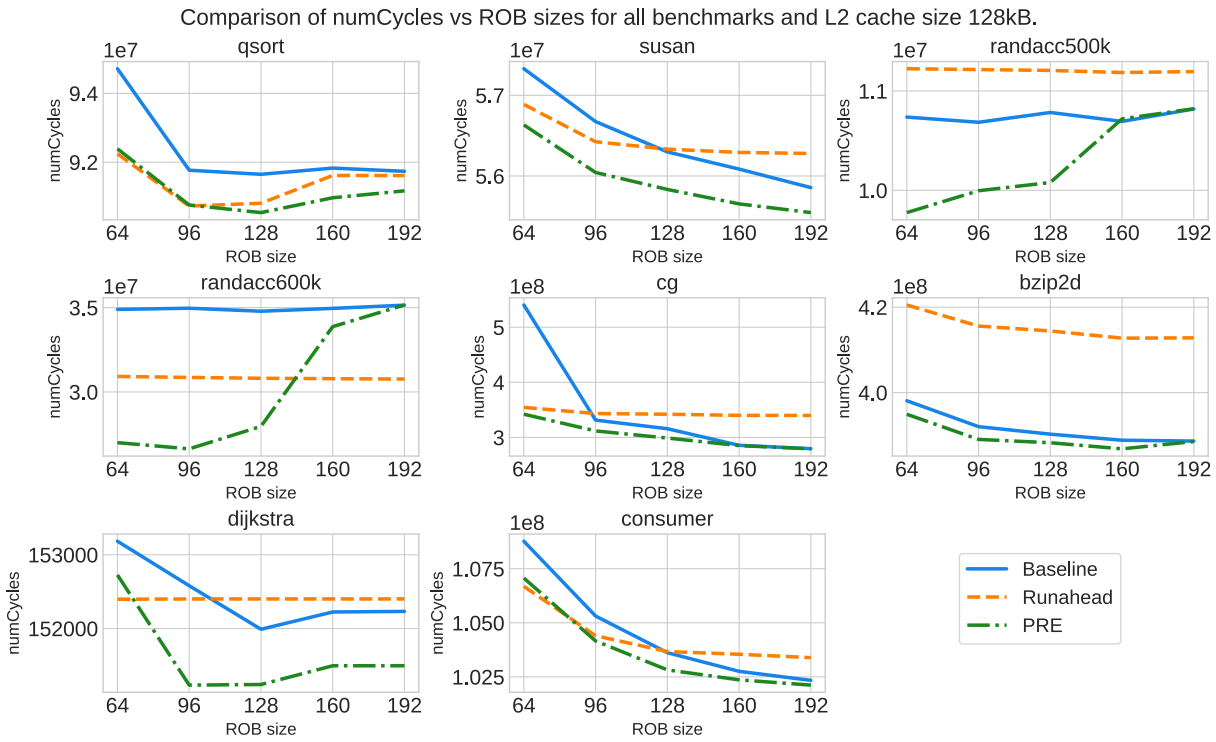


Figure (4.4) The number of clock cycles each benchmark requires to execute across ROB sizes. Interestingly, PRE runs on benchmarks: qsort, randacc and dijkstra with a ROB size of 96 by far outperform the baseline runs with a ROB size of 192. This again indicates that the method is a good alternative to big instruction window sizes.

4.3.2 Throughput

The average number of instructions executed in each clock cycle — Instructions Per Cycle (IPC) — indicates what is the overall throughput of the system. This is another metric where runahead methods' improvement becomes apparent.

Firstly, I compare the IPC across different ROB sizes. Figure 4.5 highlights that PRE always achieves better or equally good throughput as the baseline. As stated earlier the best performance gains are achieved on ROB sizes smaller than 160.

Secondly, in figure 4.6 I investigate how the size of the L2 cache impacts the throughput of the system. Comparing IPC across varying L2 sizes I determined that a bigger cache does usually aid performance but does not affect the amount of improvement resulting from my runahead implementations. Hence, the rest of my plots present data for a L2 cache of size 128kB.

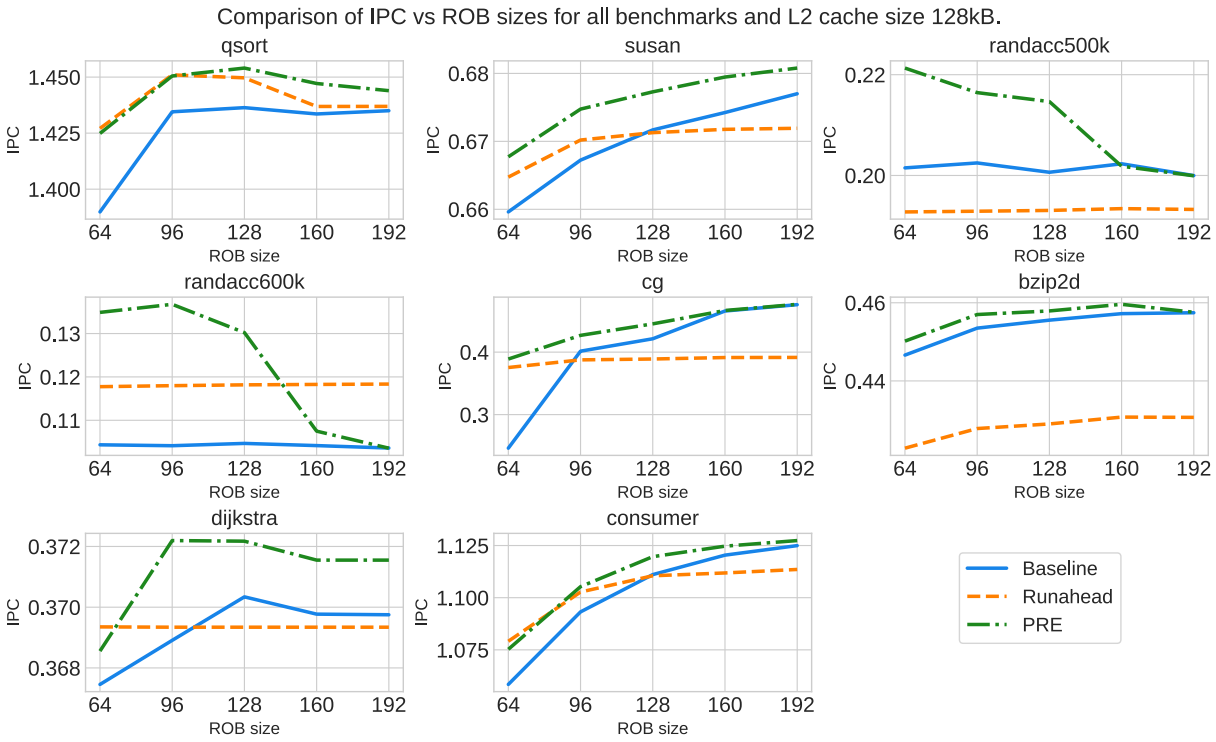


Figure (4.5) Instructions Per Cycle (IPC). The higher IPC the bigger throughput the system has. Runahead performs far worse than PRE on benchmarks randacc and bzip2d, although both these programs spent around 60% of execution time in full-window stalls. This may be caused by little useful prefetches made in runahead mode. PRE only executes instructions that hit in the SST hence making better use of the CPU resources.

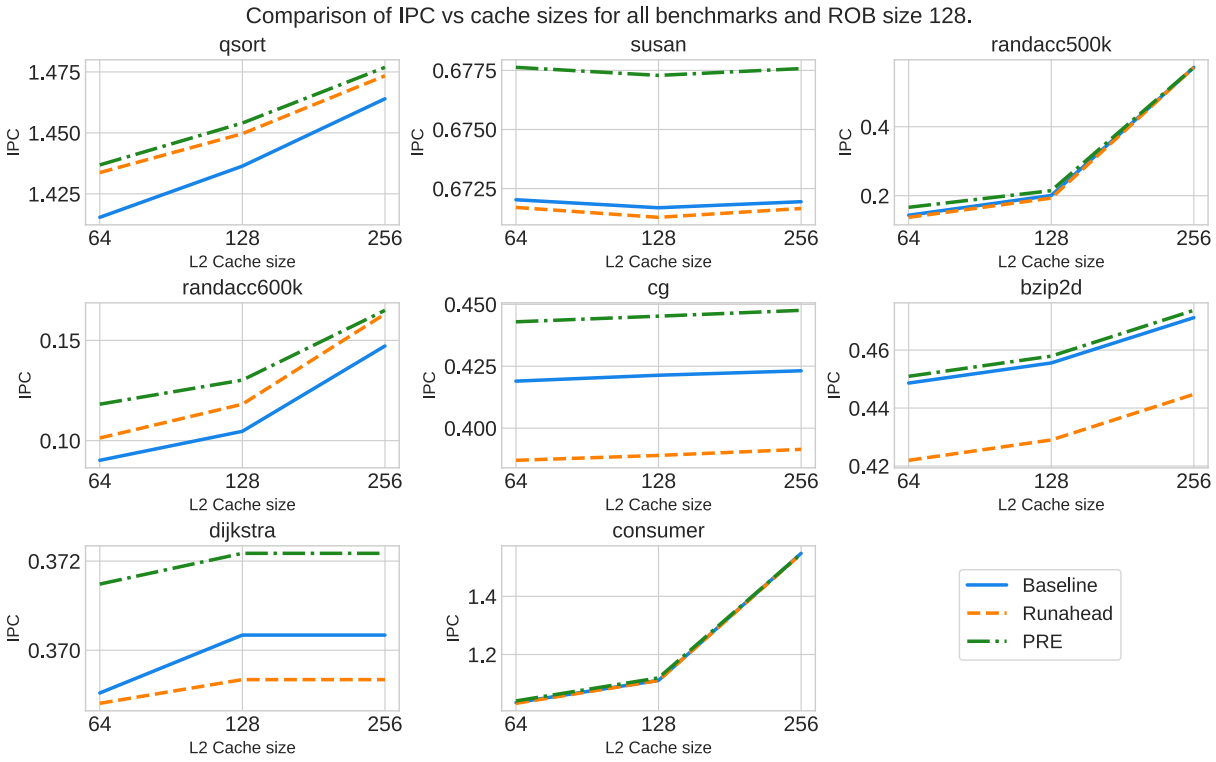


Figure (4.6) IPC comparison across different L2 cache sizes with ROB size of 128. Bigger caches enhance performance as they allow more data to be stored in L2 and therefore the accesses to main memory are less frequent. The variation in cache sizes does not visibly affect the amount of improvement stemming from runahead.

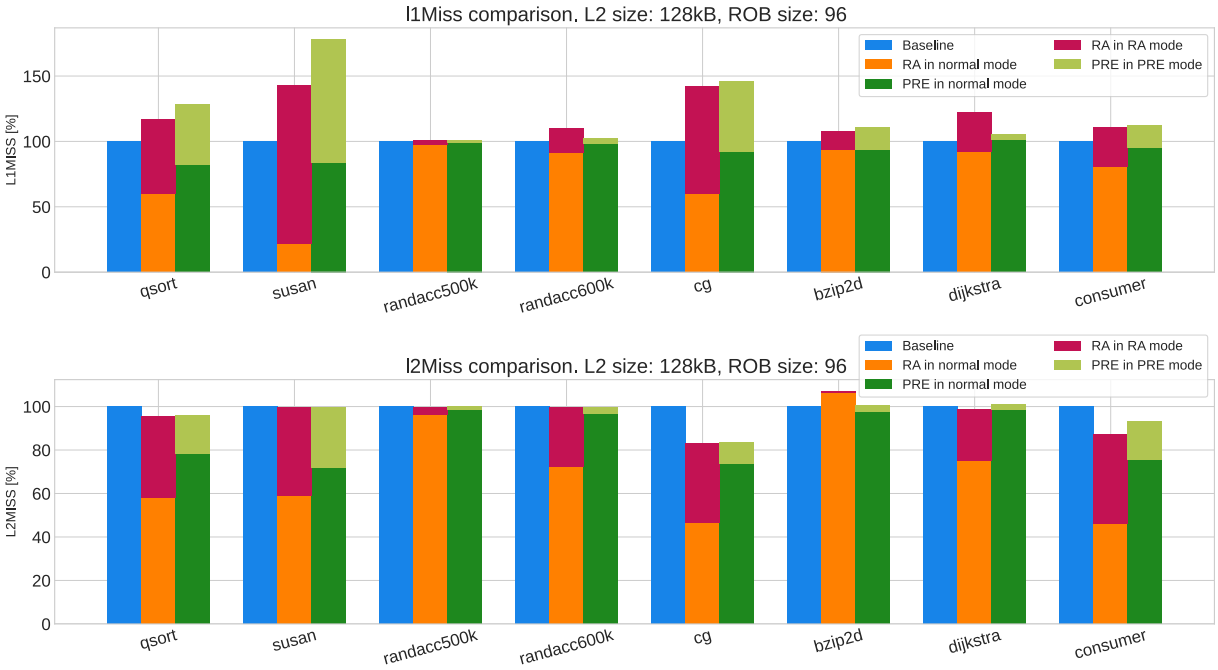


Figure (4.7) Percentage of misses happening in L1 and L2 caches in normal and runahead modes relative to the baseline. Interestingly, runahead implementation improves L2 miss counts for all benchmarks but bzip2d — this is a program where runahead has decreased the performance in all evaluated metrics. PRE usually does not result in fewer misses in normal mode than the runahead implementation (apart from bzip2d), which is due to the fact that PRE mode is entered less frequently and hence there are not as many opportunities for improvement. Nevertheless, PRE does usually improve on the baselines L2 miss counts.

4.3.3 Cache misses

The last important metric I wish to evaluate is the number of cache misses taking place in L1 and L2 data caches. Runahead methods aim to decrease the number of cache misses appearing in normal mode. However, they may increase the overall number of cache misses as they also aim to create as many data prefetches in runahead mode as possible. For this reason, I analyse normal and runahead modes separately. Figure 4.7 presents the percentage of cache misses in L1 and L2 respectively, normalised to the misses taking place in the baseline.

Runahead implementations exhibit a higher total miss count in the L1 cache for almost all benchmarks. However, the percentage of misses happening in normal mode is always below 100%, i.e. there are always fewer L1 cache misses in normal mode than in the baseline.

L2 misses exhibit a different pattern on average. Not only do cache misses in normal mode always decrease in comparison to the baseline, but also the number of cache misses in runahead and normal modes in total rarely exceeds 100%. This brings me to the conclusion that both runahead implementations are effective at prefetching data into caches during their speculative modes of execution. Thereby they achieve the goal of leveraging unused CPU resources to generate accurate data prefetches.

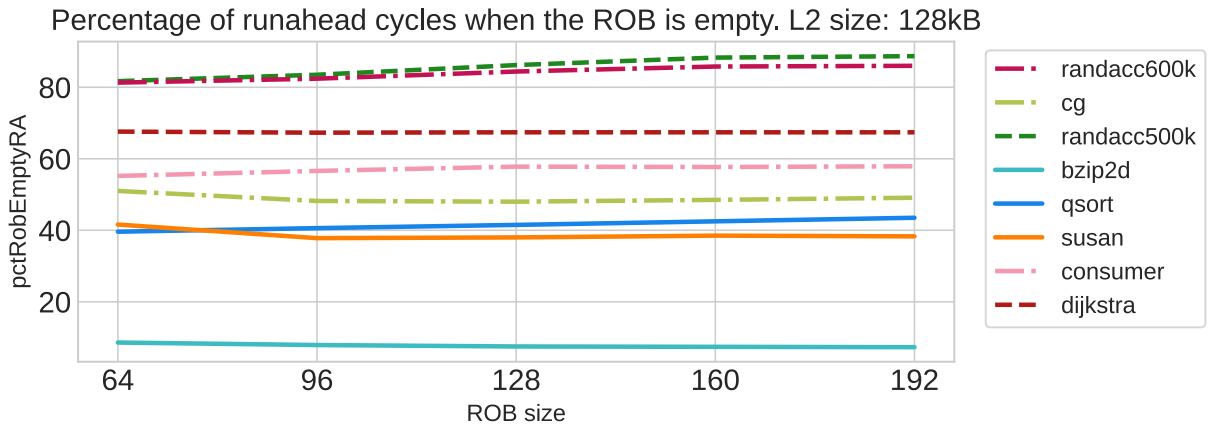


Figure (4.8) A high percentage of cycles with empty ROBs in runahead mode indicates that there is still space for improvement over runahead. Bzip2d is the only benchmark with a low percentage of empty ROB cycles. However, we have already seen that runahead decreases performance on this benchmark, so this is not a particularly interesting workload for this method.

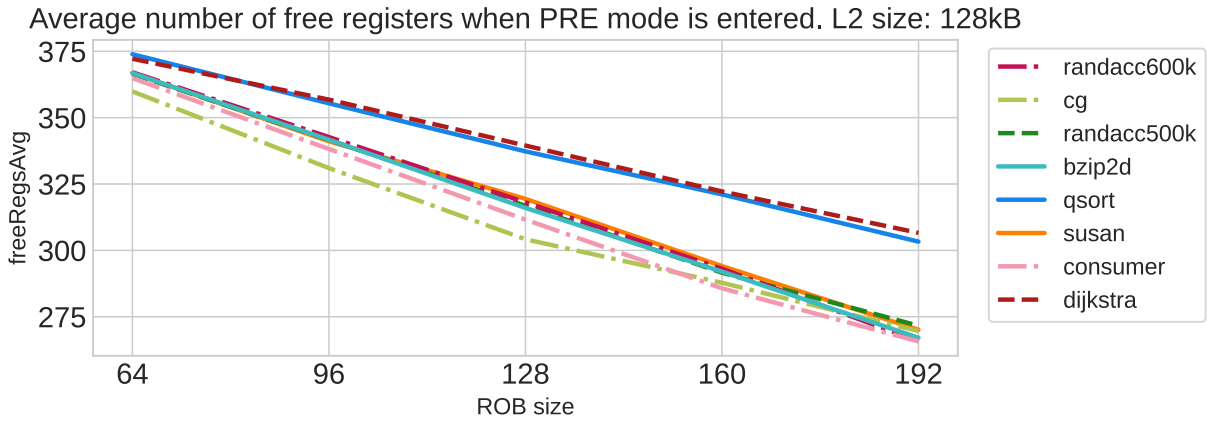


Figure (4.9) The average number of free registers when PRE mode is triggered visibly diminishes as the ROB size increases. This evaluation metric is counted by summing up the numbers of free registers on each entry into PRE mode and dividing this total by the number of PRE intervals.

4.3.4 Limitations

During my evaluation, I noticed that the CPU is often idle in runahead mode. This is observable when the ROB is empty and, hence, the resources are not being used to their full capacity. Figure 4.8 presents that the percentage of all runahead-mode cycles when the ROB is empty is on average 60%.

During PRE mode, on the other hand, the ROB is always full, but performance is limited by the number of physical registers available when PRE mode is triggered. Figure 4.9 highlights that the bigger the size of the ROB the fewer registers are free upon entry to PRE mode. This is because more instructions are present in the ROB and hence more registers are occupied. Nevertheless, from my observations, the average number of free registers never goes below 260 registers. This means that there are sufficiently many register resources for an average PRE interval to execute at least 260 instructions.

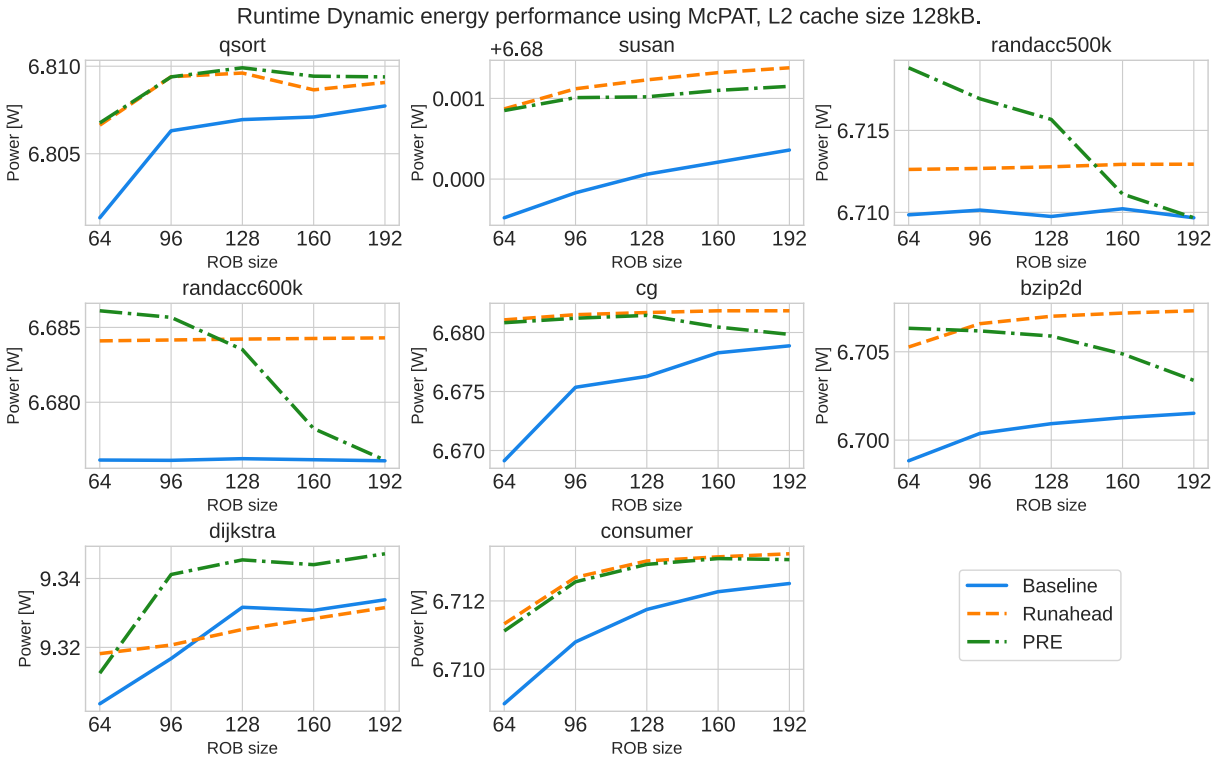


Figure (4.10) Power consumption measurements using McPAT power modelling framework. PRE's power consumption is nearly equal to the baseline's for bigger ROB sizes because it gets few opportunities to enter the speculative mode. It is crucial to notice that values on the y-axis increase very slowly. In fact, neither runahead nor PRE ever exceed the power consumption of the baseline by more than 0.3%.

4.4 Power consumption

To measure power consumption I used a power modelling framework — McPAT [6]. It quickly transpired that there is no version of the tool that is compatible with the version of gem5 that I used. Integrating these two required some reverse engineering of McPAT's inner workings and adjusting XML templates used to produce the measurement. Figure 4.10 presents power consumption costs of all three implementations, i.e. energy used divided by the time elapsed during the simulation. The evaluation proved that my runahead methods impose negligible power overheads of 0.1% on average.

Chapter 5

Conclusions

I have satisfied all success criteria stated in my project proposal (Appendix A). These included setting up project's environment and baseline for experiments, implementing runahead and PRE methods. The main evaluation goal was to compare the implementations in terms of performance and energy consumption.

My experiments have shown that on memory-intensive workloads (e.g. randacc500k, randacc600k, cg) with a ROB of size 96, PRE achieves an average improvement in IPC of 32% over the baseline and 12.7% over runahead. Moreover, when evaluated on other benchmarks and other configurations, PRE never decreases the system's performance. At the same time the energy overhead from implementing PRE is negligible, equalling 0.1% on average. This reasonably reproduces the expected system performance described in the papers ([5], [10]).

Furthermore, I have exceeded the evaluation success criteria by exploring features that make programs better suited for runahead optimisations such as the percentage of full-window stalls, cache access patterns and choosing memory-intensive workloads. I have also verified the hypothesis that runahead execution is an alternative to designing ever-growing instruction windows, as my PRE implementation on a ROB size of 96 usually exceeds the performance of the baseline with a ROB size of 192.

5.1 Lessons learnt

Never before have I worked with such a huge software tool, this was itself a valuable lesson. Most importantly, I have gained a profound understanding of processors' inner mechanisms, broadened my knowledge about computer architecture research and simulation software. However, I still have space for improvement in my project management skills.

Firstly, my initial timeline was quite optimistic. Introducing what seemed like simple changes often manifested with errors in seemingly unrelated chunks of code. For instance, I expected that the implementation of the squashing mechanism in PRE would closely resemble the one I used in runahead. After roughly 3 weeks of scrutinizing the debugging logs, I finally managed to overcome the challenges that I described in §3.3.7.

Secondly, having experienced multiple implementation issues, as well as delays due to my health condition I have learned how to choose between nice-to-have features and absolute essentials. This, I believe, constitutes a useful lesson for any future software projects I will engage in.

Nevertheless, having faced multiple struggles I have delivered all desired milestones and implemented methods that exhibit promising performance improvements. Furthermore, I believe that getting to understand the source code progressively was a good working methodology. It would have been impossible to comprehend the whole simulator at the outset of the project.

5.2 Future work

One of my implementation choices in runahead was to ignore the writeback phase of all instructions i.e. instructions write their results to the registers but not to other levels of the memory hierarchy. This was done to avoid any of the speculative values being stored in a cache and later accessed in normal mode. However, this approach limits the number of instructions that can be executed during runahead. It would be interesting to implement a runahead cache that is only used during runahead mode and allows communicating store values between speculative instructions. This would clearly increase the hardware overhead and energy consumption, these tradeoffs would constitute an interesting evaluation metric.

Lastly, PRE — being an already great improvement on runahead — is still imperfect. Modern programs exhibit highly indirect memory access patterns, and these cannot be predicted by PRE, nor by hardware prefetchers. Thus, another method — Vector Runahead (VR) — was proposed [22], to improve on PRE's constraints; prefetch data along indirect chains of memory accesses and fully exploit CPU resources in runahead mode by issuing multiple independent loads in parallel. Implementing VR and comparing its performance to runahead and PRE is an interesting direction for future work.

Bibliography

[1] *MOS Technology 6502.* URL: https://en.wikipedia.org/wiki/MOS_Technology_6502.

[2] *A Look Back at Single-Threaded CPU Performance.* URL: https://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance.

[3] David Patterson by John Hennessy. "Computer Architecture A Quantitative Approach". In: (), p. 73.

[4] *Intel Core i7-8700 Processor.* URL: https://www.intel.co.uk/content/www/uk/en/products/sku/126686/intel-core-i78700-processor-12m-cache-up-to-4-60-ghz/specifications.html.

[5] Onur Mutlu et al. "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.* (2003), pp. 129–140.

[6] Sheng Li et al. "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures". In: (2009), pp. 469–480.

[7] Nathan Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718.

[8] *The Alpha 21264 CPU.* URL: https://en.wikipedia.org/wiki/Alpha_21264.

[9] *Source code for the gem5 simulator.* URL: https://gem5.googlesource.com/public/gem5.

[10] Ajeya Naithani et al. "Precise Runahead Execution". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), pp. 397–410.

[11] *Spiral development model.* URL: https://en.wikipedia.org/wiki/Spiral_model.

[12] *pCloud — Cloud storage service.* URL: https://www.pcloud.com/eu.

[13] *Visual Studio Code Remote - SSH extension.* URL: https://code.visualstudio.com/docs/remote/ssh.

[14] *sshfs(1) - Linux man page.* URL: https://linux.die.net/man/1/sshfs.

[15] *BSD licenses*. URL: https://en.wikipedia.org/wiki/BSD_licenses.

[16] *BSD 2-Clause "Simplified" License*. URL: https://en.wikipedia.org/wiki/BSD_licenses#2-clause_license_(%22Simplified_BSD_License%22_or_%22FreeBSD_License%22).

[17] *Documentation and installation instructions for gem5 simulator*. URL: https://www.gem5.org/documentation/learning_gem5/introduction/.

[18] https://www.gem5.org/documentation/general_docs/cpu_models/O3CPU.

[19] Tien-Fu Chen and Jean-Loup Baer. "Reducing Memory Latency via Non-Blocking and Prefetching Caches". In: *SIGPLAN Not.* 27.9 (1992), pp. 51–61. ISSN: 0362-1340. DOI: 10.1145/143371.143486. URL: https://doi.org/10.1145/143371.143486.

[20] *objdump(1) — Linux manual page*. URL: https://man7.org/linux/man-pages/man1/objdump.1.html.

[21] *Miscellaneous registers on x86*. URL: https://en.wikipedia.org/wiki/X86#Miscellaneous/special_purpose.

[22] Ajeya Naithani et al. "Vector Runahead". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), pp. 195–208.

[23] Sam Ainsworth and Timothy M Jones. "Software Prefetching for Indirect Memory Accesses". English. In: CGO '17 (Feb. 2017). International Symposium on Code Generation and Optimization (CGO) 2017 ; Conference date: 04-02-2017 Through 08-02-2017, pp. 305–317.

[24] Grigori Fursin et al. "MiDataSets: Creating the Conditions for a More Realistic Evaluation of Iterative Optimization". In: (2007). Ed. by Koen De Bosschere et al., pp. 245–260.

Appendix A

Project proposal

Running ahead of memory latency - processor runahead. Project Proposal

Marta Walentynowicz (mmk64) - Queens' college

October 14, 2021

1 Description of the Project

Since the mid-1980s the microprocessor architecture has witnessed rapid advancements. This was achieved thanks to technology scaling, shortening of critical paths (e.g. by pipelining) and a decrease in instruction count. Since 2002, however, the impact of memory latency, limitations to instruction-level parallelism and power consumption have contributed to a significant slowdown in the improvements.[1]

Main memory access still remains a bottleneck for modern CPUs. A cache miss results in hundreds of processor cycles waiting on the data to be retrieved from memory. In order to mitigate these stalls, an out-of-order mode of execution was introduced. An OoO processor uses a re-order buffer (ROB). Instructions enter the ROB, they can speculatively execute out-of-order, and they retire from the ROB in-order. The set of instructions in the ROB is called an instruction window.

In case of a cache miss, all non-stalled instructions in the ROB can execute while the CPU is waiting for the data to come back from the main memory. Thus, the size of the ROB determines how much latency the processor can tolerate before experiencing a performance decrease. This has lead to designing ever-growing in size instruction windows, hence significantly increasing power consumption and complexity.

Runahead execution was proposed as an alternative to unreasonably large instruction windows. When a long-latency instruction is encountered (e.g. last level cache miss), the stalling instruction is tossed out of the instruction window, the registers' state is checkpointed, and the CPU enters the runahead mode. In runahead, the following instructions in ROB are executed, but their results are not committed, they pseudo-retire from the instruction window. Once the blocking operation is completed, all results from the runahead mode are thrown away, checkpointed values are retrieved and the CPU resumes normal execution. The data and instructions which were fetched into the cache during runahead, constitute very accurate prefetches.

According to the paper, which proposed runahead [1], modern computers can spend even 70% of all cycles in full instruction window stalls. Furthermore, a machine with runahead and a 128-entry ROB can perform within 1% of a machine with a 384-entry ROB and no runahead.

My project will aim to implement runahead and an improvement to this technique - Precise Rrunahead Execution (PRE) [2]. I aim to evaluate the two techniques against each other and, time permitting, to compare these to a further enhancement - Vector Runahead [3]. The design will be created and evaluated using a computer system simulator platform - gem5.

2 Starting Point

I have no previous experience with designing hardware architecture, neither have I ever worked on a project using the simulator platform gem5. The starting point of the project is hence nearly from zero. Over my summer break, I have read all three aforementioned research papers about runahead, I have downloaded gem5 and run it on sample configuration scripts.

[1] Performance of microprocessors between 1985 and 2002 was improving at a stunning rate of 52% per year. This rate was diminished to around 20% after 2002. `http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf`

3 Key Components

The core part of the project will consist of the following four parts:

3.1 Setting up the simulation environment

Familiarising myself more with gem5. Creating an architecture that will serve as a baseline for comparison to the runahead techniques. I will measure the performance in terms of the number of clock cycles per instruction (CPI), as well as the energy consumption. Tests will be carried out on multiple architectures. I will also need to decide on what benchmark programs to use.

Key challenge: Working flexibly with gem5.

3.2 Implementing runahead

The first substantial piece of work will be to implement runahead in its original form (as described in the introduction). This will already require some fluency in using the simulator. I will test the execution of my design using benchmark programs and compare it to how well the baseline design performed.

Key challenge: Understanding runahead in detail.

3.3 Implementing precise runahead execution (PRE)

Runahead comes with some drawbacks itself, the two main ones are:

- Limited prefetch coverage - the number of useful instructions which are fetched in runahead. Any instruction which does not lead to a long latency load can be considered a waste of execution cycles and CPU resources. The original runahead proposal executes all encountered instructions, leaving lots of space for improvement.
- Significant overhead introduced by switching between normal and runahead modes. In runahead instructions pseudo-retire from ROB, hence when the normal mode is resumed, the processor needs to flush and refill the pipeline, re-fetch and re-process all instructions which were in ROB before entering PRE. The incurred cost may be as high as 56 cycles[2]!

My task is to implement PRE, which improves on runahead by:

1. Increasing prefetch coverage by executing only the chains of instructions leading to a stalling load. For this purpose, PRE introduces a fully associative cache - Stalling Slice Table (SST) - which will store all instructions in a backward slice of a long latency operation.
2. Minimizing the overhead when switching from runahead to normal mode. This is achieved by not releasing and flushing the processor's state, instead, PRE uses the available physical registers and issue queue resources.
3. PRE also introduces a novel (at least at that time) method of register reclamation (RRR), which is a way of managing free physical registers.

Key challenge: Understanding the microarchitecture supporting PRE (especially SST, RRR).

3.4 Evaluating runahead and PRE together

The goal is to compare the baseline design, runahead and Precise Runahead Execution in terms of performance and energy consumption. I am expecting to find that PRE achieves an improvement of around 18% in comparison to runahead. Again, I will test on various architectures and benchmarks.

Key challenge: Running simulation on appropriate benchmarks, summarizing results in graphs.

[2] According to the PRE paper: for a 192-entry ROB the performance penalty is 56 clock cycles, assuming we ignore any overhead from saving and restoring the architecture register file: 8 cycles for refilling the pipeline, 48 cycles for refilling ROB and re-dispatching 192 instructions (with dispatch width of 4 instructions).

3.5 Extensions:

3.5.1 Implementing Vector Runahead

PRE is already a great improvement on runahead but, just like anything in computer science, it is imperfect. Modern programs exhibit highly indirect memory access patterns, these cannot be predicted by hardware prefetchers, nor by PRE itself.

For example, lets take code which accesses three arrays: $C[B[A[0]]]$. Access into B is determined by the previous load from array A, and access to array C is determined by previous access to array B. PRE is only able to prefetch the value stored at $A[0]$. On the other hand, PRE enhanced with a hardware prefatcher will go one level deeper - the prefetcher will bring value from $A[0]$ into the cache so that during runahead the value at $B[A[0]]$ will be fetched. But this is still not sufficient! After exiting PRE, the CPU will hit another stalling load - access to array C.

The indirect load instructions on average stall the CPU for over 61% of execution time[3]! Thus Vector Runahead was proposed, to improve on the following constraints of PRE:

- Inability to prefetch all loads in a chain of dependent loads.
- Limited rate of issuing prefetches while in runahead.
- Significant use of processor back-end resources.

My first, and very substantial extension idea is to implement Vector Runahead, which improves on the PRE method by:

1. Remaining in runahead mode until all loads along a chain of dependant instructions are speculatively executed.
2. Vectorizing the instruction stream to increase the fetch/decode bandwidth (e.g. assume our $C[B[A[0]]]$ operation happens to be within a for loop, VR will issue multiple loop iterations simultaneously). Additionally, one vector instruction will occupy only one issue queue entry, as opposed to many entries occupied in the scalar equivalent. This leads to huge savings in the back-end resources required.
3. Introducing vector unrolling - sequentially issuing multiple runs of vector runahead, which effectively brings more future data accesses into the cache.
4. Introducing vector pipelining - a method that reorders independent load instructions so that multiple memory accesses can be issued simultaneously.

3.5.2 Implementing an extended micro-op queue (EMQ) for PRE

EMQ is an optional component of PRE, which allows the processor to store all instructions decoded in runahead mode, and reuse them after resuming normal execution. It would be interesting to compare how well PRE performs with various sizes of EMQ, what are the gains (or perhaps losses?) in performance and energy consumption.

4 Success Criteria

The project is a success if the aforementioned key components have been delivered. To summarise:

- The simulation environment is set up, a baseline configuration runs on all benchmarks.
- Runahead is implemented as described in the source paper.
- Precise Runahead Execution is implemented (without the EMQ extension).
- Runahead and PRE are compared against the baseline, evaluated in terms of CPI and energy consumption, and the results are summarised in a graphical form.

[3] As indicated in the research paper proposing Vector Runahead

5 Evaluation

As stated above, the metrics for evaluation will be the number of CPU clock cycles per instruction and energy consumption. The simulation environment is gem5 - a platform for computer architecture research, which allows modelling of the entire system, as well as evaluation on various CPU models.

For my benchmarks I intend to use: *The HPC Challenge (HPCC) benchmark suite* [4] and *The NAS Parallel benchmarks* [5].

The HPCC benchmarks consist of 7 tests and measure a range of memory access patterns[4].
The NAS benchmarks were developed in order to evaluate the performance of highly parallel supercomputers[5]. They contain memory access patterns which will make the benefits of Vector Runahead more apparent since this technique benefits from parallel execution. Furthermore, this suit was also used to evaluate the original Vector Runahead proposition, hence it should be easier for me to verify whether my implementation is performing as expected.
Were I to discover that more benchmarks are needed, I can additionally use SPEC CPU benchmarks.

6 Work Plan

I have divided the time until the dissertation submission deadline into 15 two-week sprints of work (some of them contain holidays). I have also left time for work on my unit of assessment - Introduction to robotics (IRO). Unfortunately, it is difficult to predict dates for the unit taking place in Lent term.
The schedule for the Michaelmas term can be a bit more packed, as I intend to choose more courses in Lent.

1. **Sprint I: Setting up the simulation & IRO 1st assignment. 18.10 - 31.10 (MT)**
   - This is a preparatory sprint, I intend to set up gem5 for the project, download and run benchmark suites and start preparing the baseline design.
   - The second half of the sprint is devoted to my unit of assessment.
2. **Sprint II: Baseline & start implementing runahead. 1.11 - 14.11 (MT)**
   - The baseline should be finished and tested on the benchmarks.
   - Implement entering into the runahead mode.
3. **Sprint III: Runahead & IRO 2nd assignment. 15.11 - 28.11 (MT)**
   - The first week will be focused on finishing my assignment. I don't plan to code then, just read again through the runahead paper and make sure I understand it thoroughly.
   - In the second week I aim to implement the execution in runahead mode, as well as exiting from it and resuming normal mode.
4. **Sprint IV: Test runahead & begin implementing PRE. 29.11 - 12.12 (MT)**
   - In the first week I'll test runahead on my benchmarks and reread the PRE paper.
   - The second half of this sprint is a week outside of term, so I aim to implement entering into PRE and identifying stalling slices.
5. **Sprint V: Execution in PRE & Christmas Break. 13.12 - 26.12**
   - I aim to implement execution in PRE and the Runahead Register Reclamation (RRR).
   - The end of this sprint is allocated to celebrate Christmas peacefully.
6. **Sprint VI: Christmas Break & Evaluation. 27.12 - 9.01**
   - Some more time left for meeting with family, after Christmas, I will implement exiting PRE, test it, and evaluate against runahead and the baseline.

[4]Details of the tests can be found here: `http://icl.cs.utk.edu/hpcc/`
[5]Source for NAS: `https://www.nas.nasa.gov/software/npb.html`

7. **Sprint VII: Buffer week & Extension: Vector Runahead. 10.01 - 23.01**
    - This sprint leaves me some time to make sure everything so far is running properly.
    - Reread the Vector Runahead paper and implement detecting dependant loads chains.
8. **Sprint VIII: Extension: Vector Runahead & Progress Report 24.01 - 6.02 (LT)**
    - Implement vectorizing the instruction stream.
    - Submit my progress report (deadline on the 4th Feb).
9. **Sprint IX: Extension: Vector Runahead. 7.02 - 20.02 (LT)**
    - Implement vector unrolling.
10. **Sprint X: Extension: Vector Runahead. 21.02 - 6.03 (LT)**
    - Implement vector pipelining.
11. **Sprint XI: Evaluation of all runahead methods! 7.03 - 20.03 (LT)**
    - Compare all implementations, produce plots to be later used in the dissertation.
12. **Sprint XII: Dissertation draft. 21.03 - 3.04**
    - Write Introduction and Preparation chapters.
    - Draft the Implementation chapter.
13. **Sprint XIII: Dissertation write up & Easter break. 4.04 - 17.04**
    - Finish the Implementation chapter.
    - Celebrate Easter holidays.
14. **Sprint XIV: Dissertation finished. 18.04 - 1.05**
    - Finish Evaluation & Conclusion chapters.
    - Polish up and incorporate all comments from the supervisor.
15. **Sprint XV: Dissertation submission. 2.05 - 13.05**
    - Putting the finishing touches and submitting the dissertation.

7 Resources declaration

I plan to use my own laptop, I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. In case of failure, my contingency plan is to use the MCS machines or another personal laptop.

I am going to commit my code to a GitHub repository regularly (after any significant change or after a day of work). This will serve both as a version control tool as well as a backup. Furthermore, I have initialised cloud storage - pCloud - which will back up all my files, automatically saving updates and allowing recovery of old files for 1 year.

I am planning to write my dissertation in LaTeXusing overleaf.com, which auto-saves the files and provides an easy way of collaboration. On top of that, I will download the dissertation source file periodically, so that it is also backed up by pCloud.

In case my laptop turns out to be too slow to run the simulation my supervisor - Dr Timothy Jones - has given me permission to use his computing resources, and he will grant me access to *Sofia*.

8 Bibliography

[1] Onur Mutlu et al. “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.* (2003), pp. 129–140.

[2] Ajeya Naithani et al. “Precise Runahead Execution”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), pp. 397–410.

[3] Ajeya Naithani et al. “Vector Runahead”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), pp. 195–208.

[4] Piotr Luszczek et al. “The HPC Challenge (HPCC) benchmark suite”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (2006).

[5] David H. Bailey et al. “The NAS parallel benchmarks summary and preliminary results”. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing ’91)* (1991), pp. 158–165.