CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

Analysis of Full Window Stalls using SPEC CPU 2017 and Processor Characterization for Runahead Execution

by

Muhammad Umer Saeed

A thesis submitted in partial fulfillment for the degree of Master of Science

in the

Faculty of Engineering
Department of Electrical Engineering

2022

I dedicate this thesis to my Father, Dr. Muhammad Saeed Akhtar, my Mother, Mrs. Zarqa Saeed as well as my Brothers, Muhmmad Ammar Saeed, Muhammad Anas Saeed and my sister Wajiha Saeed. Finally, I would like to thanks my supervisor and HOD Electrical Engineering CUST for providing all the resources, that were used during the period of this study.

CERTIFICATE OF APPROVAL

Analysis of Full Window Stalls using SPEC CPU 2017 and Processor Characterization for Runahead Execution

by

Muhammad Umer Saeed

(MEE181020)

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External	Dr. Muhammad Najam ul Islam	BU, Islamabad
(b)	Internal	Dr. Muhammad Tahir	CUST, Islamabad
(c)	Supervisor	Dr. Muhammad Ashraf	CUST, Islamabad

Dr. Muhammad Ashraf
Thesis Supervisor
February, 2022

Dr. Noor Muhammad Khan
Head
Dept. of Electrical Engineering
February, 2022

Dr. Imtiaz Ahmad Taj
Dean
Faculty of Engineering
February, 2022

Author's Declaration

I, **Muhammad Umer Saeed** hereby state that my MS thesis titled "**Analysis of Full Window Stalls using SPEC CPU 2017 and Processor Characterization for Runahead Execution**" is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/abroad.

At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my MS Degree.

(Muhammad Umer Saeed)
Registration No: MEE181020

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled "**Analysis of Full Window Stalls using SPEC CPU 2017 and Processor Characterization for Runahead Execution**" is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS Degree, the University reserves the right to withdraw/revoke my MS degree and that HEC and the University have the right to publish my name on the HEC/University website on which names of students are placed who submitted plagiarized work.

(Muhammad Umer Saeed)
Registration No: MEE181020

Acknowledgement

All praise to Allah almighty, the most Gracious, the most Merciful who made the human being super creative, blessed me with knowledge and without his help and blessings I was unable to complete this project. I would like to thanks my supervisor. Dr. Muhammad Ashraf for his guidance, supervision, and wise counsel during the tenure of this research. I'm also grateful to my family members, especially my parents, for their continuous encouragement as well as my brothers and sisters for their ethical support. I am also thankful to all my fellow MS scholars especially for their useful advices and encouragement during my studies and research. I would like to say, special and extraordinary thanks to my friend Mr. Saqib Madni, who always helped me morally at every step and every stage during execution of my Project.

(Muhammad Umer Saeed)

Abstract

Micro-architecture design of a processor plays an important role to meet power, memory latency and execution time requirements of modern applications. Memory latency is one of the major challenges which directly effects the efficiency of the microprocessor. Prefetching is the predictive technique that is used to populate the data in cache from main memory to solve the latency issue. Runahead execution is a prefetching technique that benefits the most where long full window cycles occurs due to last level cache misses.

Full window stall was previously calculated for SPEC CPU2006 benchmark and runahead execution technique is being used in the application area where stall value is very large. During the evolution of runahead execution from 2006 to 2021, the hardware has been modified in the microprocessor to make this technique more efficient. This modified hardware has increased the design complexity as well as the power consumption. The need of finding the suitability of using runahead execution technique for modern applications has emerged.

SPEC CPU2017 is the successor of SPEC CPU2006 that represents the modern applications i.e., compiler, artificial intelligence, discrete event simulation etc. The algorithm of finding the full window stall has been proposed and validated by regenerating the stall values from SPEC CPU2006 and comparing it with the previous findings. Afterwards, full window stall cycles are calculated using SPEC CPU2017 for almost all the integer benchmarks i.e., *gcc, omnetpp, perlbench, mcf* etc. Sniper simulator is being used and modified to calculate the stall cycles by running the latest SPEC CPU2017 benchmark simulation points. Our results indicated a very high percentage of full window stall cycles in some specific benchmarks like *gcc (67%) and omnetpp (31%)* but mostly the stall cycles are very low for rest of the benchmarks i.e., *perlbench (12%), xz (6%), x264(5%), mcf (4%), leela (2%).* The allocation of resources to a microprocessor is being done based on the calculated full window stall percentage. Based on the characterization, the simulation results shows that if the processor is to be used in the discrete event simulation or compiler based application domains, a runahead enabled processor is

recommended for better performance. On the other hand, if the domain of application resembles mostly with compression, artificial intelligence and combinational optimization then runahead enabled processor will not be the optimal choice.

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
CPI	Clocks Per Instructions
CPU	Central Processing Unit
FP	Floating Point
FIFO	First In First Out
GCC	GNU Compiler Collection
GHB	Global History Buffer
GPU	Graphics Processing Unit
HMC	Hybrid Memory Cube
INV	Invalid
IPC	Instructions per Cycle
INT	Integer
LLC	Load-Line Calibration
ML	Machine Learning
MLP	Memory Level Parallelism
MEM	Memory
OOO	Out-of-Order
ROB	Re-order buffer
RF	Register File
RAT	Register Alias Table
SMT	Simultaneous Multithreading
SPEC	Standard Performance Evaluation Corporation

Chapter 1

Introduction

1.1 Background

The use of computer controlled automated systems is increasing day by day in almost all type of industries around us. The main controlling element of these automated systems remains a small chip known as "Microprocessor". The use of these microprocessors does not remain limited to the industrial control but also surrounding us in our living environments. If we look around ourselves, various types of Microprocessors are present like in our mobile sets, personal computers/Laptops, automated washing machines, kitchen appliances, cars, trains and airplanes etc.

Multidisciplinary natured requirements of the industry from the microprocessors causes an extensive research on these controlling devices to introduce new featured elements of high performance, low power, compactness, resourcefulness, security and other specialized customizations. The research in computer architectures has radically changed the world by offering a range of devices, from simple types of hand-held computing gadgets to scalable supercomputers.

Many high-performance computing centers are now moving to heterogeneous solutions consisting of general purpose CPUs along with streaming accelerators like GPUs and reconfigurable devices. Other heterogeneous architectures like Microprocessors inside reconfigurable logic inside Microprocessors have also gained a

wide range of marketplace and attention from the scientific community.

Main memory latency is the key bottleneck to improve processor performance. Out of order processing tolerates these long latencies, provided that instruction commit will be done in program order to handle precise exceptions, by buffering the instructions in the instruction window. However, instruction window eventually becomes full if top of the instruction window is filled by long-latency instructions resulting in stalling of processor. This is known as full-window stall.

Several researchers have proposed runahead execution technique to upswing processor performance when long latency cache misses are encountered [1], [2], [3], [4], [5], [6], [7], [8]. Whenever a full window stall occurs due to a last level cache miss, the processor saves the execution state i.e., architectural registers, branch history register and return address stack. The processor marks the load instruction that caused the stall as invalid (INV) and its dependent instructions are also removed from the instruction window.

The processor speculatively executes the independent stream of instructions to request maximum data prefetching from the main memory. Once the load instruction that caused the stall is being serviced with the data that was requested from memory controller, the processor exists the runahead mode and starts the normal execution mode.

The performance improvement is demonstrated by comparing the response of processor under full-window stall and during runahead mode. All types of instruction, that is load, store, branch and execute are used in analysis. Simulations are done using out of order processor with 128 reorder buffer (ROB) entries and 2048 entries, so that effect of the reorder buffer size on processor performance can be analyzed for full-window stall, by using SPEC CPU2006 and SPEC CPU2017 benchmark suites.

Benchmarking is mostly use by computer architects to validate designs. For computer architects the Standard Performance Evaluation Corporation (SPEC)'s 5th generation benchmark suite, known as SPEC CPU2006, are the de facto benchmark suites for their processor design analysis [9]. However the newly released SPEC CPU2017 benchmark has gained attention because of its large workloads and instruction mix [10].

TABLE 1.1: Integer VS floating point benchmarks

S. No	Integer benchmarks	Floating-Point benchmarks
1	Integer operations per instruction	Floating point operations per instruction
2	L1 instruction cache misses per instruction	Memory references per instruction
3	Number of branches per instruction	L2 data cache misses per instruction
4	Number of mispredicted branches per instruction	L2 data cache misses per L2 accesses
5	L2 data cache misses per instruction	Data TLB misses per instruction
6	Instruction TLB misses per instruction	L1 data cache misses per instruction

SPEC CPU2017 has remodeled benchmarks of CPU2006 and also added new workloads to meet modern application demands. The coverage area of CPU2017 is much more than CPU2006 in terms of workload design space [11]. These complex workloads are responsible of very high percentage of full-window stall. Instruction mix and memory intensive workload provided in SPEC CPU2017 helps in performance characterization of processor for full-window stall and runahead mode.

1.2 Benchmarks

A benchmark is the testing program that perform the set of well defined operations and it is used To evaluate the performance of a processor architecture. A performance matrix defines how the tested architecture performs a particular benchmark. Standard benchmarks are being used worldwide for comparison of different architectures. We used standard benchmarks i.e., *SPEC CPU2006* and *SPEC CPU2017* to evaluate the performance of processor architectures.

This method is used to not only find the subsets from any benchmark but also to check the similarities and the differences in a suite to get the cluster having similar features as a potential suspect for experiment or study [12]. These benchmark suites are released by *Standard Performance Evaluation Corporation* in 2006 and 2017.

1.2.1 Why Use SPEC06 ?

The SPEC CPU 2006 benchmark suites was the industry standard before SPEC CPU 2017 which was used for the comparison of high performance computers. These are the platform independent benchmarks so that it can be tested on different high end processors by putting the same workload to get the better comparison. There are some examples of different high end processor which are being ranked after using SPEC i.e., AMD Opteron, Intel Xeon, Intel Itanium, SPARC, IBM Power7.

Due to their target independency these benchmarks are very easy to use as it does not require any additional investment in environment apart from the cross compilation. These benchmarks can also be used to investigate the multicore architecture behavior like some cores are faster than others so, some processor with few of faster cores works more efficiently than the processor having more cores but slower. By just having the knowledge processor speed and the cores with it one can't determine the speed of processor. SPEC CPU 2006 benchmark compares the performance of processors on two different basic workload categories as mentioned in Table 1.2.

Table 1.2: SPEC CPU2006 Benchmark Details

S. No	Benchmark	Source Code	Application Area
1	astar	C++	Path Finding Algorithms
2	mcf	C	Combinational Optimization
3	omnetpp	C++	Discrete Event Simulation
4	perlbench	C	Programming Language
5	xalancbmk	C++	XML Processing
6	sjeng	C	Artificial Intelligence: chess
7	hmmer	C	Search Gene Sequence
8	h264ref	C	Video Compression
9	libquantum	C	Physics / Quantum Computing
10	bzip2	C	Compression
11	gcc	C	Compiler
12	gobmk	C	Artificial Intelligence: Go

The first workload category is the one in which the performance of processor is evaluated on the basis of compute-intensive integer mathematical operations which

are mostly used in business applications. Other workload category is compute-intensive floating-point mathematical operations, these type of workloads refers to the applications which are related to the scientific calculations. SPEC CPU 2006 was famous industry level simplistic benchmark to evaluate the performance of various processors with different architecture having same workload.

As SPEC CPU 2006 benchmark has two main categories as described above, for intensive-integer mathematical operations category workloads there are 12 benchmarks named as SPEC int2006 which were used for testing performance. Processor can be evaluated by speed which usually runs on a single core processor or throughput rate which usually checks the performance of the whole processor e.g. speed benchmarks consists of tasks which are independent of each other and the core with faster execution will be able to perform better while on the other hand overall throughput can be calculated by assigning a dependent sets of tasks to whole processor having multiple cores in it.

SPEC CPU 2006 uses some of workloads to compare the processor speed to complete a single task assigned to a single core e.g. SPECint 2006. While some other workloads are used to compare the number of tasks a processor can complete in specified time span e.g. SPECint_rate 2006 benchmark. Contrary to this SPECfp(Floating point benchmarks) is designed which evaluates processor performances with floating point work load. For intensive-floating point mathematical operations, SPECfp 2006 has 17 benchmarks for this category for testing purposes [4], [5].

1.2.2 Why Use SPEC17 ?

SPEC CPU2017 is most popular and modern industry standard benchmark suite released by Standard Performance Evaluation Corporation in 2017 as shown in Figure 1.1. It is also designed for the same purpose as SPEC CPU 2006 to evaluate the performance by testing to stress on a system's processor. The real world scenario applications like artificial intelligence workloads has been added to better analyze the modern processors whether they meet the modern requirements or not.

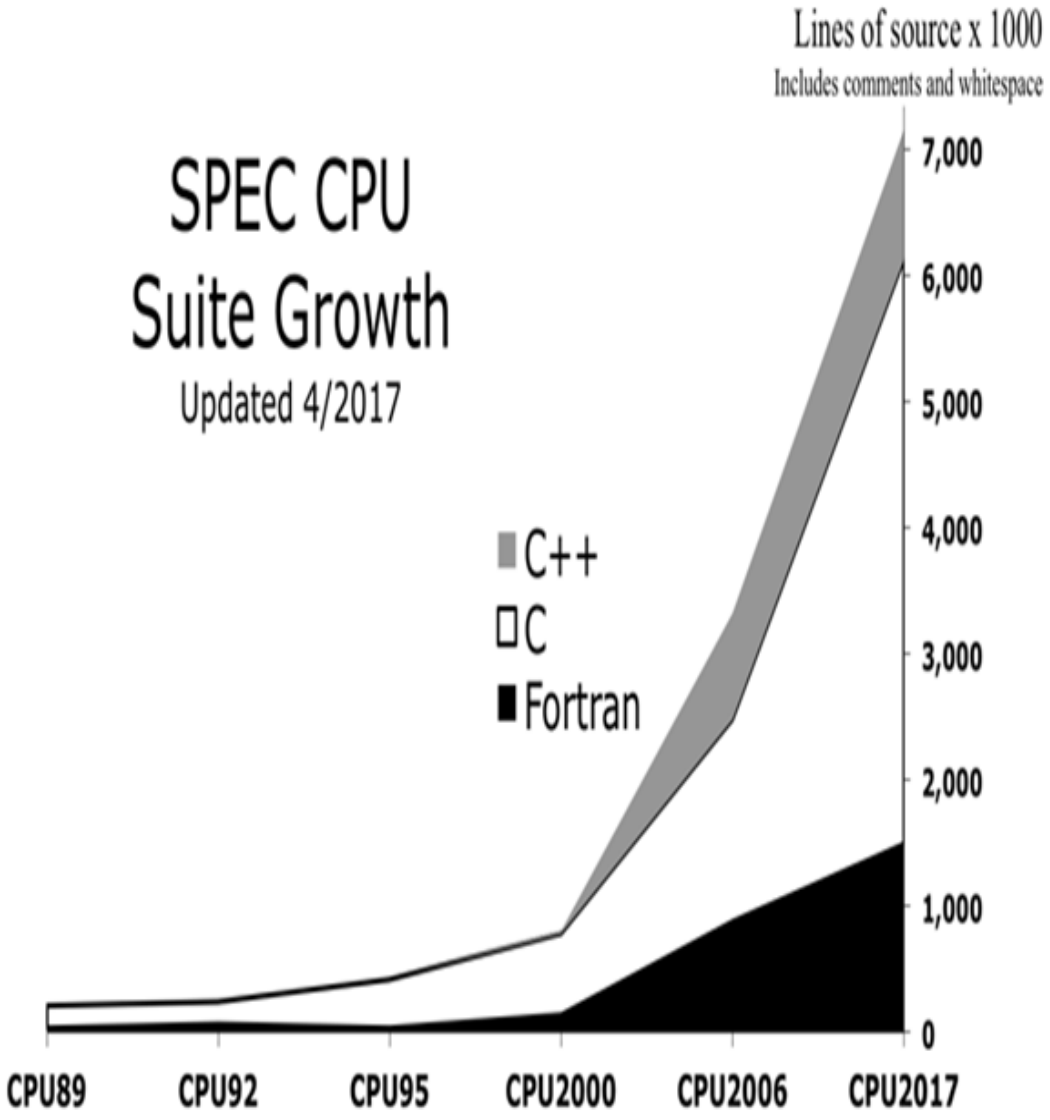

FIGURE 1.1: SPEC Benchmark Evolution

These workloads can be ported to architectures of different designs and can be tested and evaluate them which could not be possible in case of machine dependent suite. Like SPEC2006, this benchmark suite also evaluate performance in different ways using different type of workloads. It also contains the compute intensive integer mathematical operations as well as floating point operations. It is divided into four categories with total of 43 benchmarks as mentioned in Table 1.3. These are SPECspeed 2017 integer, SPECrate 2017 integer, SPECspeed 2017 floating-point and SPECrate 2017 floating-point. SPECspeed 2017 integer are used to calculate the performance of single cores which includes the integer operations. Likewise SPECrate 2017 integer benchmarks are used to calculate the performance

TABLE 1.3: SPEC CPU2017 Benchmark Details

S. No	Benchmark	Source Code	Application Area
1	exchange2	Fortran	Recursive solution generator (Sudoku)
2	gcc	C	GNU C compiler
3	leela	C++	Monte Carlo tree search (Go)
4	mcf	C	Route planning
5	omnetpp	C++	Discrete Event simulation
6	perlbench	C	Perl interpreter
7	x264	C	Video compression
8	xz	C	General data compression

of whole processor which includes multiple cores in it. The same is the case for the rest of the two categories but the operations used in these ones are floating point.

These suites are different because of compilation rules, run rules and memory consumption etc. Tester can run a benchmark in each suite, and this can be done on multiple architecture with multiple cores. SPEC CPU 2017 is considered as major update from SPEC in past 10 years. So, this benchmark allow user to use OpenMP with parallelized architecture to measure the performance on multiple cores and also provide a metrics to calculate power consumption. SPEC2017 has four suites as described above, SPECspeed measure how much time is taken by the processor to complete a task in each suite. While SPECrate demonstrate the number of tasks done in specific time by the particular processor which includes all the cores in it.

1.3 What Is Runahead Execution ?

Memory latency wastes a lot of processor cycles in getting the desired data from main memory in case of last level cache misses. To reduce the memory latency, prefetching techniques are being used that predicts the data which is likely to be used in future is fetched from main memory and place it into cache as shown in Figure 1.3. Out of all the prefetching techniques, one of the most accurate prefetching techniques used for prefetching is runahead execution.

It is a technique which saves the processor stalling issue which is caused by the

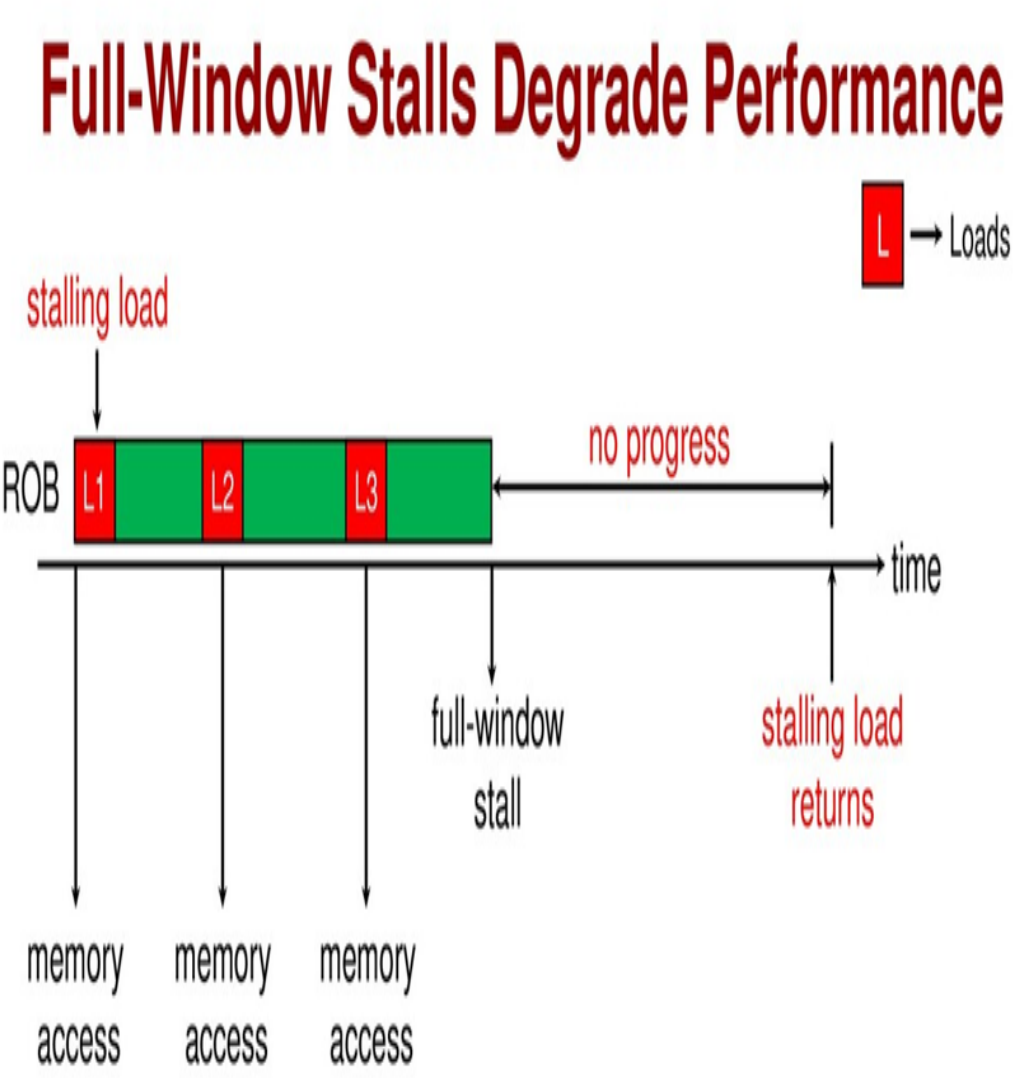

FIGURE 1.2: Full Window Stall Degrades Performance

load instruction miss from last level cache as shown in Figure 1.2. Whenever the instruction windows gets filled by processor stalling state, the processor enters into the runahead mode and execute the next instructions speculatively to execute the load instructions as much as it can, so that maximum requests can be sent to the memory controller for any missing load instruction. In this way, the execution time gets saved from accurate data prefetching before it is even used by the program. It is initially used with in order microprocessor [1] and later on as the processors internals get modified as out-of-order execution introduced this technique is being further modified and used [13].

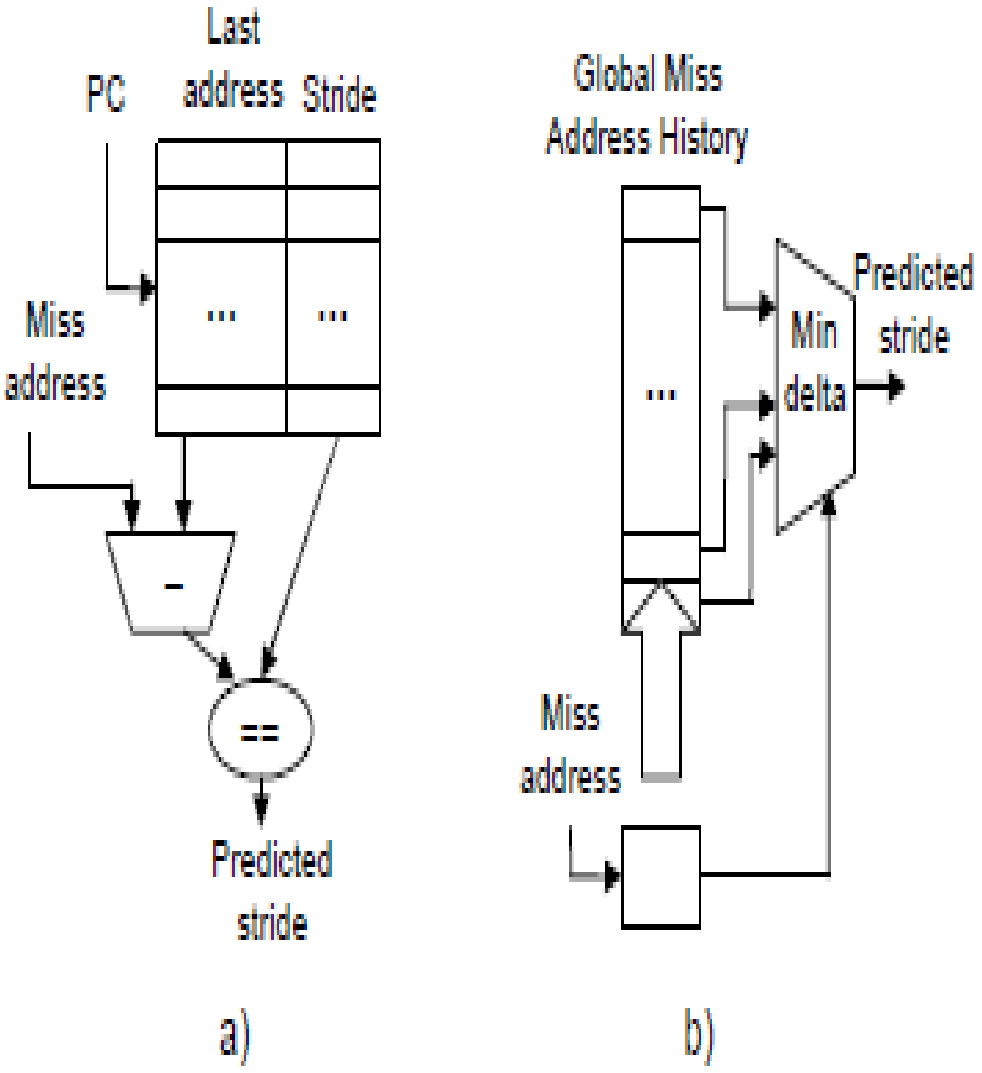

FIGURE 1.3: (a) prefetching (b) cache misses

Several researchers have proposed runahead execution technique to upswing processor performance when long latency cache misses are encountered. The processor follows the long latency queues whenever the stall occurs, to avoid the stalling processor checkpoints the architectural register state, return address stack and branch register and remove the long-latency instruction (which caused the stalling) from instruction window.

All subsequent long latency dependent instructions are identified as invalid (INV) while only independent instructions are buffered in instruction window. Once the instruction which caused the full-window stall is being serviced by the memory

controller the processor exits the runahead mode. The performance improvement is demonstrated by comparing the response of processor under full-window stall and during runahead mode.

1.4 Thesis Objective

This dissertation focuses on the usage of SPEC CPU2006 and SPEC CPU2017 benchmark suites for the investigation of full-window stall. To achieve the aim of the work, the following objectives are outlined:

1. To calculate the full window stall in SPEC 2017 benchmark.
2. To analyse SPEC 2017 benchmark with respect to runahead execution.
3. To perform processor characterization based on Full Window Stall.
4. To propose the usage of runahead execution in application specific processors.

1.5 Research Contribution

This thesis makes the following contributions:

1. To find the full window stall we have proposed the algorithm which calculates the stall cycles when the instruction window is full. We used SPEC CPU2006 and SPEC CPU2017 benchmark suite to investigate full-window stall and compare the two benchmark suite for performance characterization.
2. Sniper simulator does not have the support to calculate full window stall. We have modified the sniper simulator to calculate the full window stall by modifying the source code of the simulator.

3. We have validated the algorithm by calculating the full window stall cycles by running SPEC CPU2006 benchmark and compared the results of full window stall from previous findings of SPEC CPU2006.

4. After the validation of algorithm, we have calculated full window stall cycles by running SPEC CPU2017 benchmark. Instruction mix is calculated for each benchmark to analyze its full window stall characteristics.

5. We have characterized the processor types based on calculated full window stall cycles in the respective application area and gave the suggestion for the usage of runahead execution.

1.6 Thesis Overview

The simulated processor uses ROB of 128 and 2048 entries to verify the statement proposed by [2] that by increasing the instruction window the problem with the stall will solve. The reading were correct and the stall problem gets solved but the solution is too much expensive as it requires a the instruction window to increase which increases the complexity of design, consume more power and the die size of the processor will increase. It proved that the need for runahead technique is more efficient for the processors spending more time in stalling condition.
Simulation environment uses two different cache types i.e., Real Last Level Cache (in which every read/write transactions may hit or miss the last level cache) and Perfect Level Cache (in which every transaction will hit in last level cache). This configuration is needed to further confirm that the stall occurs every time when the last level cache is missed. The experimentation proves that the stall problem gets resolved when the perfect last level cache was used.
Furthermore, the experimentation is done using SPEC CPU 2006 and SPEC CPU 2017 on the above mentioned processor configurations and found out the full window stall percentage for both of the benchmark suites. In this way, it can be found out that whether we still need the runahead technique for the modern workloads or it is not an optimal solution for modern applications. We have selected 8 integer benchmark of CPU2017 among them 4 (*leela*, *exchange2*, *x264* and *xz*) are newly

added [14]. Likewise we selected 12 integer and 1 floating point benchmark from CPU2006.

The results shows that the benchmarks of SPEC CPU 2017 specially GCC and Omnetpp have high runahead opportunities to enhance the performance of the processor as the full window stall percentage is very high i.e., *gcc (67%), omnetpp (31%)*. But for the other benchmarks used for the experimentation has less than 10% full window stall occurrences. It is because the other benchmarks that were being under experimentation are related to compression and artificial intelligence application domains. This shows that the modern application does have an increase need of AI/ML based applications which has the data placed in the memory with sequential locality.

As a result of which, the normal prefetchers like stream prefetcher works well. This is the reason why in modern applications, full window tall occurrences are low. On the other hand, in SPEC CPU 2006 the opportunity of using runahead enabled processors is very high as the full window stall occurrences are very high in these benchmark suites. It is because, it is an old benchmark suites which does not have more AI/ML application domains.

As a conclusion, we have divided the types of processors based on the application usage i.e., we suggest to use runahead enabled processors which will be used for compiler and discrete event simulation application domain. On the other hand, if the application domain is AI/ML based which includes the training of data sets, using runahead enabled processors will not be an optimal choice.

Chapter 2

Literature Review

2.1 Introduction

There is a lot of prior work done in which runahead execution is used to enhance the performance of processors which was being analyzed by running different SPEC benchmarks i.e., SPEC CPU 2000 and SPEC CPU 2006. Enhancing the interval of runahead execution can improve the performance as examined by running SPEC CPU2006 benchmark suite after implementing the technique [8]. Benefit from Memory Level Parallelism (MLP) can be enhanced by storing dependence chain of cache misses and execute the load instructions that is coming in the path of the program [7].

As in the runahead mode, there is a limited time in which maximum load instructions should be executed and cache miss should occur as much as possible which will increase the performance. Reference [15] tackles the short coming of prior work and suggest a novel approach to utilize free processor resources to execute the dependence chain of long latency cache miss operation in runahead mode to avoid the pipeline flush.

In SPEC CPU2006, about *70%* of the time the processor is in full window stall mode [2] also shown in Figure 2.1. Also, with different machine configuration about *47%* of the time the processor spends its time doing nothing [15]. There are a lot of other researchers that calculate the full window stall for SPEC CPU2006

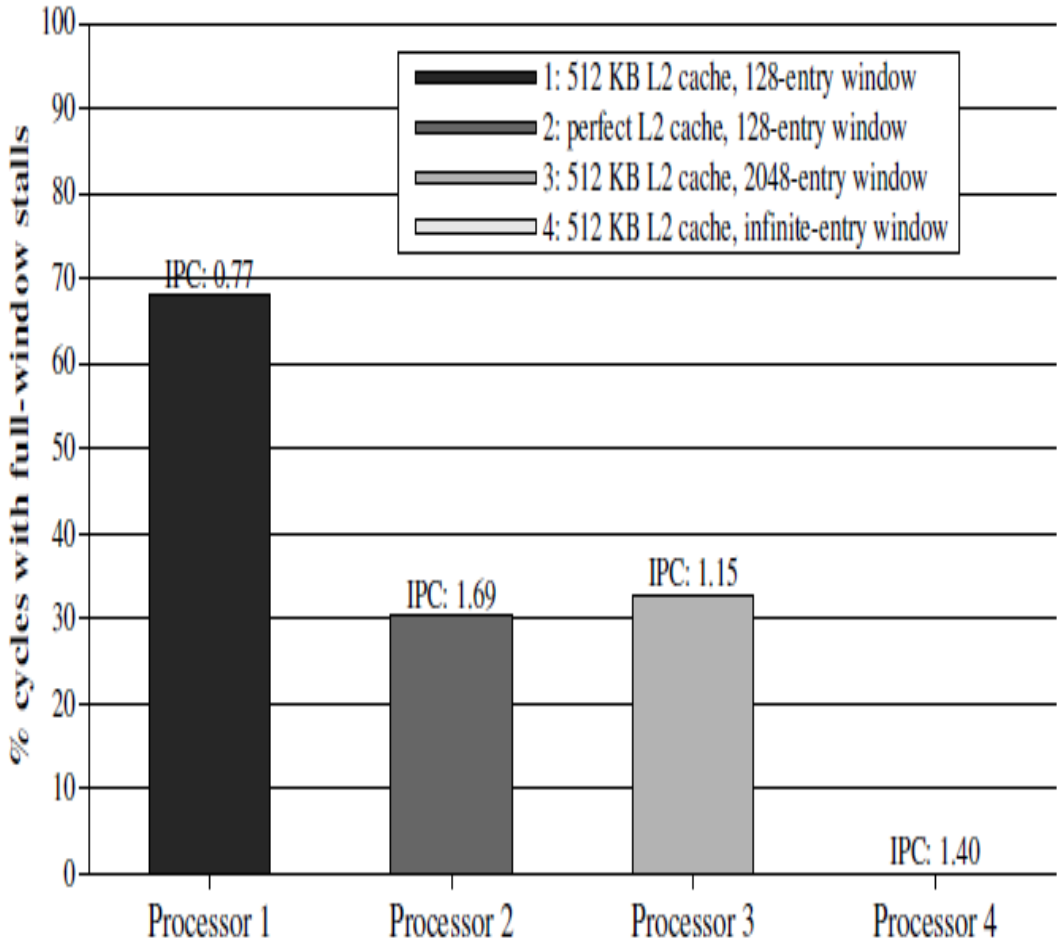

FIGURE 2.1: SPEC06 Full Window Stall

[7] [8] [5] [16]. An initial implementation of runahead execution has been proposed by [2] and later on the runahead interval has been further optimized by [3].

As shown in Figure 2.2 a basic initial architecture has been proposed and then further optimized by the later technique. It has been proposed to use this technique in gcc (compiler) and mcf (combinational optimization) domain. Then [7] proposed the hybrid technique to improve the runahead execution furthermore by calculating and storing the dependence chains and executed them only while in runahead mode. The flow diagram of this hybrid policy can be seen in the Figure 2.6.

Furthermore [8] introduced a novel hardware engine which calculates and stores the stalling loads and this improves the runahead mode usage. The proposed accelerator is shown in figure 2.3. Later on, in 2018 [15] further increased the performance of runahead execution technique by efficiently saving and restoring

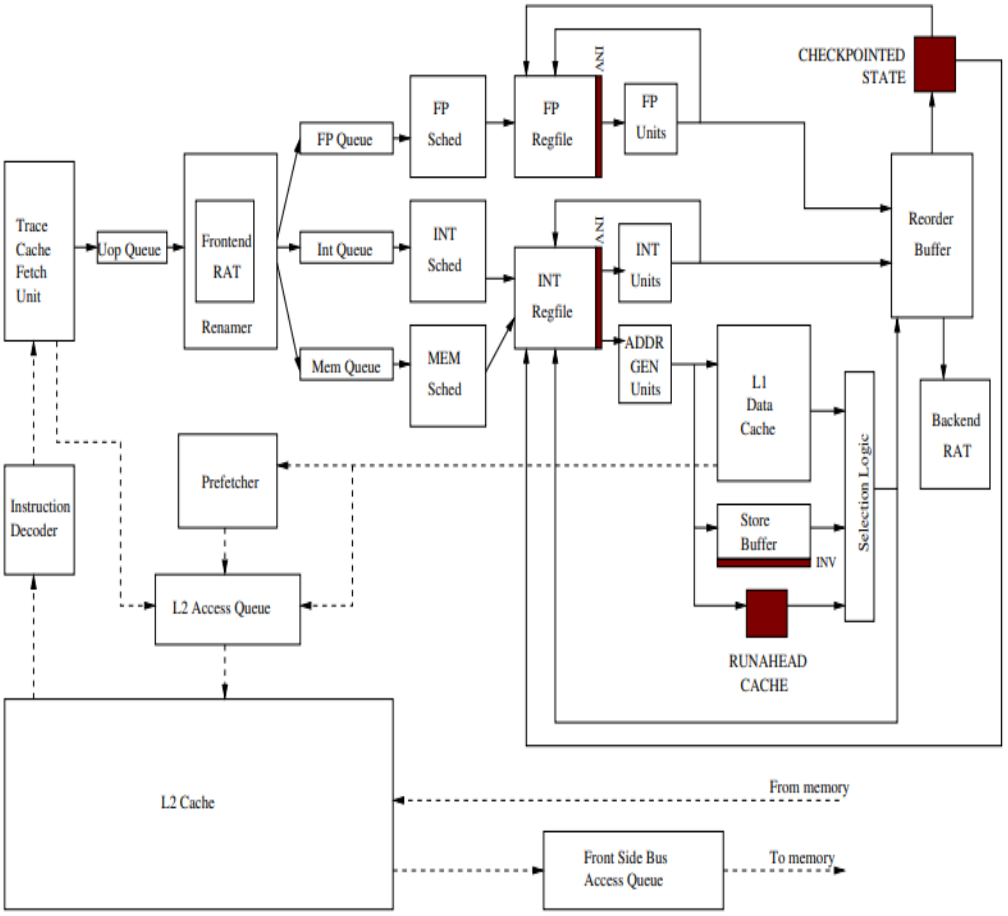

FIGURE 2.2: Initial Runahead Execution Architecture

the pipeline of microprocessor. The block micro-architecture proposed above is shown in Figure 2.4. In 2021, [17] proposed a technique to accumulate all the independent stalling loads into a vector and stall the processor at once so that the prefetching can be done in more accurate way. The proposed micro-architecture block diagram is shown in Figure 2.5.

All of the techniques discussed above has been evolved from 2006 to 2021. This evolution increase the hardware complexities as well as the power consumption and die size. These techniques has been tested and validated on the older benchmarks i.e., SPEC CPU2006 which got retired in 2018 and does not represent modern application needs [10] [11]. These techniques has been used and improved the microprocessor efficiency in the area of mcf (combinational optimization), gcc (compiler), perlbench (programming language), libquantum (quantum computing), omnetpp (discrete event simulation), xalancbmk (xml processing) and bzip2 (compression) of SPEC CPU2006 benchmark.

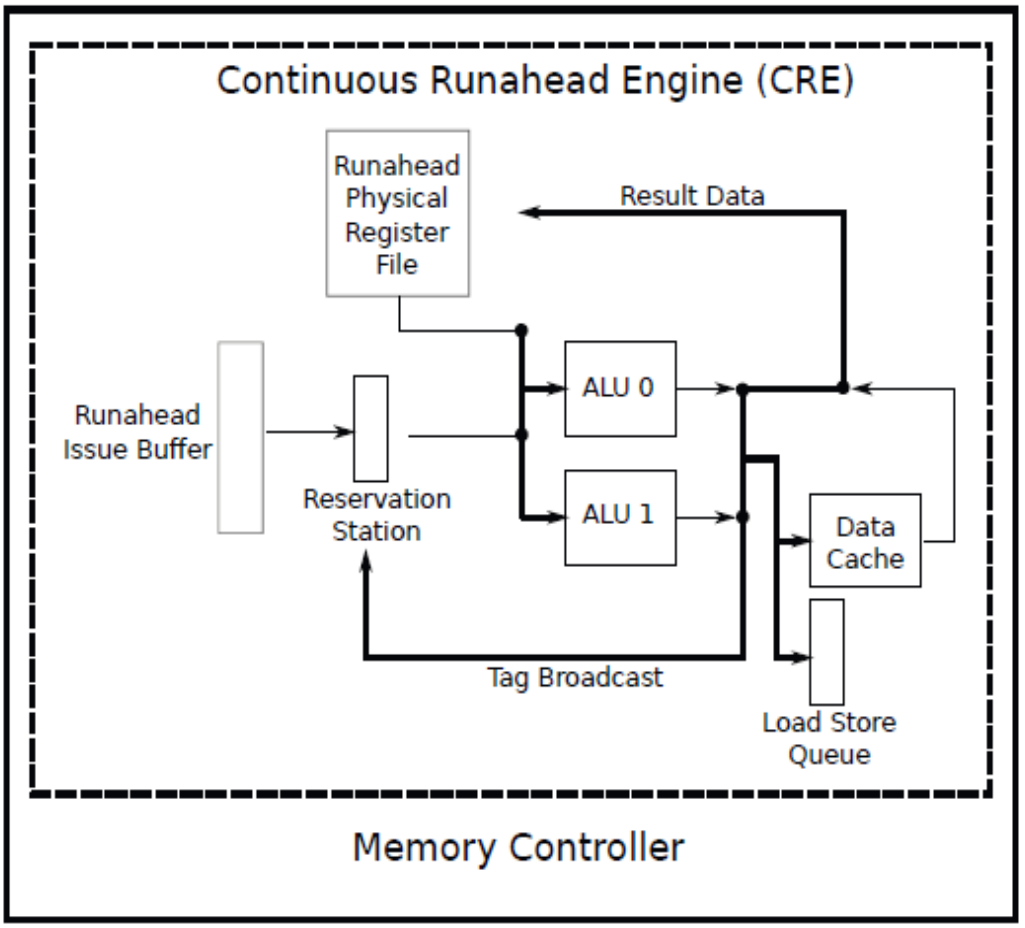

FIGURE 2.3: CRE Proposed Hardware Engine

Previously, runahead execution is evaluated on memory intensive SPEC CPU95, CPU2000 and CPU2006 benchmark suite. The latest variant from SPEC corporation is SPEC CPU2017 benchmark suite that is characterized by the instruction mix, branch prediction and cache memory behavior. The recent release of CPU2017 contains 43 benchmarks with about 10X times more dynamic instruction count than CPU2006, which can greatly increase the simulation time. Mostly subset of benchmark suite is used for analysis to reduce simulation time i.e., Simpoints is used for the experimentation.

As a result of our research, we come to know that is one of the significant attempt to find the full window stall using SPEC CPU2017 suite. Using the revamped workload of CPU2017 and CPU2006 we have furnished a comparison of full-window stall using both benchmarks. We analyze the similarities and difference of CPU2006 and CPU2017 benchmark suites. We explore the opportunities

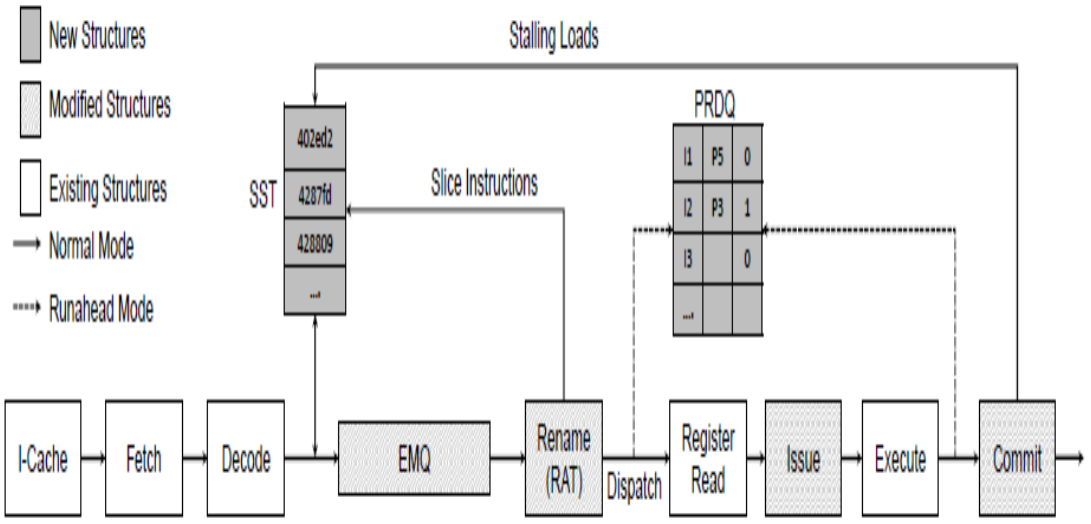

FIGURE 2.4: Precise Runahead MicroArchitecture

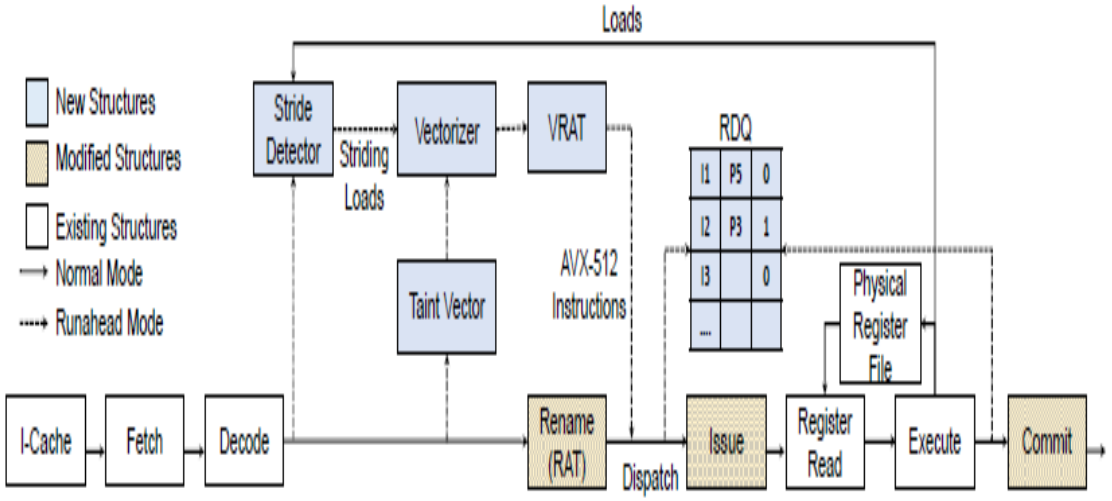

FIGURE 2.5: Vector Runahead Micro-Architecture

of performance improvement using runahead execution.

This is the first work to produce the results of full-window stall using SPEC CPU 2017 benchmark suite. As the SPEC CPU 2017 is being recently released on which there is a gap of improving or analyzing the benchmarking results of processor. For this purpose, full window stall calculation is being done on using this benchmark suite and the opportunity or feasibility of using the runahead technique is being explored. So that, with modern workload the processor design should be optimized to meet the new requirements [13].

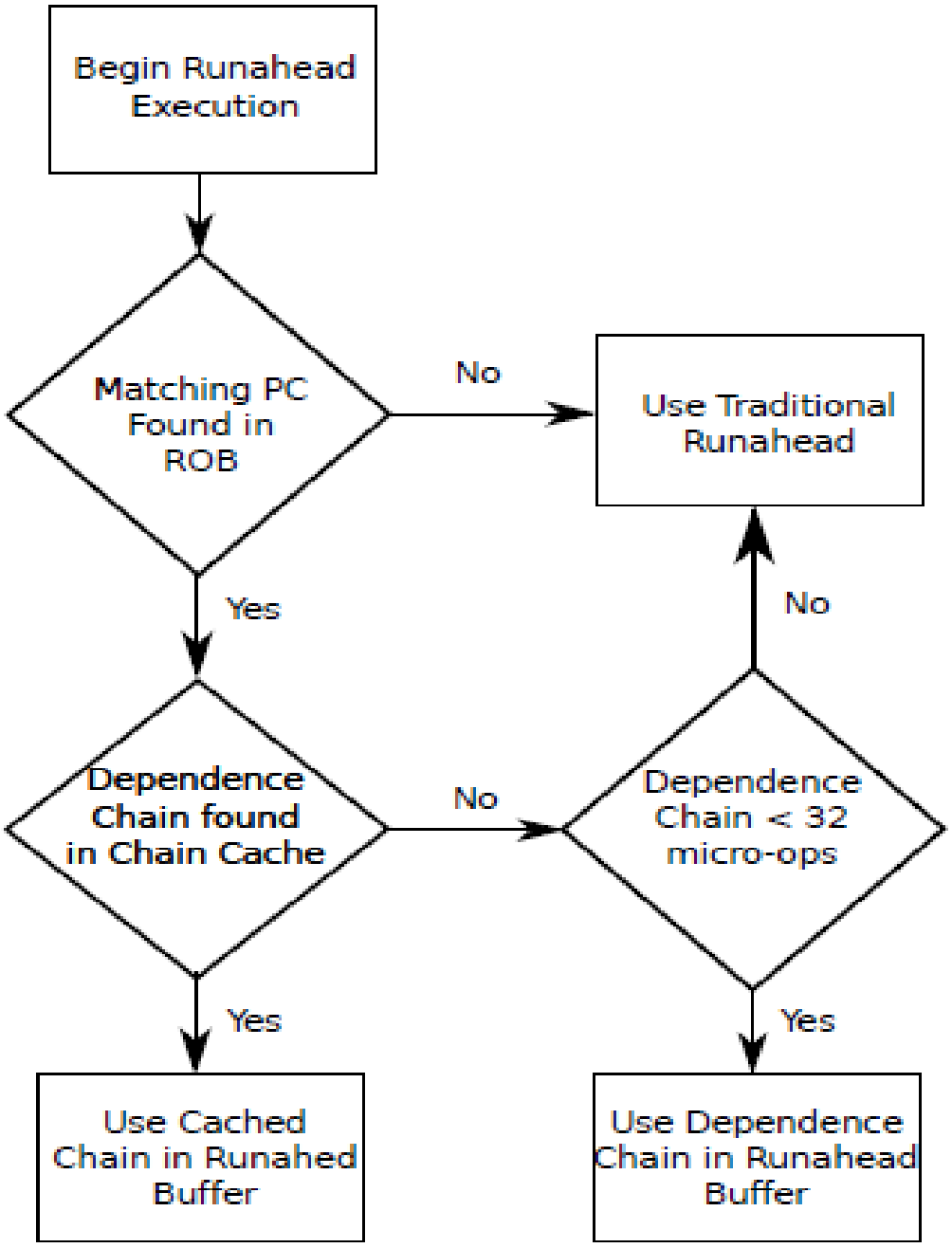

FIGURE 2.6: Hybrid Policy Flow Chart

2.2 Survey of Simulators

Numerous simulators have been created by computer architects over few decades, and the number is expected to keep rising. Simulators are classified according to the amount of detail in the simulation, the scope of the objective, and the input to the simulator.

2.2.1 Functional vs. Timing Simulators

Functional simulators represent simply the target's functioning; they do not replicate microarchitectural details. Simics [18] and SimpleScalar [19] both have 'sim-safe' and'sim-fast' models that support x86.
An accurate simulation of a computer system's timing/performance is provided by a timing simulator (sometimes referred to as an operational simulator). When a benchmark is running on a simulated CPU, cycle-level simulators maintain track of every single clock cycle. As a result of the time and resources required to keep track of so much information, functional simulators and other kinds of timing simulators are faster [20].

2.2.2 Application-Level vs. Full-System Simulators

An Application/User Mode Simulator is able to run just target programs rather than a whole operating system. As a result, they must model a CPU with just a few peripherals. Requests for system services made by simulated applications/benchmarks skip the user-level simulation and instead are handled by the underlying host OS. Because the execution of system-level code takes up so little time in computationally complex benchmarks like SPEC CPU [21], relying only on the simulation of user-level code may not be a concern. Simulators at the application level are often simpler, although they may exhibit errors owing to the absence of system-level code support. Full system simulators, on the other hand, may create a virtual copy of the target's OS. Simulated I/O devices are also used to boot an operating system.

2.2.3 Trace-Driven vs. Execution-Driven Simulators

Trace files are used as inputs in trace-driven simulators. Streams of instructions from a program's execution on actual hardware may be found in these files. Simulators that don't need to imitate the target's ISA are reasonably easy to build. There is a drawback to trace-driven simulators that their produced trace files may

be extremely big in size, and reading these huge files from disc can be sluggish. Utilizing trace sampling and reduction methods, this problem may be remedied [22].

Whereas in an execution based simulator, the simulated machine immediately executes the benchmark's instructions. At the same time as the software is running, several performance-related factors are being analysed. These simulators, as opposed to those that rely on tracing, are capable of simulating erroneous code paths. Due to rapid forwarding of time, they are more difficult to execute than trace-driven simulators and might need more time if statistical sampling is utilised.

2.3 Sniper Simulator

Sniper simulator is being used to simulate the behavior of the processor. Using the simulators makes the testing easy for proposed changes as it does not require fabricating new processors. New changes can be incorporated into the simulator and can be tested by running SPEC benchmarks on it. Once the experimentation will be done and tested, the proposed changes can be actually fabricated to the microprocessor chips.

Mostly simulation is used for high-risk experiments, safety test and scientific exploratory experiments. This simulator also incorporates the 3D visualization of thermal maps for machine under test.With the help of simulation environment the phenomena can be tested before it is being used in real world scenarios. Also by having simulation before the fabrication process of processors also helps find the potential problems which might occur in future.

Sniper is high speed, parallel and accurate x86 next generation simulator [2]. It provides the functionality to conduct simulations for both multicore programming (OpenMP) in multiple core systems. This flexibility allows us to test the benchmarks for single core as well as multicore processors. It gives fast and accurate simulation for same (homogeneous) and different type (heterogeneous) of architectures if compared with existing simulators.

It relies on internal simulation which increases the level of abstraction that allows

faster development and testing or proposed architectural changes and consume less evaluating time by jumping between intervals [23]. Another benefit of using sniper is that it allows the visualization of CPI stacks, which represents the consumption of each cycle and make the tester to be able to better understand the actual performance of each component and to measure the effectiveness of each component on net performance. SPEC CPU2017 benchmark suite is characterized by the instruction mix, branch prediction and cache memory behavior.

It is very difficult and hard to visualize the parallel performance of sniper in insightful way. So, sniper's parallel execution can be visualize by two representations. The first one is CPI stacks and the other is Thread-state timelines. We have used CPI stacks to understand the cycle consumption.

The processor performance can be represented by total number of instructions that specific processor can execute in one clock cycle of processor or it can be expressed by the CPI which means average cycles an instruction may takes. CPI further can be divided into many small components which individually express the performance of its own [24].

2.4 Overview of CPU Benchmarks

Benchmarks facilitate comparisons across various CPUs by evaluating their performance against a defined set of tests, and they are beneficial in a variety of situations such as when purchasing or constructing a new computer.

Benchmarks developed in the past have been unable to adequately describe the performance of current computer systems. Several of those benchmarks quantify component performance, while others are commonly presented as system performance [25].

2.4.1 Simple CPU Benchmarks

Simple CPU benchmarks are classified into three categories: kernel, synthetic, and application [26].

1. Kernel benchmarks are based on study and the understanding that in the majority of situations, 10% of code consumes 80% of CPU resources. Performance investigators have obtained and exploited these code parts as benchmarks.

2. Synthetic benchmarks are created based on the expertise and understanding of performance experts about instruction mix. Dhrystone and Whetstone are two examples of synthetic benchmarks. Certain previous assumptions about instruction mix are rendered outdated by new technology and instruction set designs. Synthetic benchmarks present challenges that are analogous to those seen in kernel benchmarks.

3. The best benchmark, from the user's viewpoint, is the user's own application software. Spice (for circuit designers) and the GNU compiler are two examples of application benchmarks (software developers using GNU environments). Regrettably, thousands of programmes exist, and the most of them are proprietary. A benchmark suite with a large number of programmes is likewise impracticable due to porting and assessment challenges, as well as the lengthy runtime.

2.4.2 Aging CPU Benchmarks

Aging CPU benchmarks are classified into three categories: Dhrystone, Linpack, and Whetstone [27] [28].

1. Reinhold Weicker invented Dhrystone in 1984. This synthetic benchmark devotes a substantial amount of time to string functions. It was created to evaluate the integer performance of basic architectures on tiny devices. RISC processors often outperform CISC machines on this test because to the high number of registers and the locations of code and data on RISC machines. Dhrystones per second is the performance statistic.

2. Linpack is a set of linear algebra subroutines first published in 1976 by Jack Dongarra. It is used to quantify a machine's floating-point performance.

This benchmark utilises a matrix of size 100x100, which was considered a big matrix in 1976. The application is small enough to fit easily in the caches of several computers. Millions of floating-point operations per second define its performance (MFLOPS). Single- and double-precision MFLOPS are used to quantify performance.

3. Whetstone is a well-known synthetic floating-point benchmark that was created in 1976 by H. J. Curnow and B. A. Wichman. It consists of eight modules that carry out various numerical calculations (e.g., arrays, trigonometric functions). The benchmark is compact and highly dependent on the sequence in which library modules are loaded and the amount of caches. Single- and double-precision Whetstones per second are used to quantify performance.

2.4.3 Overview of SPEC Benchmark Suites

The performance of current computer systems cannot be accurately measured using conventional benchmarks. For example, some benchmarks assess component-level performance, while others measure overall performance. For a long time, suppliers have used a wide range of ambiguous metrics to describe the performance of their systems. Confusion is exacerbated by a scarcity of reliable data on performance, a lack of consensus among competing providers, and a lack of clear leadership [29].

In October 1988, Apollo, MIPS Computer Systems, Hewlett-Packard, and SUN Microsystems teamed together with E. E. Times to create the Standard Performance Evaluation Corporation (SPEC). SPEC develops, manages, distributes, and approves a standardised collection of application-oriented programmes for benchmarking purposes [25].

The performance of a computer system cannot be quantified in terms of a single metric or benchmark. Using a single benchmark to characterise a system's performance is analogous to the proverbial blind man describing an elephant. However, many users (decision makers) want a single-number assessment of performance. The client is confronted with a bevvy of perplexing performance data and the

press's unwillingness to share detailed performance and configuration details. No question has a straightforward answer. However, both the press and the consumer must be educated of the danger and foolishness of depending on a single performance metric or a single benchmark.

Customers have been unable to assess and compare rival systems because computer manufacturers could not or would not agree on a consistent set of benchmarks. Vendors, software developers, and system designers have all had difficulties due to the absence of benchmarking standards. When comparing benchmarking results, there is no such thing as an absolute truth.

These problems were first addressed by SPEC through the selection and development of real application benchmarks that put major system components to the test. Do we require only one benchmark or are there a number of other ones that we may use? What kind of workload is optimal for showcasing a certain system? What is system performance, and how can we objectively assess the performance of various computer systems?

Many different techniques, architectures, implementations, memory units, I/O subsystems and operating systems were compared in order to see how they affected system performance as well as clock rates and bus protocols. With many CPUs and peripherals, additional issues had to be taken into account. Other aspects, like as graphics and networking, exacerbated the problem.

The performance of raw hardware is determined by a number of elements: CPUs, floating-point units (FPUs), I/O, network and graphics accelerators, memory and peripheral systems. All benchmarks had to be converted to all SPEC members' computers in order to ensure that the same source code (machine-independent) would execute on all SPEC members' workstations. This proved to be a difficult undertaking.

Conflicts about portability are handled during SPEC bench-a-thons by SPEC members. Engineers from SPEC member firms participate in a five-day bench-a-thon to create benchmarks and tools that can be used on any operating system or platform. To conduct the test, SPEC used a simple metric known as elapsed time. It was only via the use of machine-independent code and a basic speed measure that we were able to conduct an accurate comparison of rival computers

[30]. Till date, SPEC has introduced number of benchmarks before reaching to the latest benchmark (i.e., SPEC CPU 2017). The SPEC CPU benchmark suites are summarised in Table 2.1.

TABLE 2.1: SPEC CPU Benchmarks

Current Benchmarks	
SPEC CPU 2017	The most recent version of SPEC's widely used CPU performance tests. It was published in June 2017.
Retired Benchmarks	
SPEC CPU 2006	The predecessor of the current series of CPU performance testing.
SPEC CPU 2000	In February 2007, it was retired as a frequently used CPU performance test.
SPEC CPU 95	CPU 2000 was released in December 1999, rendering it obsolete.
SPEC CPU 92	CPU 95 was released in August 1995, and SPEC CPU 92 immediately became obsolete.
SPEC CPU 89	Successor to the CPU 92 suites, which are now extinct.

2.5 Memory Latency

Memory latency is the amount of time it takes for a processor to retrieve a requested byte or word from memory. Interferences between requests coming from different cores may increase the Latency of Memory accesses, affecting the overall system throughput. To address the issue, many techniques for reducing latency are available, which are mentioned below.

2.5.1 Reducing Latency By Prefetching

Generally, the distribution of hardware prefetching is divided into two categories. Prefetchers as shown in Figure 2.7 that indicate bases of future address that is based on memory access patterns and other ones are the prefetchers that depend on pre-execution of code segments provide by the application. So, the second type of prefetchers are discussed here that uncover steams [14] that require a small hardware which can increase the cost and complexity of the hardware. To decrease the data access latency for show-able data access patterns, pre-fetchers are

used.

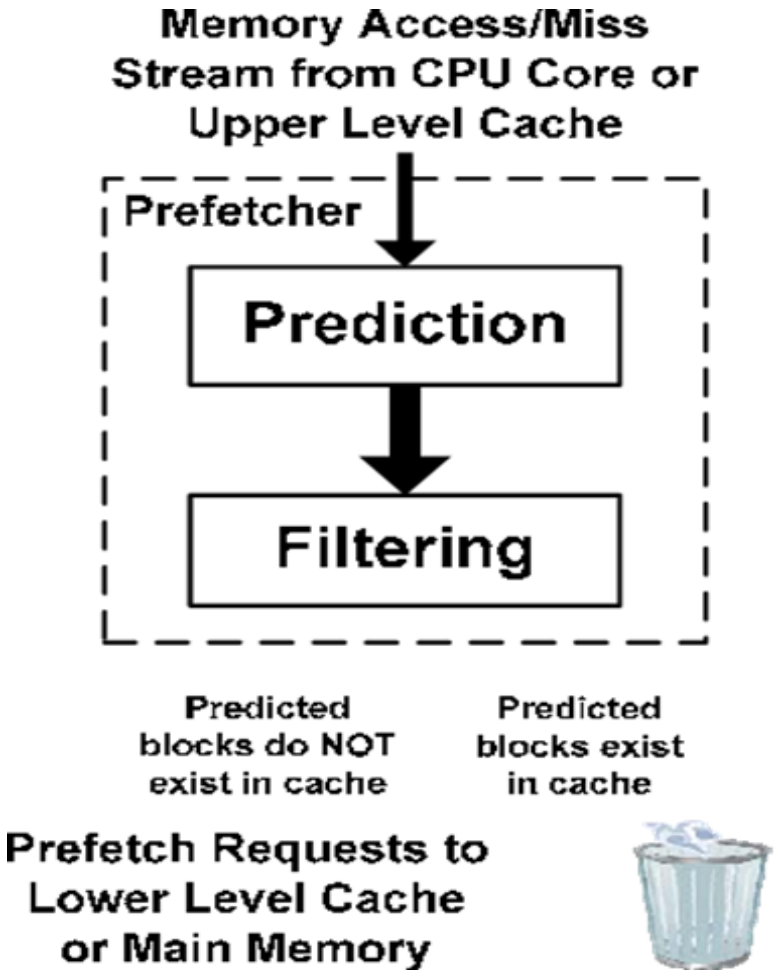

FIGURE 2.7: A Simple Prefetcher Concept

The addition of pre-fetchers cannot be used to handle complex access patterns which leads to the inaccurate pre-fetch requests and results in wastage of memory bandwidth and cache pollution. There is also the need of advance hardware pre-fetchers techniques like correlation pre-fetching aims to decrease the average memory accesses level for more unpredictable misses [14] [4]. These pre-fetchers are used to maintain work at large on clip tables that are used to relate past cache miss addresses in the future cache miss addresses.

The two-level indexing scheme is used to decrease the need of correlation tables and for that purpose global history buffer (GHB) was formed. For decreasing the need of on chip storage some pre-fetching techniques are used [14]. These proposals are used to incur additional cost of transmitting meta data over the memory bus. The focus of the technique is to increase the on chip mechanisms to decrease

memory access latency. On the other hand,, hardware mechanisms target that points that is useful to lead cache misses. During the traverse of linked data structure, jump pointer is used to create the memory level parallelism [14].

They also identified the difference between pointer on the basis of stable dependence patterns and store this information in correlation table. On another side content directed pre-fetching does not need any store pointer of additional state, it can work through pre-fetching de-referencing values that could be memorized addresses.

In that case all the cache miss addresses are not easily predicted by pre-fetching and the word dependent on the accelerating dependent target addresses are difficult to pre-fetch. According to this research at creation of independent cache misses, pointers are dynamically used for the application code to pre-fetch.

2.5.2 Reducing Latency By Pre-Execution

The backend of processor is stalled because of reorder buffer in runahead Execution [14]. In this state, the backend is stalled and the frontend continues to do its part i.e., fetch the instructions. This is because, the speculation execution of the future instructions will save the cache misses in the future i.e., memory level parallelism will be more utilized in this case. Traditional runahead Execution always needs front end to fetch the instructions which are likely to not create the cache misses in the future.

It means that, when the processor enters in runahead mode it speculatively executes the instructions (including all the instructions which are not useful for the future execution). This consumes processor power and resources. It is showed that the memories can be stored in a separate buffer by solving the dependence chain in which case the processor will execute the instructions that are needed to be executed.

This allowed to create EMC all times from the continues work of runahead but it does not only work when the core is stalled. Pre-execution is done via compiler or hand generated code segments. All the memory pages are attempted according to pre-fetch by the using of compiler or hand-tuned portions of code to create the

demand access stream [5] [14].Special hardware is used to save these threads or on a different core of a multi-core processor.

Compiler is required to free the hardware thread which will be used to analyze and create helper thread contexts or execute them. In other words, helper threads can also be created through manual work. To generate helper Threads Kim and Yeung also discussed the similar concepts proposed in techniques for the static code compilation of Executor schemes, whereas the computation loop is preceded by an inspector loop that is used to prefetch data.

The techniques of dynamic compilation and hand-generated helper code have also been designed to run on idle-cores of a multi-core processor. These all are based on the idea of statically generated pre-execution proposals of decoupling the memory access stream from the application of execution stream [3].

According to these ideas and methods these mechanisms are proposed that allow the dynamic generation of dependence chains. With this chain we do not need resources like free hardware cores or free thread-contexts.

Memory controller will be preferred to contain the specialized functionality and it required to execute these dependence chains.

Speculation via automatically generated helper threads is limited. It also needs to execute from the main-thread. This work is done by using a proper version of the main-thread by the help of this version the helper-thread can run faster than the main-thread so there is no need of informative files, that is why they are removed. First, there are two processors that are used to execute an application slipstream. This process works through the related version of the application ahead of the R-stream. After that, A-stream is used to communicate performance hints such as branch-directions or memory addresses for prefect tolerance.

However, these instructions that are being removed from slipstream are generally simple. Slipstream only removes unusual writes like the stores that are never referenced and stores that do not modify the state of a location. For the completion of other work, they have used a similar two-processor architecture but it does not allow the A-stream to stall on cache misses.

Second, Collins et al. proposes a dynamic scheme to automatically extract helper threads from the back-end of a processor. To do so, they require large additional

hardware structures including a buffer that is twice the size of all retired operations that are through this buffer. Once the helper threads are generated, they must run on full SMT thread contexts. This requires the front-end to fetch and decode operations and the SMT thread contends with the main thread for resources. An 8-way SMT core is used in their evaluation [8].

Third, Annavaram et al. [14] add hardware to extract a dependent chain of operations that are likely to result in a cache miss from the front-end during decode. These operations are prioritized and execute on a separate back-end. This reduces the effects of pipeline contention on these operations, but limits runahead distance to operations that the processor has already fetched. They also proposed a lightweight solution to dynamically create a dependence of it.

2.5.3 Reducing Latency By Near Memory Computation

The data through pelagic and memory fabricated is also used to same process as the data used in rechecking of reducing data. The proposed performance computation inside the layer of logic word as a 3d stacked ram but none of them is specialized to accelerate as target depends on cache misses. The proposal is again vested by Micron's 3D-stacked Hybrid Memory Cube (HMC) to propose performing graph processing in an interconnected network of HMCs by changing the programming model and architecture forfeiting cache coherence and virtual memory mechanisms [14].

The data elements for prefetching are further to decide clip to build the large prefetching correlation for the memory if using the memory side logic proposed by Alexander et al. and Solihin. This is generally known as split computation for the one hand chip and other chip is allocated to compute, due to which the cost of data and data across the ram bus increased. This memory bus has comparatively smaller access latency as compared to dram access latency. The computation as shown in Figure 2.8 is located as first point as the data entry chip and also the memory controller is an attractive and unexplored research area in this direction. The prior work has been proposed automatically by combining the arithmetic loads by making and transferring the data as well as migrating same proposal closer to

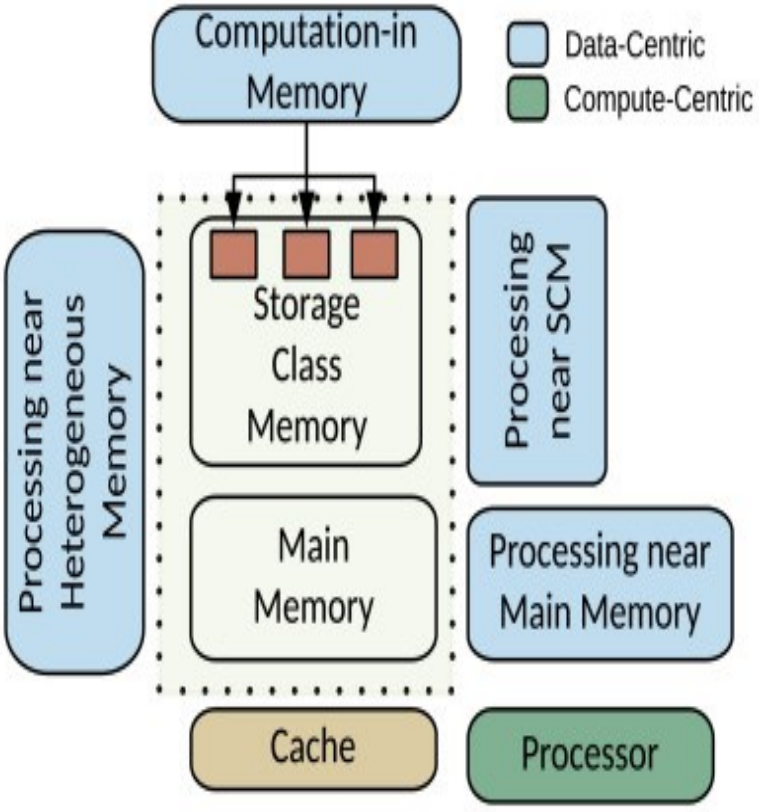

FIGURE 2.8: A Simple Near Memory Computation Model

the on chip cache. Whereas the data is safe to use for migration which reduces main memory access but not the cache access latency.

2.5.4 Reducing Latency By Memory Scheduling

According to the order of memory requests services due to the dram row buffer bank contention there is a large effect on the latency of memory, due to this reason it may affect on bank contention buffer that purpose the optimization of row buffer hit rate and data should be bank mapped and also orthogonal to the top memory scheduling according to their research as shown in Figure 2.8.

2.6 Research Gap

Table 2.2 provides the tabular overview of literature work on the implementation of runahead execution along with the benchmarks used.

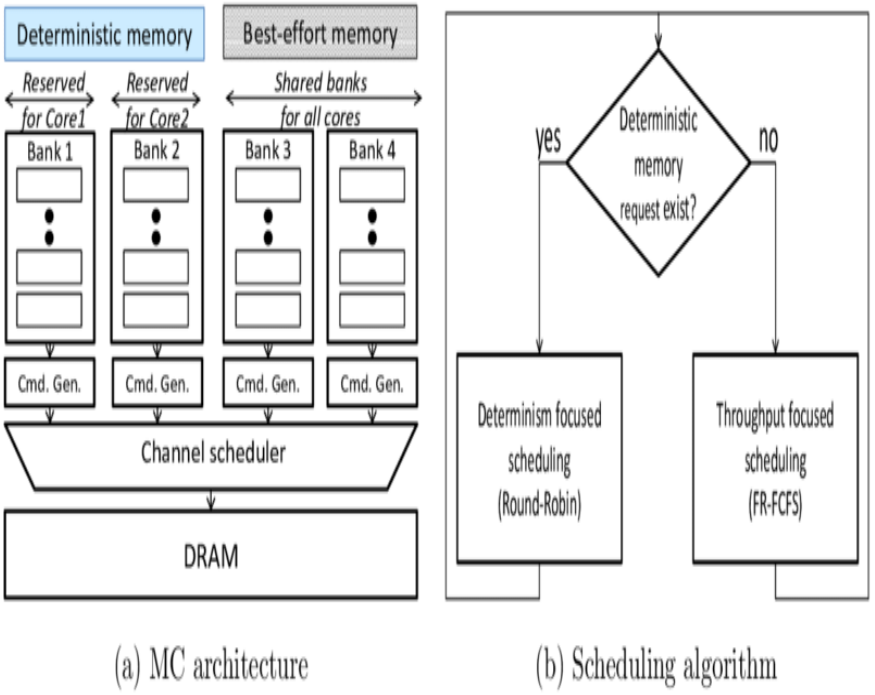

FIGURE 2.9: Memory Scheduling In RAM Controller

The limitation of the work (mentioned in Table 2.2) regarding the runahead execution implementation is that, it has been implemented only on SPEC CPU 2006. For Recent benchmark, such as SPEC CPU 2017, there is a need to determine the usage of runahead execution in order to improve performance by computing complete window stall cycles. As a result, Full Window stall has to be computed for the current applications and the processor must be characterized based on the latest benchmark values.

2.7 Problem Statement

The strategies mentioned in previous section for improving latency are capable of providing outstanding results, but it is every important to keep in mind that they have a big downside (i.e., substantial resource or processor consumption). Understanding the importance of resource utilization, there is room to come up with a method that mitigates the stated limitation. Recognizing the significance of optimizing the use of available resources, it is necessary to devise a solution (i.e.,

TABLE 2.2: Runahead execution based related studies

Year	Work	Benchmark and Description
2006	Initial Implementation By Onur Mutlu [2]	Initial Implementation of runahead execution. Full Window Stall has been calculated on SPEC CPU 2006.
2006	Efficient Runahead Execution By Onur Mutlu [31]	Efficiently utilized the runahead period when processor is running in runahead mode by disabling FPU and using prefetchers as well. The processor performance has been categorized based on SPEC CPU 2006 IPC values.
2015	Filtered Runahead Execution By Milad Hashemi [7]	Runahead execution is optimized by adding runahead buffer and storing dependence chain of load in it and executing it. The processor performance has been calculated based on SPEC CPU 2006 IPC values.
2016	Continuous Runahead By Milad Hashemi [8]	Continuous runahead engine is used to remove the short runahead interval by continuously executing the cache miss chain. The IPC on SPEC CPU 2006 has been improved.
2018	Precise Runahead Execution By Ajeya Naithani [15]	Precisely executes those instructions in runahead mode that causes full window stall by instruction slicing. In this technique, normalized execution stall time has been calculated on SPEC CPU 2006
2021	Vector Runahead By Ajeya Naithani [17]	Prefetching entire load chain when the processor is stalled and execute them in an efficient way. Extra Vector storage buffers are used. Other Benchmarks are used for IPC improvement.

finding the opportunity to use runahead execution) that addresses the aforementioned drawbacks. The need of finding the suitability of using runahead execution technique in modern processors has emerged for better performance by calculating full window stall cycles from modern benchmarks i.e. SPEC CPU 2017. However, it represents latest application challenges and FW stall needs to be calculated for latest applications.

In this research, full window stall cycles has been calculated and analysed for SPEC CPU 2017 benchmark suite and the processor characterisation is done based on these stall values. The recommendation of using runahead execution technique for the specific application area is also proposed.

2.8 Summary

This chapter contains the background of different techniques used in this study. Survey on simulators in which comparison of different type of simulators including (Functional vs. Timing Simulator), (Application-Level vs. Full-System Simulators) and (Trace-Driven vs. Execution-Driven Simulator) is discussed in detail. Additionally, an overview of several CPU benchmarks is provided, as well as an in-depth discussion of the SPEC CPU benchmark. Additionally, it demonstrates similar work and its limits, which influenced our choice of the suggested study technique.

Chapter 3

Research Methodology

3.1 Introduction

This chapter aims to describe the proposed research methodology to analyze the opportunities to enhance the performance using Run-ahead execution in SPEC CPU 2017.

3.2 Proposed Research Methodology

The flow of the proposed research methodology is illustrated in Figure 3.1. We have started the experiments with the simulator selection where we performed the theoretical comparison of number of best simulators and selected the most suitable simulator (i.e., Sniper 7.0) for further experimentation. After the selection, next step is to setup the simulator on Linux machine for which we used Ubuntu 16.04. After that, next step is to deal with the benchmark suites where we have SPEC CPU 2006 (for validation and verification purpose) and SPEC CPU 2017 for the implementation of the proposed research methodology.

In order to proceed with the experimentation, the original SPEC CPU 2017 benchmark suite which was purchased from SPEC corporation was run inside the simulator. However, we faced few limitation of using the original benchmark including

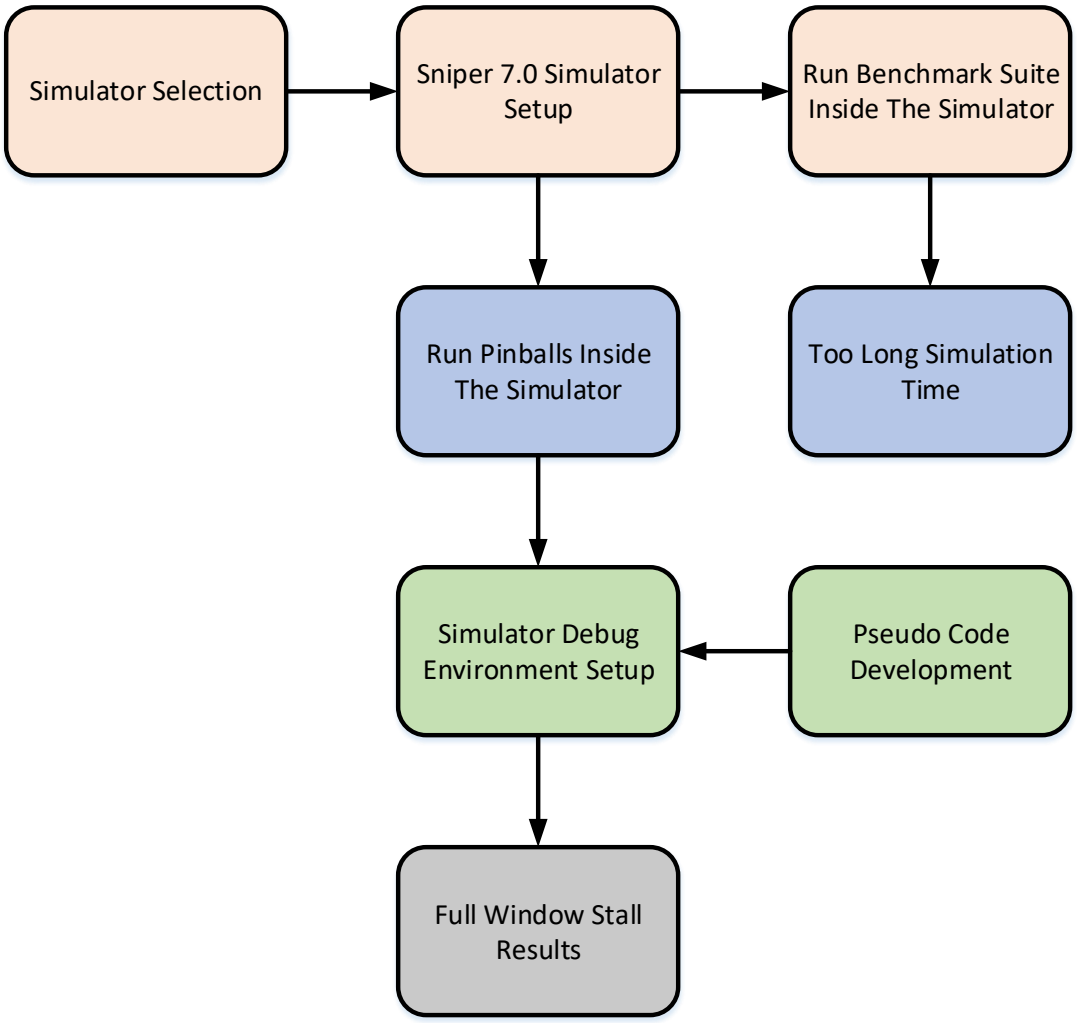

FIGURE 3.1: Workflow for the proposed methodology

(i) Simulation time for complete benchmark may take hundreds of hours, (ii) The process needs to be restarted if the simulation is interrupted and (iii) Computer resource utilization is very high. Pertaining to the issues associated with the use original benchmark, we introduced Pinballs or simulation points in our work, where instead of running the complete benchmark, simulation points are produced for both SPEC 2006 as well as for SPEC 2017 benchmark suites. Due to the reason that simulation points are the subset of the complete benchmark which mimics the same behavior, it gives us benefits in terms of time and computer utilization. Following, we worked on pseudo code development in order to find the full window (FW) stall and carried out the simulator debug environment setup where we performed some modification task to support the FW stall calculations. As a result,

we obtained the full window stall outcomes for the specific benchmarks.

We conducted our experiments using accurate interval core model in x86 Sniper simulator. Sniper simulator is used to run SPEC CPU2006 and SPEC CPU2017 benchmarks on different machine configurations to check the performance.

A simulator is a program that can have customized targeted output to explore various aspects of a design like functional behavior of the design blocks, resource utilization summary, power estimations, time and latency estimations etc.

A simulator enables us to check performance of a "To Be Designed" machine prior to its hardware implementation and an ideal simulator enables an architect to quickly investigate the design choices and precisely determine their effects on execution.

The main purpose of a simulator in case of a microprocessor design is to estimate the performance of new architecture before its fabrication to save time and money. Simulators are being used for microprocessor customization since the introduction of these new devices.

The design space exploration through simulator gives us the idea about the new design flexibility and throughput. Simulators make processor design and improvement faster and easier. Architecture simulator is irreplaceable for processor architects as a tool of processor structure research and golden module for logical design.

Sniper simulator tool is used in our work with PinPoint (simulation points) to reduce simulation time as the actual benchmark takes months to run. The simulated processor use ROB of 128 and 2048 entries. Simulation environment uses two versions of cache, Real LLC: in which every read/write transactions may hit or miss the last level cache and Perfect LLC: in which every transaction will hit in last level cache.

We used SPEC CPU2006 benchmark suite and SPEC CPU2017 benchmark suite to analyze performance enhancement opportunities in newly released CPU2017 benchmark in comparison with CPU2006 benchmark. We selected 8 integer benchmarks of CPU2017 among them 4 (*leela*, *exchange2*, *x264* and *xz*) are newly added [10]. We selected 12 integer benchmarks and 1 floating point benchmark from CPU2006.

3.3 Simulator Selection

3.3.1 Overview of Available Simulators

We selected four simulators: MARSSx86, PTLsim, ZSim and Sniper for breif comparison. Although these simulators use a variety of different simulation models, they all come under the umbrella of timing simulators. Except for PTLsim, all are contemporary simulators under active development. Although PTLsim is no longer being developed, it is still in use today. Each of these simulators is compatible with the x86 and other main architectures. Additionally, they are capable of doing thorough simulations on specified portions of any benchmark.

3.3.1.1 PTLsim

PTLsim [32] is a cycle-level simulator capable of simulating a whole operating system using the Xen hypervisor [33]. It employs co-simulation or a direct execution approach, as previously stated. It has the ability to modify a superscalar OoO core. It does not provide a thorough representation of the IO pipeline. The default core model in PTLsim is based on the features of many real-world systems, including Intel's P4 and Core 2 processors, and AMD's K8 CPU.

3.3.1.2 Sniper

Sniper [34] is a high-performance parallel simulator that makes advantage of the interval simulation technique outlined before [35]. Sniper is built on the Graphite platform, which allows a variety of one-IPC modes. Sniper is capable of simulating both OoO and IO pipelines.

3.3.1.3 ZSim

For x86-64 architectures, ZSim [36] is an application-level timing simulator for ZSim [36]. It started out as a model for ZCache [37] but has evolved into a more

capable simulator. Memory hierarchies and multiple core heterogeneous systems (single-ISA) are the primary emphasis. Modeling of OoO and IO pipelines is supported. It is able to run at very fast speeds because of its extensive usage of dynamic binary translation. A ZSim validation on an Intel Westmere core found an average inaccuracy of 10%. In multi-threaded workloads, the average absolute error is 11.2 percent.

3.3.1.4 MARSSx86

MARSSx86 is a cycle-level x86 full-system simulator [38]. MARSSx86's comprehensive pipeline model is based on PTLsim [32]. Additionally, several optimizations were introduced to improve efficiency and versatility. MARSSx86 simulates unmodified operating systems using a full-system emulation environment based on QEMU [39]. It is compatible with both out-of-order and sequential (IO) pipeline architectures. MARSSx86 is capable of simulating settings that are diverse. Additionally, it permits the emulation of real-time input/output devices.
Among the above mentioned simulators, Marss x86 and sniper simulator have been shortlisted for further consideration.

3.3.2 Marss x86 Vs Sniper 7.0

Comparitive description of Marss x86 simulator and Sniper 7.0 simulator is given in Table 3.1
From the table, we can see that the Sniper 7.0 simulator is much efficient and flexible and have better specifications in terms of support and facilities, it will be used for further research experimentation in our work. Brief overview of sniper simulator is given in next section.

3.3.3 Sniper Simulator

Sniper is high speed, parallel and accurate x86 next generation simulator [2]. It has the functionality to conduct simulations for multi-core processor architectures.

TABLE 3.1: Simulator Comparison: Marss x86 vs Sniper 7.0

S. No	**Marss x86**	**Sniper 7.0 (x86)**
1	Slow (Functional simulator)	Fast (Interval Simulator)
2	Kernel level modification	Runs in user space
3	Less flexibility	More facilities (Performance graphs etc.)
4	Low support/EOL	More support is available
5	Supported OS: Ubuntu	Supported OS: Ubuntu
6	Testing Computer: Core i5, 4th Gen	Testing Computer: Core i5, 4th Gen

It gives fast and accurate simulation for homogeneous and heterogeneous architectures compared with the existing simulators. Sniper relies on internal simulation which increases the level of abstraction and enables faster simulation and consume less evaluation time by jumping between intervals [24]. Due to these advantages we have selected sniper simulator for our experiments.

Sniper simulator is used to run standard benchmarks on different machine configurations to check the performance for comparison among different architecture designs. Sniper simulator provides visualization of the design contents using the 3D images of different parts of the design like thermal heat maps. Mostly simulation is used for high-risk experiments, safety test and scientific exploratory experiments.

Furthermore, Sniper allows the CPI stacks, which presents the number of lost cycles due to system characterization that enable the architect to better understand the actual performance of each component, and to measure the effectiveness of each component. SPEC CPU2017 benchmark suite is characterized by the instruction mix, branch prediction and cache memory behavior.

In Sniper simulator, parallel execution can be visualized by two representations. One is CPI stack and the other is Thread-state timelines. We mostly used CPI stack representation in our work. The processor performance can be represented by total number of instructions that a specific processor can execute in one clock cycle, or it can be expressed in term of CPI (Cycles per instruction) which means average number of cycles an instruction may take. In both ways, we calculate the execution time of a program (benchmark) to estimate the performance of the machine [24].

3.4 Benchmarks Availability

We have selected two SPEC CPU benchmarks (i.e., SPEC 2006 Benchmarks and SPEC 2017 Benchmarks) for the implementation and validation of the proposed methodology.

3.4.1 SPEC 2006 Benchmarks

Generally, a benchmark is used to to evaluate the performance of a machine. A benchmark characterizes application sets that the end user want to run on the machine. A benchmark perform the set of operation that are well defined and according to standards. These benchmarks are generalized form of many real applications. The end user can easily predict the performance of a machine by analyzing the performance of that machine on standard benchmarks.

We used SPEC CPU2006 and SPEC CPU2017 benchmarks to evaluate the performance and finding the opportunities to enhance the performance by implementing runahead execution in SPEC CPU2017. In our work, we selected 12 integer benchmarks and 1 floating point benchmark from CPU2006. Figure 4.8 shows load, store and branch instruction percentage in Instruction window. Different colors are used to clearly identify the relative percentages of these instructions. Blue color represents load instructions, red color shows store instructions while green color reveals branch instructions along with their respective percentages.

Figure 4.8 is helpful in presenting the comparison of these benchmarks with that of SPEC CPU2017. Performance enhancement opportunity using runahead execution is more in the benchmarks in which more load instructions are being used.The processor fetches the data from memory using the load instruction whenever there is a last level cache miss. More the load instructions are in the application, more the beneficial will it be to use the runahead execution.

It can be seen from the figure that highest percentage of load instructions is in *hmmer* benchmark having *47.36* percent load instructions while the lowest percentage of load instructions can be seen in *perlbench* benchmark having *27.99* percent load instructions.

3.4.2 SPEC2017 Benchmarks

SPEC CPU2017 benchmark is the latest released benchmark by Standard Performance Evaluation Corporation that generalized a number of advanced application set and computing algorithms to check the performance of machines having the load of many memory subsystem and intensive computation of industry-standardized applications. A number of changes occurred with reference to advancement in applications in SPEC CPU2017 benchmark compared with SPEC CPU2006 benchmark. Performance enhancement analysis after implementation of runahead execution in SPEC CPU2006 is discussed in literature review.

Our work focused on finding the opportunities of runahead execution for improved performance in SPEC CPU2017. We observed that performance index depends on number of load, store instructions that excess the memory. Whenever there is a high level cache miss, the processor has to access main memory and causes stall while, runahead execution prevents these stall by running ahead in the instruction window and executing multiple memory instructions in parallel, so that the data from memory is prefetched before its requirement.

We selected 8 integer benchmark of CPU2017 among them 4 (*leela*, *exchange2*, *x264* and *xz*) are newly added [10]. In the core comparison, we used Mcf, Omnetpp, Perlbench and Gcc benchmark of SPEC CPU2006 and SPEC CPU2017 which can be visualized in this graph.

This graph shows load, store and branch instruction percentage in Instruction window. Different colors are used to clearly identify the respective percentages of the instructions. The load instruction directly effect the LLC, so higher load instruction percentage will result in higher efficiency via runahead execution mode.

3.4.3 Drawbacks of Using Original Benchmarks

There are a few limitations to employing the original benchmarks for research purposes. Listed below are the drawbacks that apply:

1. Simulation time for complete benchmark may take hundreds of hours.

2. The process needs to be restarted if the simulation is interrupted.

3. Computer resource utilization is very high.

Pertaining to the above mentioned constraints, we will move toward the concept of Pinball or simulation points.

3.5 Pinball/ Simulation Points

In general, the Simulators require a binary scheme as an input. Numerous big, fascinating programmes, on the other hand, need complicated execution environments that are difficult to configure in combination with a simulator. Additionally, it can sometimes be impossible to simulate whole runs of long-running applications in detail. To address these problems, the "Pinball" (i.e., the user-level check-point structure) plays a vital role.
Pinball is generated and consumed via the use of a framework called PinPlay [40] that is based on dynamic instrumentation. It consists of two pin-tools: (i) a logger that records the initial architectural state and non-deterministic events happening during programme execution in a collection of files termed a pinball; and (ii) a replayer that operates on a pinball and repeats the recorded programme execution. The replayer may be used in conjunction with a simulator to simulate a game using a pinball rather than a binary set of instructions. A pinball may be built for the whole of a program's execution (a whole-program pinball) or for any interesting portion of execution (a region pinball). Using region pinballs to simulate huge programmes may significantly cut simulation time. It is compatible with both 32- and 64-bit x86 operating system binaries and no source code or special linkage is required.
There are numerous fascinating aspects to the pinball format:

1. A pinball is system-independent.

2. A pinball is self-contained, which means that it does not need the software binaries, input files, or specific licensing for playback.

3. A pinball is a compact system as compared to the overall simulator.

Computer architects often run the SPEC CPU2006 and SPEC CPU2017 benchmarks, which employ sniper simulators. Single-threaded CPU-intensive applications make up the simulators. The natural run-time of these programmes is in the range of a few seconds to a few minutes, depending on the inputs. Simulator applications that need cycle-accurate simulations might take months to execute on existing CPUs.

We utilised the PinPoints technique to identify portions of these programmes that were reflective of their simulations and used the PinPlay logger to construct pinballs for those places. In order to determine the quality of representative area selection, sniper simulator was used to compare whole-program simulation results with those anticipated by PinPoints pinball simulation. It saves us both time and money in the long run.

3.6 Calculation of Full Window Stalls

Full window stall occurs when the instruction at the top of the ROB blocks it due to a LLC miss [2]. We have designed the program to calculate the full window stall cycles by getting to know the architectural state of the processor when the stall occurs. Flow diagram of the algorithm to calculate the full window stall cycles is shown in Figure 3.2. Initially, when the benchmark starts execution inside the sniper, the program will check whether the last level cache is missed or not. If it is missed, then it will check whether the instruction gets full and is blocked or not. If the instruction window is not blocked then it will wait for the return of the cache miss and check the instruction window blocking. If the instruction window is blocked, it will check for the instruction types which are present at the top of the ROB (The instruction which is blocking the queue) and check if it is a load instruction. We have only checked the load instruction because it is verified from our experiments that almost 98% of the time, it is load instruction that is causing the full window to get stalled in terms of cycles. When the program detects that the load instruction is blocking the queue, it will check the time and store it in

terms of cycles. When the cache miss is serviced, the cycles count is calculated and stored.

This process is repeated until the end of the program and all of the cycles are accumulated that are being calculated when the ROB is full. From the flow diagram shown in Figure 3.2 the C program to find the full window stall is written and is shown in Figure 3.3. In this way, the full window stall cycles are calculated by executing the SPEC CPU 2006 as well as SPEC CPU 2017 benchmark suites.

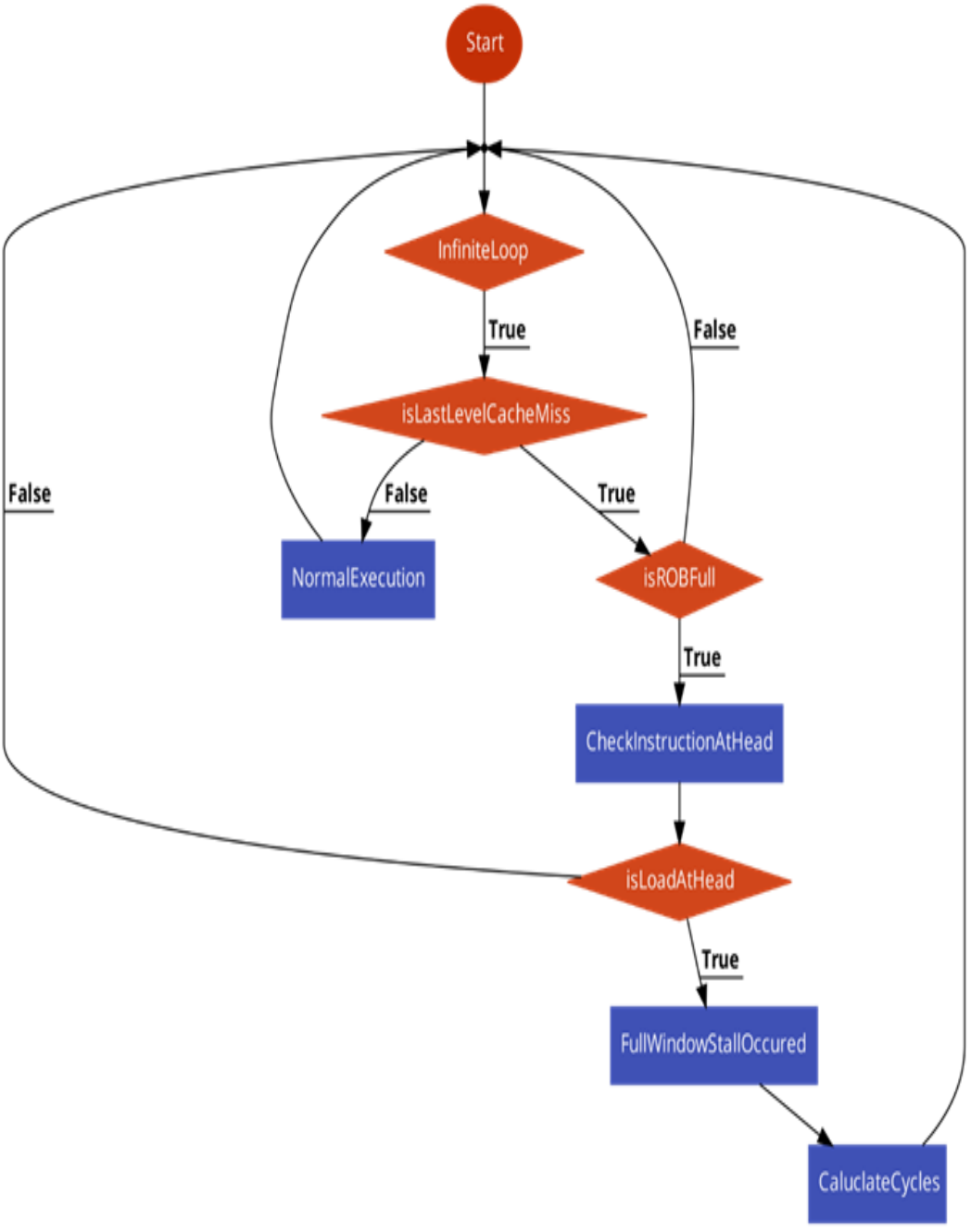

FIGURE 3.2: Flow Diagram of Algorithm

```
Start;
while(InfiniteLoop)
{
if(isLastLevelCacheMiss) {
  if(isROBFull)
  {
    CheckInstructionAtHead;
    if(isLoadAtHead)
    {
      FullWindowStallOccured;
      CaluclateCycles;
    }
  }
}
else
{
  NormalExecution;
}
}
```

FIGURE 3.3: C Code Snippet

3.7 Summary

The opportunities of runahead execution in the benchmarks are related to the proportion of memory access instructions that causes the stalls in instruction window. In order to determine the severity of these stalls in different benchmarks, their analysis is performed and described in this chapter with the help of graphical representations. The graphs shows each benchmark with their respective percentages of load, store and branch instructions. The Full window stall percentages of the benchmarks are discussed along with their graphs. The usefulness of runahead execution is summarized in this chapter.

Chapter 4

Simulation Model and Results

This section analyzes the performance of processors with different machine configurations with respect to full-window stalls using SPEC CPU2006 and SPEC CPU2017 integer benchmarks. Machines with 128 and 2048 ROB entries along with real and perfect LLC is being tested in sniper simulator [23]. SPEC CPU2006 benchmark suites shows that it spends up to *68.1%* of its cycles in full-window stall with respect to total cycles.

The results of full window stall in SPEC CPU2006 benchmark verifies the outcomes from [2]. Performed same experiment but now by reducing the memory latency by making LLC perfect and established that full-window stall cycles are now *78.5%* [5]. This indicates that main memory access latency is the major bottleneck in processor performance. We also investigated that how much of load, store, execute and branch instructions are causing full-window stall.

If we look at the instructions stucked at the top of ROB whenever full window stall occurs then it looks like *29.8%* of load instructions, *12.43%* of store instructions and *15.21%* of branch instructions are responsible for full-window stall for real LLC. If we talk about the number of cycles an instruction spends blocking the instruction window then we see that *98%* of the time, the full window stall is caused by load instructions in which processor access main memory and takes hundreds of times more cycles than other instructions. While for perfect LLC the number of full-window stall cycles are less with same configuration parameters, calculated number of full window stall cycles using SPEC17 benchmark suite.

Evaluation of new micro-architectural designs is highly important. Different benchmarks are used by computer architects to validate and improve the micro-architectural designs.

For computer architects the Standard Performance Evaluation Corporation(SPEC)'s 5th generation benchmark suite, known as SPEC CPU2006 and SPEC CPU2017, are the de facto benchmark suites for their processor design analysis [9]. However the newly released SPEC CPU2017 benchmark has gained attention because of its large workloads and instruction mix [10].

SPEC CPU2017 has remodeled benchmarks of SPEC CPU2006 and also added new workloads to meet modern application demands. The coverage area of SPEC CPU2017 is much more than that of SPEC CPU2006 in terms of workload design space [11]. These complex workloads are responsible of very high percentage of full-window stall. We analyzed the runahead execution opportunities for performance enhancement in SPEC CPU2017, compared it with the SPEC CPU2006 integer benchmarks and investigated full-window stall by increasing the size of ROB to 2048 entries and by making the LLC perfect.

4.1 Algorithm Validation

The values of full window stall has been calculated for SPEC CPU 2006 to validate our algorithm by comparing our results with the older values that researchers has produced. We have compared our obtained values with the findings reported by Naithani et al. [15].

4.1.1 Machine Configuration

This section contains the machine configurations used for the experiments. These configurations describes the parameters to build the machine environment in which the sniper simulator worked to find the basic run ahead execution opportunity for SPEC CPU 2006. The machine configuration used is the same as mentioned in the [15] so that our simulation environment matches each other.

TABLE 4.1: Machine configuration as mentioned by Naithani et al. [15]

Parameter	**Value**
Frequency	2.66 GHz
Type	Out of order
ROB Size	192
Issue Queue Size	92
Load Queue Size	64
Store Queue Size	64
Micro-op Queue Size	28
Pipeline Width	4
Pipeline Depth	8 stages (front-end only)
Branch predictor	8 KB TAGE-SC-L
Functional units	3 int add (1 cyc), 1 int mult (3 cyc), 1 int div (18 cyc), 1 fp add (3 cyc), 1 fp mult (5 cyc), 1 fp div (6 cyc)
Register file	168 int (64 bit) 168 fp (128 bit)
L1 I-cache	32 KB, assoc 4, 2 cyc
L1 D-cache	32 KB, assoc 8, 4 cyc
Private L2 cache	256 KB, assoc 8, 8 cyc
Shared L3 cache	1 MB, assoc 16, lat 30 cyc
Memory	DDR3-1600, 800 MHz Ranks: 4, banks: 32 Page size: 4 KB, bus: 64 bits tRP-tCL-tRCD: 11-11-11

4.1.2 SPEC CPU 2006 Comparison

Figure 4.1 shows that the value of benchmark libquantum is *75%*, whereas Figure 4.2 shows that the value of libquantum is *78%*, which is almost same. Additionally, the benchmark mcf and omnetpp values are *9%* and *47%*, respectively (see

Figure 4.1).

These numbers are almost identical to the benchmark values reported in the literature (see Figure 4.2), i.e., *8%* and *50%* for mcf and omnetpp, respectively. Since the implemented work produces the same results as the literature when equivalent parameters are used, the method is successfully validated.

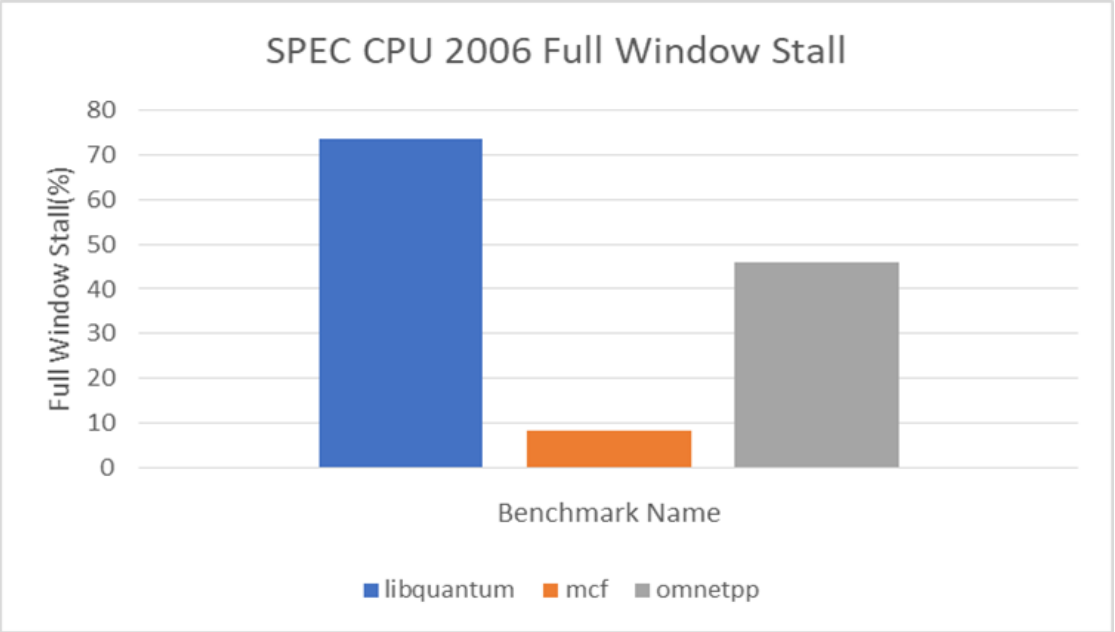

FIGURE 4.1: SPEC06 Full Window Stall

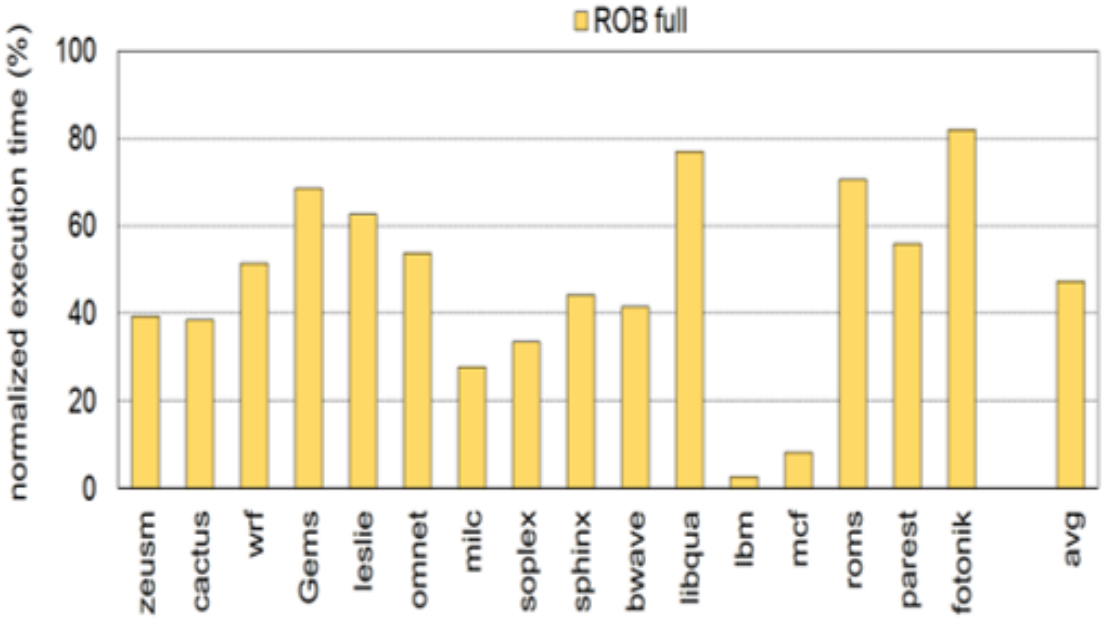

FIGURE 4.2: Precise Runahead Execution Stall Values [15]

4.2 SPEC CPU 2017 and SPEC CPU 2006 Full Window Stall

This section provides the full window (FW) stall estimation for SPEC CPU benchmarks (i.e., 2006 and 2017). In addition, it also incorporates the machine configurations used for the experiments. These specifications provide the parameters that will be used to create the machine environment necessary to identify run ahead execution opportunities for both benchmarks. Additionally, a portion gives a comparative study of FW stalls for both benchmarks.

4.2.1 Machine Configuration

Machine configuration in terms of processor parameters and memory parameters for full window stall estimation using SPEC CPU 2006 and SPEC CPU 2017 are given in below sections. We have taken the machine configuration same as mentioned in [2].

4.2.1.1 Processor Parameters

Processor configuration parameters are shown in table 4.2.

TABLE 4.2: Processor Configuration Parameters [2]

S. No	Parameter	Value
1	Processor Frequency	4 GHz
2	Instruction Window Size	128
3	Branch Misprediction Penalty	29 stages
4	Fetch/Issue/Retire width	3
5	Scheduling Window Size	16 int, 8 mem, 24 fp
6	Load Store Buffer Size	48 load, 32 store
7	Branch Predictor	1000 entry
8	Data prefetcher	16 streams
9	Memory Disambiguation	perfect

4.2.1.2 Memory Parameters [2]

Table 4.3 shows the memory configuration in finding the importance of runahead execution opportunities in SPEC CPU2006 and SPEC CPU2017 benchmarks.

TABLE 4.3: Memory Configuration Parameters

S. No	Parameter	Value
1	L1 Data cache	32 KB, 8-way
2	L1 Data cache Hit Latency	3 cycles
3	L1 Data cache bandwidth	512 GB/s
4	LLC Cache	512 KB, Real
5	Bus latency	495 processor cycles
6	Bus bandwidth	4.25 GB/s
7	Max pending Bus Transactions	10

4.2.2 SPEC CPU 2006 Full Window Stall

The instruction stream or list of operations can not be retired from the Instruction Window until the long latency instructions are not executed completely. So if the instruction window is not large enough it will be full and cause a stall in case of fetching data from main memory. Once the instruction window is full, new instructions cannot be placed into the window. It is called full window stall.

Figure 4.3 represents the full window stall situation for the 3 integer benchmarks which are selected from SPEC CPU2006. The Full window stall along with load instruction percentage of individual benchmark makes the way for runahead mode [13]. The Runahead execution paradigm is presented in this report which is a micro architectural technique to improve the performance of processor by utilizing long latencies.

The work presents the runahead execution in context of efficient high performance out-of-order execution processors, its pros, cons and limitations are presented in this report. It is a technique which allow the processor to detect the cache miss cycles instead of stalling which is done by data stream prefetches instructions

through this it detect the miss cycle before the actual execution of that instruction so the processor may save from stalls.

Figure 4.2 is taken from [15] which shows that the full window stall generated from old research is same as shown in Figure 4.3. In this way, we have validated our algorithm to find the stall from different benchmarks.

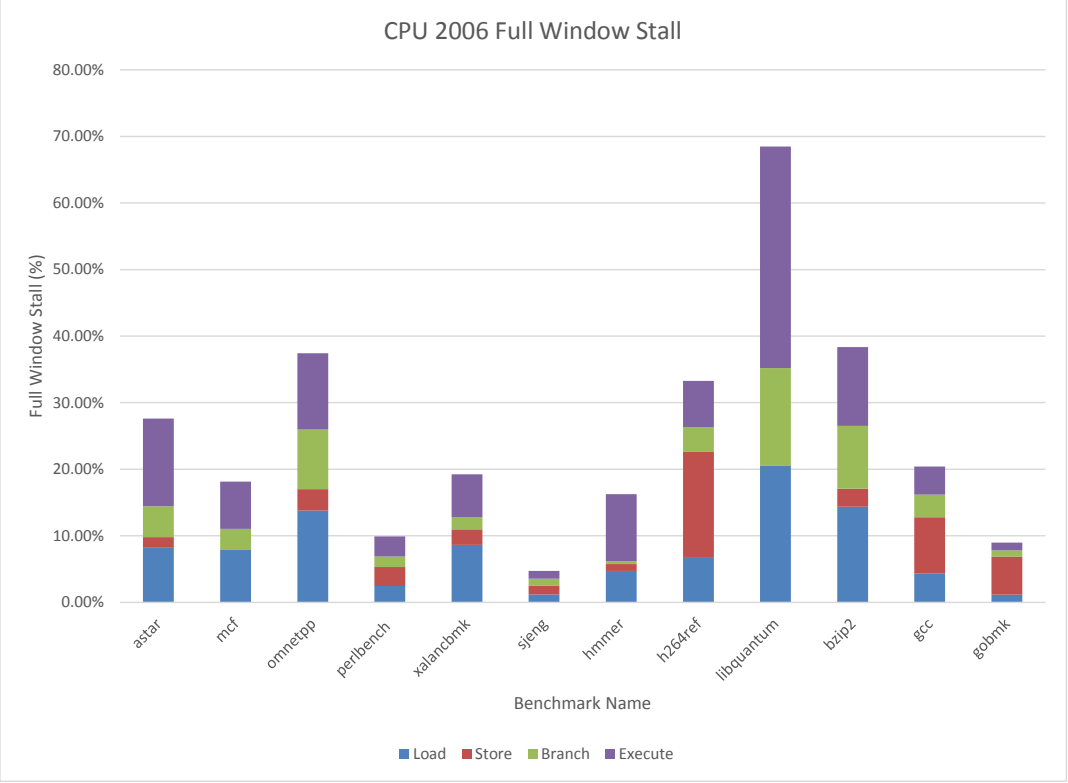

FIGURE 4.3: SPEC CPU 2006 Full Window Stall

4.2.3 SPEC CPU 2017 Full Window Stall

The full window stalls shown in Figure 4.4 can be avoided by using the runahead execution technique. Figure 4.4 is the visualization of 8 benchmarks full window stall which are selected from SPEC CPU2017. Four of them are newly introduced in 2017 while other four including mcf, Perlbench, Omnetpp, gcc are used for the comparison between SPEC CPU2006 and SPEC CPU2017.

This representation enables us to visualize the percentages of load, store and branch instructions in SPEC CPU2017 for the above-mentioned benchmarks. The

benchmark having high LLC and higher Full window stall percentage make more opportunity for run ahead execution.

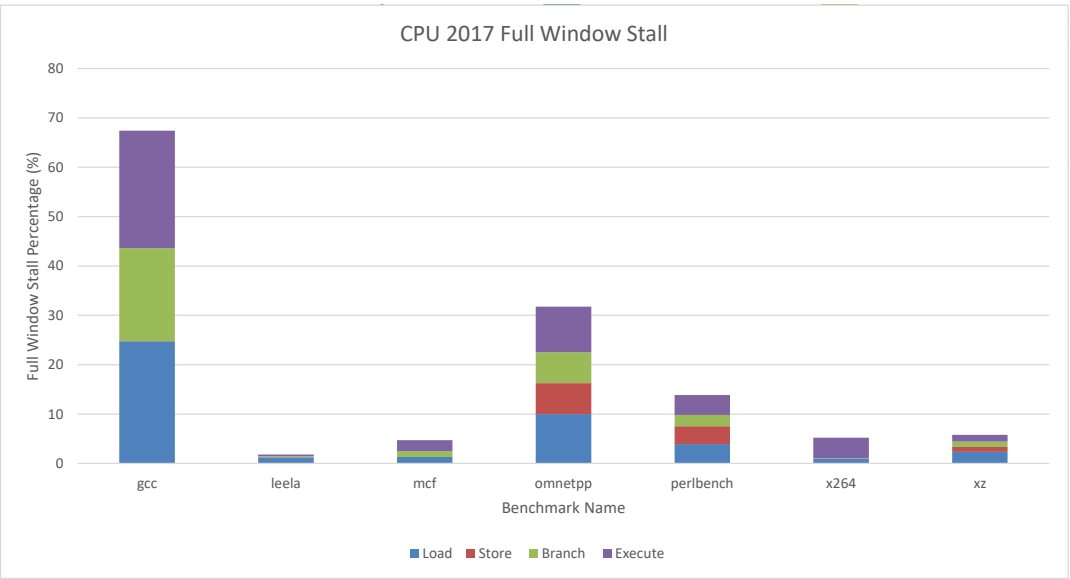

FIGURE 4.4: SPEC CPU 2017 Full Window Stall

4.2.4 FW Stall Comparison of SPEC06 vs SPEC17

This section shows the full window stall percentages of the selected integer benchmarks. Main memory access latency is the key bottleneck to improve the performance of a processor. Out of order processing tolerates these long latencies, provided that instruction commit will be done in program order to handle precise exceptions, by buffering the instructions in instruction window. The instruction stream or list of operations could not be retired from the Instruction Window until the long latency instructions are not executed completely.

So if the instruction window is not large enough it will be full. Once the window is full, new instructions cannot be placed into the window and it causes a full window stall. It stops the processor to move to the instructions that are not dependent on the long latency in the window.

One way to resolve it is to increase the size of instruction window but it becomes challenge-able because of complexity of design, difficulty in verification, power consumed by large instruction window and increased cost [5].

It also has drawback of critical path increased in quadratic way with the instruction window size due to the delays and complexity of many hardware structures [5]. Those out of order processor needed VLIW (Very Large Instruction Window) to handle memory lateness [11]. However instruction window eventually becomes full if top of the instruction window has long-latency instructions resulting in full-window stall. Figure 4.5 describes the comparison of the integer benchmarks of SPEC CPU2017 and SPEC CPU2006.

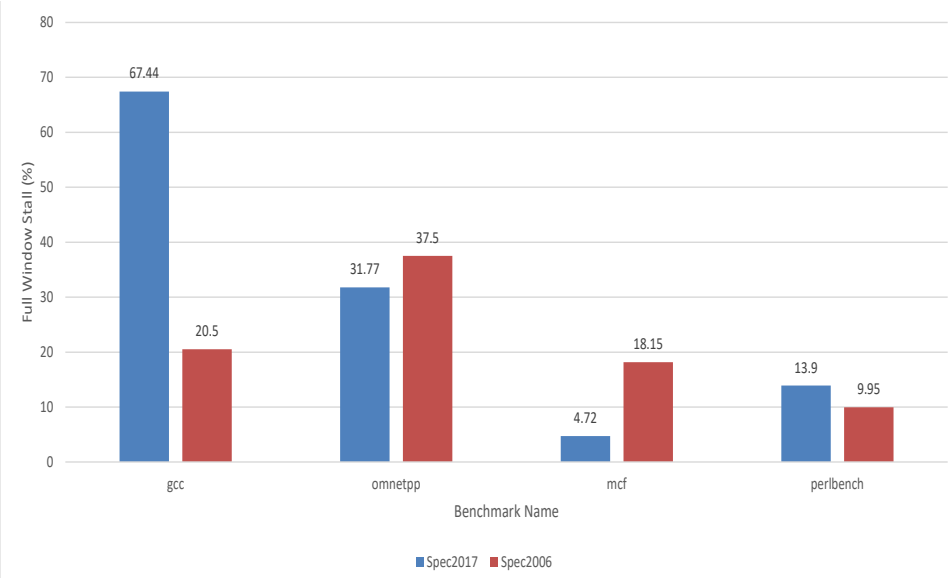

FIGURE 4.5: Full Window Stall Comparison of SPEC06 and SPEC17

In case of GCC benchmark, the percentage of full window stall is *67.44%* in SPEC CPU2017 while in SPEC CPU2006 the full window stall percentage is *40.32%* as shown in Figure 4.3. The reason is that GCC benchmark has high percentage of cache misses which in result force the processor to fetch data from main memory and leads to full window stall in instruction window stream.

In perlbench the load instruction percentage in SPEC CPU2017 is *27.34%* while that of in SPEC CPU2006 is *28.21%* which is slightly greater then that of SPEC CPU2006 as shown in Figure 4.4. However, results shows that SPEC CPU2017 have high percentage of full window stall i.e., *13.9%* than that of SPEC CPU2006 i.e., 9.9. So perlbench is the benchmarks which has more full window stalls in SPEC CPU2017 and has more runahead execution opportunities in SPEC CPU2017 to increase the processor performance. The perlbench of SPEC CPU2017 has *13.9%* full window stall cycles which is almost same as of SPEC CPU2006.

Omnetpp has *34.71%* of load instructions in SPEC CPU2006 while 22.76% of load instructions in SPEC CPU2017 as shown in the figure 4.5.
The mcf benchmark has *37.99%* of load instructions in SPEC CPU2006 whereas 18.55% of load instructions in SPEC CPU2017.
Omnetpp has *31%* full window stall as compared to which is almost the same as in SPEC CPU2006. But, mcf has low full window stall for SPEC CPU2017 as compared to SPEC CPU2006. It is not recommended to use runahead execution enabled processor for mcf based applications.

4.2.5 Instruction Type Occurrences When ROB is Stalled

Another experiment has been carried out to better understand the differences between SPEC CPU2006 and SPEC CPU 2017. Research conducted by previous academics has shown that the sole reason for stalling is due to load instructions that causes LLC misses [2].
We confirmed this by doing an experiment to see whether the instances of the instruction type at the top of the ROB are saved every time the processor goes into halted mode.
As seen in Figures 4.3 and 4.4, the execute instruction is often located at the top of the ROB most of the time. The reason for this is that the entire window stall happens for a little amount of time as well.
However, we computed the total number of window stall cycles in terms of instruction types and their cycle count. We discovered that 98 percent of full window stalls are caused by load instructions. Other instruction instances are brief and may be disregarded.

4.3 SPEC CPU2017 IPC Comparison

The purpose of this section is to validate the obtained result of instructions per cycle (IPC) from SPEC CPU 2017 benchmarks with the available results in literature [31] using the machine configurations stated below.

4.3.1 Machine Configuration

Table 4.4 details the machine setup used for the SPEC CPU 2017 IPC comparison. It contains information on parameters such as a shared front-end, shared memory system, out of order (OoO) back-end and In order (InO) back end.

TABLE 4.4: Machine configuration parameters for IPC analysis [31]

S. No	Parameters	
1	Shared Front End	4-way fetch and decode
		G share branch predictor
		32 KB 4-way L1 instruction cache
		128-entry ITLB
2	Shared Memory System	32 KB 8-way L1 data cache (3-cycle)
		256 KB 8-way L2 data cache (9-cycle)
		2 MB 16-way L3 data cache (35-cycle)
		Main memory (66 ns)
		64-entry L1 DTLB
		512-entry L2 TLB
		L1 and L2 stride data prefetcher
3	OoO Back End	4-way out-of-order back end
		128-entry ROB
		48- entry load queue
		32-entry load queue
		36- entry load queue
4	InO Back End	4-way in-order back end

4.3.2 IPC Comparison

Figure 4.7 is obtained from previously published work [31], where the researcher has used nine different benchmarks to analyze IPC in both out of order and in order processors. However, we used only 4 benchmarks (i.e., perlbench, gcc, mcf

and omnetpp) for the analysis of IPC for OoO processor only with SPEC CPU 2017 (see Figure 4.6).

As seen from Figure 4.6, perlbench has the value of 1.58 which is closer to the value (1.65) reported by the previous work as seen in Figure 4.7. Similarly, gcc, mcf and omnetpp from our work in Figure 4.6 has the value of 0.9, 0.6 and 0.42 against the almost equal values of 0.8, 0.7 and 0.5 for gcc, mcf and omnetpp, respectively, as mentioned in reference study (see Figure 4.7).

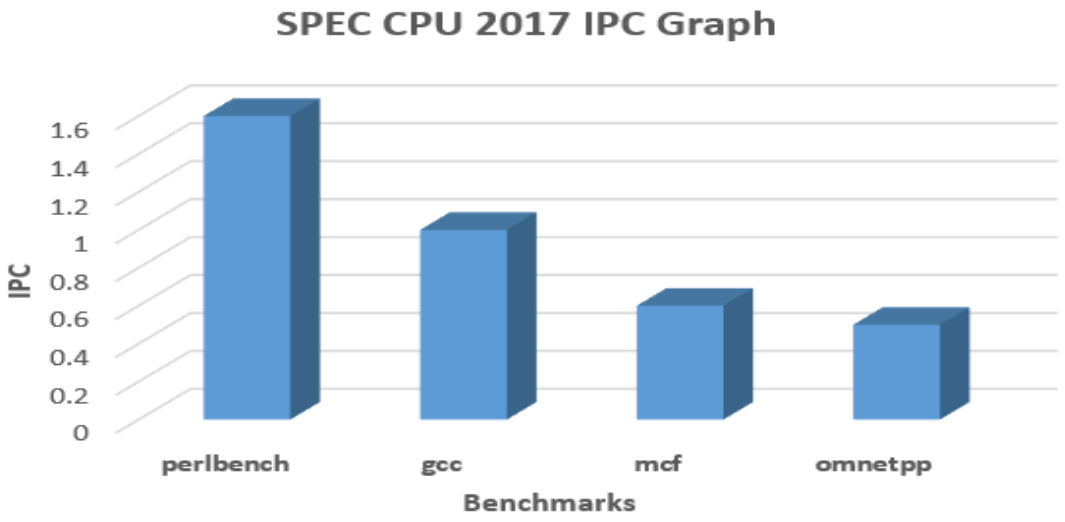

FIGURE 4.6: SPEC CPU 2017 IPC Graph

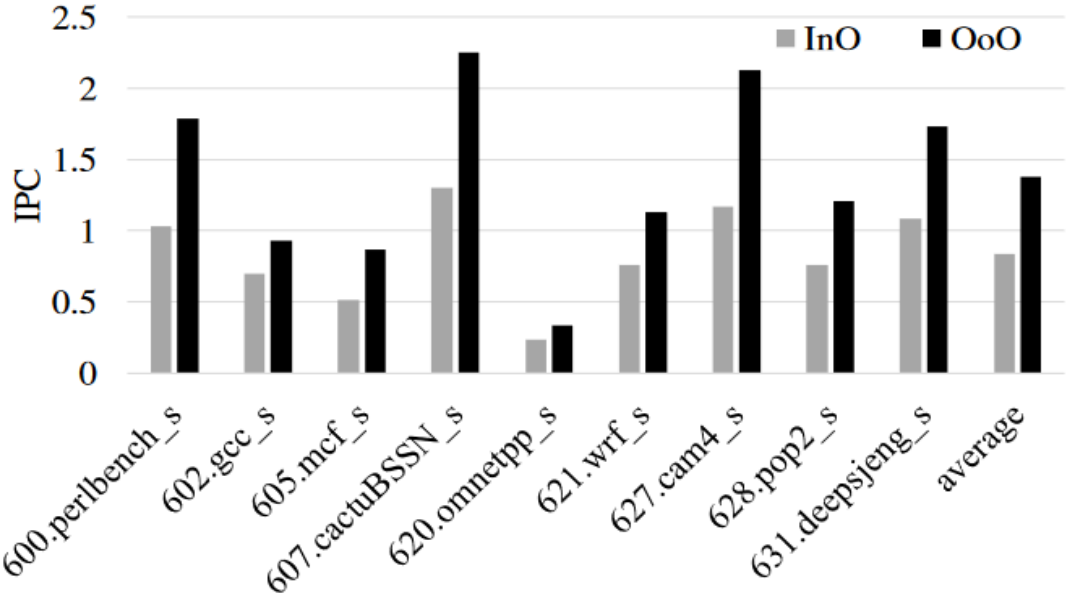

FIGURE 4.7: IPC of an InO an OoO processor with-out RAE [31]

4.4 Full Window Stall With Different Machines Configuration

This section provides the machine configuration for FW stall to verify the results reported in the literature. The processor configuration parameters and the memory configuration parameters are given in Table 4.5, and Table 4.6, respectively.

TABLE 4.5: Processor Configuration Parameters

S. No	Parameter	Value
1	Processor Frequency	4 GHz
2	Instruction Window Size	2048
3	Branch Misprediction Penalty	29 stages
4	Fetch/Issue/Retire width	3
5	Scheduling Window Size	16 int, 8 mem, 24 fp
6	Load Store Buffer Size	48 load, 32 store
7	Branch Predictor	1000 entry
8	Data prefetcher	16 streams
9	Memory Disambiguation	perfect

TABLE 4.6: Memory Configuration Parameters

S. No	Parameter	Value
1	L1 Data cache	32 KB, 8-way
2	L1 Data cache Hit Latency	3 cycles
3	L1 Data cache bandwidth	512 GB/s
4	LLC Cache	512 KB, Perfect
5	Bus latency	495 processor cycles
6	Bus bandwidth	4.25 GB/s
7	Max pending Bus Transactions	10

We have conducted another experiment to verify the results from literature work [2]. He has used 2 machine configurations in his experiments. One is the real

machine and another one is the perfect one. We have used the perfect machine configuration here for SPEC CPU2017 benchmarks to validate the findings. For this purpose, ROB size is increased to 2048 by keeping LLC to real, we have seen that the full window stall cycle will become zero. Additionally, if we optimise the LLC and limit the ROB to 128 entries, whole window stall cycles will be eliminated. Thus for SPEC CPU 2017, it is deduced that if we increase the ROB size or make the LLC cache to be perfect, the full window stall problem will be resolved.

4.5 Instruction Mix Comparison of SPEC17 and SPEC06

The reason behind the comparison of instruction mix is the huge difference of full window stall values between SPEC CPU 2006 and SPEC CPU 2017. As previously, we have verified that the full window stall cycles are *98%* percent of the time is due to load instruction. One might think, that the benchmark that does have a large load instructions it it will have large full window stall value. But it is not as expected, the benchmark with large load instruction can also have low full window stall values. So, it is better to have a comparison of instruction type percentage in each of the benchmark for SPEC CPU2017 and SPEC CPU2006 suites. In this way, we can better analyze the characteristics of the benchmark.
The opportunity to get enhanced performance using runahead execution relies mainly on the proportion of memory access instructions in the benchmark. Figures 4.10, 4.11, 4.12 and 4.13 presents the individual comparison of these benchmarks. In the core comparison we used Mcf, Omnetpp, Perlbench and Gcc benchmark of SPEC 2006 and SPEC 2017 which can clearly be visualized in this graph. This section contains the load instruction percentage of each benchmark and compare SPEC CPU2017 with SPEC CPU2006.
The GCC has *40.32%* load instructions in SPEC CPU2017 and *26.52%* load instructions in SPEC CPU2006. In the same manner, Omnetpp has *22.71%* load instructions in SPEC CPU2017 and *34.71%* load instructions in SPEC CPU2006

[5]. Likewise, Perlbench has *27.2%* load instructions in SPEC CPU2017 and *27.99%* in SPEC CPU2006. Similarly, the Mcf has *18.55%* in SPEC CPU2017 and *37.99%* in SPEC CPU2006 [14].

This graph is derived from the individual benchmark graph for instruction mix. This method is used for not only finding the subsets from any benchmark but to check the similarities and differences in these benchmark suite to find the potential of improvement in performance [12]. It will be used to conclude individual benchmark support for runahead execution opportunities.

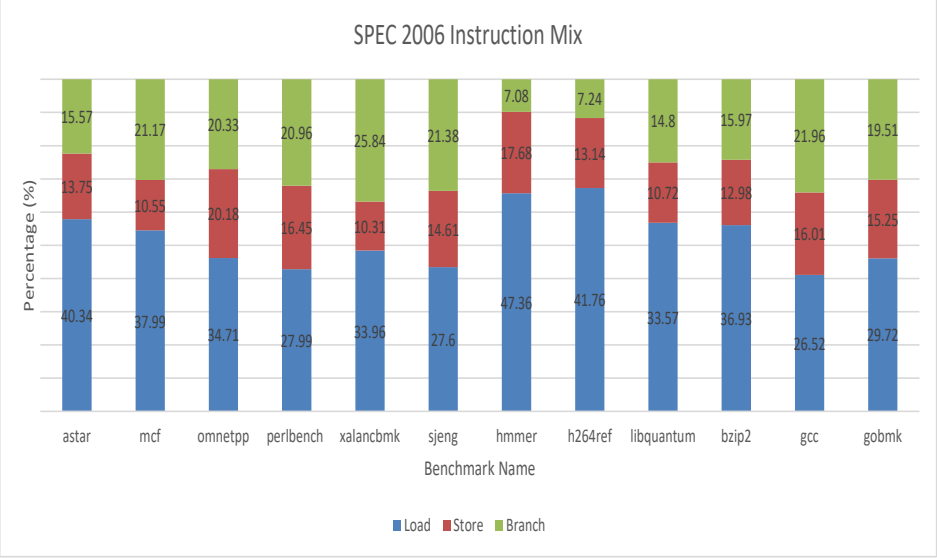

FIGURE 4.8: Instruction Mix of SPEC06 Int benchmarks

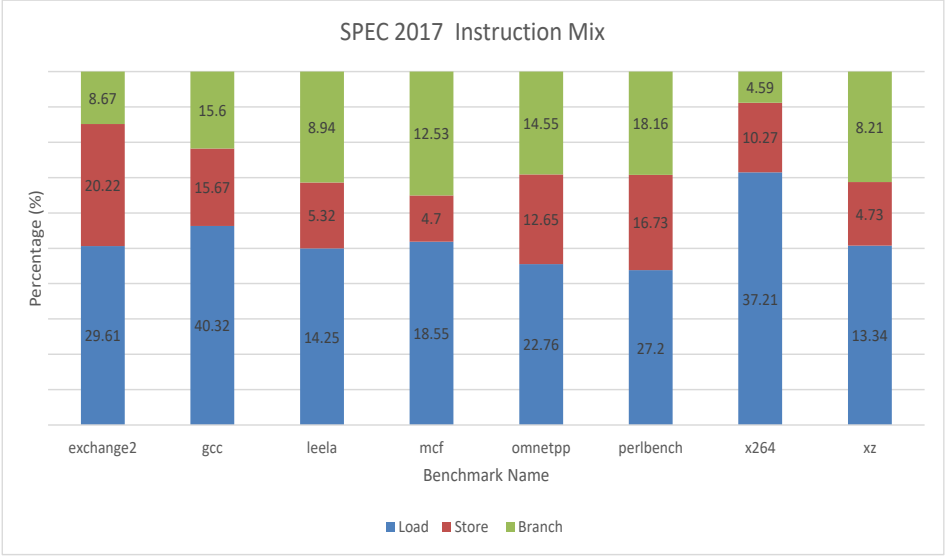

FIGURE 4.9: Instruction Mix of SPEC17 Int benchmarks

4.5.1 MCF Instruction Mix Comparison

Mcf benchmark of SPEC CPU2006 has *37.99%* of load instructions while SPEC CPU2017 has *18.55%* of the load instructions. So, Mcf suite of SPEC CPU2006 provides more runahead execution opportunity than that of SPEC CPU2017. Statistics of other memory access instructions are also high in SPEC CPU2006 than in SPEC CPU2017 e.g. in case of mcf, the percentages of store and branch instructions in SPEC CPU2017 are *4.7%* and *12.53%*, respectively. While, in case of SPEC CPU2006, the percentages of store and branch instructions are *10.55%* and *21.17%*, respectively. Runahead execution opportunities can be seen in Figure 4.10.

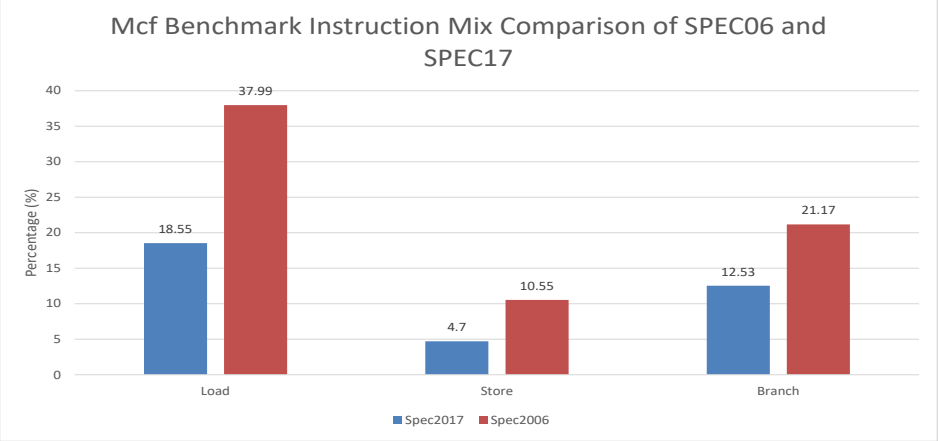

FIGURE 4.10: Instruction Mix Comparison of MCF

4.5.2 Omnetpp Instruction Mix Comparison

This benchmark contain *34.71%* of load instructions in SPEC CPU2006 while *22.76%* of load instructions in SPEC CPU2017. So this benchmark has more chance of LLC in SPEC CPU2006 while using runahead execution prefetching execution strategy, we got 34.71 percent usage of load instructions. Similarly, the percentages of store and branch instructions are *20.18%* and *20.33%*, respectively, in SPEC CPU2006 while the percentages of store and branch instructions in SPEC CPU2017 are *12.65%* and *14.55%*, respectively. The comparison of omnetpp in both SPEC CPU2006 and 2017 can be visualized in the Figure 4.11 for analysis of performance enhancement using runahead execution.

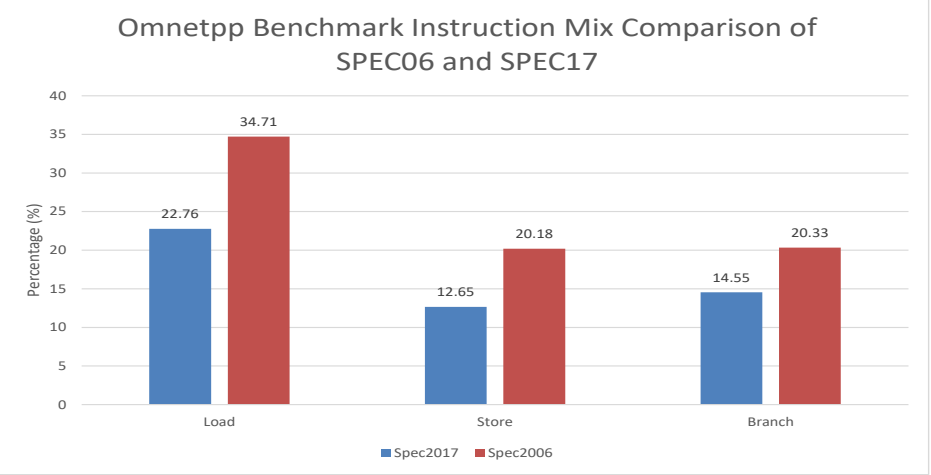

FIGURE 4.11: Instruction Mix Comparison of OMNETPP

4.5.3 PERLBENCH Instruction Mix Comparison

Perlbench benchmark of SPEC CPU2006 has *27.99%* of load Instructions while SPEC CPU2017 has *27.2%* of load instructions in perlbench. So, perlbench benchmark of SPEC CPU2006 provides more runahead execution opportunity than that of SPEC CPU2017. On the other hand, the percentages of store and branch instructions are *16.73%* and *18.16%*, respectively in SPEC CPU2017.
The percentages of store and branch instructions in SPEC CPU2006 are *16.45%* and *20.96%*, respectively. Runahead execution opportunities depending upon proportion of memory access instructions can be visualized from the Figure 4.12 for both SPEC CPU2006 and 2017.

4.5.4 GCC Instruction Mix Comparison

GCC benchmark contain *40.32%* of load instructions in SPEC CPU2017 where as the percentage of load instructions in GCC benchmark of SPEC CPU2006 is *26.52%*. so, runahead execution for pre-fetching execution from the main instruction window will be more beneficial in SPEC CPU2017 in case of GCC benchmark. Similarly, GCC for SPEC CPU2006 has *16.01%* of store instructions and *21.96%* of branch instructions whereas GCC for SPEC CPU2017 has *15.67%* of store instructions and *15.6%* of branch instructions. It can be visualized in Figure 4.13.

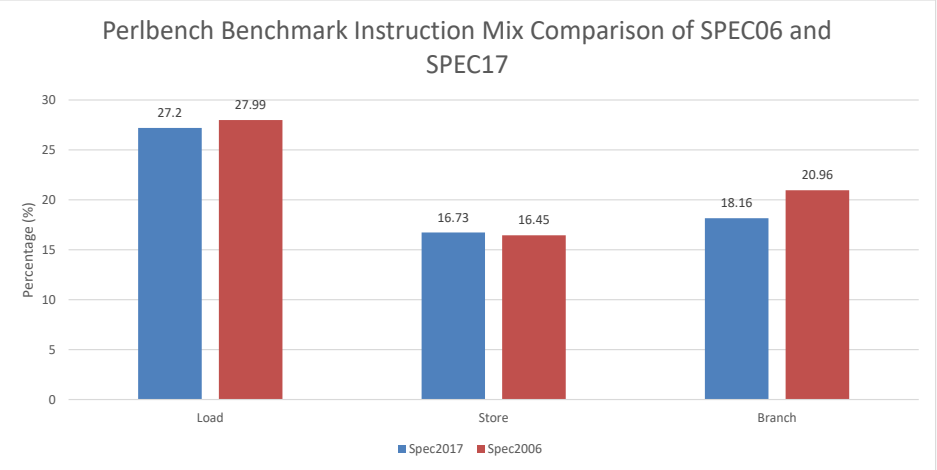

FIGURE 4.12: Instruction Mix Comparison of PERLBENCH

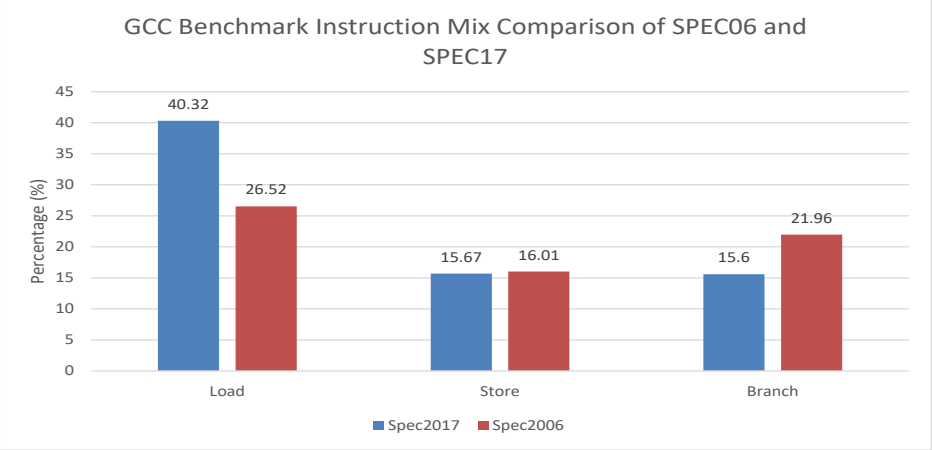

FIGURE 4.13: Instruction Mix Comparison of GCC

4.6 Processor Characterization for FW Stall Values

We have seen the huge difference between the full window stall values of SPEC CPU2006 and SPEC CPU2017 benchmarks. As SPEC CPU2017 benchmark is the successor of SPEC CPU2006. Therefore, the full window stall values for SPEC CPU2017 needs to be considered for processor characterization. We are proposing a processor characterization table in which the recommendation of using the runahead execution enabled processor is given based on the FW Stall values.

Table 4.7 provides the recommendation of runahead execution in accordance with the FW stall and the application area. Using the Omnetpp, *31%* full window stall

has been reported and runahead execution has been recommended. While *67.44%* FW stall was reported by GCC and runahead execution has been recommended. In contrast, Leela, X264 and MCF has the full window stall of less than *3%*, less than *5%* and less than *5%*, respectively. So, the runahead execution has not been recommended for any of them.

Table 4.7: Runahead execution recommendation table

S. No	**Application Area**	**Full Window (FW) Stall**	**Runahead Recommendation**
1	Omnetpp (Discrete Event Simulation)	31%	Yes
2	GCC (Compiler)	67.44%	Yes
3	Leela (Artificial Intelligence)	<3%	Not Recommended
4	X264, xz (Compression)	<5%	Not Recommended
5	Mcf (Combinational Optimization)	<5%	Not Recommended

4.7 Summary

This chapter contains the simulator configuration with all its configuration parameters for the machine which have used for the experimentation. Results of individual benchmarks are presented and explained in this chapter. By reviewing the chapter, it is concluded that there are some benchmarks like *gcc, omnetpp and perlbench* have full window stall occurrences greater as compared to other benchmarks due to sparse data locality and accesses. All the other benchmarks does not have too much full window stalls which implies that the runahead technique for these type of applications will not be suitable.

Chapter 5

Conclusion and Future Work

SPEC CPU 2017 is the successor of SPEC CPU 2006 that represents the modern applications i.e. compiler, artificial intelligence, discrete event simulation etc. Previously there was no work available for the processor characterization based on the full window stall cycles for latest benchmark suites i.e., SPEC CPU 2017. We have reproduced the full window stall values for the old benchmark i.e. SPEC CPU 2006 and validated it by comparing the values with the previous literature. After the validation of the algorithm we used for the full window stall calculation, the stall values are then calculated for SPEC CPU 2017 benchmark suite. This thesis concludes that the full window stall occurrences in SPEC CPU2017 is less as compared to the SPEC CPU2006. Our results indicates that these modern workloads have a very high percentage of full window stall cycles in some specific benchmarks like *gcc (67%) and omnetpp (31%)* but mostly the stall cycles are very low for rest of the benchmarks i.e., *perlbench (12%), xz (6%), x264 (5%), mcf (4%), leela (2%).* The characterization of the microprocessor is done based on the calculated full window stall percentage. It concludes that if the processor is to be used in the discrete event simulation or compiler based application domains, a runahead enabled processor is recommended for better performance. On the other hand, if the domain of application resembles mostly with compression, artificial intelligence and combinational optimization then runahead enabled processor will not be the optimal choice. As a result of it, the resource utilization of the processor will be more efficient and also the power consumption will be optimized.

5.1 Future Work

We were limited to use the provided simpoints for the specific benchmarks for SPEC CPU2017 at that time. Due to which the experimentation is being done on selected benchmarks from SPEC CPU2017. Once the simpoints will be available for other benchmarks which includes floating point benchmarks too, can also be used for the experimentation. In this way, all the benchmarks for SPEC CPU2017 can be tested and full window stall can be calculated for all of them. Feasibility of using runahead technique can also be further explored for these benchmarks.

Bibliography

[1] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th international conference on Supercomputing*, pp. 68–75, 1997.

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

[3] O. Mutlu, H. Kim, J. Stark, and Y. N. Patt, "On reusing the results of pre-executed instructions in a runahead execution processor," *IEEE Computer Architecture Letters*, vol. 4, no. 1, pp. 2–2, 2005.

[4] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro*, vol. 23, no. 6, pp. 20–25, 2003.

[5] O. Mutlu, H. Kim, and Y. N. Patt, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE Micro*, vol. 26, no. 1, pp. 10–20, 2006.

[6] Onur Mutlu, Hyesoon Kim, and Y. N. Patt, "Techniques for efficient processing in runahead execution engines," in *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 370–381, 2005.

[7] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 358–369, 2015.

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

[9] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[10] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 41–42, 2018.

[11] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a decade: Did spec cpu 2017 broaden the performance horizon?," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 271–282, IEEE, 2018.

[12] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, "Measuring program similarity: Experiments with spec cpu benchmark suites," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pp. 10–20, IEEE, 2005.

[13] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero, "Runahead threads to improve smt performance," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 149–158, IEEE, 2008.

[14] M. Hashemi, "On-chip mechanisms to reduce effective memory access latency," *arXiv preprint arXiv:1609.00306*, 2016.

[15] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise runahead execution," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 397–410, IEEE, 2020.

[16] O. Mutlu, H. Kim, and Y. N. Patt, "Address-value delta (avd) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pp. 12–pp, IEEE, 2005.

[17] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector runahead," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 195–208, 2021.

[18] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[19] P. S. Magnusson, F. Larsson, A. Moestedt, B. Werner, J. Nilsson, P. Stenström, F. Lundholm, M. Karlsson, F. Dahlgren, and H. Grahn, "Simics/-sun4m: A virtual workstation.," in *Usenix Annual Technical Conference*, pp. 119–130, 1998.

[20] A. Akram and L. Sawalha, "A comparison of x86 computer architecture simulators," 2016.

[21] L. Eeckhout, "Computer architecture performance evaluation methods," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–145, 2010.

[22] P. Crowley and J.-L. Baer, "On the use of trace sampling for architectural studies of desktop applications," in *Workload Characterization: Methodology and Case Studies. Based on the First Workshop on Workload Characterization*, pp. 15–24, IEEE, 1998.

[23] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, pp. 1–25, 2014.

[24] W. Heirman, T. E. Carlson, S. Sarkar, P. Ghysels, W. Vanroose, and L. Eeckhout, "Using fast and accurate simulation to explore hardware/software tradeoffs in the multi-core era," in *Applications, Tools and Techniques on the Road to Exascale Computing*, pp. 343–350, IOS Press, 2012.

[25] K. M. Dixit, "Overview of the spec benchmarks.," 1993.

[26] R. P. Weicker, "A detailed look at some popular benchmarks," *Parallel Computing*, vol. 17, no. 10-11, pp. 1153–1172, 1991.

[27] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *The Computer Journal*, vol. 19, no. 1, pp. 43–49, 1976.

[28] J. J. Dongarra, "Performance of various computers using standard linear equations software in a fortran environment," *ACM SIGARCH Computer Architecture News*, vol. 11, no. 5, pp. 22–27, 1983.

[29] K. Dixit and J. Reilly, "Spec developing new component benchmark suites," *SPEC Newsletter*, vol. 3, no. 4, pp. 14–17, 1991.

[30] J. L. Henning, "Spec cpu suite growth: An historical perspective," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 65–68, 2007.

[31] S. Mashimo, R. Shioya, and K. Inoue, "Energy efficient runahead execution on a tightly coupled heterogeneous core," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pp. 207–216, 2020.

[32] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pp. 23–34, IEEE, 2007.

[33] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[34] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2011.

[35] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, IEEE, 2010.

[36] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[37] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 187–198, IEEE, 2010.

[38] A. Patel, F. Afram, and K. Ghose, "Marss-x86: A qemu-based microarchitectural and systems simulator for x86 multicore processors," in *1st International Qemu Users' Forum*, pp. 29–30, 2011.

[39] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX annual technical conference, FREENIX Track*, vol. 41, p. 46, Califor-nia, USA, 2005.

[40] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 2–11, 2010.