國立交通大學

資訊科學與工程研究所

碩士論文

以低複雜度延伸有效的指令窗以容忍資料讀取失誤延遲

Tolerating Load Miss Latency by Extending Effective

Instruction Window with Low Complexity

研 究 生： 黃勁霖

指導教授： 鍾崇斌 博士

中華民國九十八年七月

以低複雜度延伸有效的指令窗以容忍資料讀取失誤延遲

Tolerating Load Miss Latency by Extending Effective Instruction Window with Low Complexity

研 究 生： 黃勁霖　　Student: Chin-Ling Huang

指導教授： 鍾崇斌 博士　　Advisor: Chung-Ping Chung

國立交通大學

資訊科學與工程研究所

碩士論文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

以低複雜度延伸有效的指令窗以容忍資料讀取失誤延遲

學生： 黃勁霖　　指導教授： 鍾崇斌 博士

國立交通大學資訊工程學系（研究所）碩士班

摘要

增加指令窗的尺寸和維持高頻率的時脈相衝突。而 run-ahead execution 可以發掘更高的記憶體平行度。然而，因為在大指令窗維持指令相依關係的困難度，在 run-ahead 狀態之下所產生的執行結果被浪費了。我們提出一個大的指令窗設計，相較於傳統亂序執行處理器，只使用簡單的結構和管理以維持高時脈。主要的概念為，當遇到一個長快取失誤指令，指令串以及部分的執行結果被暫時地移到一個大容量且快速的保留指令佇列。因此，指令窗可以在發生長快取失誤的同時繼續發掘未來的指令平行度。

實驗結果指出，在一個有一千個欄位的保留指令佇列的四路處理器下，相對於原本的 run-ahead 設計，對於 SPEC INT2000 以及 SPEC FP2000 分別有 5%與 10%的加速效果。

Tolerating Load Miss Latency by Extending Effective Instruction Window with Low Complexity

Student: Chin-Ling Huang Advisor: Dr. Chung-Ping Chung

Department of Computer Science and Information Engineering

College of Electrical Engineering College of Electrical Engineering and Computer Science National Chiao Tung University

Abstract

The confliction between increasing the instruction window size and keeping the clock cycle time small is getting worse. The run-ahead execution eases this problem by exploring higher memory level parallelism (MLP). However, the execution results produced in the run-ahead state are wasted due to the difficulty to maintain dependency across large instruction window. We propose a large instruction window design with the cycle time of simple structures and easy management. The main idea is, while a long latency load miss happens, the instructions with execution results are sequentially driven into a large and fast preserving buffer.

With a 1K-entry preserving buffer, the experimental results show that a 4-way processor with our design can achieve speedups of 5% and 10% over the original run-ahead execution design for SPEC INT2000 and SPEC FP2000.

致　謝

感謝我的指導老師 鍾崇斌教授，在我研究所的這段日子耐心地予以指導和教誨。讓原本基礎很差的我，漸漸可以瞭解一些計算機架構的內容。老師經常幫助我發現自己應該補強的點，讓我有機會一次一次的改善自己的問題。另外，在表達能力上面，透過老師的指導，讓我慢慢的進步，我得以順利完成此論文，幾乎完全仰賴老師的幫助。

另外，感謝李元化學長、喬維豪學長以及我的同學們，在過程中不斷的予以指導和建議，這篇論文，如果沒有他們的幫忙，是不可能完成的。特別是李元化學長，總是在繁忙的工作之後，拖著疲勞的身體，耐心的聽我一字一句的講說，並且幫我解決很多困難。

感謝我的家人，在過程中給予我鼓勵，讓我可以堅持到最後。

最後，僅向所有支持我、勉勵我的師長與親友，獻上最誠摯的祝福，謝謝你們。

黃勁霖

2009. 7. 12

Contents

List of Figures

List of Tables

1. Introduction

1.1 Problem of Load Misses

The processors can explore ILP in an instruction window through out-of-order execution, but they usually become idle due to load misses in the low level caches. As a processor encounter load misses with long latencies, the miss dependent instructions and subsequent instructions waiting for retirement are blocked. Therefore, the instruction window is occupied until the load misses are resolved.

Load instructions are the most frequent and common long latency operations. Store instructions can be completed off-line by the memory system. Other long latency operations, like division, are not common to all programs and less frequent to happen. Therefore, we focus on the load miss problem.

There are two situations that the pipeline will be eventually stalled due to long latency load misses.

1. Structural blockage due to control dependency: This happens because the long latency load misses stop all remaining instructions from retiring. (Figure 1) The load misses move to the head of ROB and wait for completion. Eventually, the ROB is full, the processor stops dispatching instructions, and then issuing of instructions eventually stops. The processor idles until the long latency operations complete.
2. Structural blockage due to data dependency: The miss dependent instructions fill the scheduler and stop subsequent instructions from entering. Thus instructions stop issuing. The processor idles until the load miss is resolved.

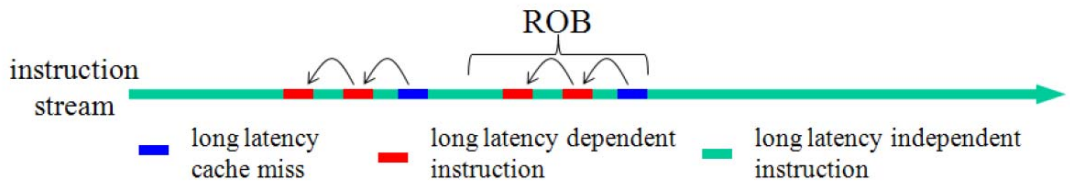


Figure 1 Structural blockage of ROB, RS and LSQ

As the technology advances, the processors tend to have deeper pipeline and higher frequency. Consequently, a load miss may take hundreds or even thousands of cycles, and will be even more serious.

Remedies for this problem are threefold, ranging from data pre-fetching, data speculation, to aggressive speculative execution (you

may find reference for these technology). Data pre-fetching and data speculation depend on execution patterns and therefore are less reliable for some programs. In this work, we study speculative execution in a large window of instructions.

Our simulation shows that the ROB and scheduler with thousands of entries are necessary to compensate the performance loss due to long latency load misses. However, it is not practical to scale up these structures to such a large size. Because large structures and long wires are used, it will be difficult to keep the clock cycle time small [1, 18].

One of the critical structures is the scheduler. The broadcast and comparison logic of the scheduler has been the bottle neck of the cycle time in a conventional out-of-order processor. Furthermore, to eliminate all anti-dependencies and output-dependencies and maintain the speculative values and to recover the processor state as branch mis-predictions or exceptions happen, the ROB should be extended to a large scale.

1.2 Related Work

The run-ahead execution continues to explore future load misses with low overhead as long latency load misses happen. The idea is to throw out the miss dependent instructions as the instruction window is blocked. Although the run-ahead execution can find more memory level parallelism (MLP), it wastes the execution results in the run-ahead state and all instructions should re-execute as it returns the normal execution state. The reason to give up these execution results is the difficulty that to maintain the dependency across hundreds or thousands of instructions.

If these execution results can be preserved in an easy way, higher performance can be achieved.

1.3 Achievement

In this work, we propose a new design to enable the processor continue to exploit future ILPs as long latency load misses happen. As a load miss instruction reaches the head of the ROB, this instruction and subsequent instructions are moved to a FIFO structure called preserving buffer (PB) in program order and leave critical structures including ROB, reservation station (RS) and load store queue (LSQ), which is shown in the figure 2. Except to explore higher MLPs, we preserve the execution

results with simple structures and easy management.

We have following achievements:

First, the execution core is no long blocked by long-latency load miss but continue to execute miss-independent instructions. When load misses, the processor switches to run-ahead mode, identify and bypass dependent instructions.

Second, execution results and bypassed instructions are all preserved and will be reused when the load miss is resolved. As the load miss instruction reaches the head of the ROB, instead of throwing them out of the ROB, instructions along with data are removed from ROB, RS, and LSQ; then they are packed into preserving buffer (PB) in program order. As the load miss is resolved, the instructions in PB are fetched into the ROB. The complete instructions with execution results are unnecessary to re-execute.

Third, miss-independent branch mis-predictions are all corrected during run-ahead state. These penalties can be hidden in the run-ahead state.

Fourth, area and management overhead is minimized. This design extends an ordinary out-of-order processor with only a FIFO preserving buffer and some simple small structures like the run-ahead register and run-ahead cache. These components can be gated off during normal execution state and waste no extra power.

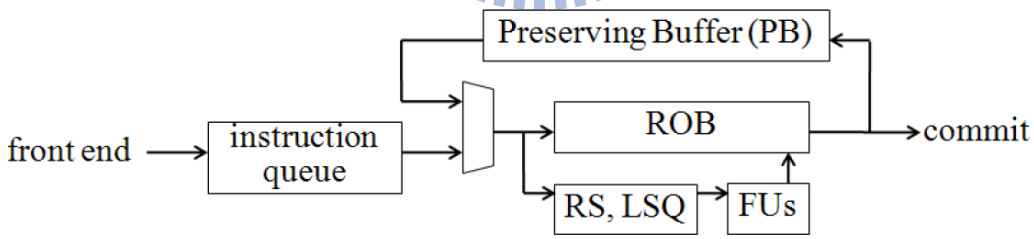


Figure 2 ROB with PB

1.4 Organization of this Thesis

The remainder of this paper is organized as follows. Section 2 provides related works of this paper and discusses their difficulty on implementation or drawbacks. Section 3 addresses our design and implementation details. Section 4 we evaluate the performance of our design and compare with conventional approaches. Section 5 summarizes this work and presents future works.

2. Background

In this section we introduce ideas of related works and then address their limitations on performance or difficulty on management.

2.1 Waiting Buffer Approaches

This kind of approach bases on the observation that load instructions causing cache misses and their dependence chains usually block instruction window. Their goal is to remove these instructions to a low-complexity structure rather than occupy the processor resources such as issue queue, ROB and register file.

Lebeck et al.'s waiting instruction buffer (WIB) [8]:

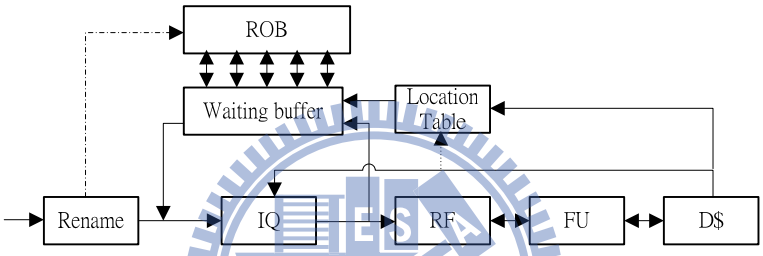


Figure 3 Waiting instruction buffer

As a load access low level memory, its destination register will be set "waiting" and its dependent instructions will observe this bit and then move into the waiting instruction buffer (WIB). The instructions move into the WIB also set their destination registers as "waiting" and their dependent instructions will observe these bits. Consequently the subsequent dependent instructions will eventually move to the WIB.

The WIB can be regarded as a simplified issue queue without wakeup-select logic. As cache misses are resolved, the locations of their dependence chain in WIB are identified by a location table, which records the WIB index of the instructions belonging to respective cache misses. These instructions then re-enter the issue queue and wait to be issued.

Difficulty:

1. The WIB design need to extend the physical register file to eliminate name dependency of all in-flight instructions, which can be extreme large. The reorder buffer should also scale up to a large size for exceptions and branch handling.

2. For handling branch mis-predictions and exceptions, WIN entries and ROB entries are correspondent one by one. Although many instructions do not depend on long cache misses, they are allocated entries in the WIB.

Cristal et al.'s slow lane instruction queue (SLIQ) [9]:

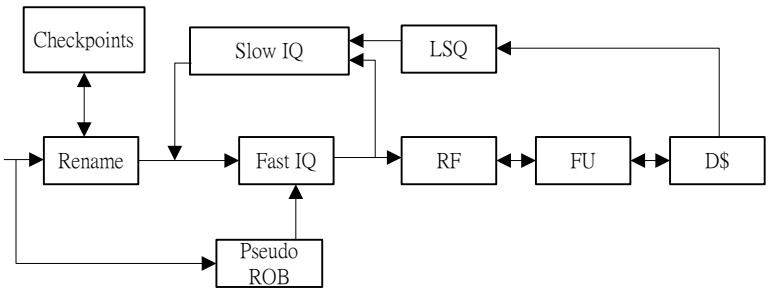


Figure 4 Slow lane instruction queue

The idea of slow lane instruction queue is similar to the WIB. However, the difference is that it detects long latency instructions by using a first-in-first-out (FIFO) pseudo reorder buffer. An instruction is inserted into the pseudo ROB before after it is decoded and removed as they retire. An instruction is considered as a long latency instructions as the reach the head of the pseudo reorder buffer and are still not to be issued. These instructions then move to the slow lane instruction queue (SLIQ) and wait.

On the other hand, this work proposes a checkpoint recovery mechanism to handle branch mis-predictions and exceptions. By eliminating the using of ROB, it can keep many in-flight instructions with sufficient checkpoints.

Difficulty:

1. The large instruction window leads necessity of many checkpoints, which may lead large hardware overhead.
2. The deallocation of physical registers may be delayed. A large physical register file is required.

Srinivasan et al.'s continual flow pipeline (CFP) [10]:

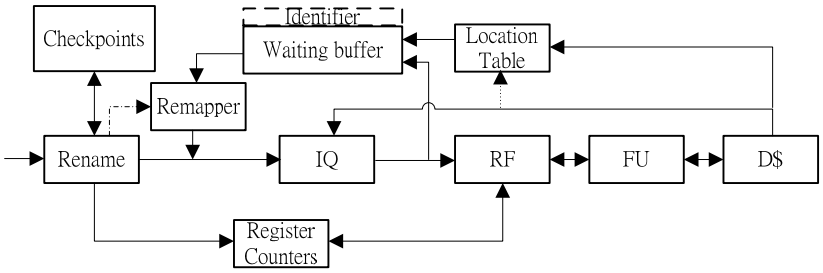


Figure 5 Continual flow pipeline

The concept of continual flow pipeline is similar to the WIB, but two new features are proposed.

1. The CFP utilizes a new mechanism to enhance physical register file management.
   - First, a counter for each registers records the number of its consumer instructions. As an instruction enters the renaming stage, it increases the counters of its source registers by one. As an instruction gets the value from a register, the counter of the register minus one. A register can be freed when its counter becomes zero and it is not a live register (of architectural registers). The instructions enter the slice data buffer will take the value of their ready source registers. This will enhance the reclamation of registers.
   - Second, the destination registers of the instructions moving to the slice data buffer will be released. They only retain the names of the physical registers to maintain the data dependency. Before they re-enter the issue queue, the remapper will allocate physical registers for them.
2. The CFP utilizes checkpoint recovery [12] to handle branch mis-predictions and exceptions. Thus the slice data buffer store only long latency instructions.

Difficulty:

1. Data dependency maintenance between instructions is very complicated. A new mechanism to early release physical registers and remap registers makes the design more complex.
2. Checkpoints recovery makes the flushing of instructions more complex.

2.2 Two-Mode Execution

Mutlu et al. ‘s Run-ahead execution [11]:

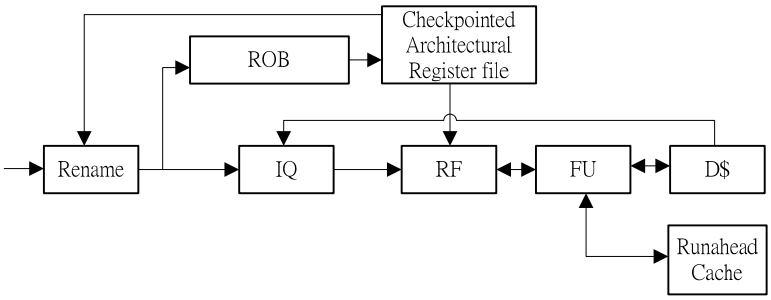


Figure 6 Run-ahead execution

The goal of this work is to use the processor idle time to execute future instructions with operands ready. Instead of preserving the execution results, the run-ahead mode only pre-fetches data for future instructions.

As a load cache miss moves to the head of instruction window and blocks, the processor checkpoints the processor state (mapping table and the contents of registers) and enter the run-ahead mode. In the run-ahead mode, the processor skips the instructions that directly or indirectly depend on load cache misses and executes the instructions with operands ready.

As the long latency operation that triggers the run-ahead mode completes, the processor restores the processor from the checkpoint and continues to execute.

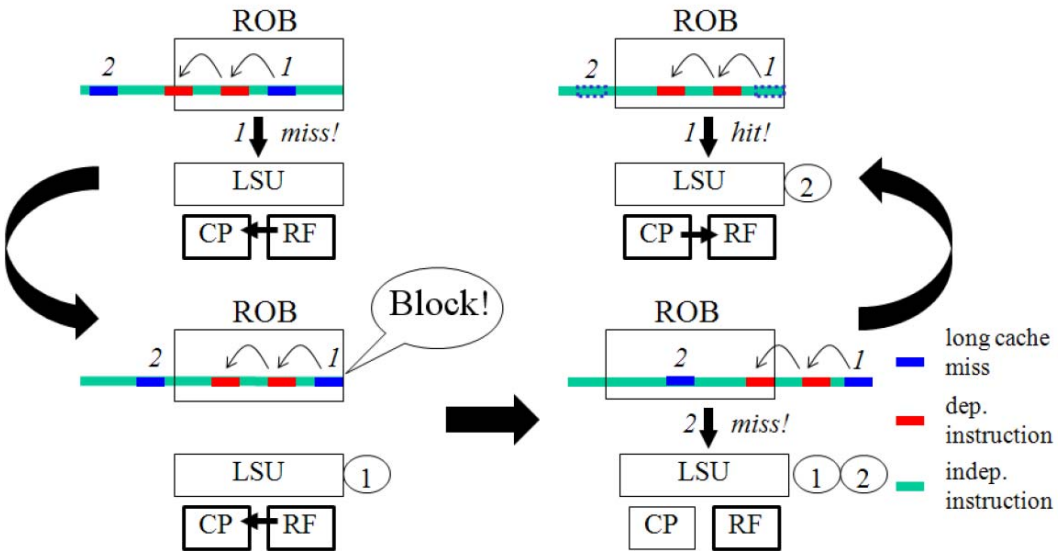


Figure 7 Run-ahead execution

Difficulty:

In the run-ahead mode, the execution results are not reserved even though they may be correct. All instructions executed in the run-ahead mode should re-execute in normal execution mode. Thus this approach causes more power waste.

Changing mode from the run-ahead mode to normal mode cause extra cycles.

Barnes et al. ‘s multi-pass execution [13]:

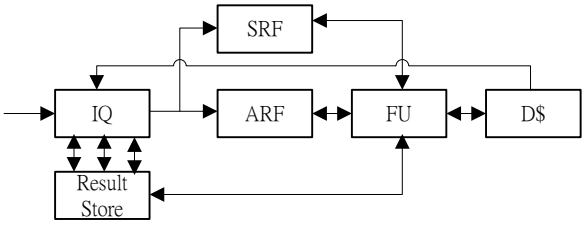


Figure 8 Multi-pass execution

The goal and idea of this work is similar to the run-ahead execution. The difference is that this design preserve execution results in advance execution mode.

Difficulty:

For easy management, this design is built on in-order processors and not only in the normal execution mode or in the advance execution mode the processor executes instructions in program order. While the execution results are reserved in advance execution mode, the performance is always limited by in-order execution.

2.3 Helper Thread Approaches

This kind of approaches speed up a single thread by using additional program context on simultaneous multithreading processors (SMT) or chip multiprocessing (CMP).

Zhou’s Dual-core execution [22]:

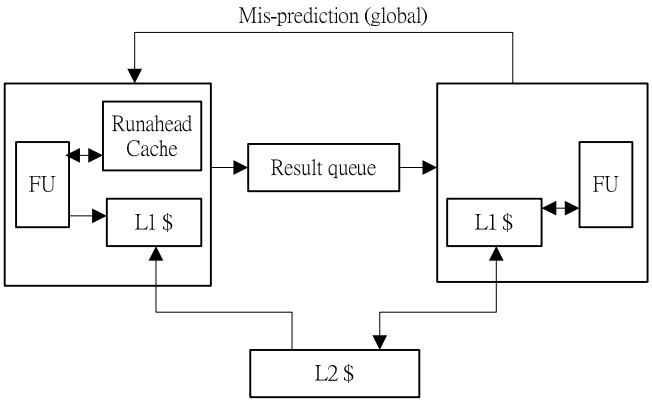


Figure 9 Dual-core execution

The idea of this work is to use one core for data pre-fetching and anther for normal execution. For the data pre-fetching processor, it is always in the run-ahead mode that executes instructions with operands ready and skips instructions depending on load cache misses. Thus it runs ahead of normal execution and continue to explore future cache misses.

Difficulty:

Similar to the run-ahead execution, the execution results produced in run-ahead mode are not reserved. All instructions should execute in normal execution processor. Thus much more power is consumed.

Sundaramoorthy et al. ‘s Slipstream Processor [14]:

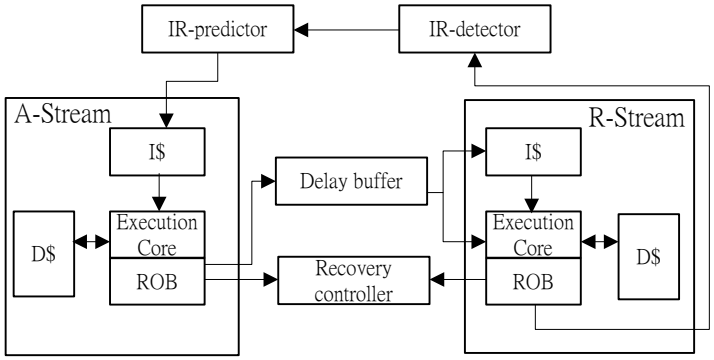


Figure 10 Slipstream processor

This approach uses one processor for speculative execution by skipping instructions that predicted to be invalid computations and provide execution results for the normal execution processor for value prediction.

Difficulty:

The processor that provides speculative values does not run far away from the normal execution processor. Thus the performance gain is limited.

3. Expanding Instruction Window with Low Complexity

3.1 Overview of our design

The major hurdle preventing run-ahead execution from making use of the execution results is that it does not manage dependency across long distance.

We conquer this difficulty by preserving long-waiting instructions and their computation results in sequential state and move them back to parallel execution when proper.

The main idea of our design is to preserve instructions with execution results in a FIFO preserving buffer (PB). Therefore, as the processor enters the run-ahead state, instructions together with execution results in ROB are sequentially driven from the ROB to the preserving buffer (PB). The preserving buffer is a simple FIFO structure with linear complexity and can be scaled to a large size, thus continuous exploration of future ILPs with value preserving is possible.

The following graph shows the overall picture of our design. The blue parts are the necessary parts to achieve our design. The main difference between our design and the original run-ahead execution is the preserving buffer to preserve the execution results.

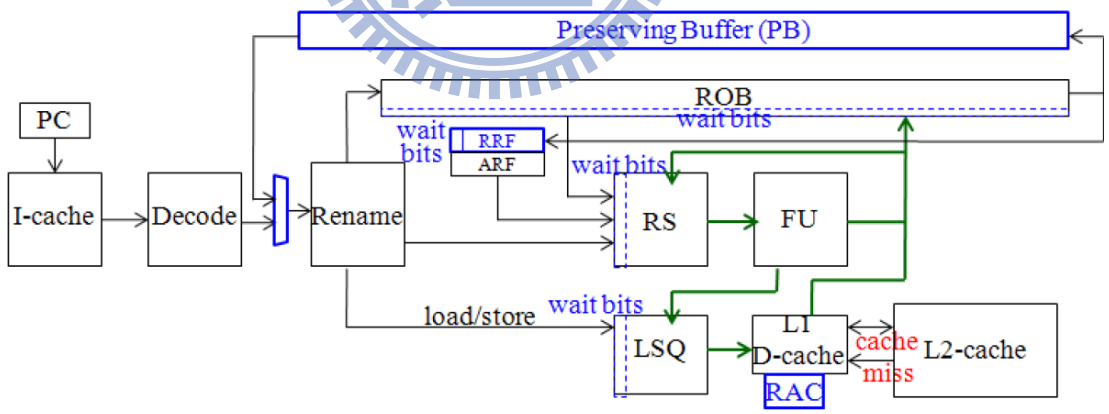


Figure 11 the overall diagram of our design

This scheme needs to solve following design issues:

1. How to revitalize the exhausted instruction window?
2. How to preserve instructions together with execution results?
3. How to resume parallel computations after the blocking reason is resolved?

4. How to manage memory access to maintain memory dependencies in the run-ahead mode?
5. How to handle exception or branch misprediction?

The procedure to handle load misses is divided into three states, which is shown in the figure 12:

1. Normal execution: The instructions execute and commit in the normal way.
2. Run-ahead execution: As the ROB is blocked, the processor enters the run-ahead state to continue to exploit future ILPs with value preserving.
3. Reusing execution: As the blocking event is resolved, the processor returns to the entry point of run-ahead execution and continue to execute instructions. The complete instructions with values are unnecessary to re-execute.

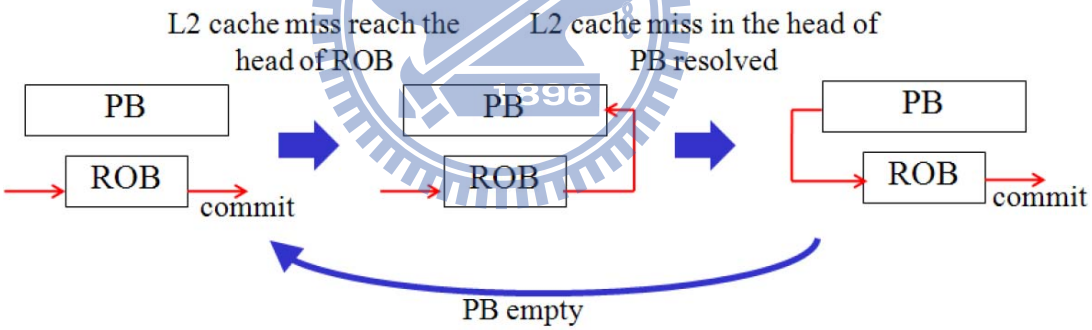


Figure 12 the state transition diagram of our design

3.2 How to revitalize the exhausted instruction window

In the normal execution mode, once a long cache miss operation reaches the head of the ROB, the processor enters run-ahead mode to prevent structural blockage.

Run-ahead execution

In the run-ahead mode, as an instruction moves to the head of ROB, the instruction that depends on a long cache miss operation moves to the PB without waiting for completion. The other independent

instructions execute, complete and move to the PB as they move to the head of ROB.
As an instruction pseudo-retire, it will not occupy critical structures like reservation station, reorder buffer and load/store queue. Thus the following instructions will not blocked by the long cache miss and the processor can continue to explore future ILPs.

Reasons that short latency operations should wait to complete in the head of the ROB

The first reason is that if they are bypassed without complete, their dependent instructions will not be executable. Therefore, there will be fewer instructions that the processor can explored in the following instructions. On the other hand, short latency operations may complete soon and wait for the long waiting instruction before. This results much longer latency for the following subsequent dependent instructions to start to execute because they cannot execute until move back to the ROB, which is harm for performance. We expect the original ROB can tolerate most of the short latencies and thus do not bypass the short latency operations directly.

The second reason is that because our design has several cycle penalties before starting to fetch instructions from the SROB, we expect that bypass short latency operations have low chances to gain performance.

Identification of dependent instructions

Our approach is similar to the Run-ahead execution, but our base architecture a ROB-based machine, which the ROB acts as the physical register file for register renaming. This makes the operations slightly different with the run-ahead execution.

We add additional bits called wait bit for each entry in the reorder buffer (ROB) and reservation station (RS) to indicate if an instruction depends on a long latency operation. The wait bit is functionally similar the ready bits in the RS. Once the waits of an instruction is set, it pretends to issue.

wait bit	complete bit	dest. (logical)	value / instruction

Figure 13 wait bit in ROB

OP	wait bit	ready bit	ready L	ready R	tag L	tag R	value L	value R

Figure 14 Reservation station fields

Initially, the ROB stores the original form of the instructions (decoded instructions). For an ROB entry, if the correspondent instruction completes, it write the result to the ROB and replace the content (decoded instruction). If an instruction is detected to depend on a long latency operation, it sets its ROB entry and remains the content (decoded instruction). The goal of maintaining the original form of an instruction is to form an equivalent instruction stream with partial values and partial instructions.

In the beginning, a load instruction that causes a memory access sets the wait bit of its ROB entry and then broadcasts its destination register tag to the RS.

An instruction whose wait bit is set pretends to issue and releases the scheduler entry immediately. The function units execute this kind of instructions without doing any computations and directly set the wait bits of the ROB.

It sets the wait bit of its ROB entry and then broadcast its destination register tag to the RS. The scheduler repeats this step and eventually detects all dependent instructions and releases their scheduler entries.

For the load and store instructions, their wait bits in the load store queue (LSQ) will be set as their effective address calculation depend on long cache misses. The load and store instructions will leave the LSQ and ROB as they reach the head of the two structures.

Moving instructions to the PB

As an instruction reaches the head of ROB, it move to the PB directly

if its wait bit is set. If its wait bit is reset, it will complete execution and then move to the PB.

The store instructions will not write values to the memory as moving to PB because the processor state should be maintained. The store instructions are treated as unexecuted and will execute as they return from the PB and commit.

Maintenance of architectural state

As an instruction moves from ROB to the PB, it should commit the execution result to the architectural register file for the following dependent instructions. However, the instructions do not retire in the run-ahead execution. Here we utilize an extra register file called run-ahead register file (RRF) with wait bits in each entries.

In the normal execution mode, as the ROB commits values, the values are only written to the original ARF. Before the processor enters the run-ahead state, the content of ARF is copied to the RRF. In the run-ahead state, the processor only updates the RRF. The original architectural register file maintains the newest version of values of the committed instructions while the RRF is established for maintaining the newest version of values of instructions moving out of the ROB.

The contents of the run-ahead register file may contain invalid values because the long cache miss operations and their subsequent dependent instructions do not complete their execution. If an instruction is not complete (depends on a long cache miss operation) while moving to the PB, the ROB does not writes the execution results to the RRF but set the destination register as waiting. If the instruction is complete, the ROB writes the execution result to the ARF. As an instruction enters the ROB, they will obtain their source values from the RRF or know that they depend on a long cache miss operation if their source registers in the RRF are set waiting and are unnecessary to enter the scheduler and set the wait bits or their destination registers.

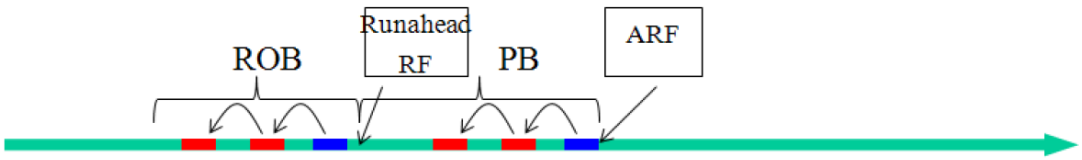


Figure 15 States of RRF and ARF on the instruction stream in the

run-ahead mode

Once the PB is full, the pipeline is stalled until the long cache miss operation in the head of the PB completes.

3.3 How to preserve instructions with execution results

In the run-ahead mode, instructions leave all the critical structures as removing from the head of ROB. Firstly, we want to preserve all execution results of instructions executed. Secondly, the dependencies should be maintained to complete the rest of instructions unexecuted and handle exceptions like the branch mispredictions.
It is known that the sequential instruction stream is the simplest way to maintain all the dependency of instructions. Therefore, as a long cache miss happens, rather than using large critical structures to achieve out-of-order execution, we use an extra buffer, which is called preserving buffer (PB) to preserve the instructions in a sequential way.

There are two design issues:
1. How to generate a sequential instruction stream?
2. How to keep the execution results?

Maintenance of instructions in ROB

The ROB can generate the instruction stream because the program order is maintained. But generally the ROB maintains only execution results and no instruction information such as source register numbers and opcodes.
To preserve the instruction with small overhead, rather than using an extra instruction queue to maintain the instruction stream, we use the value field of ROB to store the instruction. As an instruction is allocated an ROB entry, the instruction is also written to the value field.

wait bit	complete bit	dest. (logical)	value / instruction

Figure 16 shared field of value and instruction

In the run-ahead mode, if an instruction depends on a long cache miss operation, its wait bit in ROB will set and the value field remains the instruction. If the wait bit of an instruction is reset, the instruction will execute and the content of value field will be replaced with the execution result.

Instructions together with execution results in PB

The instruction stream with partial execution results is generated from the ROB. We use the preserving buffer (PB) to store this instruction stream for reusing.

The PB has similar fields with ROB except the wait bits and is an FIFO buffer with only sequential access.

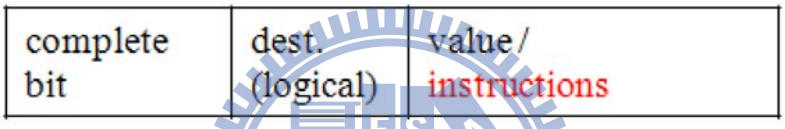


Figure 17 fields of PB

Here we compare the ROB and PB.

structure	functions
ROB	Access stages: register renaming stage, register read stage, commit stage (the 3 stages access the ROB at one cycle) Access types: sequentially access in register renaming stage and commit stage, random access in register read stage R/W ports requirement: (assume issue width is N) Register renaming stage needs N ports. Register read stage needs N*3 ports. (N instructions write to N destinations, N instructions read from N*2 sources) Commit stage needs N ports.
PB	Access stages: register renaming stage, commit stage

	(one of these 2 stages accesses the PB at one cycle) Access type: sequentially R/W ports requirement: (assume issue width is N) Register renaming stage needs N ports. Commit stage needs N ports.

We observe that the PB is much simpler than the ROB and is possible to scale to a large size. The larger the PB, the longer time the processor can run-ahead with value preserving.

Instructions with speculative values

Some instructions, like unresolved branches depending on L2 cache misses, should bring the speculative values for correction verification. If the speculative values do not equal to the execution results, the recovery mechanism should be activated.

A long cache miss dependent branch will execute until returning from the PB and the prediction target should be maintained. Because the ratio of the speculative instructions is low, we do not use an extra field in PB that causing much overhead. Here we propose to use the continuous next entry of the unresolved branch in PB to store the prediction target.

As an instruction is fetched from the PB, it will be checked if it is an unresolved branch. If so, the next entry contains the prediction target.

Instructions depending on the conditional register

For the long cache miss dependent instructions that depend on the conditional register, they should bring the value of conditional register with them. Like the unresolved branches, we propose that the value of the conditional register can store in the next entry of the instruction in PB.

3.4 How to resume the parallel computations after the blocking reason is resolved

As the long cache miss operation in the head of the PB is resolved, the ROB stops filling the PB and the processor will start to fetch instructions from the PB for reusing.

Leaving from run-ahead mode

For moving the instructions in PB back to ROB as soon as possible, all the instructions in the ROB are flushed (including RS, LSQ, RAC, function units). The operations in the cache system are set as data pre-fetch. The working register file is switched to the ARF. And then the instructions in PB are fetched sequentially.

Another alternative design does not flush the instructions in the ROB. As the long cache miss resolved, the instructions in the RS, LSQ and function units are invalidated. The operations in the cache system are set as data pre-fetch. The content of RAC is flushed. The working register file is switched to the ARF. An extra pointer for ROB called stream head pointer if set to the position of the tail pointer in ROB. And then the instructions in PB are fetched while those in ROB continue to move to the PB. As the head of instruction stream reaches the head of the ROB, the working register file is switched to the ARF.

Value reusing for complete instructions

If an instruction does not have execution result (depend on a long cache miss operation), it is taken from the value field of the PB entry and executes in the normal execution process. If the instruction has the execution result, it is unnecessary to wait for operands ready. The value is taken from the value field of the PB entry. It writes the value to the destination register directly.

Returning to the normal execution

As the PB is empty, the ROB continues to fetche the following instructions from the front end and the processor returns to the normal execution mode.

Entering the run-ahead mode in the reusing mode (not implement)

Dependent cache misses are possible to block the ROB as the processor is in the value reusing state. For tolerating dependent long cache misses, it is possible to enter the run-ahead mode while fetch instructions from the PB in the meantime. We will research this advanced design in the future.

3.5 How to manage memory access

In the run-ahead mode, the ready for execution load instructions

must know if they depend store instructions leaving the LSQ to the PB.

There are two problems to be solved:
1. How does a load instruction in ROB know if it depends on a store instruction in PB?
2. How does a load instruction obtain the value of a complete store in PB if it depends on the store?

Run-ahead cache (RAC)

The run-ahead cache (RAC) is an extra cache-like structure used in the run-ahead mode. The store instructions leave the ROB without writing values to the L1 d-cache while writing values to the RAC. As a load instructions look up the LSQ for checking memory dependency, it also accesses the RAC to determine if it depends on a previous store. And, if the store is valid, the value of the store is forwarded from the RAC.

Run-ahead cache operations

In the run-ahead mode, the load will access both the LSQ and the run-ahead cache. A matching address in the LSQ will have higher priority and the value will be forwarded from the LSQ in original way. If no address matching in the LSQ, the RAC accesses are valid.

Except the original cache fields, the RAC has a wait bit for each data entries for propagate the wait bit among stores and loads. If a store instruction depends on a long cache miss, as it writes the RAC, it sets the wait bit in the RAC. A following load that accesses the same data will see the wait bit and then sets its wait bit in the ROB. If the wait bit is reset, the data is valid and the load obtains the value and then writes the value to the ROB.

If a store has an unknown effective address, all the following load will be regarded to depend on this store. The contents of RAC is flushed and the effective address (EA) unknown bit is set. If a following load accesses the RAC and can't find a matching address, it sees the EA unknown bit to judge if it should set the wait bit in the or not.

As a store write the RAC to a cache set with address collision and the replacement should be performed, the evict bit of the set is set. If a following load accesses the set without address matching, it depends on the evict bit to set its wait bit in ROB.

If a load accesses both the LSQ and RAC without any matching address and the evict bit and EA unknown bit are reset, it then accesses the original L1 data cache.

tag	wait bit	data	evict bit	store EA unknown bit
	0	0x00000001	0	0
	1	Invalid		

Figure 18 the run-ahead cache

3.6 How to handle exception and branch misprediction

Because the PB is an in-order structure, the exception handling is simple by nullifying the younger instructions following the mis-predicted branch instruction.

Exception handling in the run-ahead mode

In the run-ahead state, the instructions in the PB are older than the instructions in the ROB. As an exception happens, the instructions following the mispredicted branch in the ROB are flushed and the renaming table is restored from the retirement mapping table. The handling rule is no difference from the original exception handling.

Exception handling in the reusing mode with the tail part of instruction stream in the ROB

In this state, parts of instructions are younger than the instructions in the PB and the others are older than the instructions in the PB. As an exception happens, this operation should be identified to be in the tail part or the head part of the instruction stream. Here we utilize the stream head pointer to achieve this. If this operation is between the head pointer of the ROB and the stream head pointer, it is in the tail part of the instruction stream. Or this operation is in the head part of the instruction stream.

As an exception happens, if the operation is in the tail part of the instruction stream, the instructions following this operation to the

stream head pointer are nullified. If this operation is in the head part of the instruction stream, the instructions between the head pointer of ROB and the stream head pointer, the instructions following the operation to the tail pointer of the ROB and the instructions in the PB are nullified.

Exception handling in the reusing state

In the reusing state, all instructions in the PB are younger than the instructions in the ROB. As an exception happens, the instructions following the mispredicted branch in the ROB and PB are nullified.

3.7 Summary of Operations in Each State

Normal state	
state transition event: if (an inst. with wait bit set reaches the head of ROB) copy the contents from the ARF to the RRF the working register file is switched to the RRF enter the run-ahead state	
components	*action*
multiplexer of instructions source	register renaming stage: insts. fetched from front end
ROB	register renamimg state: renaming logic allocates each inst. an entry renaming logic stores the inst. in the value field execution stage: if (an inst. with wait bit set is forwarded) FUs set the wait bit of the inst. in the ROB, no value writing commit stage: ROB commits values to the ARF
ARF or RRF	register read stage: read operand values from ARF commit stage: ROB commits values to the ARF
RS	issue stage: if (a broadcast with wait signal)

	if (any source operands match)
	RS sets the wait bit (with ready bit) of the inst.
	RS selects wait/ready insts. to issue
	execution stage:
	if (an inst. with wait bit set is forwarded)
	FUs send the destination tag and wait signal to RS

function unit	execution stage:
	if (an general ALU inst. with wait bit set issues)
	FUs do null operation and forwarding
	if (a load/store inst. with wait bit set issues)
	LSUs do null operation, forwarding
	LSUs set the wait bit in LSQ

D-cache/ memory system	execution stage: if (a load causes an L2 cache miss) D-cache sends cache miss signal and dest. tag to ROB and RS D-cache sets the load operation as data pre-fetch
LSQ	execution stage: if (a load finds address matching with a store with wait bit set) set the wait bit of the load in the ROB
RAC	idle
PB	idle

Table 1 operations in the normal state

Run-ahead state	
state transition event: if (a L2 cache miss is resolved) the insts. in the ROB are killed (including RS, LSQ, FUs, RAC) the cache access requests in the cache are set as pre-fetch operations the working register file is switched to the ARF enter the next state	
components	*action*
multiplexer of instructions source	register renaming stage: insts. fetched from front end
ROB	register renamimg state: allocate each inst. an entry and store the inst. in the value field execution stage: if (an inst. with wait bit set is forwarded)

	FUs set the wait bit of the inst. in the ROB, no value writing
	commit stage:
	ROB commits values to the RRF
	if (an inst. with complete bit set)
	move to the PB
	else
	if (an inst. with wait bit set)
	move to the PB
	else
	wait for completion and move to the PB

ARF or RRF	register read stage: read operand values from the RRF commit stage: ROB commits values to the RRF

RS	issue stage:
	if (a broadcast with wait signal)
	if (one of the source operands match)
	set the wait bit of the inst.
	select wait/ready insts. to issue
	execution stage:
	if (an inst. with wait bit set is forwarded)
	FUs send the destination tag and wait signal to RS

function unit	execution stage:
	if (an general ALU inst. with wait bit set issues)
	do null operation, writeback directly
	if (a load/store inst. with wait bit set issues)
	do null operation, writeback directly
	set the wait bit in LSQ

D-cache/ memory system	execution stage: if (a load causes an L2 cache miss) send the cache miss signal and destination tag to ROB and RS set the load operation as data pre-fetch
LSQ	execution stage: if (a load finds address matching with a store with wait bit set) LSUs set the wait bit of the load in the ROB
RAC	execution stage: a load issues and search the LSQ and RAC if (find a store with marching address in LSQ)

	if (the store is complete)
	use the data
	else
	if (the store's wait bit set)
	set the wait bit
	else
	stall, wait until the store to complete or set wait
	else if (find a store with matching address in RAC)
	if (the store's wait bit is not set)
	use the data
	else if (the store's wait bit is set)
	set the wait bit
	else if (the evict bit of the accessed set is set || the effective unknown bit is set)
	set the wait bit
	else
	access the I-cache
	
	commit stage:
	if (a complete store)
	if (the set is not full)
	write the RAC with wait bit reset
	else
	write the RAC with wait bit reset and replacement
	set the evict bit of the set
	else
	if (effective address known)
	if (the set is not full)
	write the RAC with wait bit set
	else
	write the RAC with wait bit set and replacement
	set the evict bit of the set
	else
	set the effective unknown bit

PB	commit stage:
	If (complete insts.)
	if (OPtype is store)
	effective address in value field

	register of store data in the destination register field
	else
	execution result in the value field
	else
	if (OPtype is branch)
	instruction in the value field
	store prediction target in the next entry
	else
	instruction in the value field

Table 2 operations in the run-ahead state

Reusing state	
state transition event: if (PB is empty) enter the normal state	
components	*Action*
multiplexer of instructions source	register renaming stage: insts. fetched from PB
ROB	(normal operation)
ARF or RRF	(normal operation)
RS	(normal operation)
function unit	(normal operation)
D-cache/ memory system	(cache miss signal is not activated)
LSQ	(normal operation)
RAC	idle

PB	register renaming stage:
	if (complete insts.)
	if (OPtype is store)
	effective address in value field
	register of store data in the destination register field
	else
	execution result in the value field
	else
	if (OPtype is branch)
	instruction in the value field
	store prediction target in the next entry

	else instruction in the value field

Table 3 operations in the reusing state

4. Experimental Results

4.1 Methodology

In the experiments, we like to know the performance gain of our design over the traditional out-of-order and the original run-ahead execution design. We also compare with the traditional design with large instruction window without concerning the cycle time constraints to observe the performance we can achieve.

We modify SimpleScalar 3.0 simulator to support our design with SPEC INT2000 and SPEC FP2000 benchmark suites. Our simulator cannot execute sixtrack and ammp.

The following is the base configurations.

Fetch/Decode/Issue/Commit width	4 / 4 / 4 / 4
RS/ROB/LSQ sizes	128
Branch prediction	Gshare, 4K-entry prediction table
L1 data cache	32 KB, 4 way
L1 inst. cache	32 KB, 4 way
L1 latency	2 cycles
L2 Unified Cache	1 MB, 8 way
L2 latency	12 cycles
Memory latency	300 cycles
TLB	32-entry, 4 way associative, 4 KB page size, 32-cycle penalty
Function units	4 integer ALUs (1-cycle), 2 integer multipliers (7-cycle), 4 FP ALUs (4-cycle),

	2 FP multipliers (7-ccyle)

Table 4 Detailed configurations

4.2 Performance Comparison with the Original Run-ahead and Base Configuration

Here we compare the performance among our design, the original run-ahead execution and the base configuration. The following table shows the configurations of sizes of critical structures. The detailed configurations are the same.

In the original run-ahead execution and our design the run-ahead cache (RAC) is perfect and will not cause any cache misses. In our design, the size of preserving buffer is unlimited to run-ahead as far as possible. The effects of the buffer sizes will be displayed later.

	O-o-O base	Run-ahead	ROB + PB
RS size	128	128	128
ROB size	128	128	128
PB size	-	-	unlimited
LSQ size	128	128 + perfect RAC	128 + perfect RAC

Table 5 Comparison with the Original Run-ahead and Base Configuration

We can see that in average our design has obvious speed-up over the baseline configuration for SPEC INT2000 and SPEC FP2000. This means that many ILPs can be exploited during the run-ahead mode as long latency cache misses are processing.

In average the performance in SPEC FP2000 of large instruction window configurations are much better than those in the SPEC INT2000. The main reason is that the L2 cache misses are usually in the critical path in SPEC INT2000. Besides, the L2 cache miss rate are higher in SPEC FP2000, the effect of overlapping long execution latencies is largely enhanced.

Our design outperforms the original run-ahead execution because

the execution results are preserved in the run-ahead mode. The more the values in the critical path preserved in the run-ahead mode, the more our design outperforms the original run-ahead execution. This depends on the benchmarks' program behaviors.

On average, our design improve performance of base out-of-order by 12% and the original run-ahead by 5% for SPEC INT2000. For SPEC FP2000, the two provide improvements of 56% and 10% respectively.

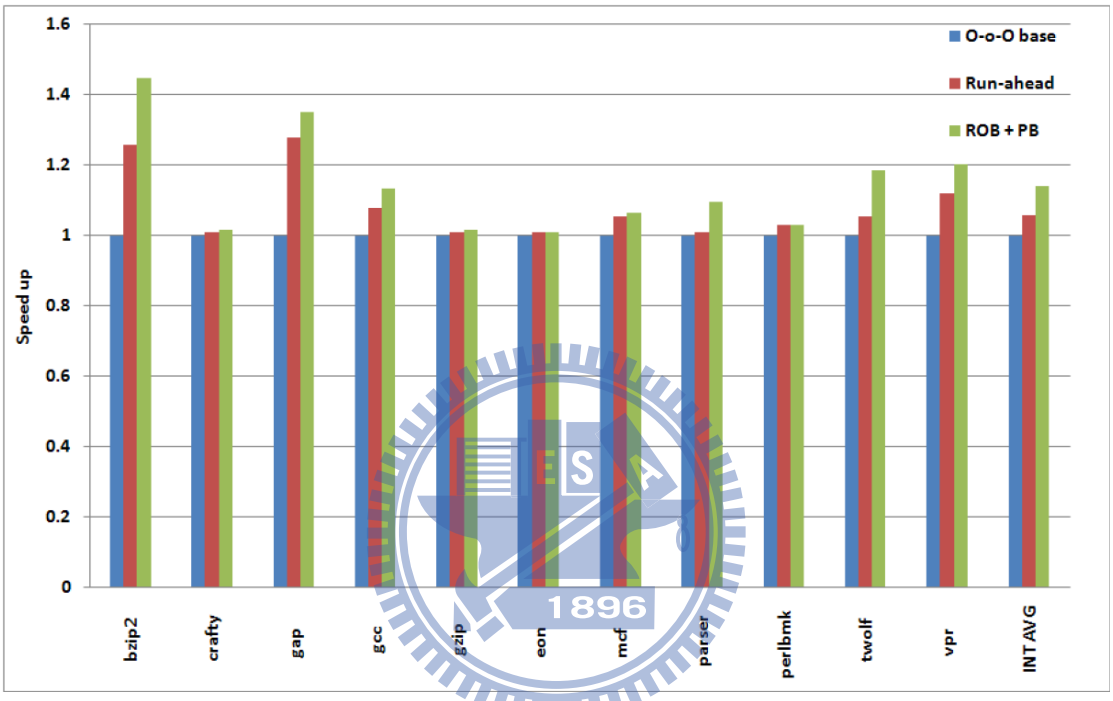


Figure 19 Overall performance (IPC) on SPEC INT2000

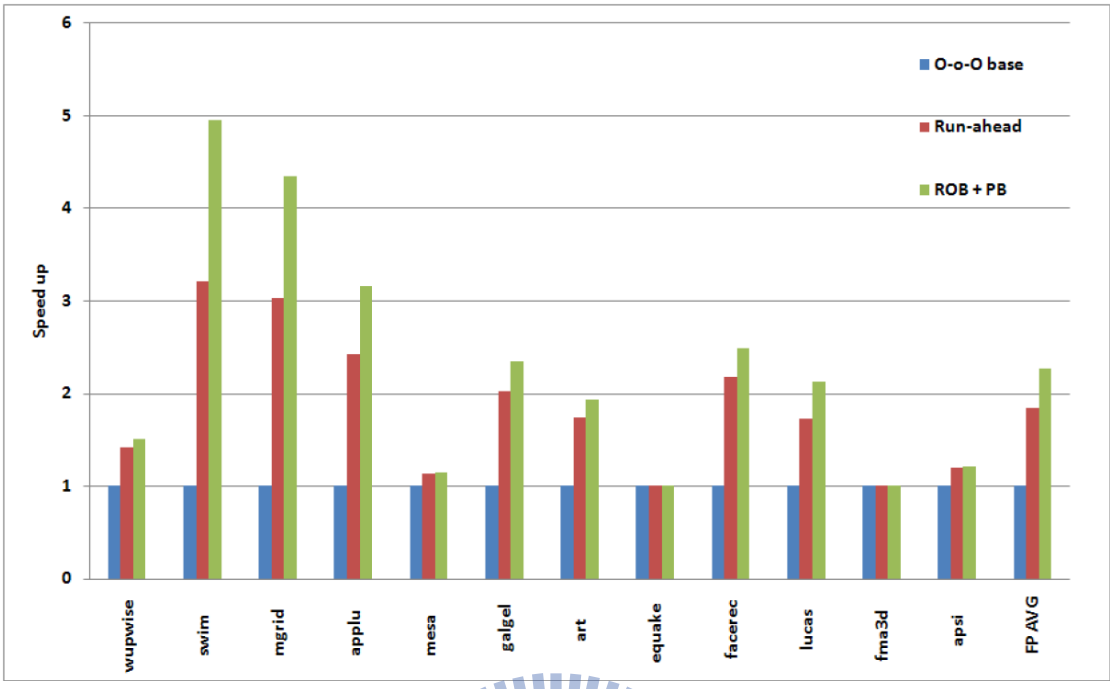


Figure 20 Overall performance (IPC) on SPEC FP2000

In the following table we can see that in average over 50 percent instructions carry execution results as they enter the preserving buffer. The more the execution results are preserved, the more chances the performance gain we may achieve.

However, high percent of instructions with values do not definitely mean higher performance gain. This is because the instructions with values are not necessary in the critical path of the program.

Benchmarks	Avg. runahead inst.	Avg. valid inst.	percent of valid
bzip2	534.19	370.96	69.44%
crafty	371.17	335.23	90.32%
gap	206.81	168.37	81.41%
gcc	641.52	498.09	77.64%
gzip	381.13	283.44	74.37%
eon	195.83	180.47	92.16%
mcf	234.02	117.87	50.37%
parser	632.20	409.58	64.79%
perlbmk	57.85	44.14	76.30%
twolf	311.86	150.07	48.12%

vpr	535.81	369.02	68.87%
wupwise	352.02	243.38	69.14%
swim	645.07	438.54	67.98%
mgrid	1161.71	664.90	57.23%
applu	909.24	487.16	53.58%
mesa	191.50	178.94	93.44%
galgel	508.93	302.28	59.40%
art	216.43	96.84	44.74%
equake	508.58	503.77	99.05%
facerec	363.76	202.24	55.60%
lucas	292.86	258.11	88.13%
fma3d	321.47	231.91	72.14%
apsi	167.31	80.78	48.28%

Table 6 percent of instructions with execution results

In the following table we can see the cache misses overlapped in the run-ahead mode. The more cache misses overlapped, the more performance gain the run-ahead mode can achieve.

However, some benchmarks, like the mcf, do not perform well. This is because the cache misses gather up in specific time points and even small window can overlap theses cache misses. Thus the effect of run-ahead mode is not obvious.

Benchmarks	L2 miss rate global	Avg. misses in runahead
bzip2	0.65%	18.63
crafty	0.02%	1.93
gap	0.26%	6.94
gcc	0.38%	4.41
gzip	0.05%	42.92
eon	0.00%	4.35
mcf	3.90%	21.68
parser	0.65%	4.91
perlbmk	0.00%	1.47
twolf	2.28%	7.87
vpr	1.59%	13.75
wupwise	1.75%	21.17

swim	10.70%	84.50
mgrid	5.43%	98.66
applu	9.38%	122.05
mesa	0.17%	8.57
galgel	5.61%	90.36
art	24.98%	43.20
equake	0.00%	1.07
facerec	2.48%	45.31
lucas	4.60%	32.56
fma3d	0.00%	18.46
apsi	1.22%	38.26

Table 7 average L2 cache misses in the runahead mode

4.3 Impact of Different Cache Sizes on Performance Comparison with the Original Run-ahead and Base Configuration

With the same configurations, we observe the trend of average performances as the L2 cache size is enlarged. Although the performance gain of the run-ahead execution decrease as the cache size increases, more than 5 percent performance gain remains as the L2 cache size reaches 8 mega bytes. The performance gain in the SPEC FP remains more than 50 percent.

Another observant is that the performance gain gradually decreases as the cache sizes are over 4 mega bytes. With the trend of enlarging cache sizes, our design is more efficient than just using very large caches.

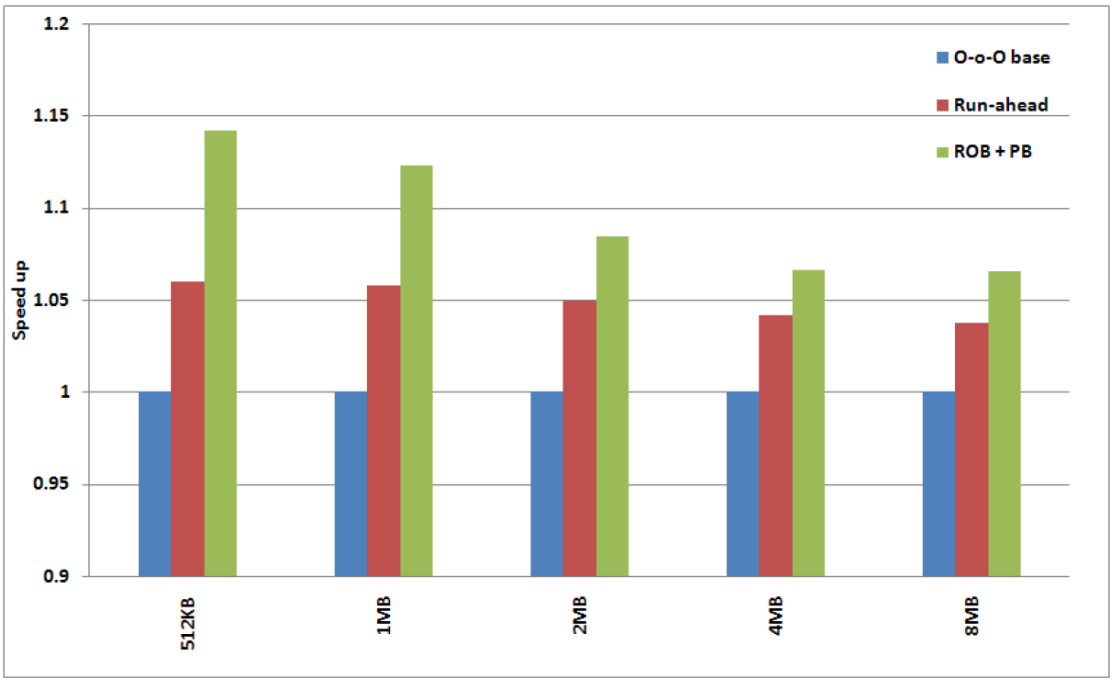


Figure 21 performance impact of different cache sizes in SPEC INT

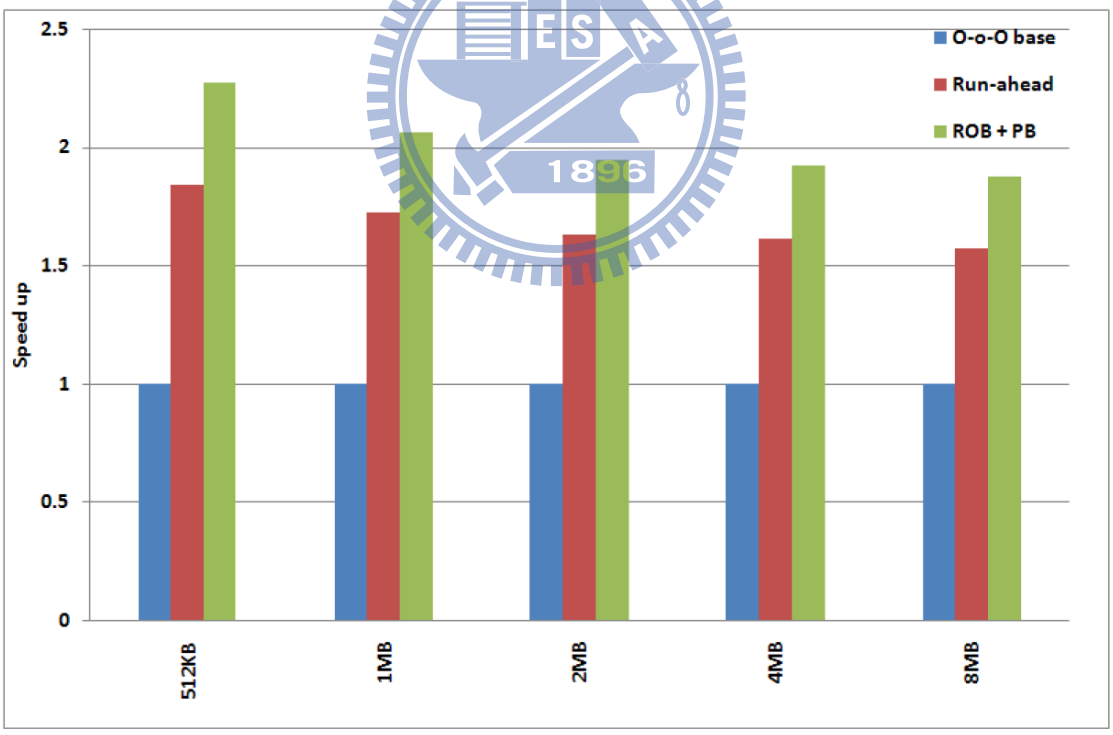


Figure 22 performance impact of different cache sizes in SPEC FP

4.4 Impact of Different Memory Latencies on Performance Comparison with the Original Run-ahead and Base Configuration

With the same configurations, we observe the performance gain in different memory latencies. The stall cycles caused by the L2 cache misses increase as the memory latency increase. In our simulation results the fact is obvious. In the configuration of longer memory latencies, the effects of overlapping cache misses are large.

However, the benefits of preserving execution results do not increase. This is because most instructions with execution results gather in the beginning of run-ahead mode. There are fewer execution results can be preserved as the run-ahead distance is getting longer and longer.

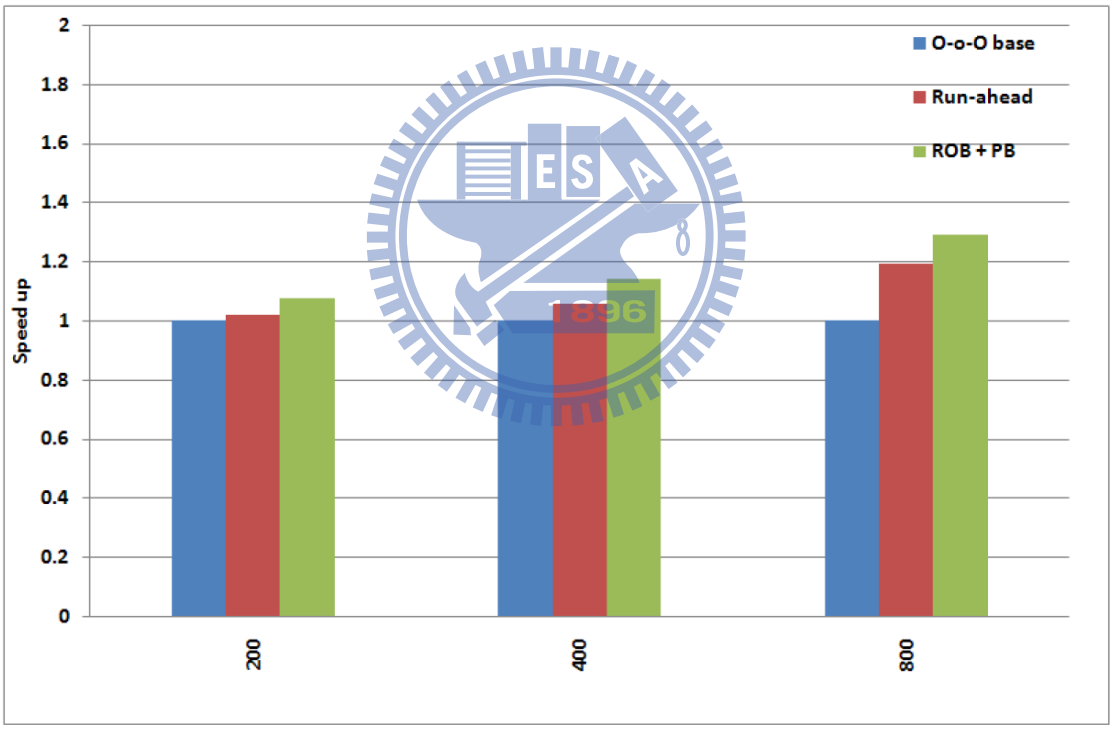


Figure 23 performance impact of different memory latencies in SPEC INT

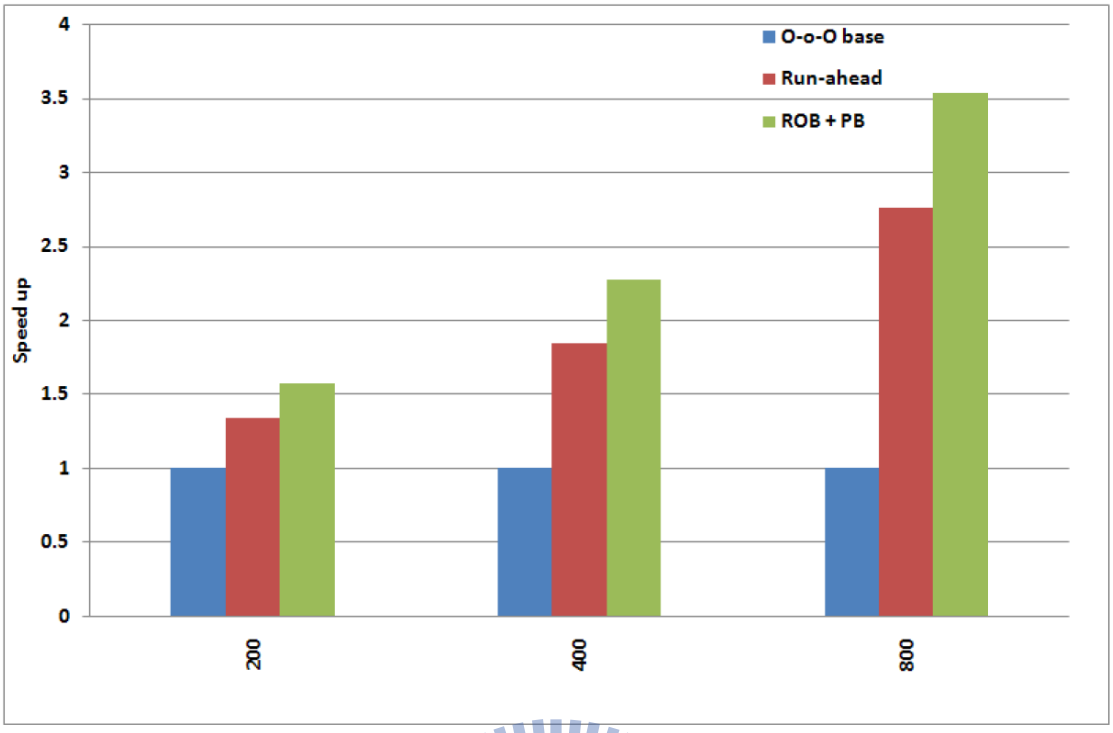


Figure 24 performance impact of different memory latencies in SPEC FP

4.5 Impact of Preserving Buffer Size in Different Memory Latency Configurations

Here we observe the effects of preserving buffer sizes on performance. The idea preserving buffer size should be the issue width multiplies the memory latency.

We can see that the performance is saturate as the instruction buffer size is up to 512 in the configuration that issue width is 4 and the memory latency is 200 cycles. However, with the growing memory latencies, the large buffer is necessary to achieve high performance. This is because there are more chances to overlap more cache misses and preserve more values.

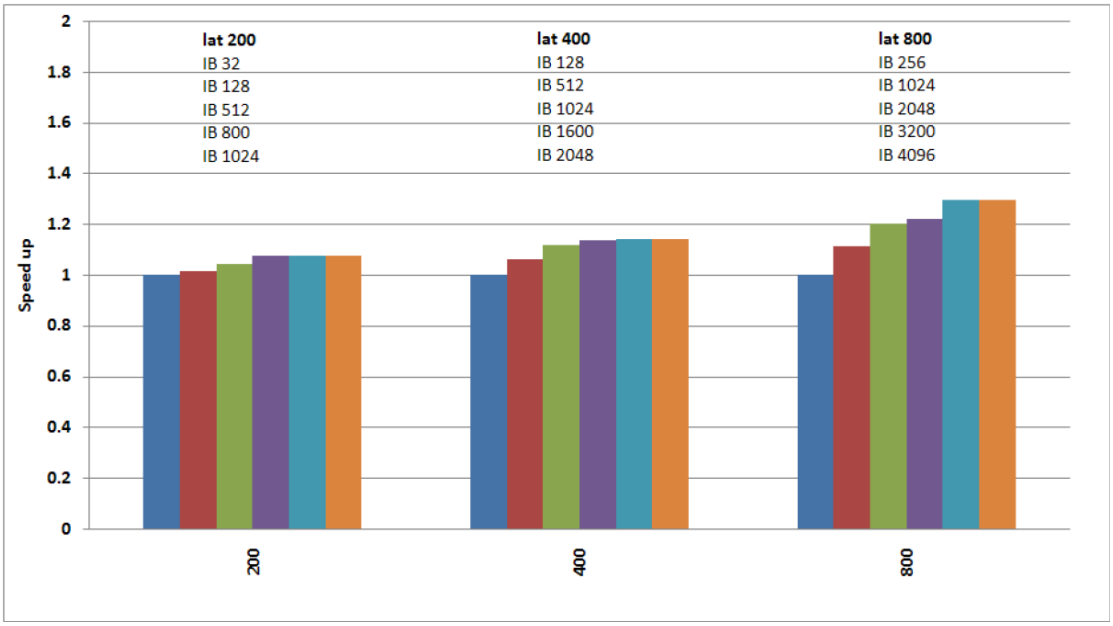


Figure 25 Impact of instruction buffer sizes on SPEC INT2000

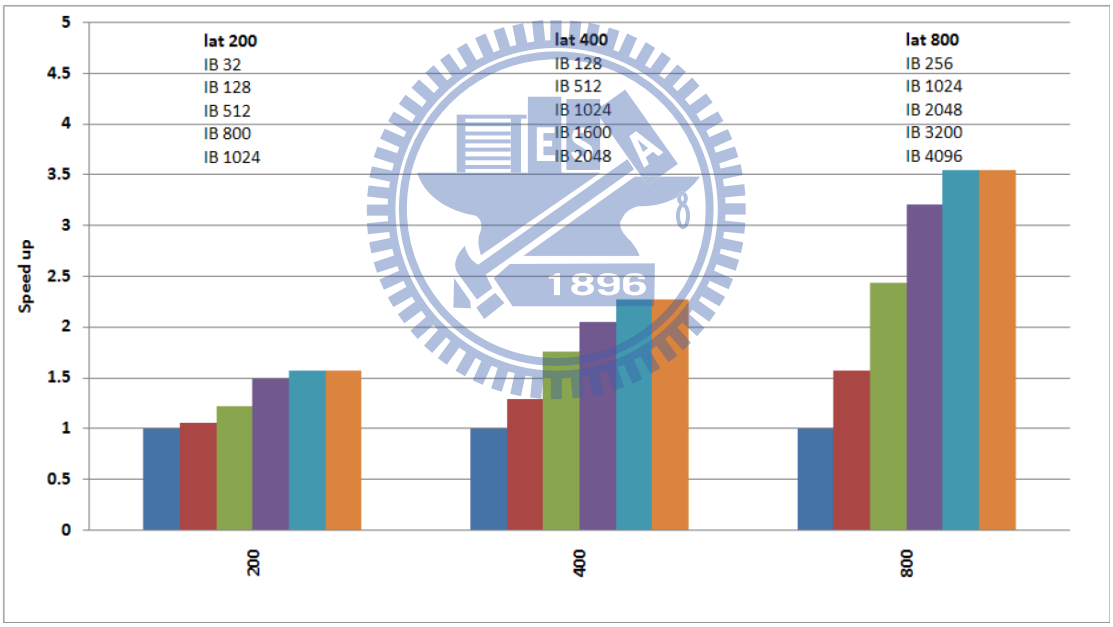


Figure 26 Impact of instruction buffer sizes on SPEC FP2000

4.6 Impact of RAC sizes

The factor that causes performance loss of RAC is the number of load instructions depend on store data evicted from the RAC. A larger RAC can relieve the situation of cache line eviction thus the better performance can be achieved.

Our simulation use different cache size configurations with fixed 4-way set associative. The results show that a RAC with 1K byte of 4-way set associative can achieve the performance close to the perfect RAC.

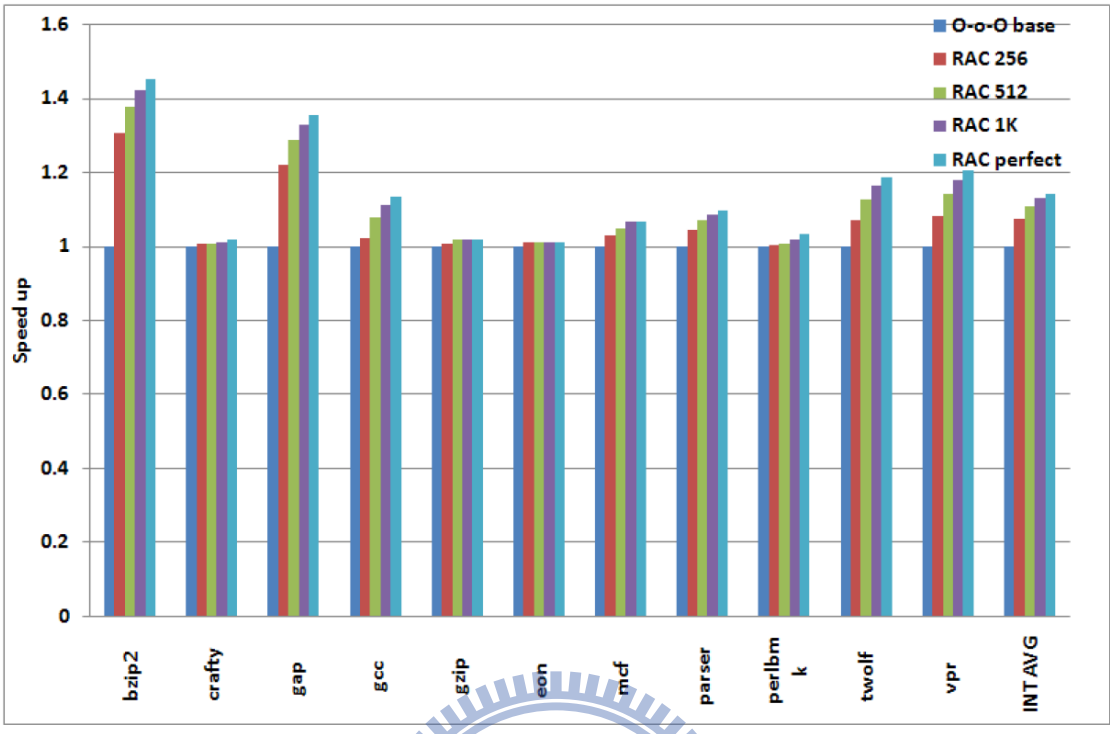


Figure 27 effect of RAC sizes on SPEC INT2000

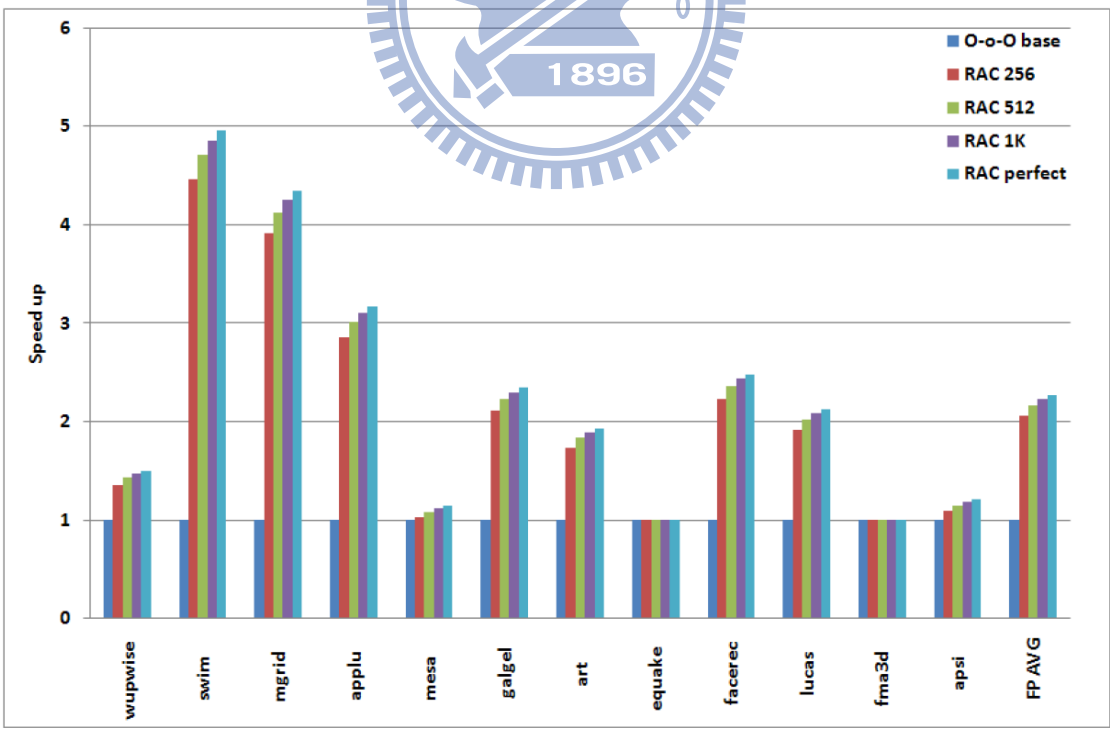


Figure 28 effect of RAC sizes on SPEC FP2000

4.7 Performance Comparison with Large Instruction Windows

Here we compare our design with the conventional design with large instruction window to see the performance we can achieve. Four large window configurations are simulated. They are o-o-o 256,512,1K and 2K, which have instruction window of 256,512,1K and 2K respectively.

In SPEC INT, the performance of our design is between the o-o-o 256 and o-o-o 512. In SPEC FP, the performance is between the o-o-o 512 and o-o-o 1K. This show the cache misses affect the performance of run-ahead execution largely.

In many of FP benchmarks, like the SWIM and the MGRID benchmarks, the performance is far from the ideal configurations. This is because the large instruction windows can always explore distant ILP while our design explores future instructions only in the run-ahead mode. Besides, the run-ahead mode seeks for future instruction gradually while the large instruction windows check for ready instructions in parallel.

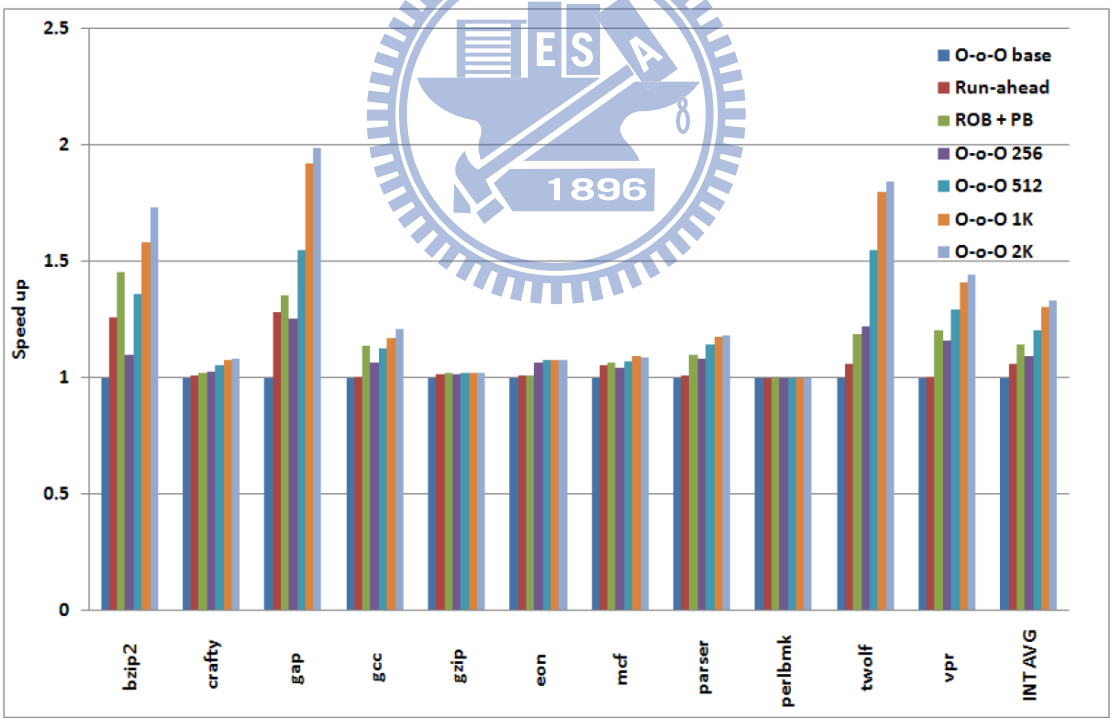


Figure 29 performance comparison with large windows in SPEC INT

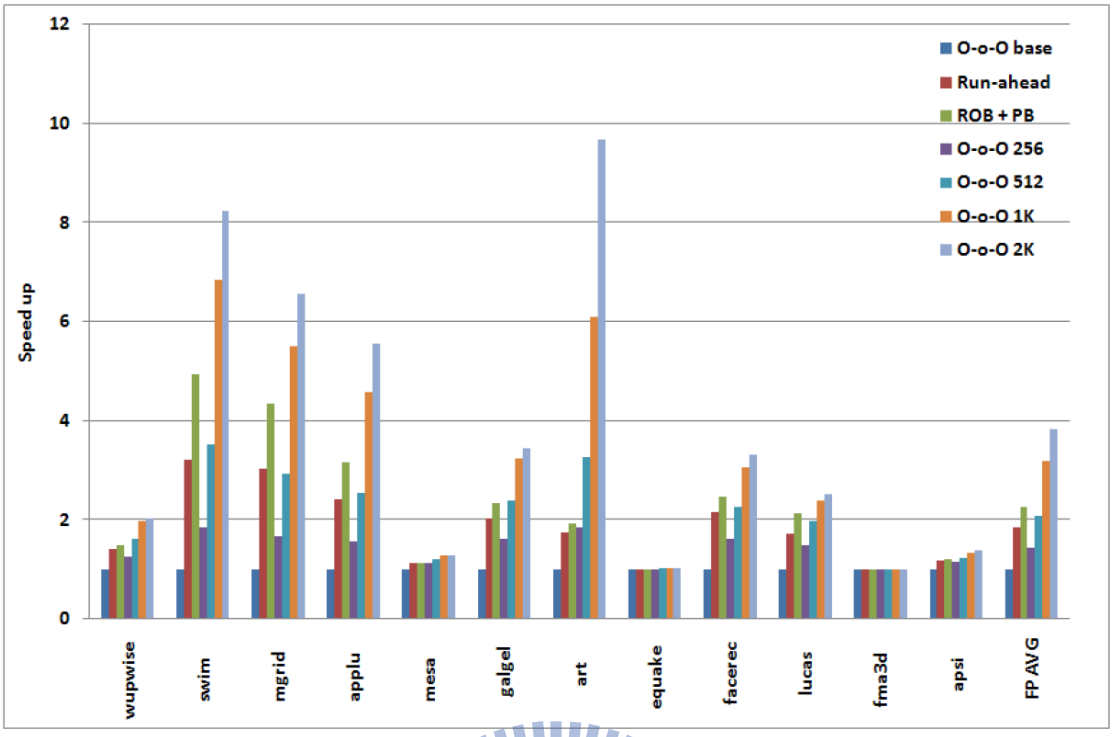


Figure 30 performance comparison with large windows in SPEC FP

4.8 Performance Comparison with Perfect Branch Prediction

In this section we simulate overall performance with perfect branch prediction. The results show that the performance of large instruction window depends on the branch prediction heavily. With accurate branch prediction, the large instruction window configuration achieves higher performance.

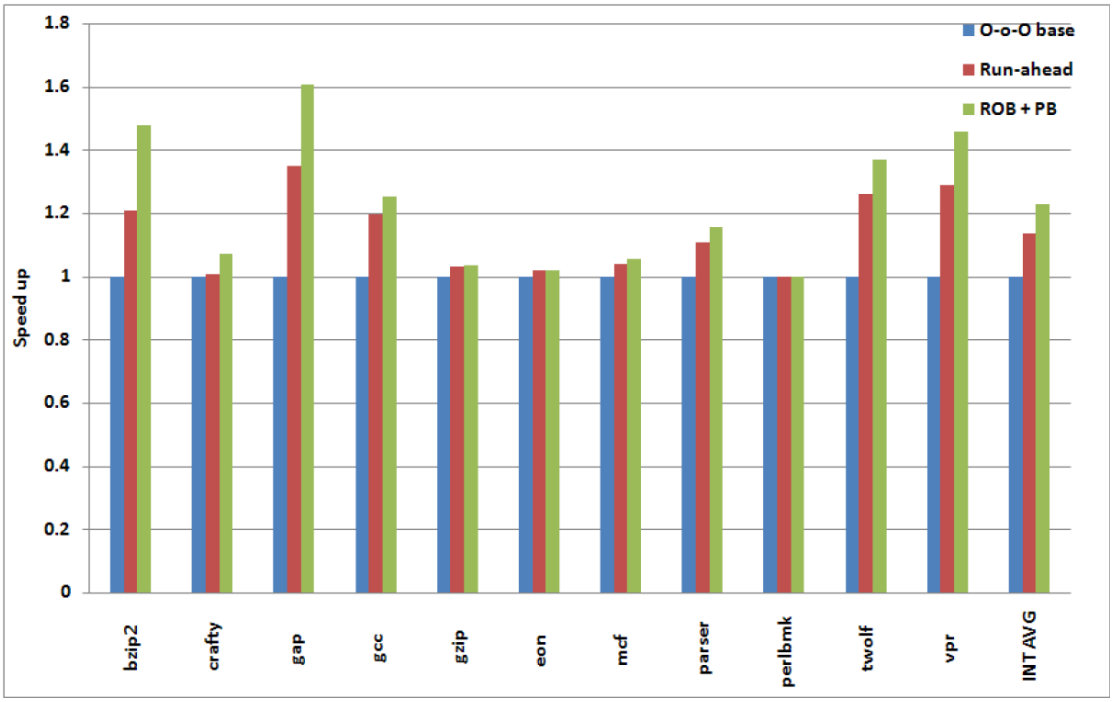


Figure 31 effect of branch prediction on SPEC INT2000

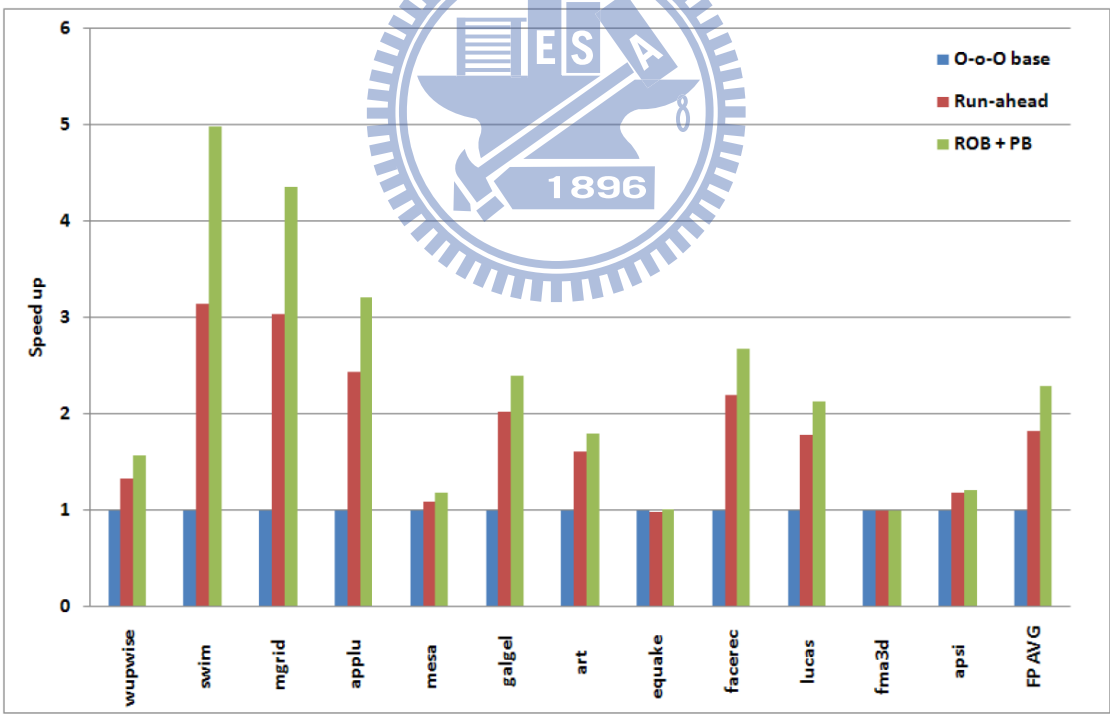


Figure 32 effect of branch prediction on SPEC FP2000

Benchmarks	Branches	Unresolved branches	unresolved + mispredicted branches
bzip2	52.90	13.33	0.19

crafty	44.93	6.34	0.34
gap	12.23	1.04	0.05
gcc	84.61	36.02	0.65
gzip	20.70	2.39	0.16
eon	14.21	0.27	0.05
mcf	56.15	26.53	0.11
parser	113.12	55.47	0.48
perlbmk	3.83	2.12	0.42
twolf	42.98	19.17	0.27
vpr	72.26	28.79	0.51
wupwise	51.79	21.16	0.17
swim	8.94	0.01	0.00
mgrid	3.24	0.00	0.00
applu	2.67	0.00	0.00
mesa	16.73	1.25	0.00
galgel	30.99	0.12	0.00
art	19.27	0.73	0.02
equake	84.17	0.43	0.01
facerec	23.76	0.61	0.00
lucas	23.57	6.91	0.03
fma3d	51.31	17.15	0.50
apsi	4.67	0.01	0.00

Table 8 branch instructions in the run-ahead mode

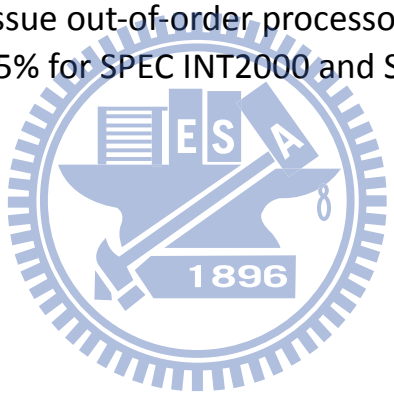
5. Conclusion and Future Work

The traditional large instruction window design can exploit higher MLPs and ILPs but it is difficult to use large structures while keeping cycle time small.

Run-ahead execution can continue exploring future instructions as encountering long latency operations, but, due to difficulty to maintain dependencies across hundreds and thousands of instructions, it causes many wasted computations.

This paper introduces a novel design that continues to explore future instructions while long latency operations happen. Based on a ROB-based processor, our design can achieve higher MLPs and ILPs by adding an easy FIFO buffer with linear complexity, an extra run-ahead register file and a small run-ahead cache. Furthermore, our design is free from complex management.

Simulations of a 4-issue out-of-order processor can produce speedups of 10% and 15% for SPEC INT2000 and SPEC FP2000 over Run-ahead execution.



6. Reference

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith, “Complexity-effective superscalar processors,” in *ISCA’97: Proceedings of the 24th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 1997, pp. 206–218.

[2] E. Riseman and C. Foster, “The inhibition of potential parallelism by conditional jumps,” *Transactions on Computers*, vol. C-21, no. 12, pp. 1405–1411, Dec. 1972.

[3] P.Michaud, A. Seznec, and S. Jourdan, “An exploration of instruction fetch requirement in out-of-order superscalar processors,” *Int. J. Parallel Program.*, vol. 29, no. 1, pp. 35–58, 2001.

[4] R. Canal and A. Gonz´alez, “A low-complexity issue logic,” in *ICS ’00: Proceedings of the 14th international conference on Supercomputing*. New York, NY, USA: ACM Press, 2000, pp. 327–335.

[5] P. Michaud and A. Seznec, “Data-flow prescheduling for large instruction windows in out-of-order processors,” in *HPCA ’01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2001, p. 27.

[6] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt, “A scalable instruction queue design using dependence chains,” in *ISCA ’02: Proceedings of the 29th annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 318–329.

[7] I. Kim and M. H. Lipasti, “Half-price architecture,” in *ISCA ’03: Proceedings of the 30th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 2003, pp. 28–38.

[8] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, “A large, fast instruction window for tolerating cache misses,” in *ISCA ’02: Proceedings of the 29th annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 59–70.

[9] A. Cristal, D. Ortega, J. Llosa, and M. Valero, “Out-of-order commit processors,” in *HPCA ’04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2004, p. 48.

[10] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, “Continual flow pipelines,” in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 2004, pp. 107–119.

[11] 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 *HPCA ’03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 129.

[12] H. Akkary, R. Rajwar, and S. T. Srinivasan, “Checkpoint processing and recovery: Toward scalable large instruction window processors” in *MICRO ’03: Proceedings of the 36th International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 423.

[13] R. D. Barnes, S. Ryoo, and Wen-mei W. Hwu, “"Flea-flicker" Multipass Pipelining: An Alternative to the High-Power Out-of-Order Offense” in *MICRO ’05: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture.* Washington, DC, USA: IEEE Computer Society, 2005, p. 319.

[14] K. Sundaramoorthy, Z. Purser, and E. Rotenburg, “Slipstream Processors: Improving both Performance and Fault Tolerance” in *ACM SIGOPS Operating Systems Review*. New York, NY, USA: ACM Press, 2000, p. 257.

[15] T. Karkhanis and J. E. Smith, “A Day in the Life of a Data Cache Miss” in *Second Annual Workshop on Memory Performance Issues*, 2002.

[16] M. Pericas, A. Cristal, R. Gonzalez, D. A. Jimenez and M. Valero, “A Decoupled KILO-Instruction Processor,” in *HPCA ’06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, p. 53.

[17] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, “The Microarchitecture of the Pentium 4 Processor” in *Intel Technology Journal*. February 2001.

[18] V. Agarwal, M.S. Hrishikesh, S. W. Keckler and Doug Burger, “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures,” in *ISCA ‘2000: Proceedings of the 27th annual International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2000, p. 248.

[19] Wm. A. Wulf and S. A. McKee, “Hitting the Memory Wall: Implications of the Obvious,” in *Computer Architecture News, 1995*.

[20] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramierz, M. Pericas and M. Valero, “Kilo-Instruction Processors: Overcoming the Memory Wall,” in *MICRO ’05: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture.* Washington, DC, USA: IEEE Computer Society, 2005, p. 48.

[21] S. P. Vanderwiel and D. J. Lilja, “Data Prefetch Mechanisms,” in *ACM*

Computing Surveys 2000.

[22] H. Zhou, “Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window,” in *PACT ’05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. p. 231-242.

[23] D. M. Tullsen, S. J. Eggers and H. M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. June 1995, p. 392-403.

[24] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer and J. Joyner, “Power5 System Microarchitecture,” in *IBM Journal of Research and Development*, 49(4), 2005.