AMERICAN UNIVERSITY OF BEIRUT

TUNING THE CONTINUAL FLOW PIPELINE ARCHITECTURE

by
KOMAL MADAIAH JOTHI

A dissertation
submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
to the Department of Electrical and Computer Engineering
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
February 2014

AMERICAN UNIVERSITY OF BEIRUT

TUNING THE CONTINUAL FLOW PIPELINE ARCHITECTURE

by
KOMAL MADAIAH JOTHI

Approved by:

Haitham Akkary

Dr. Haitham Akkary, Associate Professor — Advisor
Electrical and Computer Engineering

[illegible]

Dr. Ayman Kayssi, Professor — Member of Committee
Electrical and Computer Engineering

Ali Chehab

Dr. Ali Chehab, Associate Professor — Member of Committee
Electrical and Computer Engineering

[illegible] on behalf of Prof. Saghir

Dr. Mazen Saghir, Associate Professor — Member of Committee
Electrical and Computer Engineering, Texas A&M University at Qatar

[illegible] on behalf of Dr. Alameldeen

Dr. Alaa Alameldeen, Adjunct Faculty — Member of Committee
Electrical and Computer Engineering, Portland State University

Date of dissertation defense: February 21, 2014

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: JOTHI KOMAL MADAIAH

Last	First	Middle
JOTHI	KOMAL	MADAIAH

○ Master's Thesis ○ Master's Project [✓ Doctoral Dissertation]

[✓] I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, **three years after the date of submitting my thesis, dissertation, or project,** to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

[illegible] MARCH 12, 2014

Signature Date

ACKNOWLEDGMENTS

I would like to thank the Department of ECE and the then chair Dr. Karim Kabalan for giving me a chance to pursue my PhD at American University of Beirut. Thanks to the committee members, Dr. Ayman Kayssi, Dr. Ali Chehab, Dr. Mazen Saghir and Dr. Alaa Alameldeen for their feedback and comments.

Many thanks to Mageda Sharafeddine for the help with the power model. Also thanks to Saadallah Itani from the IT department for all the help with the servers. This work involved running a lot of simulations and the dedicated machines eased a lot of pressure. Thanks to Rabab Abi Shakra from the ECE Department and Lara Touma from the Dean's office for all the administrative help. Thanks to Nadine Knesevitch from the AUB library for her patience and help with formatting the final report.

Special thanks to my parents and family for their understanding and always being there for me.

These four years of my PhD were the happiest years of my life. I am very thankful to all my friends in Beirut for having made this such a memorable journey. Finally, I am most thankful to my advisor Dr. Haitham Akkary for what has been a thoroughly enjoyable ride.

AN ABSTRACT OF THE DISSERTATION OF

Komal Madaiah Jothi for Doctor of Philosophy
Major: Electrical and Computer Engineering

Title: Tuning the Continual Flow Pipeline Architecture

One of the main factors that impacts performance of general purpose computer processors is misses to the data cache. Conventional techniques used in modern processors - building wide superscalars and large instruction buffers to hide the latency of these misses and keep the processor units busy - are not suitable for present and next generation processors that need to cater to high energy efficiency demands.

Continual Flow Pipeline (CFP) allows a processor core to handle hundreds of in-flight instructions without increasing cycle-critical pipeline resources. When a load misses the data cache, CFP checkpoints the processor register state and then moves all miss-dependent instructions into a low complexity waiting buffer to unblock the pipeline. Meanwhile, miss-independent instructions execute normally and update the processor state. When the miss data returns, CFP replays the miss-dependent instructions from the waiting buffer and then merges the miss dependent and independent execution results.

CFP was initially proposed for cache misses to DRAM. In that work, the miss-independent and miss-dependent instructions execute at different times separated by a pipeline flush in between, based on the timing of the load miss event and the data arrival event.

In this thesis, we focus on reducing the execution overhead of CFP by avoiding the pipeline flush and executing dependent and independent instructions concurrently. The goal of these improvements is to gain performance by applying CFP to L1 data cache misses that hit the last level on-chip cache.

However, we see that when CFP is applied to L1 data cache misses, many applications or execution phases of applications incur excessive amount of replay and/or rollbacks to the checkpoint. This frequently cancels benefits from CFP and reduces performance.

We mitigate this issue by using a novel virtual register renaming substrate, and by tuning the replay policies to eliminate excessive replays and rollbacks to the checkpoint. We describe these new design optimizations and show, using Spec 2006 benchmarks and microarchitecture performance and power models of our design, that our Tuned-CFP architecture improves performance and energy consumption over previous CFP architectures by ~10% and ~8%, respectively. We also demonstrate that our proposed architecture gives better performance return on energy per instruction compared to a conventional superscalar as well as previous CFP architectures.

CONTENTS

ILLUSTRATIONS

TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Over the last 20 years, the range of applications that can be run on a computer processor have evolved considerably. Some tasks which were considered impractical in the past, for example, video conferencing between two parties or streaming high quality multimedia, are being done with ease. So with the never ending demands placed by upcoming applications on processors, the onus is on computer architects to design better performing processors. In the past, the goal of a computer architect was fairly straight forward. A processor was designed to run as fast as possible without any compromises. These days with the battery life of mobile devices being critical and large budgets dedicated to cooling servers, energy consumption is an equally important aspect.

The need for energy efficient performance has led to the proliferation of multi-core processors, with the first multicore processor released by IBM in 2001. The energy efficiency of multi-core designs comes from building distributed structures as against centralized ones, thus avoiding the quadratic costs associated with scaling the centralized structures for single-thread performance. For instance, four structures of size N consume less energy than one structure of size 4N [38]. Similarly, four structures with one port each consume less energy than one structure with four ports, and four cores operating at 1GHz consume less energy than one core operating at 4GHz [1]. Multi-core processors are efficient either for throughput performance or when there are explicitly written parallel programs available to take advantage of multiple execution resources. In addition to parallel applications there are many important single-threaded applications already in use which are difficult to parallelize. Future multi-core processors should be able to run both parallel and single-threaded applications efficiently. No matter how many parallel cores are integrated on a chip, it is the performance of single thread programs, i.e. the slowest portion of code, which will dominate overall performance according to Amdahl's Law [16]. Hence it is critical to obtain single-thread performance using energy efficient methods.

So in view of these points, computer architects are presented with a new challenge: how to provide energy efficient single-thread performance for applications that are hard to parallelize, while placing on a single die as many cores as possible for high throughput performance.

One of the main factors that impact processor performance is data cache misses. Figure 1 shows the example of an execution sequence of eight instructions *A* to *H* in an in-order processor. Instructions *A* and *H* miss the cache and hence take a long time (shown as 150 cycles) to complete. Even though the rest of the instructions are independent of the cache miss and can complete soon, the overall execution time is dominated by the time needed to service the cache misses. Figure 2 shows the speedup of a processor core with perfect cache over a core with practical cache configuration. Many memory bound programs like gcc and twolf benefit immensely from not having to stall because of cache misses. How can these stalls be avoided?

Since the introduction of Intel P6 architecture in 1996 [39], the capacity of instruction buffers, such as reorder buffers, reservation stations and load and store queues [44], and the size of on-chip caches have kept increasing with every new generation of out-of-order core. The motivation behind this continuous evolution

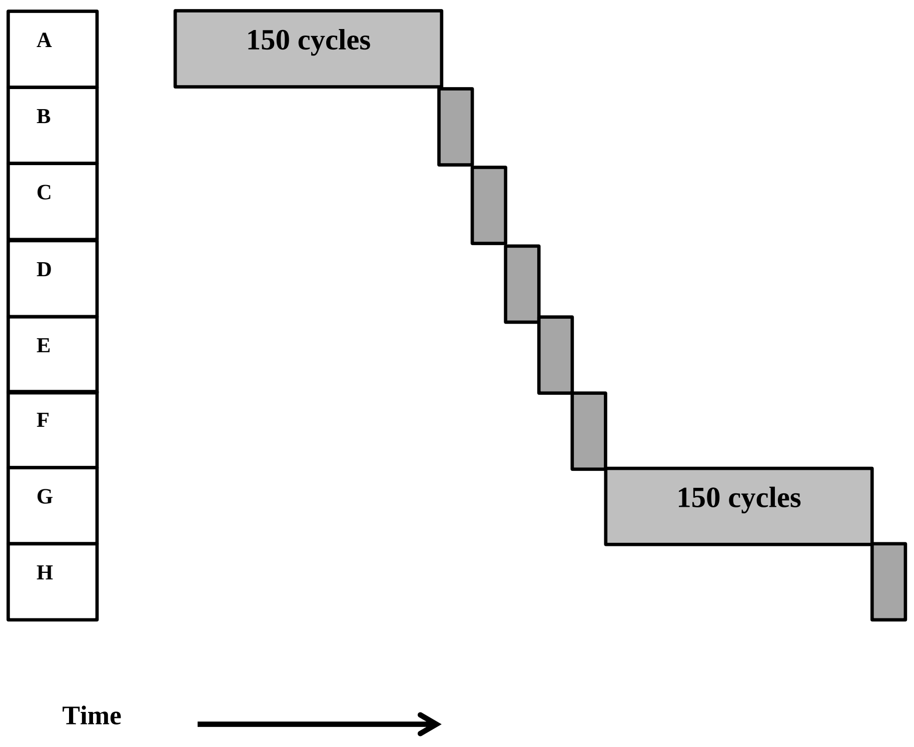


Figure 1. Execution sequence of an in-order processor when instructions miss the cache

towards larger buffers and caches has been performance of single-thread applications. Achieving performance this way has come at the expense of area, power, and complexity. At the circuit level, larger L1 data caches and multi-ported, timing critical instruction buffers, such as the store queue, the reorder buffer, the register file and the reservation stations, have become increasingly difficult to design while maintaining high clock rates. Designers have had to increase the degree of pipelining to meet cycle time, for example adding pipe stages to the first level data cache and the store queue.

The critical circuit paths introduced by larger instruction buffers and the increased pipelining have led to high complexity in the logic design as well. In fact, core complexity has risen so much that adding almost any new performance feature

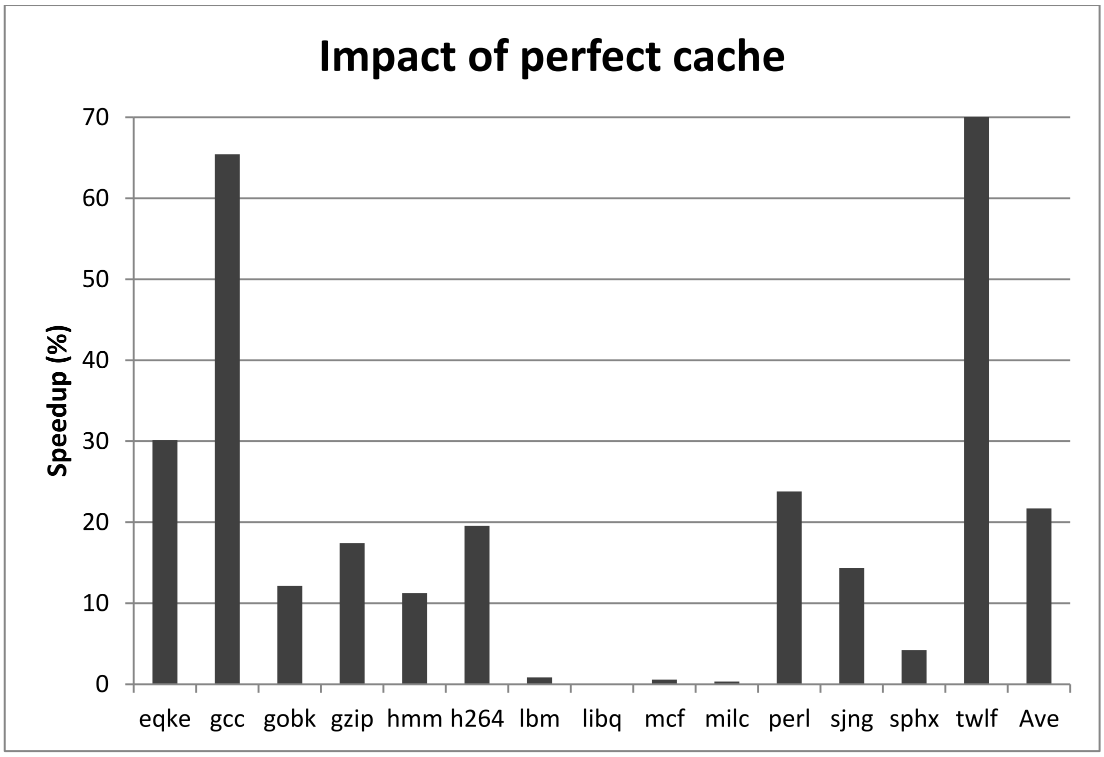


Figure 2. Impact of a perfect cache on performance

requires significant design validation effort. By increasing instruction buffer sizes designers have been seeking performance benefits from three different sources:

1. With larger buffers, the scheduling hardware has a larger pool of instructions from which to dynamically identify and schedule independent instructions concurrently, thus taking advantage of the wide pipeline and multiple functional units.

2. Increased instruction level parallelism from larger instruction windows reduces the impact on performance of stalls from multi-cycle instructions, e.g. floating point, load, multiply, or divide instructions.

3. Finally, larger buffers allow hardware to find instructions to execute behind very long latency instructions, such as loads that miss the data cache, thus reducing the impact of cache miss data hazards on performance.

For two decades, increasing instruction buffers has provided good performance improvement due to the benefits listed above. However, we have reached an end point [1]. Current buffer sizes are more than sufficient for code that hits the L1 data cache (benefits 1 and 2 above), and way too small for code that misses the L1 data cache (benefit 3 above). Load latency to the last level cache on current multicore processors is more than 20 clock cycles, and latency of load miss to DRAM, even in the best case of on-chip DRAM controller, is significantly more than hundred cycles. It is simply not practical to increase instruction buffer sizes to the capacity necessary to handle long load latencies to the last level cache or to DRAM.

Figure 3 shows the percentage of reorder buffer stalls and reservation station stalls over total execution time in a typical superscalar processor. From the figure it is clear that even with reasonably sized buffers, conventional processors fill up soon and quickly run out of instructions to retire in case of the reorder buffer, or execute in case of reservation stations and spend a considerable fraction of execution time sitting idle.

A different design strategy would be to size the instruction buffers to the minimum size necessary to handle the common case of L1 data cache hit and use less circuit critical and power hungry mechanisms to handle code that misses the L1 data cache, assuming such mechanisms exist. In fact, there have been various studies of latency-tolerant architectures that target reducing the impact of data cache misses on performance, without having to increase instruction buffer sizes [7][9][10][17][32][37][46]. These architectures have common characteristics, but vary in implementation details. They all tolerate cache miss latencies by managing buffer resources in a non-conventional way. Instructions that depend on a cache miss do not block the execution pipeline. Instead, they move with any ready source registers into a

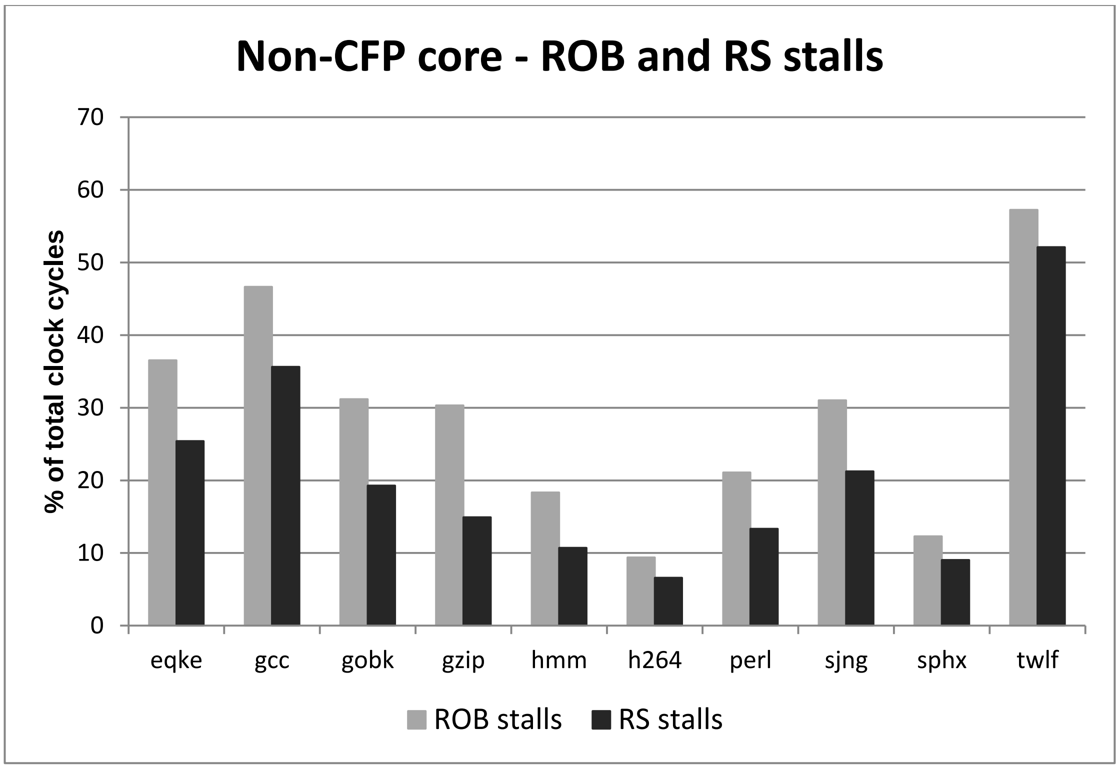


Figure 3. Reorder buffer stalls and reservation station stalls in conventional processors

buffer outside the execution pipeline, implemented as SRAM FIFO buffer to minimize area and power, thus allowing look-ahead execution and pseudo-retirement of independent instructions to continue without stalling the pipeline. When the miss data is loaded into the cache, the miss dependent instructions are dispatched back from the FIFO buffer into the execution pipeline. When the execution of the miss dependent instructions completes, results of the independent and dependent instructions are merged together and then execution resumes normally until another load instruction misses the data cache. Since independent instructions complete and pseudo-retire before older instructions that depend on the data cache miss, all these architectures use checkpoints [2][3] to recover from branch mis-predictions and exceptions that depend on the miss.

All previous latency-tolerant architectures either target last level cache misses to DRAM [9][10][32][46] or use small in-order cores for high-throughput many-core processors [7][17][37]. Unfortunately, in-order architectures provide limited single-thread performance as compared to out-of-order cores, and latency-tolerant performance techniques that target cache misses to DRAM provide less benefit nowadays than on previous generations of processors. This is because large on-chip caches and integrated on-chip memory controllers, typical on current main stream processors, reduce (but not completely eliminate) the overall impact of accesses to DRAM. Still unexploited in latency-tolerant out-of-order core proposals are misses to the L1 data cache that hit L2 and L3 on-chip cache. These misses cause shorter delays than misses to DRAM, but they occur a lot more frequently and consequently have as much impact on performance as misses to DRAM on many applications. Therefore, new out-of-order cores that can handle not only cache misses to DRAM, but cache misses at all levels of the cache hierarchy, without having to increase buffer sizes, are needed to increase energy efficiency and performance of single-thread applications.

Continual Flow Pipeline architecture [46] is one such proposal that attempts to tolerate long latencies of only last level cache misses that go to DRAM. In that work, the miss-independent and miss-dependent instructions execute at different times, based on the timing of the load miss event and the data arrival event. Switching between the two executions is costly because it involves a pipeline flush, making this proposal unsuitable for L1 misses that hit the on-chip cache. To avoid this costly pipeline flush, we propose to execute the independent and dependent instructions simultaneously, thus making this proposal, which we call Simultaneous-CFP, more suitable for first level data cache misses. However, in Simultaneous-CFP, many applications or execution phases of applications incur excessive amount of replay and rollbacks to checkpoints because of miss-dependent branches that mis-predict. Excessive replay increases the chance of replaying mis-predicted branches and consequently rollbacks to the checkpoints. This frequently cancels any desired improvement resulting from S-CFP handling of L1 data cache misses and can even cause performance degradation. In order to overcome these limitations, we use a novel virtual register renaming substrate [24][41] and fine tune the replay policies to mitigate excessive replays and rollbacks to the checkpoint. We call this Tuned-CFP processor core architecture.

1.2. Dissertation Contribution

In this dissertation, we propose a processor core architecture that overcomes the limitations of previous latency tolerant processor architectures and make the following contributions –

1. We introduce novel out-of-order execution algorithms that extend Continual Flow Pipelines [46] to allow L1 data cache miss independent instructions and dependent instructions, widely separated within a single-thread program, to execute simultaneously in the core pipeline. This allows our architecture to handle effectively first level cache misses that hit the higher level on-chip cache, as well as cache misses to DRAM. We call this architecture Simultaneous Continual Flow Pipeline (S-CFP).

2. To support the Simultaneous-CFP core architecture, we present novel algorithms for performing fast register results integration and load-store memory ordering while executing simultaneously in the core pipeline non-contiguous miss dependent and independent instructions.

3. We see that Simultaneous-CFP incurs a lot of circuit activity due to excessive replay of load miss dependent instructions. In an attempt to fine tune the replay policies, we keep the miss-dependent instructions in the reservation stations as long as possible after the miss before they are evicted to the waiting buffer. This reduces the number of miss-dependent instructions that are replayed in case of medium latency load misses, which are those loads that miss the L1 data cache but hit the L2 data cache.

4. We remove the reorder buffer from the replay loop in order to reduce the replay latency of miss-dependent instructions that are evicted to the waiting buffer and thus reduce the total execution time. For this, we use an order list of instructions in the reservation stations to order miss-dependent instructions when they have to move into the waiting buffer.

5. On CFP architectures, all miss-dependent instructions have to be moved into the waiting buffer and then replayed, once the load miss is moved into the waiting buffer. This is necessary since when the miss load pseudo commits and moves into the waiting buffer, it releases its renamed destination register ID (also called tag). This breaks the dependence links between the miss load and its dependents, requiring the full dependence chain to be replayed and renamed again to re-establish the dependence links. In this dissertation, we use virtual register renaming [41], which allows partial replay of the miss load and its dependents, thus significantly reducing the number of replayed instructions and the total execution time.

6. On CFP architectures, mis-predicted branches that depend on load misses have very high mis-prediction penalty. The reason is that CFP recovers from these mis-predictions by rolling back execution to a checklepoint taken at the load miss. Reducing

the number of replayed instructions reduces the chance of encountering miss-dependent branch mis-predictions and their high recovery penalty. This improves performance significantly on benchmarks that have many branches that depend on load misses.

7. Branch instructions that move into the waiting buffer and later found to be mis-predicted not only degrade performance but also increase power consumption significantly because of the large number of wrong path instructions burning processor resources wastefully. This dissertation presents a hardware predictor that is used as a branch confidence mechanism to identify miss-dependent branches that are likely to mis-predict, and stalls the pipeline when such low confidence branches are moved into the waiting buffer. This prediction mechanism reduces execution waste and as a result power consumption on benchmarks that have many mis-predicted branches that depend on load misses.

8. Finally, using a microarchitecture performance simulator and architectural level power model, it shows that the optimized CFP architecture, which we call Tuned-CFP, improves execution time and energy consumption by 10% and 8% respectively over S-CFP architecture. This dissertation also shows that as the instruction buffer sizes are scaled over a wide range, the Tuned-CFP architecture consistently provides better performance return on energy per instruction compared to conventional reorder buffer superscalar architecture and previous latency tolerant architectures.

1.3 Dissertation Related Publications

1. Simultaneous continual flow pipeline architecture. In *Proceedings of 29th IEEE International Conference on Computer Design.* IEEE, 127-134. Oct 2011.

2. Virtual Register Renaming: Energy Efficient Substrate for Continual Flow Pipelines. In *Proceedings of the 23rd Great Lakes Symposium on VLSI.* ACM, 43-48. May 2013.

3. Tuning the Continual Pipeline Architecture. In *Proceedings of the 27th International Conference on Supercomputing.* ACM, 243-252. June 2013.

4. Streamlining the Continual Flow Processor Architecture with Fast Replay Loop. In *Proceedings of the IEEE International Conference on Computer as a Tool.* EUROCON. IEEE, 1821-1828. July 2013.

5. Tuning the Continual Pipeline Architecture with Virtual Register Renaming. In *ACM Transactions on Architecture and Code Optimization.* ACM, Accepted November 2013.

1.4 Document Organization

The rest of the document is organized as follows. Chapter 2 talks about related work on latency tolerance for cache misses. Chapter 3 talks about Simultaneous-CFP architecture and its limitations, and presents a limit study to measure the upper bound on performance when all its limitations are addressed. Chapter 4 talks about Tuned-CFP architecture which is built on a virtual register renaming substrate and also fine tunes many replay policies of Simultaneous-CFP. Chapter 5 details the memory organization of an architecture that supports simultaneous execution of miss-dependent and miss-independent instructions. Chapter 6 talks about the experimental setup. Chapter 7 presents experimental results. Chapter 8 makes qualitative comparisons between Tuned-CFP and earlier latency tolerant proposals. Chapter 9 concludes and talks about the scope for future work.

CHAPTER 2

BACKGROUND WORK

The objective of this dissertation is to provide energy efficient load latency tolerance for better performance. In this chapter we will talk about the techniques that have been proposed in literature and used in practice to provide load latency tolerance in processors. The first section will talk about the conventional techniques used in commercial processors and explain why future processors cannot employ the same techniques to gain performance. The second section will talk about unconventional techniques proposed in literature to support large instruction windows without scaling cycle critical buffers. The third and fourth sections will talk about two concepts that aid the application of CFP to first level misses and keep the processor energy efficient – one is simultaneous multithreading that allows miss-dependents and miss-independents to execute concurrently; the other is virtual register renaming, an essential substrate on which our latency tolerant architecture is built.

2.1 Conventional Mechanisms Used in Commercial Processors

As mentioned before, commercial processors have been building wider machines and larger instruction buffers and on-chip caches to provide performance by exposing instruction level parallelism (ILP) and memory level parallelism (MLP) [44]. ILP occurs when the processor finds some independent instructions to execute while the load miss data is being fetched from the higher level memory. MLP occurs when the processor overlaps the memory access latency of independent load misses. Both forms of parallelism are limited to how far the processor can look ahead in the program to seek instructions that can be scheduled for execution. The way this works is explained with an example below.

Figure 4(b) shows the working of a conventional out-of-order processor. To understanding its working, the execution sequence of an in-order processor is also shown above it in Figure 4(a). In Figure 4 there are eight instructions *A-H*. *A* and *G* miss the last level cache and hence take a long time to execute, with their latency shown as 150 cycles. The other instructions are of short latency and can finish quickly. Instruction *D* depends on *A* as shown by the arrow from *A* to *D*. In an in-order processor, there is no re-ordering of instructions based on them becoming ready, so the misses serialize and in the end the 8 instructions take a long time to finish.

An out-of-order processor can achieve some performance gains as shown in Figure 4(b). When *A* misses the cache and while its data is being fetched, independent instructions *B, C, E, F* and *H* are scheduled for execution. The MLP comes from the fact that both misses *A* and *G* can be overlapped together, so that *G* does not spend additional time waiting for its data. At the same time, some ILP comes from the fact that *B, C, E, F* and *H* can be executed while *A* and *G* wait for their data to arrive. From this example, it can be seen that a large out-of-order window benefits performance. However the structures that support out-of-order execution are not energy efficient.

The organization of a conventional reorder buffer based superscalar processor will be explained next.

2.1.1 Microarchitecture of a Conventional Superscalar

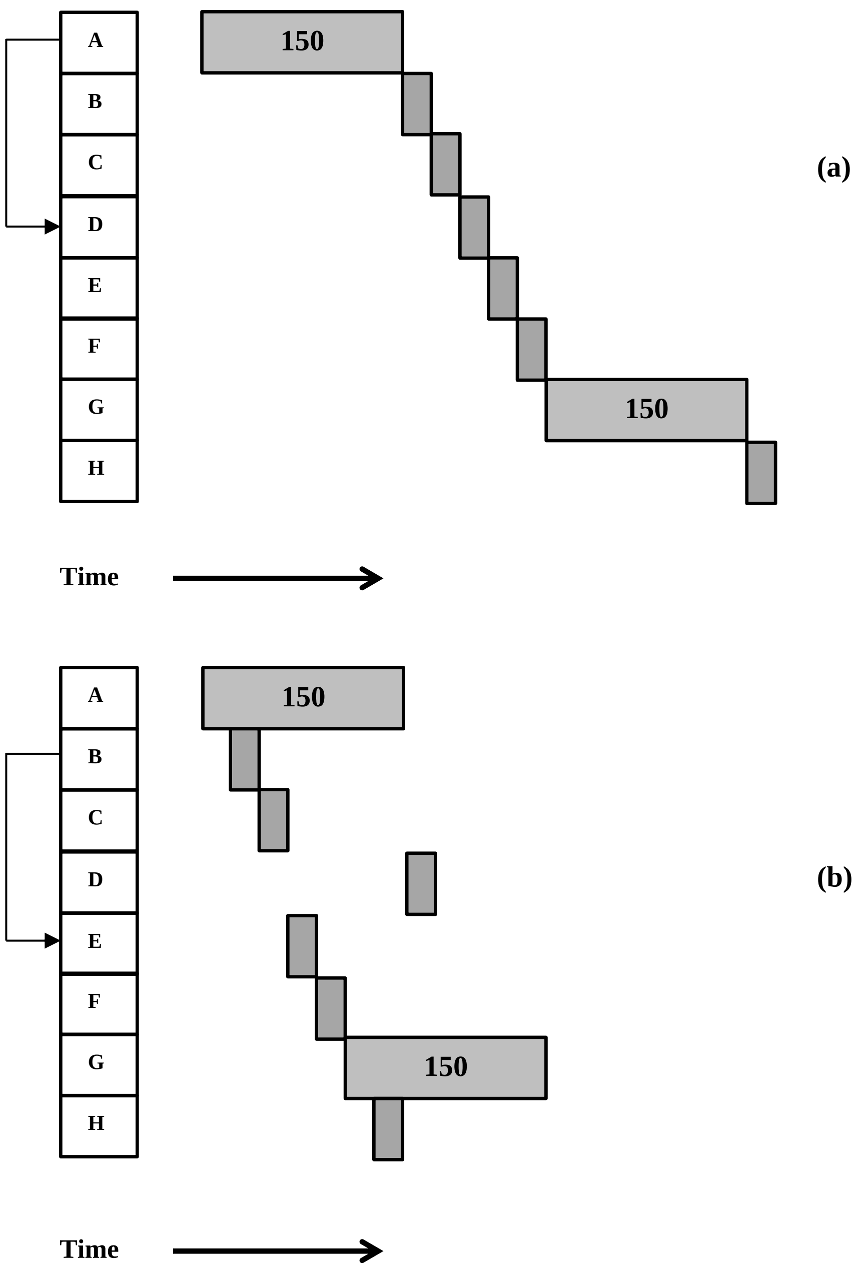
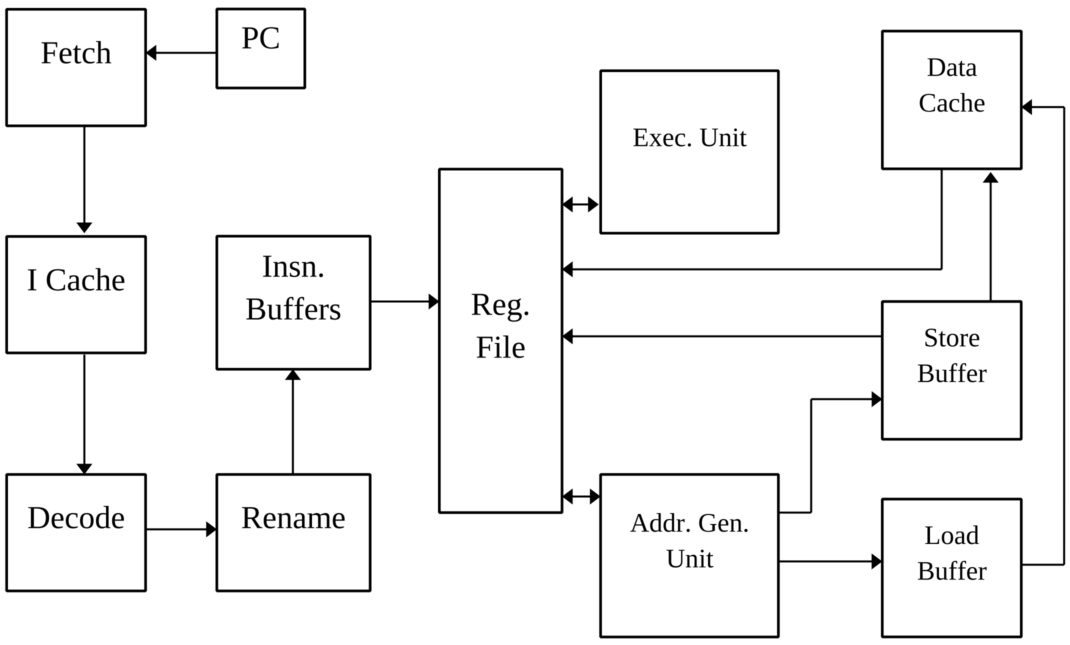


Figure 4. Execution sequence of an out-of-order execution processor

A conventional superscalar architecture and its pipeline are shown in Figure 5. This superscalar processor fetches a block of instructions every cycle. The fetch unit controls fetching using branch prediction, a branch target buffer (BTB), and a procedure



Fetch	Decode	Rename	Issue	Register Read	Execute	Retire

Figure 5. Block diagram and pipeline of a conventional superscalar processor

return address stack (RAS). Fetched instructions are decoded and passed to the rename unit, which maps logical registers into a set of physical registers, removing false dependencies. Instructions are then placed in the issue queue where they remain until they get a chance to execute, i.e. their source operands become available. The instructions execute out-of-order based on when their sources become ready and release their issue queue entries upon completion. The instructions are retired in program order after completing execution, freeing physical registers that are no longer needed. Loads and stores are issued to address generation units. Stores are then written into a store buffer, speculatively, where they remain until retirement. At retirement, stores update the data cache, but this time non-speculatively in program order. A load is sent with its address to the load buffer, store buffer, and the data cache unit. Its data is written back from the store buffer or cache to the register file, speculatively, if there is no address resolution conflict in the store buffer. If there is an address conflict or the memory dependence predictor detects a dependency, the load waits in the load buffer until the conflicting address in the store buffer is resolved. If a load is found to have been prematurely scheduled, the load does not have to re-fetched, since it a correct instruction that has executed with incorrect data. It is simply reissued for execution or *replayed* with correct data. If a branch is found to be mis-predicted, the pipeline is flushed and all speculative state following the branch is discarded.

The main cycle critical structures in a superscalar processor are physical registers, issue queue and load store queues.

2.1.2 Problems with Scaling Physical Register File

In some implementations [39], the reorder buffer [43] is used for the purpose of register renaming, so we will discuss the energy efficiency issues of a reorder buffer with respect to the physical register file array [43]. Register renaming is a technique employed by out-of-order processors where a larger set of physical registers in addition to the architectural registers are used to overcome false dependencies between instructions that come as an artifact of limited logical registers in the instruction set architecture (ISA) [43].

Assuming each instruction has two source registers and one destination register, the register file needs two read ports and one write port for every instruction. So, on a 4 wide machine, it needs approximately 8 read ports and 4 write ports. In some implementations this can be reduced to a lower number because some instructions get their data from the bypass network and instructions like branches and stores do not need the write port as they do not produce results that need to be written to the register file. In any case, the complex implementation and power inefficiency of the large number of ports needed to support a wide machine and large instruction window makes the register file a difficult structure to scale.

2.1.3 Problems with Scaling Issue Queue

The reservation station (RS) or issue queue holds all the instructions that have been renamed but are yet to execute [45]. The working of a reservation station involves two actions – 1) finding instructions that can be scheduled for execution. 2) After an instruction finishes execution, waking up other instructions which are waiting for the result produced by the completing instruction. Scheduling ready instructions is typically implemented as a priority encoder. The wakeup logic is implemented as content - addressable memory (CAM) matching logic where every entry in the reservation station matches its source ID with the ID of a completing instruction to find out if its source is ready. This CAM logic is associative and as a result, the RS implementation involves a lot of dynamic activity and power consumption. At the same time, The RS logic is on the critical path and cannot take multiple cycles because dependent instructions need to be woken up and scheduled for execution in consecutive clock cycles to ensure good performance. For this reason the wakeup logic is usually designed to have minimum possible latency. These two reasons make the implementation of large reservation stations very difficult.

2.1.4 Problems with Scaling Load Store Queue

In addition to instructions that access registers, the instruction window is also made of instructions that access memory. In order to support a large window, conventional processors use a load and store queue [43]. With out-of-order execution and stores completing out-of-order, it is necessary to make sure that a load reads its data from the correct store. Stores write to the cache only in program order for correctness reasons. It hurts performance if a load has to wait for every store to update the cache and then read its data. Just like register data can be forwarded to its sourcing instruction through the bypass network much before the data reaches the register file, there is provision for a store to forward data to its dependent load before it reaches the cache. In order to achieve this store-to-load forwarding, conventional processors use a store queue that holds the address and data of all stores in the instruction window. When a load executes, it searches the entire store queue for an older store to the same address. If there is a hit, it uses the data from the store queue; otherwise, it uses the data from the cache. Typically the store queue and the cache are searched for a matching address in parallel to save on time. The store queue is implemented using CAM logic to match the store address with that of the load. The latency of this associative match should not be worse than that of a cache hit; otherwise the load scheduling logic will become more complex. All these reasons make it difficult to build a large store queue to support a large instruction window for performance.

In conventional processors, the store-to-load forwarding is done speculatively. With out-of-order execution, a load address may become ready before older stores complete. It is very conservative to let a load wait until all stores before it complete, which does not help performance. Typically, processors use a memory dependence predictor to decide whether to stall such a load or schedule it [8]. In some cases a load may read the data from an incorrect store, leading to a memory ordering violation [8]. In order to avoid such memory ordering violations there is a load buffer that holds all the loads in the window in program order. Each retiring store checks with this load buffer if any load has executed prematurely with incorrect data. If a memory ordering violation is detected, corrective action has to be taken, for instance the load will have to be replayed with correct data [19]. This matching of the store address with the load buffer entries is also an associative search, making it non-scalable like the previously discussed structures.

Sections 2.1.2 to 2.1.4 explain the reasons why scaling the instruction buffers to large sizes in order to support a large number of in-flight instructions is neither practical nor energy efficient.

2.2 Mechanisms in Literature

So far it has been established that a large instruction window is needed for good single thread performance, but large instruction buffers cannot be used to achieve this because of their energy inefficiency. Is there an alternative to simply scaling the sizes of instruction buffers for performance?

Earlier work [29][46] on ILP have shown that in an instruction window comprising of both blocked and miss-independent instructions, the blocked instructions are far fewer than independent instructions. As shown in Figure 6, beyond a few dependent instructions, there are a lot of independent instructions which can be scheduled for execution without having to wait for the load miss. The amount of ILP

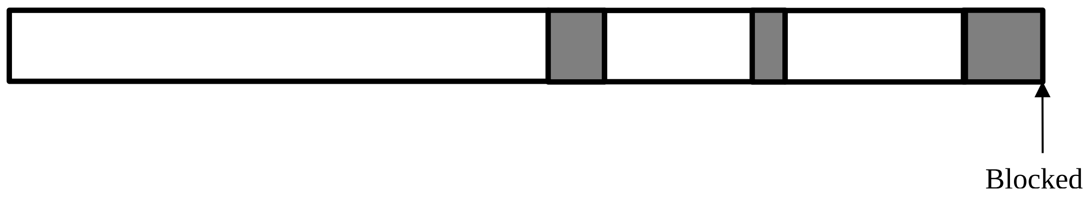


Figure 6. Typical behavior of programs showing ILP behind blocked cache misses

that can be exposed by a typical out-of-order superscalar processor in the event of a last level load miss is not even comparable to the amount of ILP that exists in programs. For instance, if the window in Figure 6 is of thousands of instructions, a state of the art commercial processor which can support an instruction window of ~200 instructions will soon run out of instruction buffers and stall much before it can reach the majority of independent instructions, thus exposing only limited ILP while dissipating a lot of power in the process. There have been several unconventional processor architectures proposed in literature which take advantage of such program behavior shown in Figure 6 and deal with load misses to the cache in a different way. They unblock the load miss and its dependent instructions and make them release their pipeline resources until the miss data arrives. Since long latency instructions do not hold on to their instruction buffers for a long time, pipeline resources are always recycled quickly, so only a reasonable number of instruction buffers are sufficient to support a large instruction window. These architectures differ from each other in terms of how this large look ahead instruction window is made available, which is explained in the next section.

2.2.1 Runahead execution processors

A Runahead execution processor [12][34] exposes MLP in programs in the event of last level cache misses. When a load misses the cache, a Runahead processor checkpoints [21] the architectural state and enters Runahead mode. The load miss and its dependents pseudo-execute and release their pipeline resources. The miss-independents also complete execution and retire from the reorder buffer. However, in Runahead mode, the retiring instructions do not update the architectural state or cache. When the miss returns the processor state is restored from the checkpoint. All the instructions, both dependents and independents are re-executed with correct data. The only difference is that the execution now is accelerated because the data required by the instructions in the current window is brought to the processor core sooner, owing to the MLP exposed by Runahead execution. All the speculative work done in the shadow of the cache miss is thrown away.

Figure 7 shows the working of a Runahead processor for the same working example from Figure 4. The darker shaded blocks indicate pseudo-execution of instructions while the lighter shaded blocks indicate second wave execution after the miss data is returned to the processor. Figure 7 shows that second wave execution is much faster and involves fewer stalls compared to first wave execution. The same figure also shows that all the instructions are re-executed during the second wave. As for the memory state needed to maintain a large window, since Runahead processors do not update the cache, memory ordering violations in Runahead mode are not fatal. Hence a small load store queue will suffice.

The advantage of Runahead execution is that it is perfectly implementable with negligible hardware overhead needed for the checkpoint. However, notice that Runahead chooses to re-execute all the miss-independent instructions redundantly. In other words, the key problem with Runahead processor is that it does not have provision

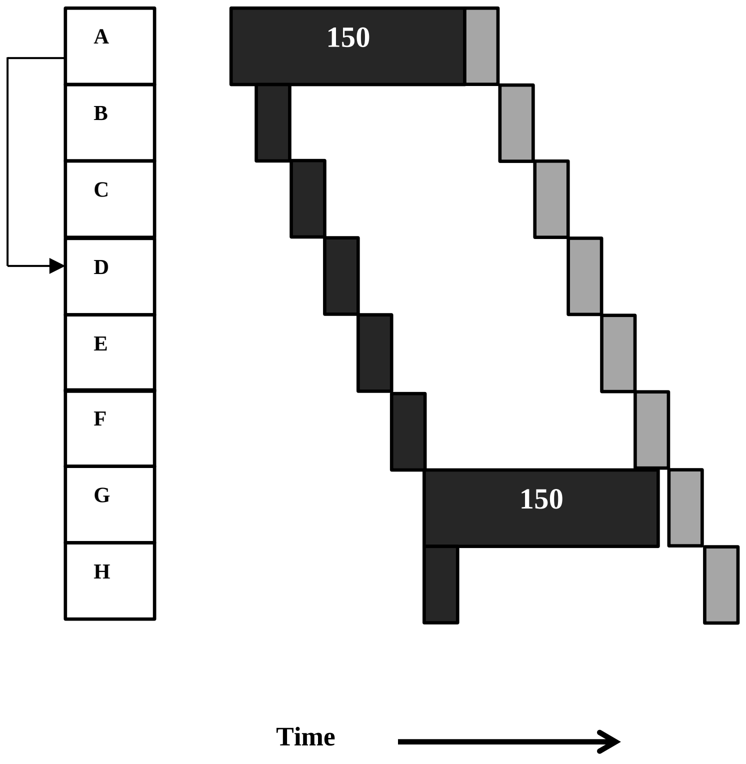


Figure 7. Execution sequence of a Runahead processor

to save the results of independent execution and thereby ends up re-executing all the non-shaded instructions in Figure 7 after the miss comes back, even though those instructions are not related to the miss. Since all the speculative work is thrown away there is limited performance improvement from Runahead execution. Moreover, there is wastage of energy because all instructions are executed twice, once in Runahead mode and again when the miss data is delivered. If Runahead cannot expose any MLP then there is no performance benefit, while the energy consumption cannot be avoided. In order to overcome this, later work [35][36], adds heuristics to determine phases where Runahead execution would not benefit. This is done by maintaining a history of load misses that have exposed MLP. If a load that is known to have not given any prefetch benefit misses again, the processor opts not to go into Runahead mode. Another heuristic makes note of the number of instructions pseudo-retired in the runahead window and makes sure atleast these many instructions retire before Runahead mode is entered once again. This heuristic avoids overlapping Runahead episodes. Overall these heuristics improve performance and energy by some amount.

In summary, the Runahead concept is applied to last level misses that typically take a long time to return. Potentially a lot of ILP can be exposed in Runahead mode while the miss is outstanding. The fact that all this work, including meaningful independent execution is thrown away after the miss returns, would be the biggest limiting factor of this proposal.

2.2.2 Waiting Instruction Buffer

The Waiting Instruction Buffer [30] does what Runahead execution processor does not - which is execute the miss-independent instructions while the load miss is outstanding and not revisit them again after the miss comes back. This is because, unlike Runahead processor, Waiting Instruction Buffer (WIB) provides the ability to retain the results of independent execution, but at the expense of inordinate amount of buffering, as will be explained further.

Typically in a superscalar processor the issue queue size is lower than the size of the instruction window because the reservation stations do not buffer those instructions that have completed but are yet to retire. As mentioned before, the issue queue is a complex structure made up of CAM logic which is also timing critical. The WIB design releases pressure on the issue queue by making long latency cache misses release their issue queue entries. The execution of WIB for the same working example

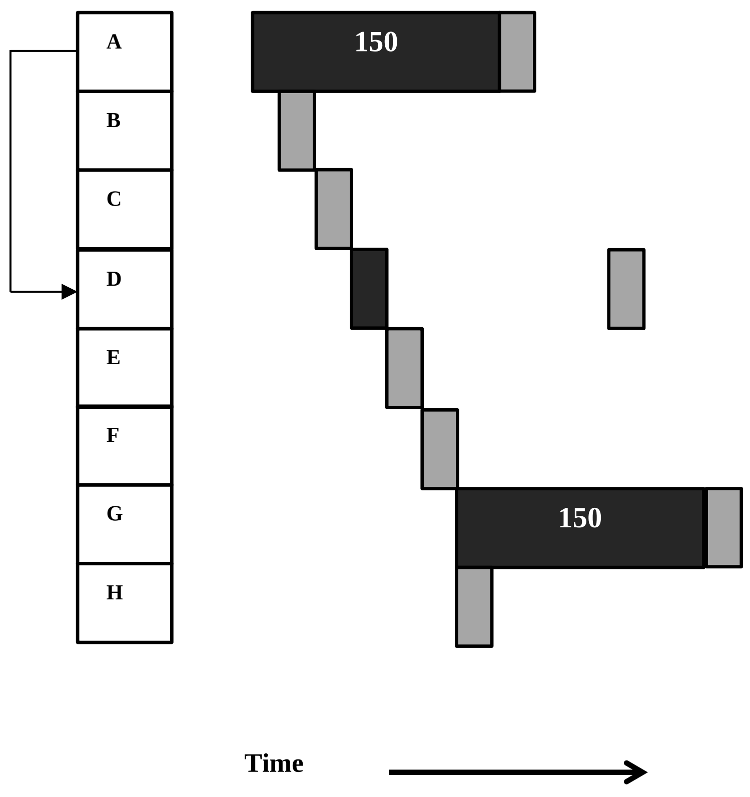


Figure 8. Execution sequence of a processor that uses WIB

is shown in Figure 8. As can be seen in the figure, the miss-independents *B, C, E, F* and *H* do not re-execute when the miss returns; only the load miss and miss-dependent instructions *A, D* and *G* do, shown by the presence of both dark and light shaded boxes alongside these instructions. WIB treats the issue queue as the only critical resource and allows miss dependents to retain other pipeline resources like physical registers and load store queue entries, simply assuming that a large number of them can be made available in future implementations. So in order to support a large instruction window, WIB physically buffers the entire window with a multi-level register file and large instruction buffers. In WIB, each instruction dependent on a load miss sets a bit vector dedicated for each outstanding miss, which allows miss-dependents to be reissued post wakeup without needing the complex broadcast logic of the issue queue. Their waiting buffer is organized as a multi-banked structure which allows miss-dependents to be reissued in any order as and when the wakeup arrives. Physically buffering the entire instruction window also allows instructions dependent on multiple load misses or miss-dependent misses to be moved in and out of the issue queue multiple times, although at the cost of excessive re-execution and energy.

In summary, WIB is able to exploit large amount of ILP by executing the miss-independents while the load miss is outstanding and does not re-execute these instructions when the miss data returns. However, it achieves this by physically buffering the entire instruction window, the implementation of which is impractical.

2.2.3 Continual Flow Pipelines

Continual Flow Pipelines [46] was the first latency tolerant proposal that included scalable solutions for the all the cycle critical pipeline resources including the register file, issue queue and load/store queues. This dissertation aims to improve on the latency tolerance techniques proposed in conventional CFP.

When a load miss occurs, CFP makes the load miss and its dependents relinquish their critical pipeline resources and wait in a low complexity FIFO buffer outside the core pipeline. Repeating the running example, CFP execution would look exactly the same as the execution of WIB shown in Figure 8. The primary difference is that CFP incorporates a scalable register management algorithm and addresses issues with scaling the load and store queue issues in addition to the issue queue.

2.2.3.1 Register Management in CFP

CFP alleviates pressure on the register file by replacing conventional ROB-based register management with Checkpoint Processing and Recovery (CPR) [2][3]. CPR operates at the granularity of checkpoints, which are created at the rename stage. Instructions are bulk committed one checkpoint at a time and recovery is permitted only to checkpoints. CPR takes advantage of this recovery restriction to aggressively reclaim physical registers between checkpoints. CPR tracks registers by associating each register with a reference count and a count of zero indicates the register is free. A register's reference count is incremented when an instruction that reads or writes the register is dispatched to the issue queue and decremented when that instruction executes or is squashed.

2.2.3.2 Slice Buffer Management in CFP

CFP releases pressure on the issue queue by making miss-dependent instructions release their issue queue entries, similar to WIB, and moving them outside the critical pipeline into a low complexity slice buffer. While CFP's slice buffer is similar to WIB, it has two important differences. First, while WIB physically buffers the entire instruction window, CFP buffers only the miss-dependents, which are far fewer in number compared to the miss-independents [28][46], as shown in Figure 6. This allows CFP to support a large instruction window with a reasonably small slice buffer. Second, in CFP miss-dependent instructions capture miss-independent register inputs when they move into the slice buffer. This decouples the miss-dependent slice from the rest of the program. This decoupling means that miss-dependent instructions can release their input registers as if they had executed, allowing CPR to reclaim them long before they are used by the slice buffer instructions.

The CFP slice buffer is maintained in execution order and instructions are allocated slice buffer entries as they come out of the issue queue. After the wakeup, CFP re-introduces the instructions from the slice buffer into the pipeline by re-renaming and re-dispatching them. Since CFP populates the slice buffer in execution order, the slice buffer cannot represent register dependences in terms of logical register names (since logical registers may have been overwritten by the miss-independent instructions) and hence must use physical register names instead. So CFP uses a physical-to-physical register renaming policy to reintroduce slice instructions into the pipeline. In physical-to-physical renaming, the physical-to-physical map table has as many entries as the processor has physical registers. An instructions input registers are renamed by indexing this map table with the physical register numbers that it received when it was originally renamed. A new destination register is allocated, and entered into the map table at the index corresponding to the destination physical register number originally assigned at rename.

2.2.3.3 Load and Store Management in CFP

CFP scales the load and store queues through the use of hierarchy [2]. To support a large window of load instructions, CFP uses a conventional fully-associative first level load queue along with a large set-associative second level load queue [2]. In this hierarchical design, the youngest loads are placed in the fully-associative queue. When the fully-associative queue fills up, the oldest load in the queue is relegated to the set-associative load queue.

CFP also uses hierarchy to manage stores [2]. Similar to the load queue design there is a conventional first level store queue and a larger associative second level store queue with slower access latency. This is known to work well because store-load pairs are usually within a short distance from each other and most of the forwarding occurs when a store is still in the first level store queue. This is the main purpose of the first level store queue while the second level store queue is used mainly to ensure memory ordering. Though hierarchical store queues perform well, the disadvantage with a large second level store queue is the area and energy inefficiency of the CAM matching circuitry [14].

To overcome the dynamic power associated with a large associative hierarchical store queue, Gandhi et. al. propose to replace it with a simple FIFO structure called the Store-Redo Log (SRL) [14] that records all the stores in the instruction window in program order. All completing stores write their data into pre-allocated locations (assigned at rename) in the SRL whenever they become ready.

While the miss is pending, miss-independent stores write to the cache speculatively and also update the SRL. Miss-independent loads which cannot forward data from the small conventional store queue get their data speculatively from the cache. When the miss returns, the speculative state in the cache is flushed and the slice buffer instructions are reissued for second wave execution. Miss-dependent stores get store queue entries to forward data to dependent loads. When miss-dependent stores complete, they fill up the holes in the SRL buffer assigned for dependent stores. When the slice buffer empties, all the holes in the SRL are filled up by either independent or dependent stores. Then all the stores are emptied into the cache in program order from SRL.

Notice that the miss-independents write to the cache twice, first time for forwarding data to independent loads and second time from the SRL in correct program order. The advantage of the SRL is that it does not take part in any forwarding and is neither a multi-ported nor an associative structure. It is primarily there to ensure store updates to the cache take place in program order, while the responsibility of forwarding is off-loaded to the small store queue and cache. This allows the SRL design to support a large window and yet remain scalable.

2.2.3.4 CFP on In-order Processors

CFP on in-order cores was first proposed in [37]. This approach is suitable for highly energy constrained computing devices but less suitable for the performance needs of conventional single-thread applications targeted by the Tuned-CFP multicore architecture.

2.2.3.5 Limitations of CFP

Figure 9(a) shows an execution sequence of instructions from *#1* to *#300*. Instruction *#2* misses the last level cache and will take a long time to execute. The cache miss dependents are shown as dark shaded boxes. All the following blocks represent the pipeline resources like reorder buffer entries and load store queue entries. Occupied pipeline resources are shown as shaded, while unoccupied resources are shown in white. The boxes on the right side show the low complexity slice buffer where load miss instructions and their dependents wait until the miss data arrives. The non-CFP ROB processor does not have a slice buffer.

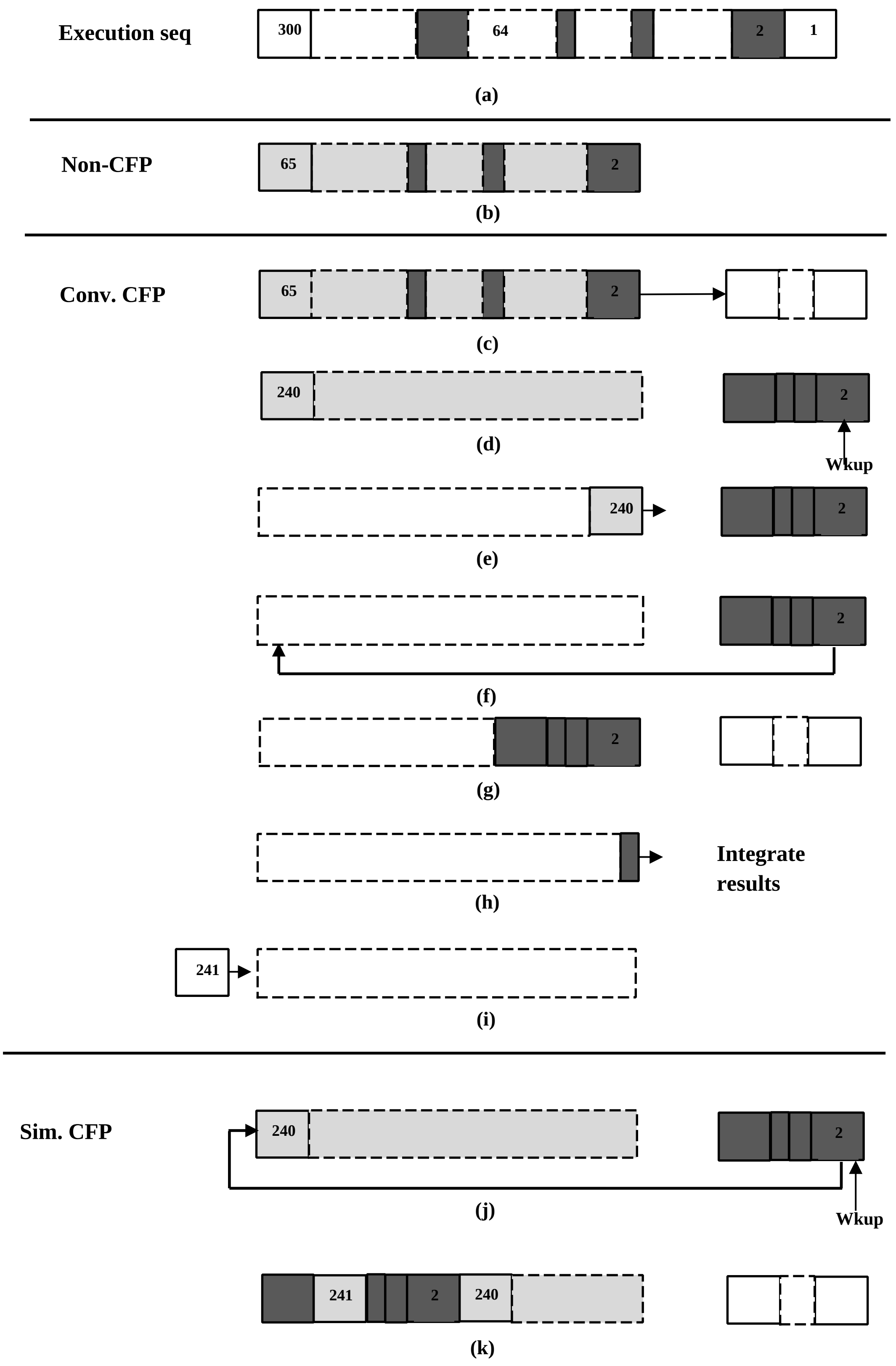


Figure 9. Execution sequence demonstrating the pipeline flush limitation of conventional-CFP

We assume a 64 entry reorder buffer for this example. Figure 9(b) shows that the non-CFP ROB processor stalls after instruction *#65* is brought into the pipeline because *#2* cannot proceed forward until the cache miss data returns. Notice that even though all non-shaded instructions between *#2* and *#65* execute in the out-of-order processor, they still stay in the pipeline holding onto their resources because *#2* is blocking them, and as a result the ILP that can be exploited by a non-CFP processor is limited.

Figures 9(c) - 9(i) show the working of a conventional-CFP processor. When the load miss instruction *#2* is found to be blocking the pipeline, it is moved into the slice buffer as shown in Figure 9(c). Figure 9(d) shows that all miss-dependents or shaded instructions, including those beyond *#64*, move into the slice buffer. Though not explicitly shown in the figure, the miss-dependents also carry their ready input sources from miss-independent execution into the slice buffer. While the miss-dependents move into the slice buffer, CFP is able to process a lot more instructions compared to non-CFP ROB, indicated by the fact that CFP is able to take processing forward up to instruction *#240* when the wakeup for *#2* arrives as shown in the same Figure 9(d).

After the wakeup arrives, the front end of the conventional-CFP processor is stalled; so no more instructions after *#240* are brought into the pipeline, indicated by the white slots in Figure 9(e). As shown in Figure 9(f), finally when the pipeline is drained, the slice instructions can be reintroduced into the pipeline. The shaded instructions are a self-contained slice and though separated in the original program as shown in the execution sequence of Figure 9(a), they can be executed together as a block in the second wave as shown in Figure 9(g).

When all dependents complete without exceptions or mis-predictions, the results of independent and dependent execution are integrated as shown in Figure 9(h). Only after this, the next instruction from the execution sequence *#241* gets a chance to enter the pipeline, as shown in Figure 9(i).

As we see from the example, the main disadvantage of conventional-CFP is the pipeline flush that separates the execution of independents and dependents leading to a lot of unused pipeline slots whenever the dependents are replayed from the slice buffer. The impact of this pipeline flush was not very pronounced when conventional-CFP was introduced because it was applied to only last level cache misses that went to DRAM. In present generation processors, with integrated memory controllers and large on-chip caches, applying CFP to only misses that go to DRAM will give less benefit, especially on CPU benchmarks. Targeting the other kind of misses, that is first level misses that hit higher on-chip caches, is also equally important. Although they do not incur as much delay as a miss to DRAM, they occur more frequently and consequently have the same impact on performance as a last level cache miss. Moreover, this allows designing processor cores with smaller instruction buffers to suit the L1 hit case.

However, conventional switch-on-event CFP cannot be applied to first level misses because the pipeline flush penalty will override the benefits of CFP execution. To overcome the limitation of conventional-CFP, in this dissertation, we propose to execute the independent and dependent instructions concurrently, similar to a simultaneous multithreading (SMT) architecture. We call this simultaneous-CFP (S-CFP) processor core architecture.

For the same working example, when the dependents block forward progress, like conventional-CFP, they are moved into the slice buffer as shown in Figure 9(c). But unlike conventional-CFP, when the wakeup arrives, the pipeline is not drained. The slice instructions are reissued while the independents are active in the pipeline as shown in Figure 9(j). Notice that the slice instructions execute as a separate thread, so they can be reintroduced into the pipeline, inter-mingled with the independents, as shown by the non-contiguous shaded instructions in Figure 9(k). The key impact of concurrent execution is that the empty or white slots are completely avoided with this approach. Avoiding the pipeline flush overhead allows CFP to be applied on first level misses as well for better performance as we will show in the results section.

2.2.4 Kilo Instruction Processors

Gonzalez et al. [15] proposed using virtual registers to shorten the lifetime of physical registers. Refer Section 2.3 for more details on virtual registers.

Kilo instruction processors [10] also used virtual renaming and ephemeral registers to do late allocation of physical registers. KILO scales the register file using a combination of virtual registers and reference counting. At rename, logical registers are mapped to virtual registers - a namespace larger than the set of physical registers. The virtual register name is then re-mapped to a physical register only when it needs a physical location – that is when the instruction produces a result after execution. KILO reference counts these virtual register names to determine when they can be reclaimed, and reclaims the underlying physical register at the same time. For the load or store queues, KILO cites various prior proposals, including those used by CFP, and states that any would be satisfactory.

Decoupled KILO-Instruction Processor (D-KIP) [40] improves the register management scheme of KILO by using two different processors – a Cache Processor for independents and a Memory Processor for dependents. In D-KIP, instructions start in the Cache Processor. The Cache Processor is a conventional out-of-order processor, except that miss-dependent instructions are forced out of its ROB after a certain number of cycles. As miss-dependent instructions leave the Cache Processor, they are placed into a FIFO queue which connects to an in-order Memory Processor. Miss-dependents capture their ready register input values as they exit the Cache Processor, so that the Memory Processor remains a self-contained slice. D-KIP maintains precise register state in the Cache Processor using checkpoints. D-KIP scales the load and store queues using hierarchy.

2.2.5 Flea-Flicker

Flea-Flicker [4][5] executes a program on two in-order back-end pipelines coupled by a queue. An advance pipeline executes independent instructions without stalling on long latency cache misses while deferring dependent instructions. A backup pipeline executes the instructions deferred in the advance pipeline and merges them with results stored in the queue from the advance pipeline. Flea-flicker executes instructions in an in-order pipeline, saves advanced instructions and results in its queue and merges results sequentially during backup pipeline execution.

2.2.6 iCFP and BOLT

iCFP [17] tolerates cache misses at all levels in the cache hierarchy, but uses an in-order pipeline, which is less suitable for the performance needs of conventional single-threaded applications.

BOLT [18] utilizes additional map tables in Simultaneous Multithreading architecture to re-rename L2 miss-dependent slice, combined with a program order slice and a unified physical register file that supports aggressive register reclamation. BOLT reuse of SMT hardware is to improve energy efficiency and performance.

2.2.7 Sun Microsystems Rock

Sun Microsystems Rock is a single die multicore processor for high throughput computing. Rock uses Simultaneous Speculative Threading [7] to defer dependent instructions into a buffer, and executes the deferred instructions from the checkpoint after the miss data returns. The deferred instructions execution uses a simultaneous hardware thread and merges the results into the scout thread future file. Rock uses an in-order pipeline, similar to iCFP and is thus less suitable for the performance needs of single-threaded applications.

2.3 Simultaneous Multithreading

Simultaneous multithreading (SMT) [48][49] is a technique that allows multiple independent threads to issue multiple instructions each cycle to a superscalar processor's execution units. SMT benefits from multiple instruction issue of current superscalars and latency hiding ability of multithreading. SMT allows threads to share resources without context switching delays by having all thread register contexts active simultaneously. SMT also works around the limited thread instruction parallelism by having instructions from multiple independent threads in the scheduling window, providing plenty of opportunity to keep the superscalar execution units at full utilization. SMT can be implemented as an extension to a superscalar in a straightforward manner. Changes necessary to support SMT on a superscalar are:

- Multiple program counters and a fetch unit arbitration mechanism.
- Separate return address stack per thread.
- Separate retirement, branch mis-prediction, and exception recovery logic for each thread.
- A register alias table for each thread in the rename unit.
- Larger register file to support logical registers for all threads as well as additional registers for renaming.

Because dependencies between instructions are established at the rename stage, instructions from different threads can share the instruction buffers, the register file, execution units and read/write ports to the cache. The issue logic does not have to keep track to which thread an instruction belongs. Execution proceeds in a dataflow manner as usual. The primary impact is the enlarged size of the register file, especially for instruction sets which have a large number of logical registers. The enlarged register file may require an additional cycle in the pipeline to perform the read, and an additional cycle to write results, as suggested in [49].

Although SMT can be very useful in increasing throughput when running multiple programs, it is still limited in some respect. Sharing the cache hierarchy and the chip pin bandwidth among multiple programs limits the efficiency of these resources in comparison to running a single program. On the other hand, multithreading a single program, does not limit cache efficiency to the same degree, especially when instructions from the two threads lie very close to each other. Since this dissertation targets CFP on first level misses and majority of first level misses hit on-chip caches, the SMT threads comprise of closely lying instructions from the single program and usually benefit from constructive cache accesses.

2.4 Architectures with Virtual Register Renaming

The mechanisms proposed in this dissertation to improve earlier CFP architectures benefit immensely from the concept of virtual register renaming. This section gives a brief account of prior work done on virtual registers.

2.4.1 Virtual Registers

The concept of using virtual registers for renaming was first introduced by Gonzalez et al [15]. It was used to shorten the lifetime of a physical register in the instruction window. After an instruction comes out of the decode stage, it is renamed with a virtual register ID, chosen from a free pool of virtual registers. A virtual map table indexed by the logical register is used by future instructions to determine their sources. A physical register from a free pool is allocated to this instruction only when a storage location is required by it, which is when a result is generated by this instruction from an execution unit. A second table called physical map table indexed by the virtual register is used for translation to physical registers. The virtual register and physical register are both freed when the instruction commits. A ROB is used to release the virtual and physical registers and to recover from branch mis-predictions and exceptions.

2.4.2 Virtual Register Renaming

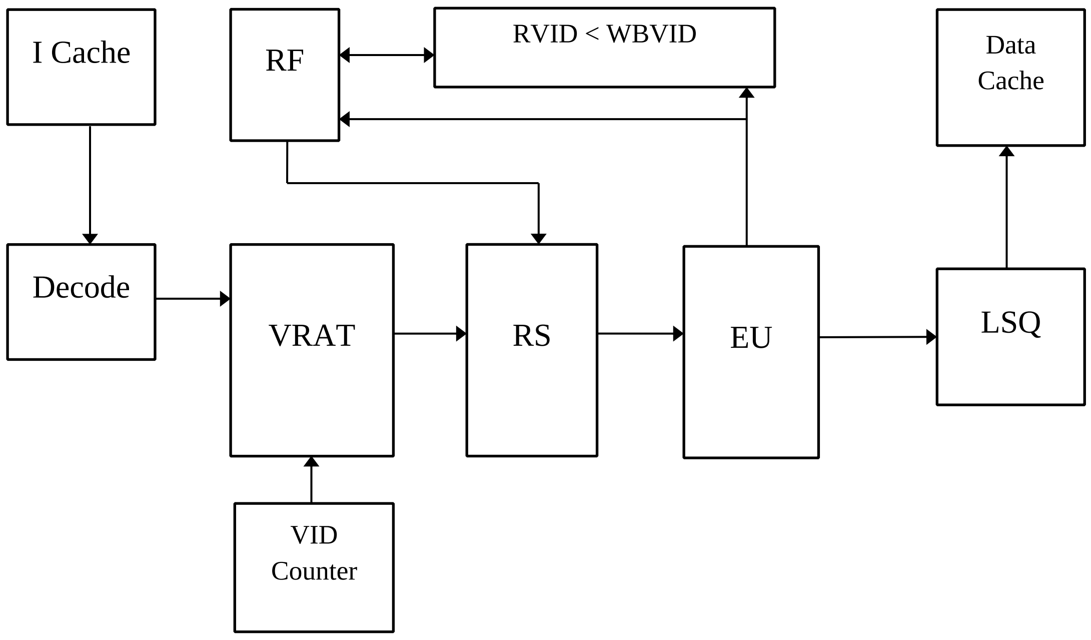


Figure 10. Virtual register renaming architecture block diagram

Sharafeddine and Akkary describe a virtual register renaming (VRR) architecture [41] that uses checkpoints which eliminates the use of physical registers and reorder buffer. Figure 10 shows the block diagram of VRR architecture. An instruction from the decode unit is allocated a virtual ID (VID) from the virtual register counter (VR_Count in Figure 10) if there is an available entry in the reservation station. This VID stays with the instruction until it completes full execution. Each entry in the RS has fields for its source designator, its data value and a valid bit. In the VRR architecture, the VID acts as the source designator. Whenever a result is generated by the execution unit it broadcasts the result along with its VID on the bus. All the entries in the RS compare this VID to that stored in their respective source designator fields and upon a match, copy the data to their RS and set the valid bit. Whenever all the source operands are valid and an execution unit is available, an instruction is scheduled for execution. The instruction takes its VID along with it for execution so that it can broadcast it after completion and consumers waiting for this data can grab it from the writeback bus. When an instruction completes, it writes its results to the RF based on the condition whether its VID is larger than the VID of the last writer to have updated the RF. This ensures sequential updates to the RF without having to use a ROB for the same purpose. Also in the VRR architecture, checkpoints are created at select low confidence branches (determined based on history) to ensure forward progress. In case of an exception or branch mis-prediction, execution rolls back to the previous checkpointed state.

This virtual register renaming architecture [41] forms the substrate over which our Tuned-CFP architecture is built.

2.5 Summary

This chapter explained the work that has been done in the past to deal with cache misses to improve single thread performance in processors. Conventional processors rely on non-scalable structures to give performance improvement, while the non-conventional proposals either provide latency tolerance on in-order processors or only target cache misses to DRAM. If latency tolerance needs to be provided for misses at all levels of the cache hierarchy, then the proposals from the past are not effective. Chapter 3 details the microarchitecture of our proposal that allows miss-dependents and miss-independents to execute simultaneously. Chapter 4 details the implementation of a processor that uses the virtual register renaming architecture explained in section 2.3 as a substrate to provide improved load latency tolerance, both in terms of performance and energy.

CHAPTER 3

SIMULTANEOUS-CFP ARCHITECTURE

Continual Flow Pipeline architecture [46] was originally proposed to tolerate long latencies of L2 cache misses that go to DRAM. In the original proposal, the miss-independent and dependent instructions execute at different times, based on the timing of the load miss event and the data arrival event. Switching between the two executions is costly because it involves a pipeline flush, making this proposal unsuitable for L1 misses that hit the on-chip cache. Simultaneous-CFP (S-CFP) executes the independent and dependent instructions simultaneously to avoid the costly pipeline flush, thus making S-CFP more suitable for first level data cache misses.

3.1 Simultaneous-CFP Microarchitecture

In this section, we describe Simultaneous Continual Flow Pipeline (S-CFP) core microarchitecture, and its performance optimizations, simplifications, and differences from previous latency tolerant core architectures.

3.1.1. Microarchitecture Overview

The S-CFP core microarchitecture is based on Intel P6 core architecture [39]. Intel P6 architecture executes x86 code after decoding the x86 complex macro-instructions into RISC-like micro instructions or micro-ops (uops). We use the word instructions to mean micro-ops for the rest of this document. Figure 11 shows a block diagram of the Simultaneous-CFP core. Unlike previous latency tolerant out-of-order

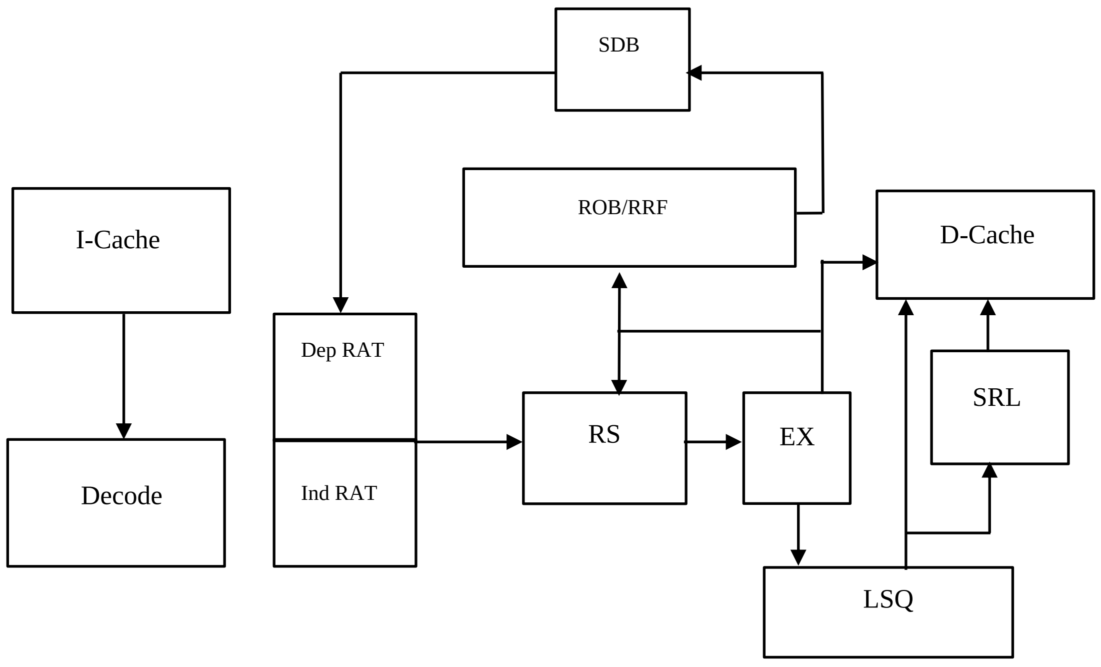


Figure 11. Simultaneous-CFP architecture block diagram

architectures, the S-CFP core executes cache miss dependent and independent instructions concurrently using two different hardware thread contexts. The S-CFP hardware is similar to simultaneous multithreading architectures (SMT) [48], except that in S-CFP, the two simultaneous threads are constructed dynamically from the same program, instead of being two different programs that run simultaneously in the same core. In order to support two hardware threads, S-CFP has two register alias tables (RAT) for renaming the independent and the dependent thread instructions. S-CFP also has two retirement register file contexts (RRF), one for retiring independent instruction results and the other for retiring dependent instruction results.

In S-CFP, execution initially starts using a hardware thread that we call the independent thread. When an L1 data cache load miss occurs, a poison bit is set in the destination register of the load. Load-dependent instructions in the reservation stations

(RS) are then woken up, as if the load completed. Poison bits propagate through instruction dependences, and identify all instructions that depend on the load miss and their descendants. Wherever register data resides or flows in the core, poison bits extend and qualify the data. For this purpose, S-CFP has new poison bits added to the reservation stations, writeback buses, bypass network, reorder buffer entries (ROB), retirement register file and store queue entries (SQ).

The miss load and its dependents, identified by the poison bits in the ROB, pseudo-retire in program order and move from the ROB into a dependent slice and data buffer (SDB) outside the pipeline. Therefore, miss-dependent instructions do not consume or occupy precious pipeline resources such as reservation stations or ROB entries, while waiting for the load miss data. This frees the pipeline resources for independent instructions to execute. Since dependent instructions do not tie pipeline resources, the core can look ahead far into the program for independent instructions to process.

Since poisoned instructions are reordered using the ROB before they are written into the SDB, the complexity of physical to physical register renaming, deadlock avoidance hardware, and rename filter, required in conventional-CFP architecture [46], are eliminated in S-CFP. Another advantage of reordering dependent instructions before writing them into the SDB, is that it allows implementing the SDB using single-ported SRAM, organized as an array of multi-instruction blocks. The width of a SDB block can be set to the width of the S-CFP core pipeline to allow writing or issuing SDB blocks of instructions, at full pipeline bandwidth.

Since the SDB needs to store any completed non-poisoned source registers with its instructions, S-CFP uses the ROB data array to propagate completed source operands with the poisoned instructions to the SDB. This is achieved with a modification of the execution units to make them write back completed source registers of poisoned instructions to the ROB. In other words, functional units execute poisoned operations as move source to destination operations. A poisoned instruction has at most one completed source since S-CFP uses RISC-like uops with at most two source registers.

When the miss-data is fetched into the L1 data cache, the dependent instructions wake up and issue again from the SDB into the pipeline using a hardware thread that we call dependent thread. The dependent thread executes simultaneously with the independent thread until the SDB is drained. Since the dependent and independent threads execute simultaneously using different register contexts, it is not necessary to flush the pipeline in order to execute the dependent instructions, as in previous latency tolerant architectures work. Without a pipeline flush penalty overhead, it becomes beneficial to pseudo-retire and defer L1 data cache load misses and their dependent instructions, even though many L1 data cache misses hit the last level cache and therefore result in shorter stalls than load misses to DRAM. When all dependent instructions re-execute and the SDB is drained, the execution results of the dependent and independent threads are integrated, with a single-cycle flash copy within the RRF. The independent thread continues execution by itself without any interruption or pipeline flush.

Since loads can depend on stores through memory, S-CFP relies on memory dependence prediction [8] to propagate poison through memory dependences. Poison bits are added to the store queue entries for this purpose. We now describe further details of the S-CFP core architecture.

3.1.2. Independent Thread Execution and Dependent Thread Construction

The independent hardware thread is the main execution thread in S-CFP. It is responsible for instruction fetch and decode, branch prediction, and memory dependence prediction. It also propagates poison bits after a cache miss to identify and remove from the pipeline miss-dependent instructions.

The independent thread executes instructions that are independent of L1 data cache misses, and pseudo-retires all instructions, miss independent as well as dependent. The retirement process copies the poison bit of each retired instruction into the retirement register file (RRF). Poison bits in the RRF are not sticky. At any time, a poison bit of a register entry in the RRF can be either true or false, depending on whether the last retired writer of this logical register was independent or dependent on the load miss.

When the SDB is empty and a load miss retires and enters the SDB, the independent thread retirement register file contains the precise state of the execution up to the load miss. S-CFP saves a checkpoint of this RRF for recovery in case of a subsequent miss dependent branch mis-prediction or exception.

Since a poisoned instruction has two source operands, it has at least one poisoned source and at most one completed source operand. S-CFP functional units write back the completed source registers of poisoned instructions to the ROB, from which they are copied to the SDB when the instructions pseudo-retire. Therefore, dependent instructions in the SDB form a self-contained program slice or thread that is completely independent of the retired miss-independent instructions.

In case of an independent mis-predicted branch, instructions before the branch pseudo retire and the dependents among these instructions enter the SDB. Instructions after the branch are flushed before they pseudo retire, thus no bogus mis-predicted instructions ever enter the SDB.

3.1.3. Dependent Thread Execution

Dependent thread execution starts when load miss data fetch completes and the load is woken from the SDB. It continues until the SDB is empty. The dependent thread execution uses a separate SMT hardware thread, with its own register alias table (DEP RAT), and retirement register file context (RRF). It executes simultaneously with the independent thread, with which it shares execution resources, such as, reservation stations, functional units, and data cache read and write ports. Dependent loads and stores carry with them unique sequence IDs assigned to them when they were originally fetched by the independent thread. These sequence IDs allow the load-store ordering hardware to identify the order of dependent and independent loads and stores within the program. The dependent thread is responsible for committing all stores, independent and dependent, in program order to the data cache.

The SDB is managed as a circular buffer with head and tail pointers. Dependent instructions enter the SDB at the tail when they pseudo-retire and deallocate from the head after they re-execute and retire. The SDB could contain at any time multiple load misses and their dependents, stored in program order. Load miss wakeup and reissue from the SDB is done in program order, even though load miss wakeups could arrive to the SDB out of order. If a load miss occurs in the dependent thread during its execution, the load stalls until the data is fetched into the L1 data cache. In other words, dependent instructions do not pseudo retire or enter the SDB more than once. As a consequence, pointer chasing code that encounter dependent misses do not fully benefit from S-CFP. However, we have chosen this approach to keep hardware simple.

If a mis-predicted branch or an exception occurs during dependent thread execution, S-CFP flushes the SDB and the pipeline, and rolls back execution to the checkpoint. Our simulation results, indicate that these flushes do not happen too frequently and S-CFP can still achieve significant performance gain in spite of these costly but infrequent rollback events.

3.1.4. Checkpoints and Results Integration

Figure 12 shows the S-CFP retirement register file cell with checkpoint flash copy support. We use a flash copy of the RRF for creating checkpoints. In two cycles every RRF bit is shifted into a checkpoint latch within the register cell (rightmost latch). The register file can be restored from the checkpoint in one cycle by asserting RSTR_CLK.

In addition to a checkpoint latch, the cell contains two context bits, one for the independent thread RRF (leftmost latch) and one for the dependent thread RRF (center latch). The independent and dependent thread results integration is a special restore cycle. At the end of dependent execution and when all instructions in the SDB have re-issued and retired, the RRF has all the live-out registers, some computed by the independent thread and some computed by the dependent thread, as determined by the poison bits in the RRF. To integrate these results back into one context, a restore cycle is performed from the dependent thread context into the independent thread context. However, not all registers are copied. Figure 12 shows that only the poisoned registers are copied by using the poison bits to enable the clock of the copy operation. A 2-to-1

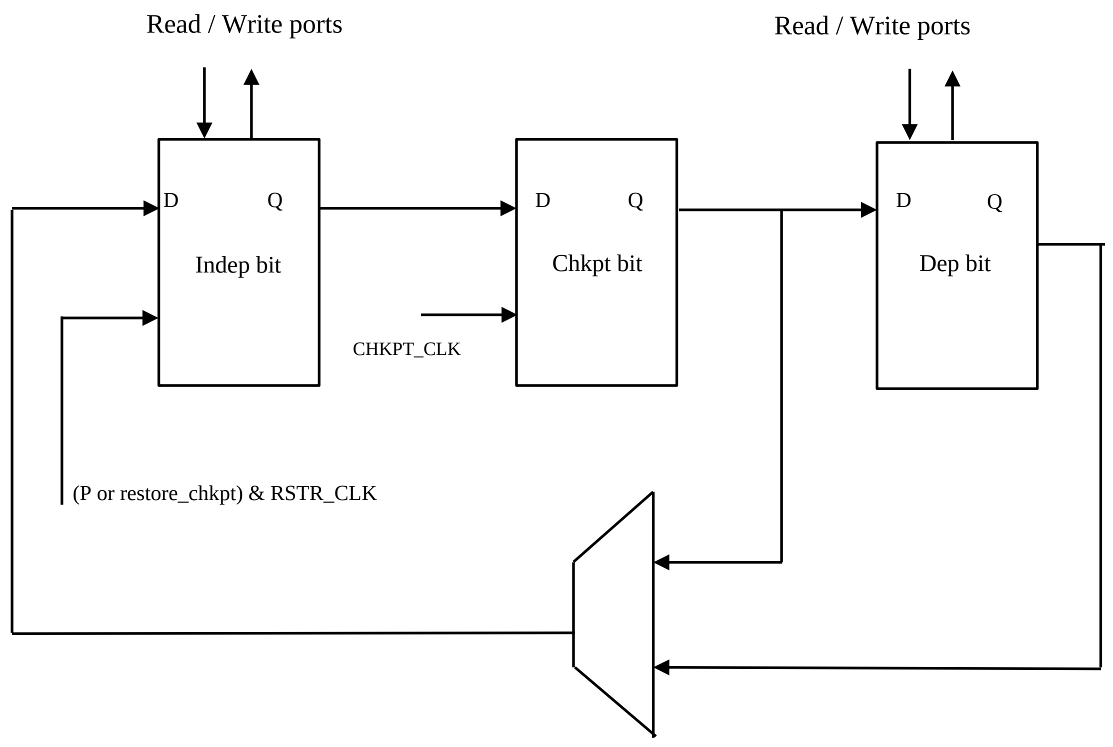


Figure 12. S-CFP register file cell

multiplexer in the cell restores either the checkpoint bit or the dependent bit during a RSTR_CLK cycle.

3.1.5. Memory Ordering and Load, Store Execution

To maintain proper memory ordering of loads and stores from the independent and dependent threads execution, we use small size load and store queues (LSQ), a Store Redo Log (SRL) [14] and a store set memory dependence predictor [8]. All stores, dependent and independent, are allocated entries (and IDs) in the SRL in program order by the independent thread. Every load, dependent or independent, carries the SRL ID of the last prior store. The SRL IDs assigned to loads and stores are unique and determine the order of memory instructions. The independent thread performs load memory dependence prediction, with a store-set predictor [8], to determine the store on which a load may depend. The predictor uses SRL IDs in the prediction and writes store poison bits in the SRL, thus allowing propagation of poison bits from stores to predicted-dependent loads. The L1 data cache support and dependent and independent threads load and store execution are explained in detail in Chapter 5.

3.1.6. Resource Sharing During Simultaneous Execution

S-CFP is different from simultaneous multithreading (SMT) in two key aspects: 1) the two simultaneous threads belong to the same single-thread program, and 2) the execution time is dominated by the independent thread, since dependent instructions form a small fraction of the overall program. S-CFP resource sharing has to favor the dominant independent thread to provide best overall performance.

The strategy we use for resource sharing is the following. When the SDB has instructions to issue, we use a round robin policy to schedule rename cycles between the dependent thread and the independent thread. This is one typical policy used in simultaneous multithreading architectures. However, we do not partition in S-CFP the reorder buffer between the two threads, and simply allow dependent and independent instructions to be interleaved in the reorder buffer. This may complicate the retirement stage to some degree. However, we believe that this policy is implementable, since dependent thread branch mis-predictions and exceptions are not taken at instruction granularity. Instead, they are handled by checkpoint rollbacks.

3.2 Limitations of Simultaneous-CFP

The problem with S-CFP is that the entire chain of the load miss and its dependents needs to be renamed and replayed from the SDB until all miss-independent and miss-dependent instructions complete execution and their results are merged. This can last for a long distance forward in the program, causing excessive replays and rollbacks. We show next in more detail examples that illustrate the drawbacks of S-CFP targeted by our optimizations.

3.2.1 Unoptimized S-CFP Execution Examples

Figures 13(a) and 13(b) show snapshots of the ROB and WB (the term *waiting buffer* can be used interchangeably with *slice data buffer*) states in S-CFP at different execution times. All shaded instructions correspond to either a load miss or a load miss dependent, both of which are potential candidates to move into the WB.

In Figure 13(a), the WB has a load miss *X* at the head waiting for its wakeup. Instruction *A* misses the first level cache and is marked as a potential candidate to be moved into the WB. When *A* reaches the head of the ROB, there are still free entries available in the ROB. Nevertheless, S-CFP eagerly pseudo-retires and moves *A* into the WB. In order to execute when the miss data is fetched, *A* has to be replayed from the WB back into the pipeline to be renamed again and allocated resources for execution.

Figure 13(b) shows that the load miss hits the L2 data cache and *A* is woken up from the L1 data cache shortly after it enters the WB. However, it is stuck in the WB behind instruction *X* that has missed to DRAM. For a long time afterwards, and until the miss data of load *X* is fetched from DRAM, many of the dependents of *A* will be poisoned and moved into the WB. Therefore, even though *A* has hit the L2 cache, it is replayed with its dependents from the WB as if it has encountered a miss in the L2 data

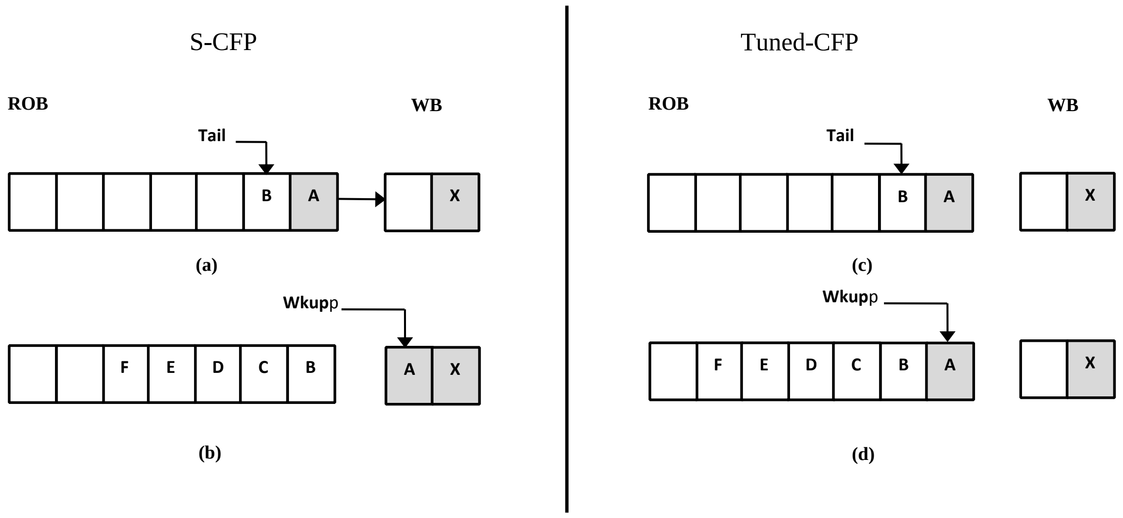


Figure 13. Execution sequence showing S-CFP moving a dependent into WB eagerly

cache and has needed to go all the way to DRAM for the data.

Figures 14(a)-14(c) show an execution sequence to illustrate a situation in S-CFP that leads to a rollback to the checkpoint. Similar to the earlier example, the WB has a load instruction *X* that has missed. Instruction *A* misses the first level cache. In this example, *F* is a branch instruction dependent on *A*. *A* is moved into the WB from the head of the ROB as shown in Figure 14(a). *F* also follows *A* into the waiting buffer, even though the wakeup for *A* arrives while *F* is still in the ROB/RS, as shown in Figure 14(b). Both *A* and *F* are replayed behind instruction *X* as shown in Figure 14(c). On replay, branch *F* is found to be mis-predicted and branch mis-prediction recovery has to be performed by rolling back execution to the checkpoint, since by then, the sequential state in the register file has been corrupted by the out-of-order pseudo-retirement of instructions during the cache miss processing.

Figures 15(a)-15(d) show another execution sequence to illustrate why S-CFP needs to replay a load and all its dependents once the load enters the WB. In this

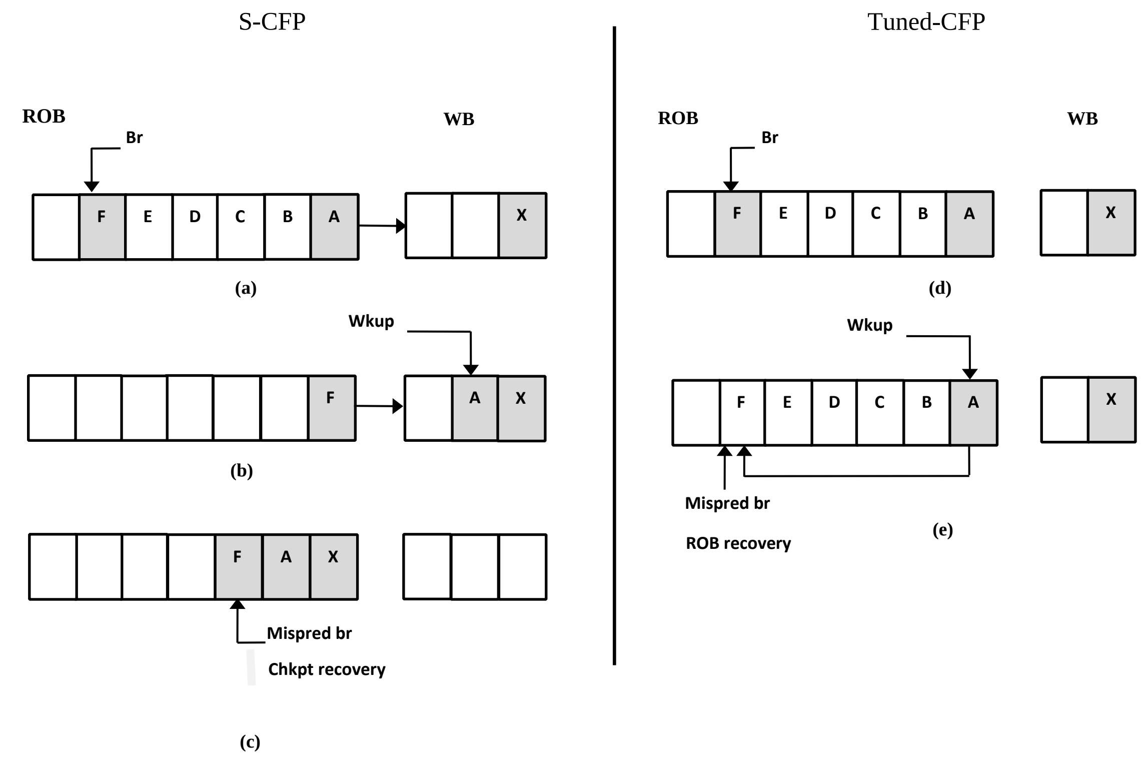


Figure 14. Execution sequence showing a scenario leading to rollback in S-CFP which is avoided in Tuned-CFP

example, *A* is a load miss and *B* is dependent on *A*. The two instructions are separated by miss independents shown as dotted lines. The figures show only the miss dependents in the pipeline for clarity. The renamed physical source and destination register tags *(src1, src2 -> dest)* are shown alongside the instruction. Also shown below their ROB entries are the ROB IDs or physical destination registers of *A* and *B*. The waiting buffer, initially empty, is also shown in the figures.

In Figure 15(a), *A* reaches the head of the ROB. It pseudo-retires and moves into the WB, releasing all its pipeline resources including its ROB ID *#3*, as shown in Figure 15(b). When the wakeup for *A* arrives and it replays, it is allocated a new entry at the tail of the ROB as shown in Figure 15(c). Notice that *A* gets a new ROB ID *#24*

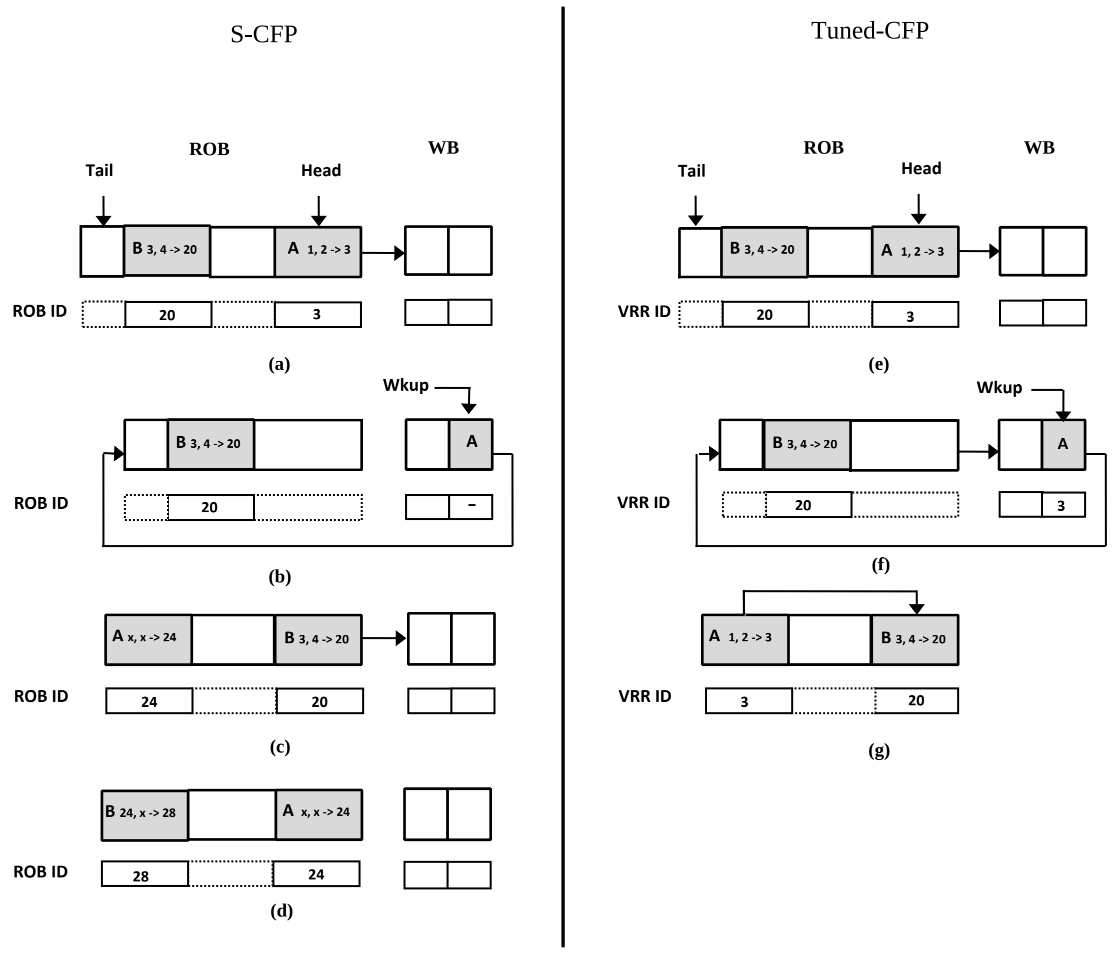


Figure 15. Comparison of full replay in S-CFP and partial replay in Tuned-CFP

when it is reintroduced into the pipeline.

Because of this new ID, even though *B* is still in the RS and the ROB while *A* is being replayed, *A*'s data writeback cannot wakeup *B*, because *B* still has the physical register destination ID *#3* as its source operand. *B* reaches the ROB head, pseudo-retires, and moves into the WB. When *B* is replayed and reintroduced into the pipeline, it goes through the rename stage, gets a new ROB ID *#28* and receives the correct physical source register ID *#24*, re-establishing its link with *A* from the dependent RAT, as shown in Figure 15(d). Notice that an ‘x’ is shown for the other source IDs to indicate that the IDs of these sources are ‘don’t care’ for illustrating this example.

3.2.2 Tuned-CFP Execution Examples

As illustrated in the above examples, S-CFP causes excessive replays and rollbacks, negatively impacting performance and energy consumption. As a next step in this dissertation, we optimize S-CFP execution to overcome the limitations discussed above. We call the optimized S-CFP architecture as Tuned-CFP processor, the implementation of which is described in Chapter 4.

Figures 13(c) and 13(d) show the ROB and WB states in the Tuned-CFP architecture. Instruction *A* misses the first level cache and is marked as poisoned. However, unlike in S-CFP, it does not release its RS entry until it becomes a blocking instruction, just in case the load hits the L2 cache providing the miss data to the CFP core shortly. *A* may reach the head of the ROB before the L2 data cache loads the data, but it will still be kept in the RS and ROB by stalling pseudo-retirement as long as there are free entries in the ROB and other instruction buffers for the pipeline to continue execution of other instructions without blocking. If the miss data arrives before the pipeline blocks, *A* is woken up from the RS and ROB by clearing its poison bits, as shown in Figure 13(d). *A* and its dependents do not need to go through the replay loop at all in this example, saving significant time delay and energy.

Figures 14(d) and 14(e) show how the rollback situation in S-CFP is avoided with the Tuned-CFP architecture. Similar to the previous example, instruction *A* stays in the ROB, even if it reaches the head, as long as it is not blocking execution. *A* gets its wakeup before it moves into the WB, as shown in Figure 14(e). Even though *F* is a miss-dependent and mis-predicted branch, it executes before it pseudo-retires. When it reaches the head of the ROB, the ROB flushes the pipeline to clear all the wrong path instructions that have been fetched after the branch, and signals to the fetch unit to restart fetch and execution from the corrected target. The costly S-CFP branch recovery from the checkpoint has been avoided.

Figures 15(e)-15(g) illustrate a partial replay in the Tuned-CFP architecture, representing the same scenario discussed earlier in Figure 15(a). In Tuned-CFP, virtual register IDs that are not associated with any physical locations are used for register renaming and in the RS wakeup and scheduling logic. The virtual register IDs of instructions *A* and *B* are shown under their ROB entries in addition to the renamed source and destination virtual register IDs. As before, when *A* reaches the head of the ROB, it pseudo-retires and moves into the WB, as shown in Figure 15(e). However, unlike in S-CFP, *A* releases its RS but carries its virtual register ID *#3* along with it into the WB, as shown in Figure 15(f). Later on, when it wakes up and replays, *A* still carries with it its original virtual register ID *#3*, still maintaining its link with its dependent instruction *B* intact. This allows the RS to schedule *B* without replaying and renaming it again, as shown in Figure 15(g).

Avoiding excessive replays and rollbacks not only saves significant execution time by miss dependents but also considerable power consumption.

3.3 Limit Studies to Quantify the Limitations of S-CFP

To quantify the disadvantages of S-CFP described in Section 3.2, we used ideal studies to eliminate each of these disadvantages and to compare the performance against a non-optimized S-CFP architecture.

3.3.1 Buffer Full Condition

As we pointed out earlier in Figure 13, S-CFP does not wait for the buffer full *(bf)* condition before moving a load miss or its dependent instructions into the WB. To measure the impact of this issue, we evaluate a model in which a load miss instruction is prevented from moving into the WB until it blocks the core execution pipeline. Figure 16 shows the performance improvement due to this optimization as column *bf*.

Note that benchmarks, such as gcc and perl, benefit the most from keeping dependent instructions in the ROB/RS until they block the execution pipeline. This is because these benchmarks have significant number of L1 cache misses that actually hit the L2 and don't encounter the very long DRAM access latency. Therefore, leaving a load miss in the RS for a little longer significantly increases the probability that the load miss will receive a wakeup signal before it moves to the WB, thus saving the entire miss-dependence chain from being replayed. From Figure 16, the average contribution of *bf* optimization over all benchmarks is 2.7%. This speedup is not very high, mainly because the window of opportunity to save unnecessary replay is limited to the time between the load miss reaching the ROB head and the time the instruction buffer fills up. If the miss data does not come back within this time window, the entire dependence chain must be replayed.

3.3.2 Miss-Dependent Branch Mis-predictions

Miss-dependent branches that are later found to be mis-predicted cause performance degradation in CFP architectures because of the following reasons. First, execution needs to be rolled back to the checkpoint taken at the load miss to recover

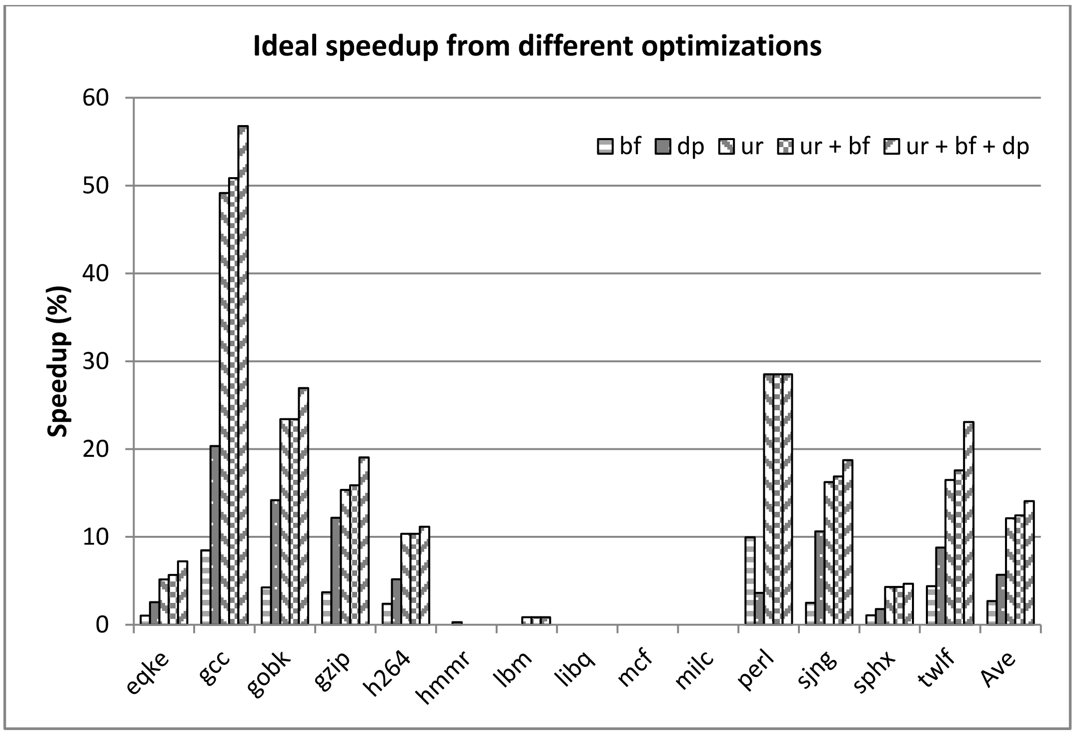


Figure 16. Limit studies showing potential speedup from buffer full, Oracle miss-dependent branch predictor, minimum replays, individually and combined

from the incorrect speculative updates made to the architectural state. Second, the miss-dependent branch itself may be resolved much later in time depending on when it replays from the in-order WB. Until a miss-dependent mis-predicted branch is resolved, S-CFP continues to fetch instructions from the wrong path. This not only wastes precious pipeline resources including cycles in the power hungry fetch and decode stages, but also exerts additional pressure on the execution pipeline, thus increasing the number of instructions that are moved into the waiting buffer and subsequently replayed.

In order to quantify the impact of miss-dependent mis-predicted branches on performance, we evaluate an S-CFP model with an Oracle predictor that stalls the processor front-end perfectly on a miss-dependent mis-predicted branch. This is the *dp* model in Figure 16. As can be seen in Figure 16 from the speedup of the *dp* model, the average impact of miss-dependent mis-predicted branches on performance is about ~5%. This is a moderate performance loss compared to that caused by replaying large number of miss-dependent instructions from the WB, as we present in the next section.

3.3.3 Unnecessary Replay Performance Overhead

In order to measure the negative impact of instruction replays on S-CFP performance, we evaluate an ideal model that eliminates unnecessary instruction replay. As described earlier in Section 3.2 and Figure 15, unnecessary replays are those caused by having to replay instructions that are still in the L1 RS when the miss load has been replayed in order to restore the dependence links. This model is called *ur* in Figure 16.

From Figure 16, it can be seen that 12% reduction in S-CFP performance comes from unnecessary replays. Virtual register renaming is therefore the most important optimization to S-CFP.

3.3.4 Combining Limit Studies

Figure 16 also shows two models *(ur + bf)* and *(ur + bf + dp)* that combine individual limit studies. As can be seen from these two ideal models, the potential performance benefit from multiple optimizations is cumulative. In particular, (*ur + bf + dp)* model, which combines the three optimizations, establishes the performance upper bound that can be achieved with a realistic Tuned-CFP machine exhibiting minimum possible replay and rollback. Importantly, it is evident from these results the importance of eliminating unnecessary replays with virtual register renaming to CFP. In the next section, we present simulated performance results for a realistic Tuned-CFP machine.

Summarizing Section 3.3, on memory intensive benchmarks, the average contribution from each optimization towards improving performance is 4% for *bf*, 9% for *dp*, 18% for *ur* and 22% when combined. None of the optimizations require power hungry structures and provide sizeable speedup to justify themselves. Since these optimizations complement each other very well, combining them to get the best possible performance would be the recommended option. The next chapter will talk about how these optimizations can be implemented on a CFP processor.

CHAPTER 4

TUNED-CFP ARCHITECTURE

Simultaneous-CFP executes both miss-independent and miss-dependent instructions concurrently while targeting L1 data cache misses. Since L1 misses occur more frequently, S-CFP ends up replaying a large number of instructions, sometimes losing the advantage gained from CFP latency tolerance. In this chapter, we will describe the core architecture of CFP with virtual register renaming (Tuned-CFP) that mitigates excessive S-CFP replay activity and execution waste.

4.1 Tuned-CFP Microarchitecture

In this section, we will describe Tuned-CFP core microarchitecture and the implementation details of the optimizations discussed in Section 3.2.2.

4.1.1 Microarchitecture Overview

Figure 17 shows a block diagram of the Tuned-CFP core. Tuned-CFP microarchitecture uses Tomasulo’s algorithm and reservation stations to perform data-driven, out-of-order execution [47].

Like other superscalar architectures, Tuned-CFP uses a reorder buffer to commit instructions and update register and memory state in program order. However, it does not use the reorder buffer for register renaming. Instead, it performs register renaming using virtual register IDs (VID) generated by a special counter. These VIDs are not mapped to any fixed storage locations in the core, thus can be large in number

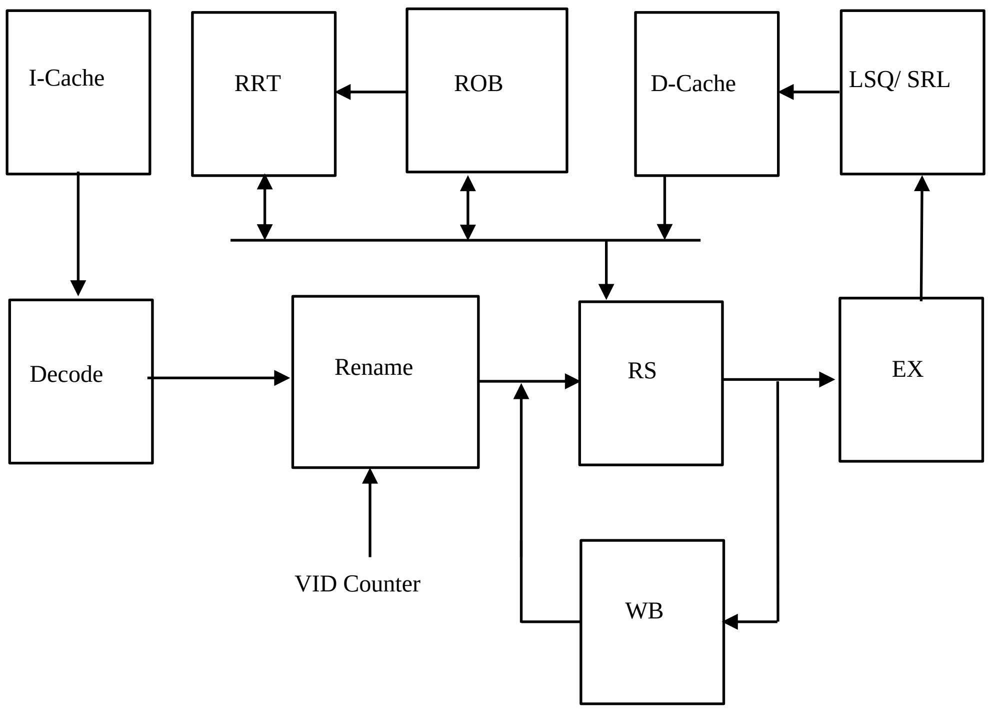


Figure 17. Block diagram of CFP architecture with virtual register renaming

and allocated to instructions throughout their life time, including miss-dependent instructions evicted to the waiting buffers. Since the VIDs are plentiful, Tuned-CFP does not run the risk of pipeline stalls resulting from miss-dependent instructions holding on to their renamed registers for a long time while waiting for the long latency load miss. The VID counter is finite in size and cannot be allowed to overflow in order to present allocating the same VIDs to multiple instructions in the pipeline. Tuned-CFP opportunistically resets the VID counter whenever it can, e.g. when the pipeline is flushed to recover from a mis-predicted branch. Otherwise, a pipeline stall and drain is forced to reset the counter when it overflows. With 10-bit VID counter, our simulations show that the impact on performance of forced pipeline stalls to reset the counter is negligible.

Virtual register renaming gives Tuned-CFP a significant advantage over previous CFP architectures by allowing Tuned-CFP to replay only a part of the load miss dependence chain.

4.1.2 Miss Independents Execution

Like previous CFP cores, The Tuned-CFP core is capable of executing cache miss-dependent and independent instructions concurrently in the pipeline, supported by two retirement register file contexts (RRF), one for retiring miss-independent instructions and the other for retiring miss-dependent instructions.

In Tuned-CFP, execution initially starts using a retirement register file context (RRF) that we call the independent retirement register file. When an L1 data cache load miss occurs, a poison bit is set in the destination reorder buffer entry of the load. Load-dependent instructions in the reservation stations (RS) capture the poison bit from the common write back data bus. They are then woken up, as if the load completed, and are scheduled by the reservation stations control logic for pseudo-execution. Pseudo-execution of poisoned instructions does not actually use any execution units. However, pseudo-execution consumes RS dispatch ports and writeback bus cycles to propagate poison bits through instruction dependences and to identify all instructions in the reservation stations that depend on the load miss data. After pseudo-execution, miss-dependent instructions stay in their reservation stations until they wake up for real execution when the load miss data arrives, or until they are moved into the waiting buffer.

4.1.3 Replay Loop and Miss Dependents Execution

Figure 17 shows the reduced replay loop in Tuned-CFP consisting of two stages: the reservation stations (RS) and the waiting buffer (WB). The waiting buffer basically acts as a second level storage for the reservation stations. With virtual register renaming, entries can be freely evicted from the RS to the WB and then loaded back again to the RS to be scheduled for execution at a later time.

In Tuned-CFP, miss-dependent instructions are evicted from the reservation stations to the waiting buffer only when their buffer resources are needed to unblock the execution pipeline. Therefore, when a miss is processed and its data fetched to the L1 data cache, the miss-dependent instructions may still be in the reservation stations. In this case, when the data is written back, the miss data is captured by all the reservation stations that have instructions that depend on the miss. The captured writeback data sets the source operand “ready” state bits and clears the poison bits of the dependent reservation station entries, making the instructions in these entries ready for scheduling and dispatch to execution.

Evicting miss-dependent instructions to the WB on resource need basis significantly reduces the number of replayed instructions, especially in the case of medium latency load misses, which are those that miss the L1 data cache but hit the on-chip L2 cache.

In case of a load miss to DRAM, it is often the case that the long miss latency causes the instruction buffers to fill up and some or all of the in-flight miss-dependent instructions to evict to the waiting buffer. When the load miss is serviced, the miss load and its dependents are re-inserted from the waiting buffer back to the reservation stations where they are scheduled for execution.

Notice that even though replayed miss-dependent instructions do not need to be renamed again, they sometimes, as in Tomasulo's algorithm, need to read source operands that have already been computed and retired from the ROB to the register file (RRF). State bits that track whether the last instructions to write logical registers have been retired are stored in a special storage structure. These state bits are checked during replay to determine if the operands are ready in the RRF, and to read them and move them into the RS with the replayed instructions.

4.1.4 Reservation Stations

Tuned-CFP uses a centralized array of conventional data-capture reservation stations [39]. Each reservation station entry is extended with a poison bit per source operand and L1-DCache-miss bit. The L1-DCache-miss bit is set to 1 if the entry contains a load instruction that has missed the L1 data cache. We say an instruction is poisoned if one of its source poison bits or the L1-DCache-bit is set to 1. A source operand of an instruction is poisoned if and only if it is the destination of another poisoned instruction. In other words, the poison bits propagate the dependences from L1 data cache misses to later instructions in the program to identify instructions that may encounter long data cache miss delays. These instructions are candidates to move to the waiting buffer to avoid pipeline stalls that could occur if any of the reservation stations, reorder buffer, load queue, or store queue arrays becomes full.

The reservation stations array is augmented with a free list and an order list. Tuned-CFP uses the free list to track the occupancy of the reservation stations. The order list tracks the program order of the reservation stations. The RS order list could be implemented as part of the reorder buffer by adding the reservation station ID of each instruction to its allocated reorder buffer entry. It also could be implemented as a special array separate from the ROB.

Four conditions are checked to determine if an instruction should be moved to the waiting buffer: 1) the instruction is at the head of the RS order list, 2) the instruction is poisoned, 3) one of the RS, reorder buffer, load queue or store queue arrays is full, and 4) every source operand of the instruction is either poisoned or ready. The last condition ensures that the miss-dependent instructions carry their non-poisoned input values with them when they are replayed, since there is no guarantee that these values would not be overwritten in the register file by replay time. In addition, each reservation station has a state bit that indicates if the instruction in the entry has been replayed once before, i.e. it has been moved earlier in time to the waiting buffer and then back to the RS array. Each reservation station also contains a load miss identifier, in case it has a load instruction that misses the data cache. An implementation could use for this purpose the ID of the L1 data cache fill buffer used to handle the load miss.

4.1.5 Waiting Buffer

The waiting buffer is a wide single ported SRAM array managed as a circular buffer using head and tail pointers. Miss-dependent RS entries at the head of the RS array moves to the tail of the waiting buffer when any of the instruction buffers fills up due to data cache misses. When a data cache miss is completed, Tuned-CFP replays the miss-dependent entries by loading them back from the head of the waiting buffer to the tail of the RS. Ideally, the width of the two buses connecting the RS and the waiting buffer would match the pipeline width. Narrower interconnect can also be used, trading some performance for simpler hardware.

A key to the efficiency of Tuned-CFP large window design is the fact that the waiting buffer has no CAM ports, connections to write back buses for capturing data operands or conventional ready/schedule logic. All these functions are handled in the RS array after the data cache miss completes and the miss-dependent instructions are replayed. Therefore, the waiting buffer array can be designed using non-tagged SRAM and made significantly larger than the RS array at much lower area and power cost than if the RS array is large enough to hold the full instruction window.

In order to wake up miss dependents from the waiting buffer and replay them, the L1 data cache fill buffer handling a load miss has to receive and save the waiting buffer ID of its load miss. When the miss is completed, Tuned-CFP replays the load miss and its dependents in program order, as described earlier, from the head of the waiting buffer back into the RS allocate/write stage of the execution pipeline.

4.1.6 Register File and Results Integration

Tuned-CFP uses the reorder buffer to handle branch mis-predictions and exceptions incurred by miss-independent instructions. On the other hand, it uses checkpoints to handle branch mis-predictions or exceptions encountered by miss-dependent instructions.

Like S-CFP architecture [23], Tuned-CFP has a specialized register file for checkpointing register state at the load miss, for later use to handle miss-dependent branch mis-predictions and exceptions. The register file also has special logic for integrating the results of independent and dependent instructions and to restore precise register state after all miss-dependent instructions execute.

In addition to the checkpoint bit and the independent RRF context bit, Tuned-CFP register file cell contains one context bit for the dependent RRF state (rightmost latch). The integration of the independent and dependent instruction results is done in one restore cycle. At the end of dependent execution, after all instructions in the WB have replayed and retired, the RRF has all the live-out registers, some of which computed by the independent instructions and some by the dependent instructions. This is determined by the poison bits in the RRF. To integrate these results back into one context, a restore cycle is performed from the dependent context into the independent context. However, not all registers are copied. Only poisoned registers are copied by using the poison bits to enable the clock of the copy operation. A 2-to-1 multiplexer in the cell restores either the checkpoint bit or the dependent bit during a RSTR_CLK cycle.

4.1.7 Load and Store Execution

To maintain proper memory ordering of loads and stores from the independent and dependent instructions execution, Tuned-CFP uses load and store queues (LSQ), a Store Redo Log (SRL) [14] and a store-set memory dependence predictor [8]. A detailed description of Tuned-CFP Store Redo Log and the speculative L1 data cache is presented in Chapter 5.

4.1.8 Miss-Dependent Branch Predictor

A key reason for CFP performance degradation is dependent branches that go into the waiting buffer and are later found to be mis-predicted. Factors contributing to this performance degradation include: a large window of wrong path instructions, re-execution of instructions between the load miss and the mis-predicted branch, and delayed resolution of the miss-dependent branch while waiting its turn to come out of the WB.

To address this problem, we use an approach similar to pipeline gating [31], except that we apply it only to costly dependent branches. We identify branches that are likely to mis-predict and take necessary action when they move into the WB. We have observed that in multiple benchmarks there is a strong correlation between the dependent mis-predicted branch and its PC value. We use a small hardware predictor of 32 entries that contains the addresses of previous dependent mis-predicted branches to estimate branch confidence. The processor front end is stalled when a branch with low confidence is moved into the WB. The front end of the pipeline is unblocked only after the load miss data is delivered to the cache and the branch is resolved. Our results in section 5.2.2 show that this mechanism reduces excessive replay and rollback execution.

The next chapter will explain the mechanisms used in our CFP architecture to manage instructions that access memory.

CHAPTER 5

MEMORY ORDERING MECHANISMS

In this chapter we will explain the mechanisms used in this dissertation to ensure correct execution of loads and stores that belong to a large instruction window. To maintain proper ordering of memory instructions from the independent and dependent threads execution, we use load and store queues (LSQ), a Store Redo Log (SRL) [14] and a store-set memory dependence predictor [8]. All stores, dependent and independent, are allocated entries (and IDs) in the SRL in program order by the independent thread. Every load, dependent or independent, carries the SRL ID of the last prior store. The SRL IDs assigned to loads and stores are unique and determine the order of memory instructions.

The independent thread performs load memory dependence prediction, with a store-set predictor [8], to determine the store on which a load may depend. The predictor uses SRL IDs in the prediction and writes store poison bits in the SRL, thus allowing propagation of poison bits from stores to predicted-dependent loads.

We next describe the architecture needed to support the SRL mechanism and speculative cache. The SRL mechanism from [14] is modified in this work to support concurrent execution of loads and stores from both independent and dependent threads.

5.1 Architecture Support for Speculative Cache and SRL Mechanism

This section talks about the L1 data cache support and how speculative load and store accesses from dependent and independent threads are handled.

5.1.1 L1 Data cache state

In order to support simultaneous execution of dependent and independent loads and stores, a data cache block has 2 new states: Speculative Independent (*Spec_Ind*) and Speculative Dependent (*Spec_Dep*). A block that is not in one of these two states is committed and would be in one of the states defined by the cache coherence protocol, e.g. Shared, Exclusive, or Modified in a MESI coherence protocol. Note that the second level cache is kept completely non-speculative, eliminating the need to have the special state bits there.

While the processor stays in CFP mode, stores from both the independent and dependent threads update the cache blocks, including the corresponding *Spec_Ind* or *Spec_Dep* bits. It could so happen that the same address might occupy two different blocks in the same set, one in *Spec_Ind* state and the other in *Spec_Dep* state. The independent and dependent state bits have to be matched along with the address tag to determine an address hit. The execution example discussed in Section 5.2 will explain how such a situation can occur.

5.1.2 Independent thread store execution

Independent stores from the independent thread are written speculatively into the first level cache after they pseudo-retire, setting the *Spec_Ind* bit of the written block. If an independent store hits a dirty block in the L1 data cache, the block is written back to the L2 cache before the store writes the block and changes it into *Spec_Ind* state. If an independent store address matches the address of a *Spec_Dep* block, it is handled as a cache miss and another cache block in the set is allocated to the independent store. Independent stores are also written in the SRL buffer at the same time they are written into the L1 data cache, as shown in Figure 18(b).

5.1.3 Dependent thread store execution

When dependent stores from the dependent thread execute, they are written into the SRL, but not in the SQ or data cache. After the dependent stores retire, the dependent thread writes all stores, dependent and independent, from the SRL into the data cache in program order, setting the written cache block to *Spec_Dep*. If a store hits a *Spec_Ind* cache block, the cache treats the store as a miss and allocates for it a new block in *Spec_Dep* state. Notice that Independent stores are written twice into the data cache: 1) speculatively by the independent thread to forward data to independent loads, and 2) by the dependent thread, interleaved with dependent stores in program order to enforce a final, correct order of memory writes. After the dependent thread executes, and the SDB and SRL become empty, *Spec_Ind* blocks in the cache are bulk flushed, and *Spec_Dep* blocks are bulk committed, leaving in the cache only ordered stores data.

5.1.4 Independent thread load execution

Independent loads read *Spec_Ind* or Committed blocks in the L1 cache, whichever they happen to hit. Independent stores therefore forward data to their descendent independent loads through the data cache, long before they are actually committed. This allows S-CFP to de-allocate stores from the reduced store queue immediately at pseudo-retirement, keeping the store queue as small as possible. An independent load that does not hit a *Spec_Ind* block but only hits a *Spec_Dep* block in the cache can also forward the data from the cache as explained in Table 1. This is a situation where the SRL has updated the cache and an independent load to the address is issued on its first wave much later. The independent load cannot read the poison bit because the SRL entry has been released. This is very similar to a partial replay case discussed in Figure 15 except that in this case, memory instructions are involved.

5.1.5 Dependent thread load execution

Dependent loads are re-issued from the SDB and execute from the data cache after all stores ahead of them in the SRL are written to the cache. Synchronizing the dependent load execution with the stores ahead of them in the SRL is performed using the SRL ID assigned to every load. The load SRL ID identifies the entry in the SRL that was assigned to the last prior store. Notice that since dependent loads execution is synchronized with older SRL stores, memory dependence mis-predictions of dependent loads are irrelevant to execution or performance. A dependent load can only read data from a committed block or *Spec_Dep* block. If the address of a dependent load matches the address of a *Spec_Ind* block, it is treated as a miss. When this load reaches the ROB head, the processor is stalled until the miss data is returned.

5.1.6 Recovering from memory dependence mis-predictions

Since dependent load execution is synchronized relative to prior stores in the SRL, only memory dependence mis-predictions of independent loads that actually depend on miss-dependent stores will cause memory ordering violations. All independent and dependent load addresses are cached in a set-associative address buffer. Dependent stores snoop this load address buffer when they are dispatched by the dependent thread to the cache. A hit to an independent load address indicates an ordering violation. In this case, the pipeline, SDB, SRL, RAT, and all *Spec_Dep* and

Spec_Ind blocks in the cache are bulk flushed. Execution then rolls back to the checkpoint. Finally, to enforce memory consistency, the load address buffer snoops committed stores from other threads.

5.1.7 Victim Cache

During CFP mode, if an instruction misses the L1 cache, a block needs to be evicted to make way for the incoming block. The only candidates for eviction are non-speculative blocks which can be written back to the next level cache. As mentioned before, speculative bits are limited to the L1 cache and cannot spill over to the next level. If there are no non-speculative blocks available in the L1 cache to be evicted, then a small associative victim cache is brought to use. A new entry is created for the incoming block, if it already does not exist in the victim cache.

If there is no space left in the victim cache and a speculative store does not find a place both in the L1 cache and victim cache then this is a point of no return; the processor state needs to be rolled back to the checkpoint. However, the situation is different for a speculative load, since it can be serviced by providing the data to the core without bringing the block into the L1 cache or victim cache. From our simulations we see that an 8 entry fully associative victim cache is enough to support all speculative blocks without needing to evict any of them and rollback execution to the checkpoint.

Table 1 summarizes all possible speculative accesses to the cache and how they are handled by the cache control logic.

5.2 Working of Speculative Cache with an Execution Sequence

Figure 18 shows an example to demonstrate the working of the speculative cache and SRL algorithm. The execution sequence is shown in Figure 18(a). The instruction numbers are shown in the figure along with the load and store information. An *L* stands for a load and *S* for a store. Assume that all instructions access the same address. The shaded instructions are miss-dependent. Instruction *#1* is a load miss, *#2* is a store that depends on this miss. Assume that the stores and loads are sufficiently far away from each other that the store cannot forward its data to a dependent load through the store queue.

The following rows indicate the states of the reorder buffer (ROB), waiting buffer (WB), store redo log (SRL) and a two way set associative L1 data cache. The additional state bits in the cache are shown just above the label L1 D-Cache as D (for speculative *Dependent*) and I (for speculative *Independent*). The tag bits in the two ways of the cache are shown as *T_1* and *T_2*.

Figure 18(b) shows the processor state when the miss-dependents have moved into the WB. The miss-independents continue to be processed in the ROB. The SRL entries for the two stores *#2* and *#41* are allocated at rename.

Table 1. Summary of speculative accesses to the cache

Access Type	Condition	Action	Eviction Priority	Comments
Spec-Independent Store	Hits *Spec-Ind* block	Return hit	-	
	Hits non-Spec block	Return hit	-	
	Hits *Spec-Dep* block	Create new block	1) non-Spec block in cache 2) non-Spec block in victim buffer	This speculative store needs a place somewhere. If there is no space in either cache or victim buffer, execution needs to rollback to the checkpoint.
	Does not hit any block			

Spec-Dependent Store	Hits *Spec-Dep* block	Return hit	-	
	Hits non-Spec block	Return hit	-	
	Hits *Spec-Ind* block	Create new block	1) non-Spec block in cache 2) non-Spec block in victim buffer	This speculative store needs a place somewhere. If there is no space in either cache or victim buffer, execution needs to rollback to the checkpoint.
	Does not hit any block			
Spec-Independent Load	Hits *Spec-Ind* block	Return hit	-	
	Hits *non-Spec* block	Return hit	-	
	Hits *Spec-Dep* block only - Does not hit *Spec-Ind* block	Return hit	-	This is a situation where SRL has updated the cache and the load comes much later. This is the partial replay equivalent for memory instructions.
	Does not hit any block	Return miss. Poison the load. Bring the data in and create new block	1) non-Spec block in cache 2) non-Spec block in victim buffer	If there is no space in both cache and victim buffer, the load miss can be serviced directly from the higher level without bringing the block in.
Spec-Dependent Load	Hits *Spec-Dep* block	Return hit	-	
	Hits non-Spec block	Return hit	-	
	Hits *Spec-Ind* block only - does not hit *Spec-Dep* block	Return miss. Do not Poison the load because this is a *miss-dependent-miss*. Stall the processor until miss data returns. Bring the data in and create new block	1) non-Spec block in cache 2) non-Spec block in victim buffer	If there is no space in both cache and victim buffer, the load miss can be serviced directly from the higher level without bringing the block in.
	Does not hit any block			

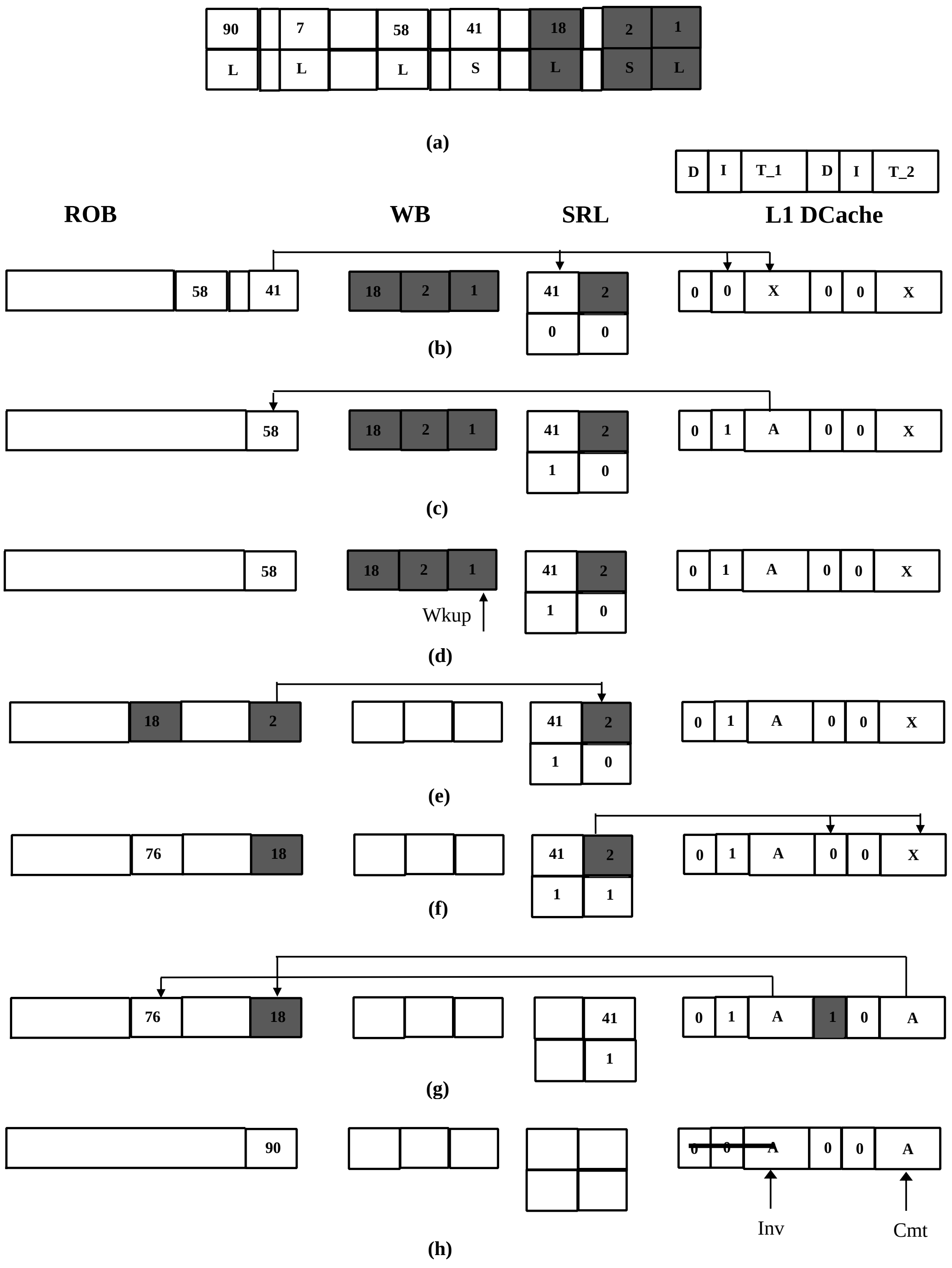


Figure 18. Working of SRL algorithm with an execution sequence

Their ready bits currently read “0” because the stores are yet to complete. The cache state is ‘don’t care’ currently and the address in question is not present in it.

When *#41* completes execution, it updates its allocated entry in the SRL with its data and sets the ready bit. At the same time it updates the cache speculatively and sets the *Spec_Ind* bit as shown in Figure 18(b). Notice that a new block is allocated to the store since it does not match a *Spec_Ind* block.

Younger speculative loads like *#58* match their *Spec_Ind* bit in addition to the tag and forward the data from the cache as shown in Figure 18(c). At some later point in time, the wakeup for instruction *#1* arrives as shown in Figure 18(d), allowing the miss-dependents to be reintroduced into the pipeline.

On replay, the miss-dependent store *#2* completes, writes its data to the SRL and updates the ready bit in the SRL as shown in Figure 18(e). However, it does not update the cache. In general, the SRL is made up of miss-independent and miss-dependent stores. The independent stores update the SRL while the miss is outstanding, while the dependent stores fill up their SRL entries on replay.

Now that the head of the SRL is ready, it can update the cache non-speculatively in program order and also set the *Spec_Dep* bit as shown in Figure 18(f). Notice that again a new block is allocated to this store access from the SRL because it does not match a *Spec_Dep* block. At the same time, load instruction *#76* enters the pipeline.

At this point in time, the same address is present in the cache in two versions, *Spec_Ind* and *Spec_Dep*. They forward their data to the respective loads by matching the corresponding bits in addition to the address tag. In other words, the shaded cache block forwards to the shaded load while the non-shaded block forwards to the non-shaded load, as shown in Figure 18(g). Finally when the waiting buffer and SRL empty and speculative updates to the cache are known to be correct, all *Spec_Ind* blocks are bulk invalidated and all *Spec_Dep* blocks are bulk committed, as shown in Figure 18(h). At this point, load instruction *#90* can read its data from the cache like a normal cache hit.

5.3 Synchronization of Loads and Stores

After a load has been moved into the WB, all updates made to the cache are speculative. Some updates made until the first branch encountered may still be correct, but the memory state should always correspond to the checkpoint taken at the load miss. For this reason, all updates made to the cache from the load miss onwards are marked as speculative. There are usually long execution phases where instructions from the WB are reissued for execution in CFP mode. Once the processor enters CFP mode, it comes out of CFP mode without needing recovery action only when all the following conditions are satisfied - 1) WB empties 2) all instructions that are reissued from the WB retire 3) SRL catches up with the ROB. Before these conditions are satisfied, if another load misses the cache and moves into the WB, then execution stays in CFP mode and waits for the next opportunity to enter normal execution mode.

The logical register file and the cache represent the architectural state of the processor. Both the register file and cache are updated speculatively and when speculative updates are found to be incorrect, the register state is recovered from the checkpoint, while the processor state is recovered by bulk invalidating all speculative blocks. Since in CFP mode, the ROB updates the register state and the SRL updates the cache state, completely independent of each other, merging of the independent and dependent states and exiting from CFP mode can happen only when the SRL and ROB catch up with each other to produce a consistent architectural state. In this work, we opportunistically look to merge the states whenever the WB empties. When the WB empties and all the miss-dependents complete, we preclude poisoning further load misses until the SRL catches up with the ROB state and the checkpoint is advanced. In practice, waiting for SRL to catch up with the register state update does not impact performance badly because all the stores would have updated the ready bit in the SRL by now; the independent stores much before in time, while the dependent stores by virtue of all WB instructions completing execution. So the SRL will update the cache as quickly as the bandwidth between the SRL and cache can support, thus providing an opportunity to advance the architectural state.

Another requirement for correctness is the synchronization of dependent loads with respect to the store ahead of it. When dependent loads which do not miss the cache but are predicted to be dependent, replay from the WB, they cannot directly forward their data from the store queue. This is because the small energy efficient store queue holds only a subset of all the stores in the instruction window. With this limited information, it is not possible to clearly disambiguate the store that will forward to the dependent load. For this reason, every load notes down the ID of the store immediately ahead of it and the dependent load is replayed only when the SRL state has reached this noted store. In other words, every dependent load which does not miss the cache waits until all stores before it write to the cache from the SRL before being re-issued. For example in Figure 18(f), the dependent load *#18* has to wait until all the stores ahead of it update the cache. Since *#2* is the latest store ahead of it, it forwards the data from *#2* from the cache and not from the store buffer.

In addition to the above requirement, stores to the cache also need to be synchronized with respect to the loads to make sure loads forward correct data from the cache. The synchronization logic has to make sure updates to the cache from the SRL only happen after all the dependent loads have read their data from the cache. For example, in Figure 18(g), even though store *#41* is ready, the SRL update of this store has to wait until load *#18* has successfully read its data from the shaded *Spec_Dep* block. If *#41* is allowed to update the cache without this synchronization, load *#18* will end up forwarding incorrect data from the cache. To support this synchronization, a simple bit vector is used to keep track of all the loads and stores in the instruction window, similar to the one suggested in [14]. A dependent load that needs to forward its data from a *Spec_Dep* block in the cache sets a bit in the SRL entry of the store immediately ahead of it to indicate to that store to hold its update to the cache. This store will then wait until all the load bits in the bit vector are set before it updates the cache.

5.4 Memory Ordering Design Choices

On a cache miss, a block needs to be evicted to make way for the incoming block. If all the blocks in the set are speculative and the victim cache is full, the processor state needs to be flushed and execution is rolled back to the checkpoint. One way to minimize the chances of such costly rollback is to victimize *Spec_Ind* blocks as well and poison *Spec_Ind* accesses that miss the cache. An *overflow* bit in the cache can be set when a *Spec_Ind* block is victimized to make sure only addresses belonging to this set are poisoned. The idea is that these *Spec_Ind* blocks would anyway get a second chance to execute with correct data when they replay from the WB. These loads could be poisoned normally just like a first wave load miss. However, these loads will need to be marked differently to wake them up when they reach the WB head.

In this case, three kinds of load misses will exist in the WB – 1) a normal load miss, 2) a load that is predicted to forward from a poisoned store queue entry and 3) a speculative independent load that hits a block with its *overflow* bit set.

A normal load miss is re-issued from the WB head only when it is qualified by the cache controller wakeup signal. A load that is predicted to forward data from a poisoned store queue entry can directly issue from the WB head without a wakeup signal because the forwarding store would have been issued before it from the in-order WB. The third kind of load miss that hits a block with the *overflow* bit set can also issue when it reaches the WB head because it can be guaranteed that the store ahead of it has been re-issued.

However, note that *Spec_Dep* blocks cannot be evicted similarly because they are given only one chance to execute.

In this work, in order to keep the design simple and also save additional hardware, we choose not to evict speculative independent blocks from the cache. Only *Spec_Ind* load misses to non-speculative blocks are poisoned and moved into the WB.

The non-speculative blocks occupy the victim cache when evicted from the L1 cache. But the speculative blocks always have priority over the non-speculative blocks when it comes to occupying a slot in the victim cache. A non-speculative block always makes way to a speculative block if need be.

CHAPTER 6

EXPERIMENTAL SETUP

We built our Simultaneous-CFP architecture model on the Simplescalar ARM ISA simulation infrastructure [51]. CFP benefits applications that suffer high data cache miss rates. However, its effectiveness in handling data cache misses is limited by mis-predicted branches that depend on data cache misses. For this reason, we used all 14 “C” benchmarks from SPEC 2000 and SPEC 2006 that we succeeded in compiling using the Simplescalar cross compiler tool. The benchmarks and their inputs are detailed in Table 3.

These benchmarks do not suffer much on average from data cache misses but have high branch mis-prediction rates which make them useful in exposing the performance limitations and glass jaws of CFP, thus making them appropriate for evaluating the effectiveness of our proposed optimizations. Our choice of ARM ISA is arbitrary and does not change the conclusions in the dissertation, since data cache miss rates and branch mis-predictions depend mainly on the application characteristics and not the ISA.

After skipping the initialization code and warming up the caches and predictors for 40 million instructions, we simulated 200 million instructions from each benchmark, consisting of four different samples manually selected from representative execution phases to display wide variation in the cache miss rate as well as the branch mis-prediction rate, both of which significantly impact CFP execution behavior. Results in chapter 7 show the average performance of these selected samples for each benchmark.

Table 2. Simulated Machine Configuration

Pipeline	Fetch to retire15 stages, rename to retire 8 stages, 4-wide
Instruction buffers	80 ROB, 40 RS, 40 LQ, 30 SQ
L1 I-Cache	16KB, 8-way, 3 cycles, 64-byte line
L1 D-Cache	8KB, 2-way, 3 cycles, 64-byte line
L2 cache	Unified, 512KB, 16-way, 32 cycles, 64-byte line
DRAM latency	150 cycles L2 to data return with on-chip DRAM controller
Branch predictor	Combined bimodal and gshare, 4K Meta, 4K bimodal, 64K gshare, 4K BTB, 16 entry Return Address Stack
Hardware data prefetcher	Stream based (16 streams)

For energy analysis, we have obtained measurements from SPICE circuit simulations of all baseline superscalar and CFP core functional blocks, using Cadence tools and 45nm process technology. We combined these with logic switching activity from our Simplescalar simulator, creating an Architectural Level Power Simulator (ALPS) [6]. All energy results reported in this dissertation account for total energy, which is the sum of dynamic and static (or leakage) energy. Our model assumes 30% of energy consumed per instruction to be due to leakage, which is typical of current high performance integrated circuits technology. The energy model accounts for the

Table 3. Details of benchmarks

Benchmark	Inputs	Comments
equake	Ref	Seismic wave propagation simulation, SPEC 2000
gcc	166, 200, c-typeck, cp-decl, expr, expr2, g23, s04, scilab	C compiler
gobmk	13x13, nngs, score2, trevorc, trevord	Artificial intelligence: go
gzip	Input	Compression, SPEC 2000
h264ref	foreman_baseline, foreman_main, sss	Video compression
hmmer	nph3, retro	Search gene sequence
lbm	ref	Fluid dynamics
libquantum	Ref	Physics/ Quantum computing
mcf	Ref	Combinatorial optimization
milc	Ref	Physics/ Quantum chromodynamics
perl	checkspam, diffmail, splitmail	Programming language
sjeng	Ref	Artificial intelligence: chess
sphinx	Ref	Speech recognition
twolf	Ref	Place and route simulator, SPEC 2000

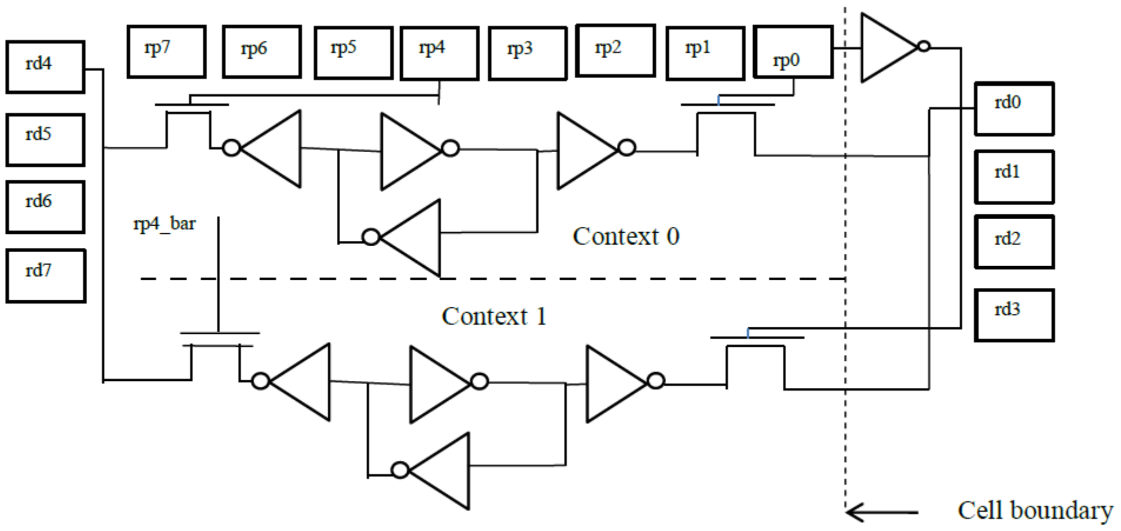


Figure 19. Single ended read ports

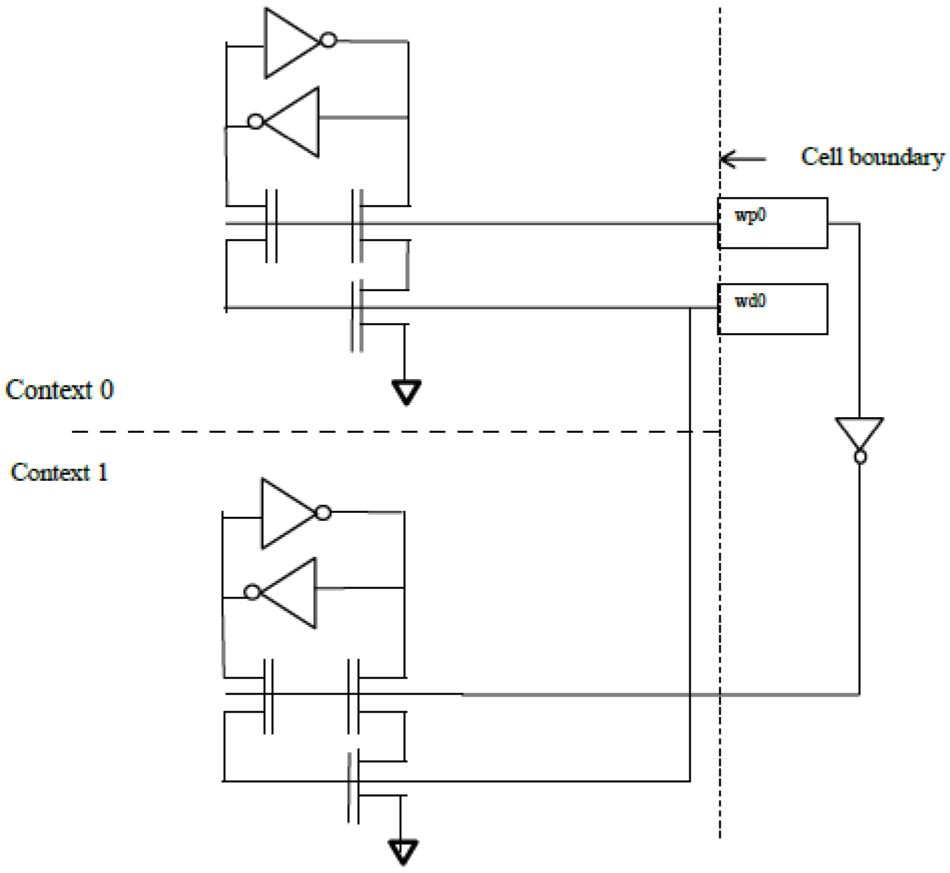


Figure 20. Write mechanism – 4 write ports

additional-CFP structures area, including 8K bytes of SRAM in the SRL, SDB, two thread contexts, checkpoint and result integration support in the RRF. Our estimate for the area overhead of CFP adds up to less than 5% of the non-CFP core configuration shown in Table 2. We used 6-transistor SRAM cell design with differential sense amplifiers for the SRL and SDB circuits and the single-sided register file cell described in [20], augmented with checkpoint [22] and results integration circuit. We used 6-transistor SRAM cell design with differential sense amplifiers for the SRL and WB circuits and a single-sided register file cell, augmented with checkpoint and results integration circuit. Figures 19 and 20 show the basic register cell circuit with eight read ports and four write ports. The cell includes the two contexts. We include logic in Figure 19 for reading from ports 0 & 4 in contexts 0 & 1 and show the load on the respective ports. In Figure 20 we show one write port connection in both contexts.

Unless specified otherwise, the average consumed energy per instruction (EPI) is reported for each evaluated core relative to the non-CFP baseline configuration.

Table 2 shows the simulated machine configuration. Since our simulations are done with a single core model, we use a two-level cache hierarchy with an L2 cache size that is representative of the L2 capacity per core of current multicore processors. We selected optimum instruction buffers and L1 data cache sizes for maximum EPI efficiency, as described in section 7.3.

CHAPTER 7

EXPERIMENTAL RESULTS

This chapter evaluates the performance and energy consumption of the baseline non-CFP, Simultaneous-CFP and Tuned-CFP cores. The analysis is split up into four sections. The first section establishes the sizes of instruction buffers and data cache for optimum energy efficiency. It introduces a metric to represent the energy efficiency of a processor core. In the second section the performance of the Tuned-CFP core is compared against that of non-CFP core and other latency tolerant cores. The third section compares the energy consumed per instruction by non-CFP, S-CFP and Tuned-CFP cores. The fourth section compares the energy efficiency of the three cores.

7.1 Analysis of Energy Characteristics of Non-CFP Superscalar Cores

This section presents an analysis of energy efficiency of a conventional 4-wide superscalar core that does not implement CFP, as the core instruction buffers and L1 cache sizes increase. The motivation is to determine appropriate instruction buffer and cache sizes by using an intuitive definition of energy efficiency. First, we define *Return on Energy* (ROE) from an added hardware feature to be the percent increase in performance divided by the percent increase in energy per instruction (EPI) resulting from the added feature. Using ROE definition, we say that core *A* is more energy efficient than core *B* if core *A* return on energy (ROE) is larger than 1. In other words, core *A* is more efficient than core *B* if *A* improves performance by a larger percentage than the percentage of additional energy it consumes relative to core *B*.

7.1.1 Energy Characteristics of 4-wide Non-CFP Superscalar Core with Ideal Data Cache

The sizes of instruction buffers and the L1 data cache significantly impact performance and energy per instruction of superscalar cores. In this section, we investigate the performance and ROE of non-CFP superscalar cores as the instruction buffers sizes increase. We simulate an ideal data cache to isolate the contribution coming from instruction buffers alone. We vary the sizes of instruction buffers, namely reorder buffer (ROB), reservation station (RS), load queue (LQ) and store queue (SQ) from a 32_16_16_12 configuration to a 192_96_96_72 configuration (ROB_RS_LQ_SQ). All other parameters like machine width, pipeline length, branch predictors, and instruction cache size remain unchanged and are as shown in Table 2.

Figure 21 shows the percentage speedup of non-CFP superscalar core with ideal L1 data cache for various buffer sizes. All speedups are reported as percentage increase in IPC relative to the minimum 32_16_16_12 configuration. Only a representative set of benchmarks are shown in the figure for clarity. It is clear from Figure 21 that for all benchmarks, speedup initially increases linearly and then saturates as the buffers sizes increase.

Figure 22 shows a plot of the return on energy (ROE) for various configurations relative to the minimum 32_16_16_12 machine, and across all benchmarks. We identify two configurations of interest in Figure 22. The maximum ROE occurs at the 48_24_24_18 configuration for all benchmarks. This *peak return on energy* configuration is the most energy efficient configuration by our definition. A second configuration of interest is the one for which most benchmarks have a value of ROE around 1. From Figure 22, this is configuration 128_64_64_48. We call this

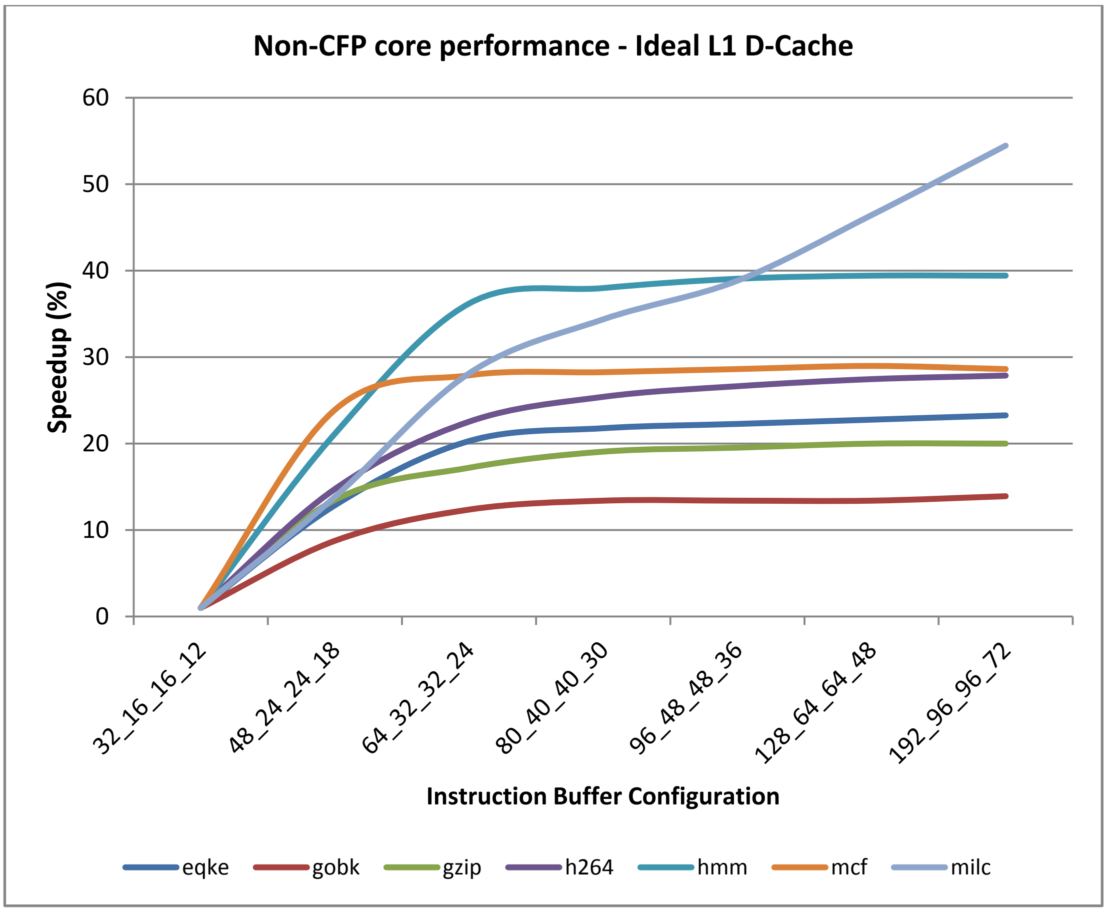


Figure 21. Percent speedup of non-CFP 4-wide superscalar cores of different instruction buffer configurations and ideal data cache over minimal configuration core

configuration the *point of diminishing return on energy*. If the instruction buffers are increased beyond this point, the resulting speedup will be smaller than the accompanying increase in energy. We next use this configuration to determine an optimal L1 data cache configuration.

7.1.2 Selecting the L1 Data Cache Size

The L1 data cache configuration is critical towards deciding the core power consumption. The data cache is typically optimized for lowest possible load hit latency with high associativity and aggressive circuit design that uses concurrent read of the

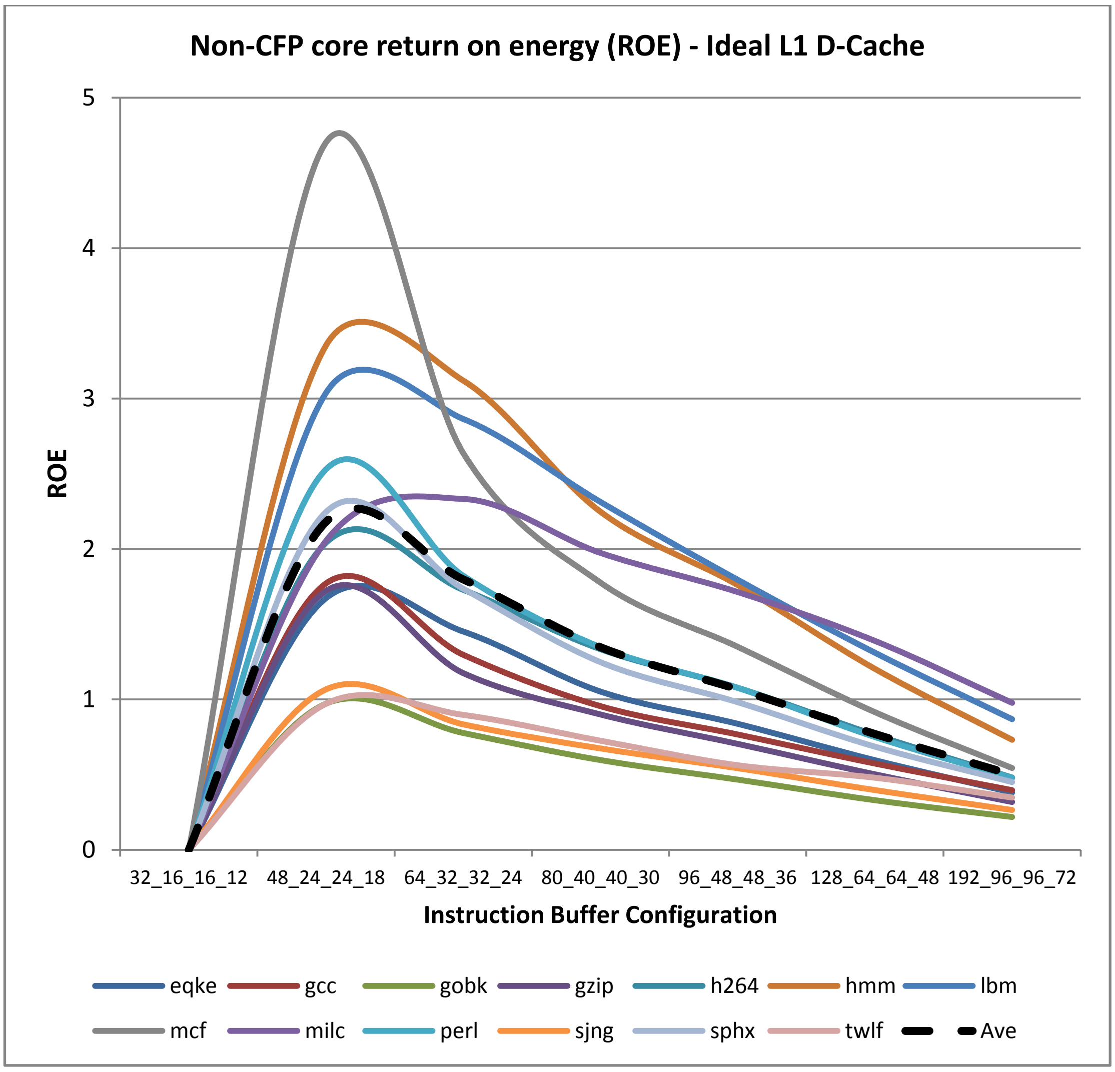


Figure 22. ROE of non-CFP core with ideal data cache for different buffer configurations

tags and data from all ways in the indexed set. To find the optimal L1 cache configuration experimentally, we use the *point of diminishing return* buffers configuration and vary the L1 data cache size from 4KB to 32KB and associativity from 1 to 8. Our simulations show that the peak ROE occurs at 2-way, 8KB L1 D-cache while the point of diminishing ROE occurs at 4-way 16KB L1 D-cache size. We do not show The L1 data cache ROE plots to avoid repetition.

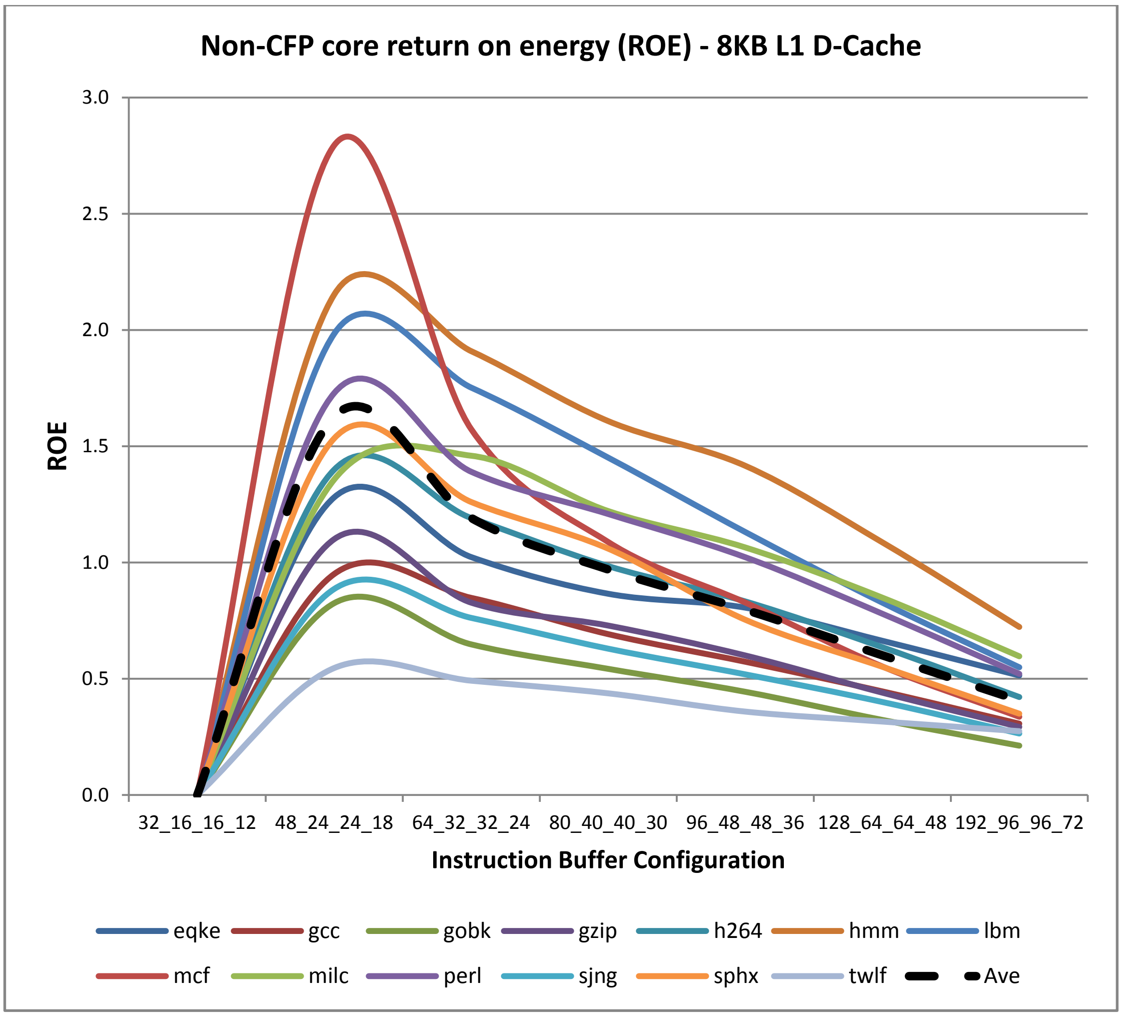


Figure 23. ROE of non-CFP core with 8KB L1 D-cache for different buffer configurations

7.1.3 Varying Instruction Buffer Sizes with Optimal L1 Data Cache

To see how the ROE curves of the non-CFP core behave when simulated with a practical cache configuration, we re-do the experiment shown in Figure 22 with the peak ROE 8-KB L1 data cache instead of ideal cache. Figure 23 shows the results of this experiment. Figure 23 shows that while the peak ROE point stays at the 48_24_24_18 configuration, the point of diminishing return moves backward from 128_64_64_48 for an ideal data cache to the 80_40_40_30 configuration. Notice that by the time the 96_48_48_36 configuration is reached, the ROE is well below unity for almost all benchmarks. This indicates that increasing the buffer sizes from 80_40_40_30 to 128_64_64_48 configuration benefits more from the increased ILP due to larger instruction window than from increased tolerance to L1 data cache misses. This observation validates our hypothesis that a better design strategy for energy efficient cores is to size the instruction buffers appropriately for code that hit the L1 data cache and use CFP to handle data cache misses.

7.2 Evaluating Performance

This section compares the IPC of Tuned-CFP to that of other latency tolerant cores. It also evaluates the IPC from the optimizations discussed in Section 3.2.2.

7.2.1 Comparing Tuned-CFP to Non-CFP Core Architectures

The above experiments point to three machine configurations suitable for different design targets:

1. A *peak return on energy* machine with 48_24_24_18 buffers and 2-way 8KB L1 data cache. This is the most energy efficient configuration so we call it EFF.
2. A unity gain ROE machine that represents the *point of diminishing return* with 80_40_40_30 buffers and 4-way 16KB L1 data cache (DIM).
3. A large machine that compromises on ROE for *high performance* (HP). This configuration uses 192_96_96_72 buffers and 4-way 16KB L1 data cache. This is what a designer might choose for best single-thread performance.

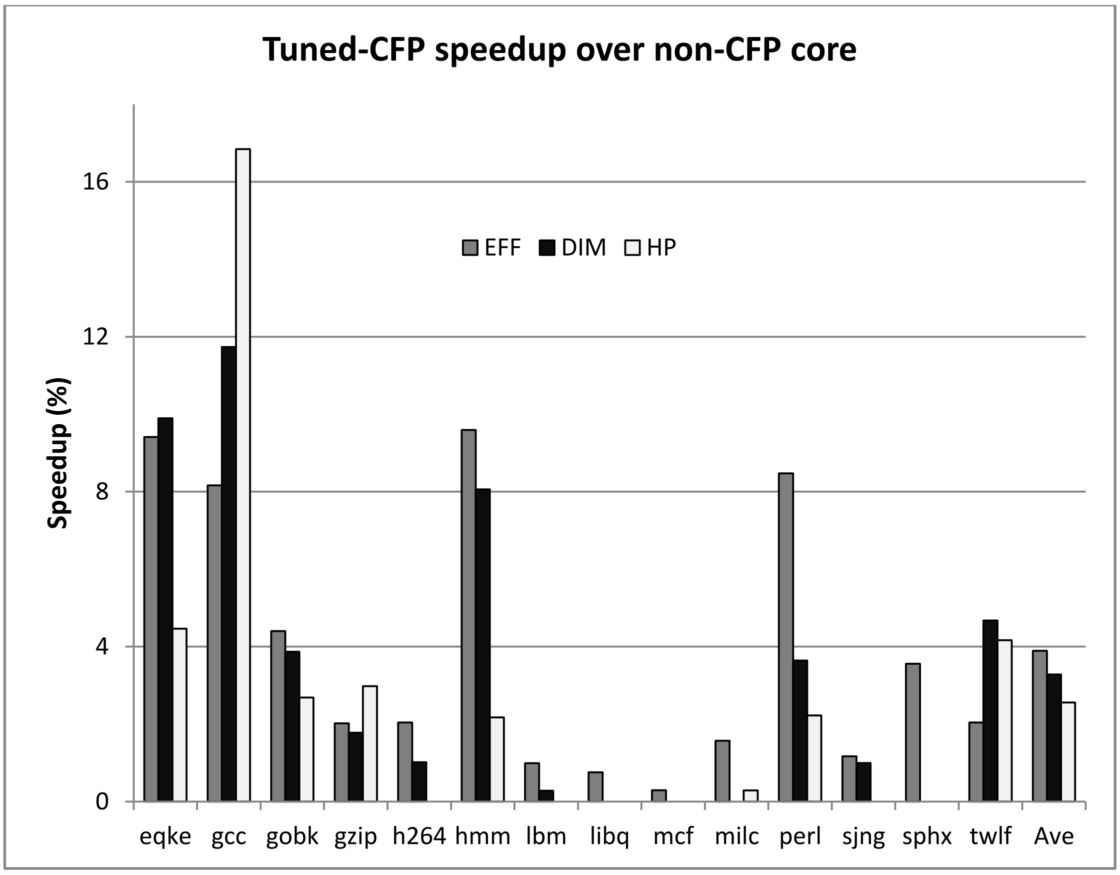


Figure 24. Percent speedup of Tuned-CFP over non-CFP core for EFF, DIM and HP models

Figure 24 shows the speedup of Tuned-CFP over a similar sized conventional non-CFP core for the EFF, DIM and HP machine configurations, when CFP is applied to L1 data cache misses. Table 4 shows various relevant execution statistics of Tuned-CFP. First observation to note is that Tuned-CFP benefits performance mostly on benchmarks that frequently miss the data cache, as to be expected. Second, Tuned-CFP, with its latency tolerance to first level cache misses, outperforms the non-CFP baseline by an average of 3-4% on all the configurations. Even though the average speedup of all benchmarks over the non-CFP baseline is modest, Tuned-CFP shows considerable performance improvement on benchmarks that frequently miss the data cache, e.g. gcc and equake. Earlier CFP work has similarly shown modest performance improvement

Table 4. Simulation statistics of Tuned-CFP

Benchmark	L1 cache load misses per 1000 instructions	L2 cache load misses per 1000 instructions	% Speedup over baseline	% Replayed instructions	Dependent branch mis-predictions per 1000 instructions
eqke	10.9	0.9	9.0	3.2	0.11
gcc	25.1	0.5	12.4	8.5	0.81
gobk	24.4	0.4	2.9	5.4	0.80
gzip	15.5	0.0	1.4	3.5	1.07
h264	11.1	0.0	1.5	2.5	0.22
hmm	7.7	0.1	7.1	2.4	0.01
lbm	2.7	0.0	0.0	0.0	0.00
libq	0.0	0.0	0.0	0.0	0.00
mcf	0.1	0.0	0.0	0.1	0.00
milc	0.0	0.00	0.3	0.0	0.00
perl	14.4	0.1	5.6	3.7	0.04
sjng	21.1	0.0	0.0	4.3	0.67
sphx	3.6	0.0	2.5	0.9	0.04
twlf	25.8	3.1	4.8	8.6	1.03

on applications like Spec 2006 that do not frequently miss the cache, but significant performance benefit on applications with high cache miss rates, such as server and workstations applications [46].

7.2.2 Tuned-CFP and Simultaneous-CFP Speedup over Conventional-CFP

In this section we will compare the performance of three latency tolerant cores – conventional-CFP, simultaneous-CFP and Tuned-CFP. All the latency tolerant cores are simulated with the machine configuration shown in Table 2 and all the cores apply CFP on first level load misses to the data cache.

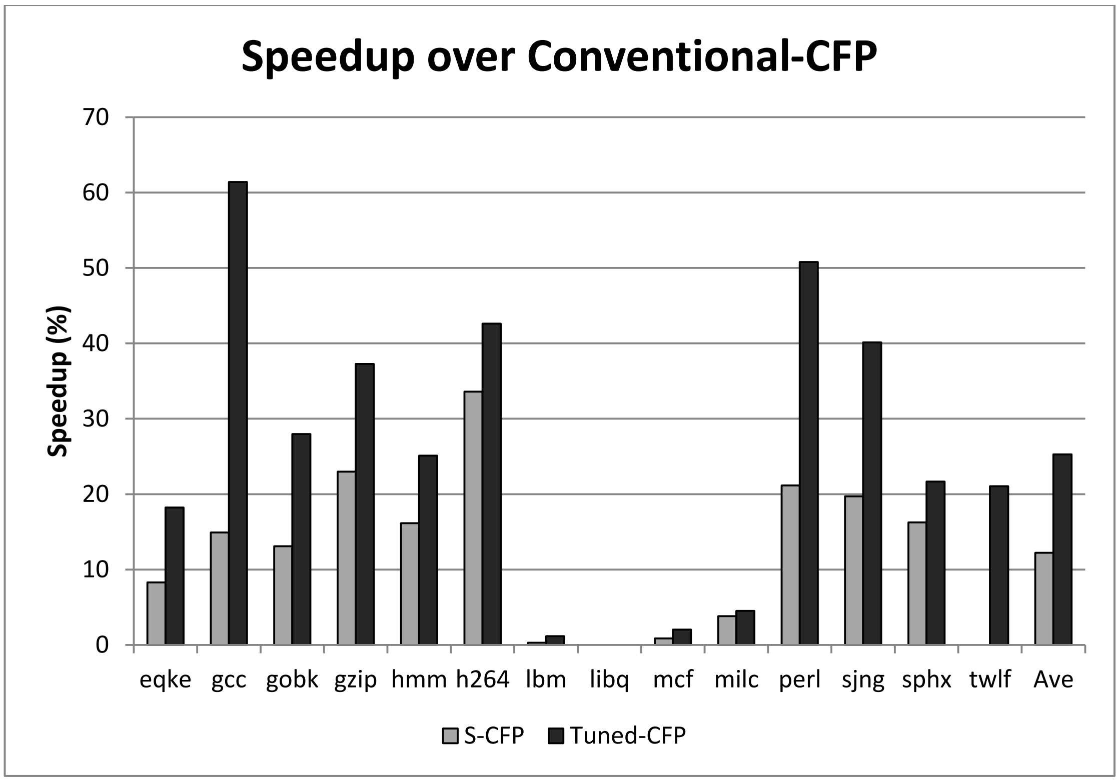


Figure 25. Speedup of Simultaneous-CFP and Tuned-CFP over Conventional-CFP

When first level misses are targeted for CFP, a large number of instructions go into the waiting buffer. The load misses in the waiting buffer will be woken up at different times based on which cache level has the required data. For example, suppose there are two loads in the waiting buffer. The first load hits L2 and wakes up within a short time while the second load has to go all the way to DRAM. If instructions from the waiting buffer are replayed based on wakeup time, conventional-CFP will have to flush the pipeline for each wakeup, which will hurt performance. The other option is to wait until all the load miss instructions in the waiting buffer get their wakeup and then flush the pipeline only once. This is sometimes too conservative because it increases the chances of a checkpoint rollback because the processor state remains speculative for an extended period of time. We have experimentally determined the best policy to reintroduce the miss-dependents in conventional-CFP is the following - wait until all the load misses receive their wakeup or the waiting buffer becomes half-full; after this if the waiting buffer continues to get populated, re-introduce the oldest entries from the waiting buffer head since there is a good chance that they would have woken up by now. Figure 25 shows the speedup of both the CFP architectures proposed in this dissertation, i.e. Simultaneous-CFP and Tuned-CFP, over Conventional-CFP. Both S-CFP and Tuned-CFP benefit in performance over conventional-CFP on most benchmarks because L1 misses frequently hit the on-chip cache and wakeup within a short time. While both S-CFP and Tuned-CFP execute the dependents and independents concurrently, the fact that the pipeline is drained frequently in conventional-CFP is the main reason for its degraded performance. This speedup of both S-CFP and Tuned-CFP over switch-on-event CFP justifies the need for this concurrent execution model.

7.2.3 Comparing Tuned-CFP to Simultaneous-CFP

This section compares the performance of Tuned-CFP to S-CFP and also isolates the contribution of each optimization scheme applied towards improving overall Tuned-CFP performance.

7.2.3.1 Tuned-CFP Speedup over Simultaneous-CFP

Figure 26 shows the speedup of Tuned-CFP over S-CFP when targeting data cache misses at all levels. Tuned-CFP, with its virtual register renaming, short replay loop and reduced replay/rollback outperforms S-CFP by an average of 10% when CFP algorithm is applied to L1 data cache misses. The maximum improvement occurs on gcc (39% speedup), which displays high rate of L1 cache misses as well as high branch

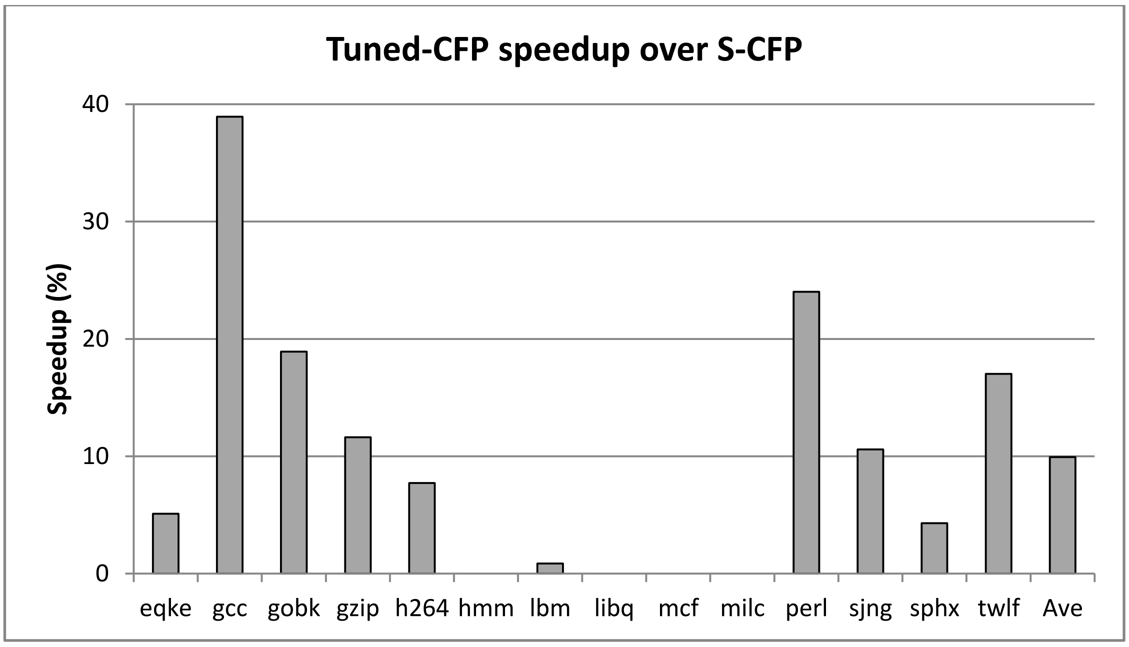


Figure 26. Tuned-CFP percent speedup over simultaneous-CFP

mis-prediction rate. The reduced number of replayed instructions in Tuned-CFP is very favourable to benchmarks like gcc and perl, since reducing the amount of replay also significantly reduces the number of costly miss-dependent branch mis-predictions. Also, the long replay loop of S-CFP is a glass jaw that is exposed on benchmarks like gcc. A carefully designed replay loop is necessary when designing CFP for general purpose processors that target many applications with widely different execution characteristics. The variation in the improvement between benchmarks is mainly due to the variation in the cache miss rates and branch mis-prediction rates of our simulation samples.

Notice that the results in this section are reported with a Tuned-CFP core having all the optimizations discussed in Section 3. With the intention to highlight the benefits from the most architecturally significant optimization (i.e. virtual register renaming), Tuned-CFP performance is compared to an S-CFP that has all the optimizations except VRR, i.e. buffer full optimization for load misses and stalling the front end based on the miss-dependent branch predictor. For this reason, Tuned-CFP speedup over S-CFP in Figure 26 is not as high as the speedup of (*ur* + *bf* + *dp)* model over un-optimized S-CFP shown in Figure 16.

7.2.3.2 Tuned-CFP Optimizations

There are mainly three optimizations featured in Tuned-CFP: 1) partial replay of the load miss dependence chain 2) moving miss-dependent instructions into the WB only when a resource or buffer becomes full and 3) dependent branch confidence predictor. We call these optimizations PR, BF and DP, respectively.

The contributions of PR and BF optimizations towards the performance benefit of Tuned-CFP performance is exactly the same as the benefit from the ideal models *ur* and *bf* shown in Figure 16. Figure 27 shows percent speedup contributed by the simple history based miss-dependent branch predictor (DP) discussed in Section 3.1.7. The speedup coming from the Oracle miss-dependent branch predictor (DP_Orc) is also shown for comparison. The (PR + BF + DP) model and the Oracle (PR + BF + DP_Orc) model contribute average speedups of 3.7% and 4.2% respectively, showing that our small 32-entry dependent branch confidence predictor achieves a speedup within 0.5% of the perfect upper bound performance of Oracle model. We observed that some benchmarks, like gcc and twolf, have a small number of static branches that contribute to a large percentage of dependent branch mis-predictions, making our small 32-entry hardware predictor very effective. Other benchmarks, like gobmk and gzip, display more complex control flow patterns with a larger set of static branches contributing to mis-predictions. For these benchmarks, our dependent branch predictor is less effective. Section 7.2.7 analyzes performance of the miss-dependent predictor.

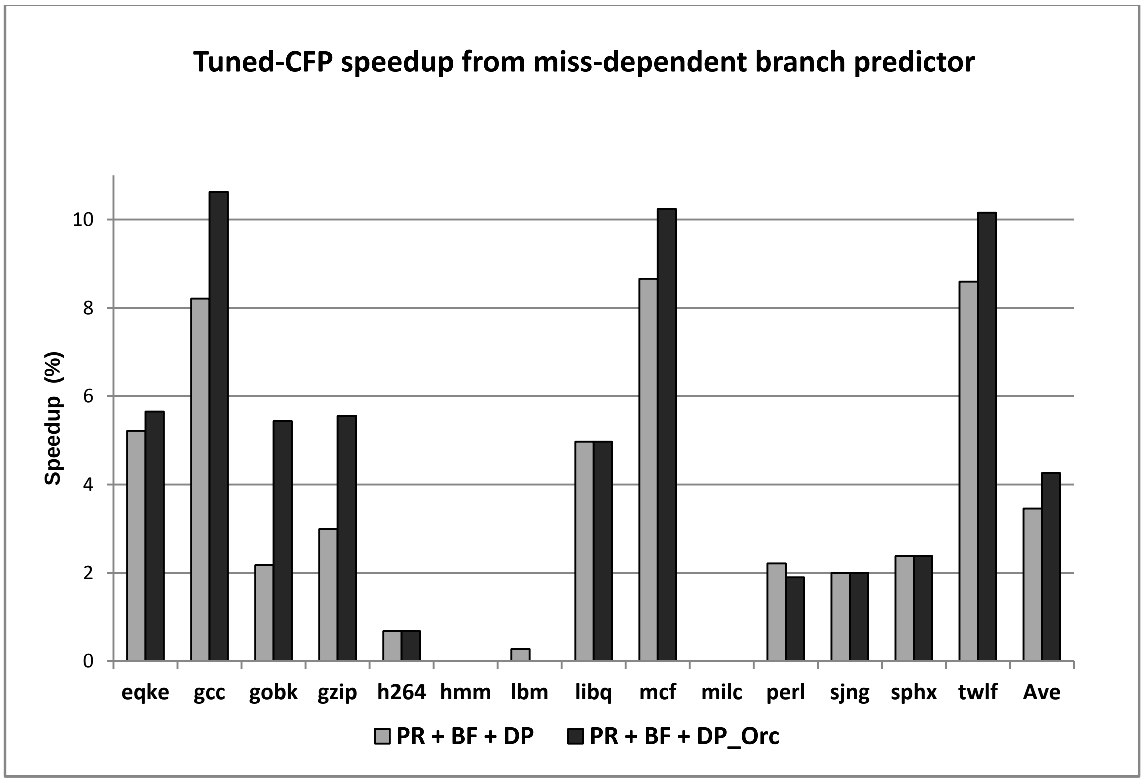


Figure 27. Percent speedup contribution from miss-dependent branch predictor

7.2.4 Performance of Cores with Ideal Branch Predictor

Branch mis-predictions negatively impact performance of any core because of wasteful execution of instructions belonging to the wrong path. Dependent branch mis-predictions have a bigger impact on performance, not only because of their late resolution but also because sometimes data from the wrong path which may not be of any use in the near future is brought into the cache. In this section we will compare the performance of non-CFP and Tuned-CFP cores with an ideal branch predictor to see what can be potentially achieved from latency tolerance alone, without any intervention from mis-predictions. Also branch prediction is a widely researched topic and state of the art academic and industrial branch predictors can be expected to be more accurate

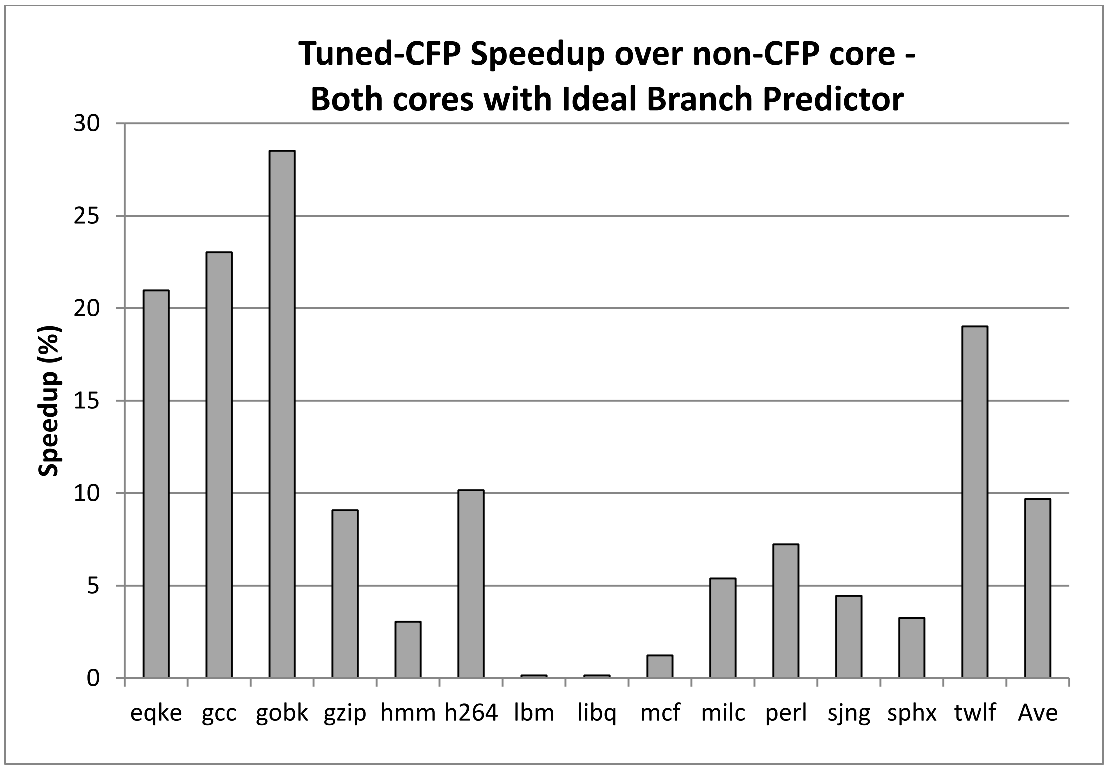


Figure 28. Speedup of Tuned-CFP over conventional non-CFP superscalar when both cores are simulated with a perfect branch predictor

than the one used in our simulation model. Table 5 shows the branch predictor statistics measured with our simulator. Figure 28 shows the speedup of Tuned-CFP processor over non-CFP core when both cores are simulated with a perfect branch predictor. Tuned-CFP benefits very well on benchmarks like gcc and twolf which miss the cache frequently and also encounter dependent branches that mis-predict more often as shown in Table 5, column 7 (poisoned mis-predicted branches per 1000 instructions). Benchmarks like equake and gobmk, even though they have fewer dependent mis-predicted branches compared to gcc and twolf, have more miss-dependent branches that go all the way to DRAM, and hence benefit very well when such branches with heavy impact are avoided.

Table 5. Branch instruction statistics of Tuned-CFP

Bmk	% Branches over total instruc-tions	%Mis-predicted branches over total branches	% Poisoned mis-predicted branches over total poisoned branches	% Poisoned branches over total branches	% Poisoned mis-predicted branches over total mis-predicted branches	Poisoned mis-predicted branches per 1000 insns
eqke	7.6	11.9	19.1	0.8	1.3	0.1
gcc	7.2	5.4	9.6	11.4	20.4	0.8
gobk	9.0	9.0	17.6	3.4	6.6	0.5
gzip	6.1	7.4	28.3	6.3	23.9	1.1
h264	5.3	6.4	19.5	2.0	6.3	0.2
hmm	2.2	0.6	8.1	0.3	4.1	0.0
lbm	0.6	0.4	0.0	0.1	0.0	0.0
libq	5.3	0.0	7.1	0.0	28.6	0.0
mcf	17.0	0.7	3.2	0.0	0.1	0.0
milc	3.7	3.4	6.7	0.0	0.0	0.0
perl	14.3	1.0	0.9	3.0	2.6	0.0
sjng	8.5	10.3	19.4	4.0	7.6	0.7
sphx	8.7	2.4	6.1	1.5	3.8	0.1
twlf	8.0	11.1	15.3	8.6	11.9	1.0
Ave	7.4	5.0	11.5	3.0	8.4	0.3

7.2.5 Performance of Cores with off-chip Memory Controller

In Section 2.2.3.5, it was mentioned that targeting CFP on only last level cache misses gives less performance benefit in current generation processors because of the integrated memory controller. Figure 29 shows the speedup of Tuned-CFP core over non-CFP core with an off-chip memory controller. From the figure, it can be seen that Tuned-CFP achieves an average speedup of 8% over non-CFP core, which is double the speedup that is achieved with an integrated memory controller as reported in Figure 24. This 8% speedup is also better than the 4% speedup of conventional-CFP on CPU benchmarks reported in [46] with similar DRAM latency.

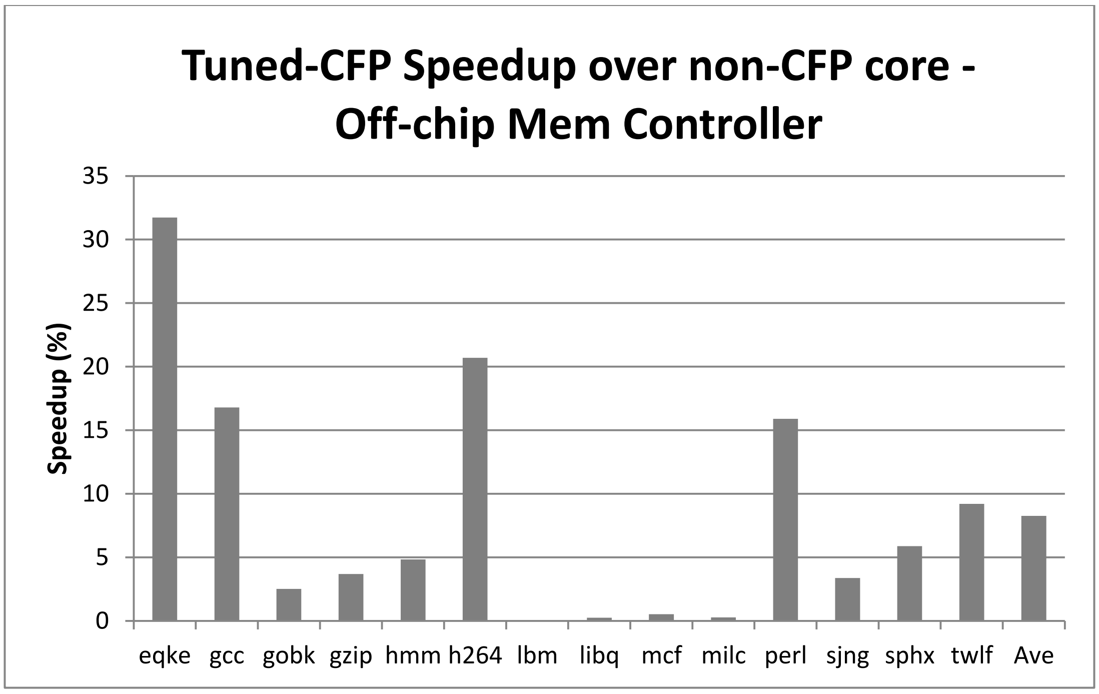


Figure 29. Speedup of Tuned-CFP over conventional non-CFP superscalar when both cores are simulated with off-chip memory controller

7.2.6 Performance of Speculative Cache and SRL

In order to support a large number of loads and stores in the instruction window, we use the modified SRL algorithm as explained in Chapter 5. This section evaluates the performance of SRL by comparing it with a large store queue. The large store queue is an impractical design because of the issues mentioned in Section 2.1.4 and is only used to model the behavior of an ideal store queue to establish the upper bound for performance and also evaluate how SRL matches up with it. In all the experiments the SRL size and large store queue size is limited to 256 entries as shown in Table 2. This size is enough to support all the stores most of the time because the instruction window size is also limited to 1024, as explained in Section 4.1.1. If the SRL or store queue fills up, the front end is stalled.

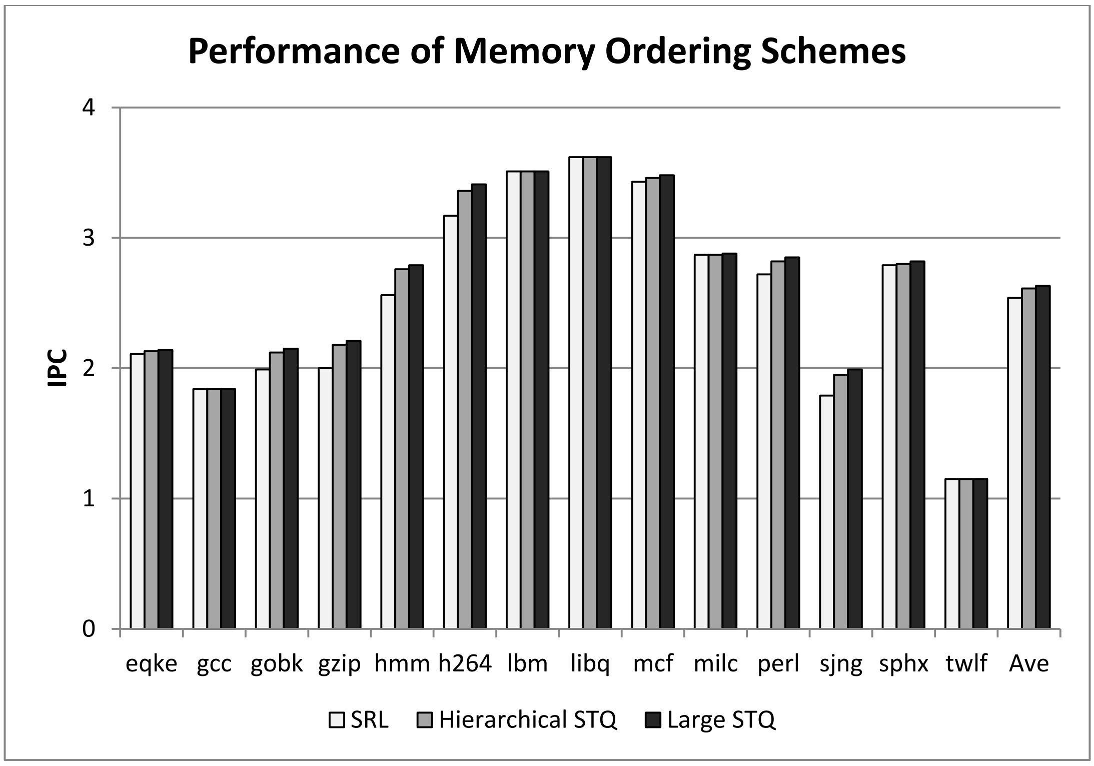


Figure 30. IPC comparison of SRL, hierarchical STQ and large STQ memory ordering mechanisms

Figure 30 shows the IPC's of Tuned-CFP core that uses SRL, hierarchical store queue [2][3] and a large store queue. On the CPU benchmarks, hierarchical store queue performs very close to the large store queue mainly because most of the store-to-load forwarding happens between closely lying load-store pairs. SRL under-performs the large store queue on some benchmarks like gobmk, gzip and sjeng mainly because of two reasons. First the speculative blocks evict non-speculative blocks from the cache, leading to more load misses. These load misses result in more replay, rollback and execution of instructions from the wrong path. Table 6 shows the increase in replay, rollback and wrong path instructions in case of SRL when compared to large STQ. The second reason is the synchronization of each dependent load with the store immediately ahead of it. A dependent load cannot clearly disambiguate the correct store from which

Table 6. Replay, rollback, wrong path and STQ forwarding statistics for SRL mechanism

Bmk	Increase in % Replay with SRL over total instructions compared to Large STQ design	Increase in % Rollback with SRL over total instructions compared to Large STQ design	Increase in % Wrong path instructions with SRL over total instructions compared to Large STQ design	% of Loads that forward from STQ over total loads – Large Store Queue Design	% of Loads that forward from STQ over total loads – Large Store Queue Design
eqke	0.72	0.30	0.75	6.98	6.34
gcc	1.34	0.00	-0.27	18.39	12.53
gobk	1.72	0.47	3.39	11.19	8.94
gzip	3.11	0.42	-0.05	13.39	9.77
h264	1.55	0.18	1.19	11.41	9.48
hmm	0.58	0.01	0.01	20.71	12.28
perl	0.48	0.12	0.78	10.13	6.83
sjng	3.33	1.28	3.25	7.85	5.39
sphx	1.18	0.10	1.33	21.50	19.91
twlf	1.14	0.48	2.12	6.91	2.91
Ave	1.52	0.33	1.25	12.85	9.44

it has to forward the data until all the stores ahead of it have written to the cache. This is because the store queue contains only a subset of all the stores in the speculative window and it is not possible to safely determine the exact forwarding store from this limited set. On the other hand, a dependent load in a large store queue can forward from the store queue immediately when the store is ready because all the stores in the window are buffered, albeit impractically, making it possible to select the exact store to forward the required data. In some benchmarks like gcc and gzip there is a reduction in wrong path instructions in case of SRL mainly because of pipeline stalls due to this synchronization. With these limitations, the SRL implementation performs within an average 3% of the large store queue.

Tables 7 and 8 show the statistics for speculative load and store accesses to the cache. Notice in some cases, speculative accesses hit non-speculative blocks (col 5, 7).

Table 7. Speculative load statistics for SRL mechanism

Bmk	% of Spec-Indep. load accesses that hit L1 cache	% of Spec-Dep. load accesses that hit L1 cache	Spec-Indep. load hits that hit Spec-Indep. blocks (%)	Spec-Indep. load hits that hit non-Spec or Spec-Dep. blocks (%)	Spec-Dep. Load hits that hit Spec-Dep. blocks (%)	Spec-Dep. Load hits that hit non-Spec blocks (%)
eqke	78.90	81.67	88.69	11.31	98.14	1.86
gcc	84.50	66.39	74.26	25.74	93.33	6.67
gobk	86.69	74.19	85.36	14.64	98.02	1.98
gzip	89.62	64.98	80.41	19.59	98.48	1.52
hmm	90.59	78.56	85.23	14.77	97.79	2.21
h264	87.30	73.49	74.05	25.95	93.76	6.24
perl	83.16	76.50	80.73	19.27	89.18	10.82
sjng	86.67	73.69	84.20	15.80	97.26	2.74
sphx	78.51	84.65	73.31	26.69	93.97	6.03
twlf	86.28	71.72	86.53	13.47	90.71	9.29
Ave	85.96	72.31	81.62	18.38	94.67	5.33

Table 1 shows the conditions to declare a speculative access as a hit. When a speculative access does not hit the cache, it is searched for in the victim cache as explained in Section 5.1.7. From our experiments a victim cache of size 8 is enough to record all speculative blocks without the need to evict any of them. If at all a speculative block needs to be evicted, the processor state is rolled back to the checkpoint.

7.2.7 Performance of Miss-dependent Branch Predictor

This section presents an example of the correlation that exists between a dependent mis-predicted branch and its PC address, a property that is exploited by the

Table 8. Speculative store statistics for SRL mechanism

Bmk	% of Spec-Indep. store accesses that hit L1 cache	% of Spec-Dep. store accesses that hit L1 cache	Spec-Indep. store hits that hit Spec-Indep. blocks (%)	Spec-Indep. store hits that hit non-Spec blocks (%)	Spec-Dep. store hits that hit Spec-Dep. blocks (%)	Spec-Dep. store hits that hit non-Spec blocks (%)
eqke	99.24	99.91	73.99	26.01	95.33	4.67
gcc	99.26	99.94	84.19	15.81	95.51	4.49
gobk	99.05	99.87	83.18	16.82	96.96	3.04
gzip	99.69	99.88	77.57	22.43	94.86	5.14
hmm	99.41	99.96	80.90	19.10	97.16	2.84
h264	96.27	91.50	83.85	16.15	93.88	6.12
perl	99.27	99.83	82.38	17.62	95.60	4.40
sjng	99.33	99.73	75.47	24.53	94.91	5.09
sphx	99.75	100.00	78.36	21.64	98.46	1.54
twlf	99.41	99.72	81.82	18.18	88.83	11.17
Ave	99.12	99.20	80.69	19.31	95.18	4.82

history based predictor discussed in Section 4.1.8. Figure 31 shows four curves for benchmark gcc where a particular branch (at a PC address) is plotted on the X-axis and different statistics of each branch is plotted on the Y-axis.

The dotted line at the top of the figure shows the number of times each branch gets poisoned and enters the waiting buffer (shown as *Into WB*). Figure 31 shows the branch predictor statistics for branches that enter the waiting buffer between 1000 and 10000 times and Figure 32 shows the same for branches that enter the waiting buffer more than 10000 times. While both the figures show the same information, they have been split into two for clarity. The data joined by the triangle markers show the number of times that particular branch is found to be mis-predicted after the load miss data

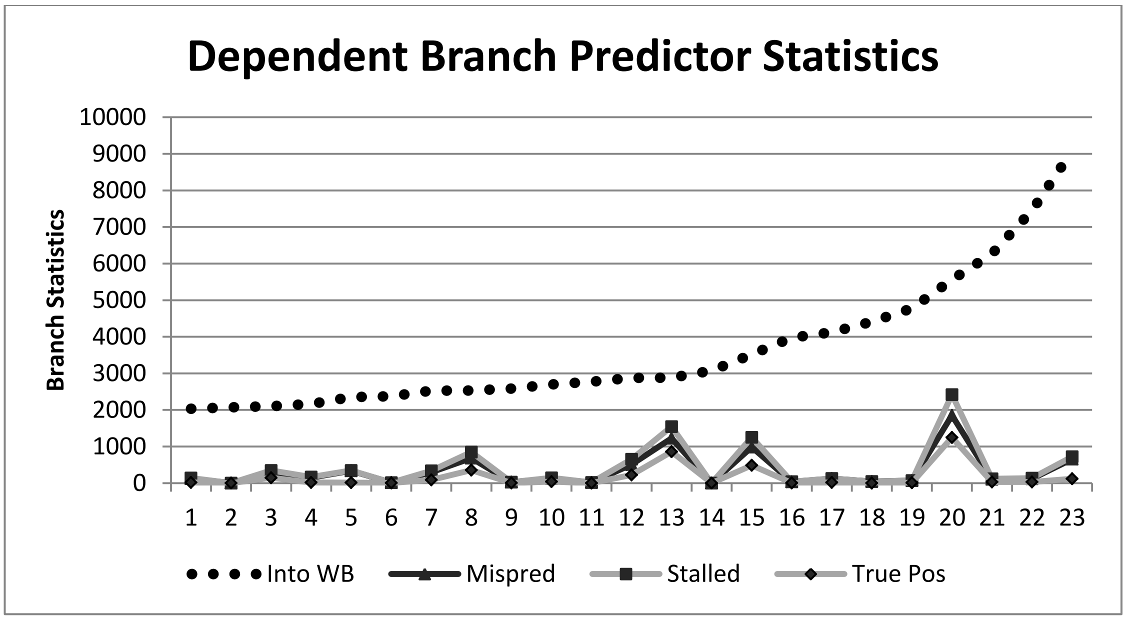


Figure 31. Correlation between dependent mis-predicted branch and its PC address for branches replayed between 1000 and 10000 times

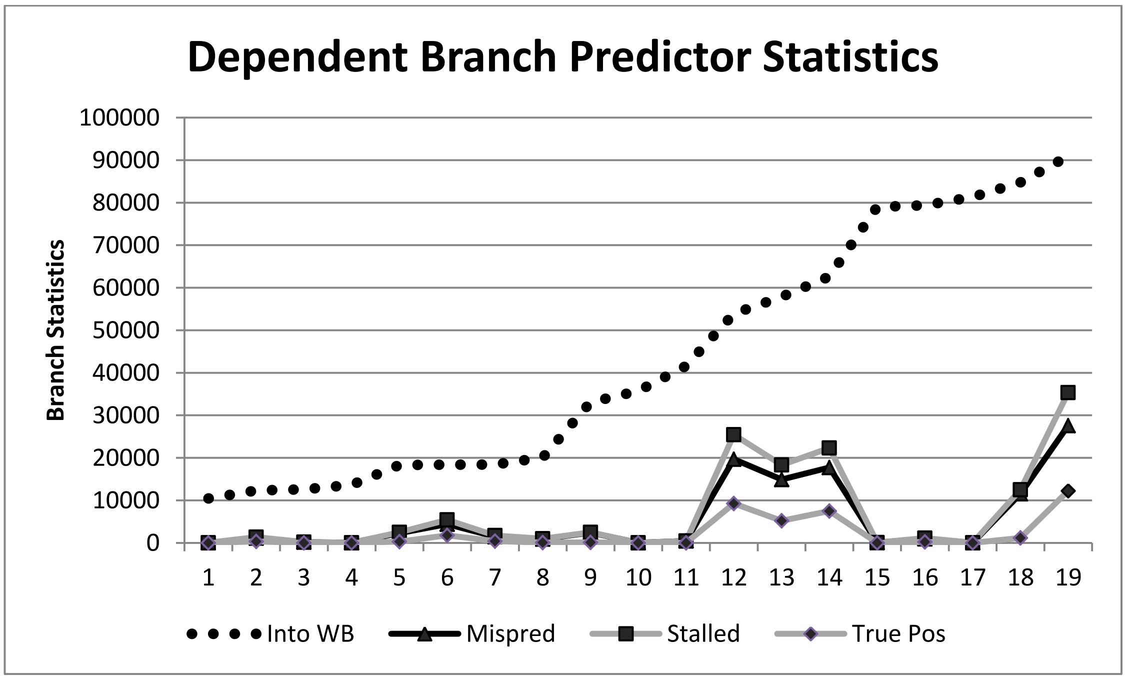


Figure 32. Correlation between dependent mis-predicted branch and its PC address for branches replayed between 10000 and 100000 times

is delivered to the processor (shown as *Mispred*). The data series joined by the square markers shows the number of times that particular branch entering the waiting buffer is predicted to be a problem branch, thereby stalling the front end (shown as *Stalled*). The series joined by the diamond marker shows the number of times the dependent predictor is successful in stalling the front end on a dependent mis-predicted branch and saving execution waste (shown in *True Pos*).

In the figures, since the three curves show similar behavior, it can be said that the miss-dependent branch predictor is able to train itself sufficiently well based on the branch history. The gap between the diamond curve and the triangle curve is the number of times the dependent branch predictor is not able to detect a mis-predicted branch, thus ending up in a checkpoint rollback. This gap is narrow for most branch PC addresses. In an ideal predictor the diamond curve should coincide with the triangle curve. The gap between the square curve and the upper dotted curve is the number of times a branch with high confidence is allowed to go through without stalling the front end. This is also considerably high for most branches; for instance, branch #17 in Figure 32 enters the waiting buffer nearly 80000 times but is very rarely found to be mis-predicted. Finally the gap between the diamond curve and square curve shows the number of times the predictor anticipates a mis-prediction but the execution turns out to be on the right path. There are a considerable number of false positives and the front end is conservatively stalled as a result, but we intentionally bias the predictor to be conservative in order to avoid excessive execution waste.

Figure 33 shows the true positive, true negative, false positive and false negative statistics of the dependent branch predictor for branches that enter the waiting buffer more than 10000 times. From the figure, for most branch instances, the dark

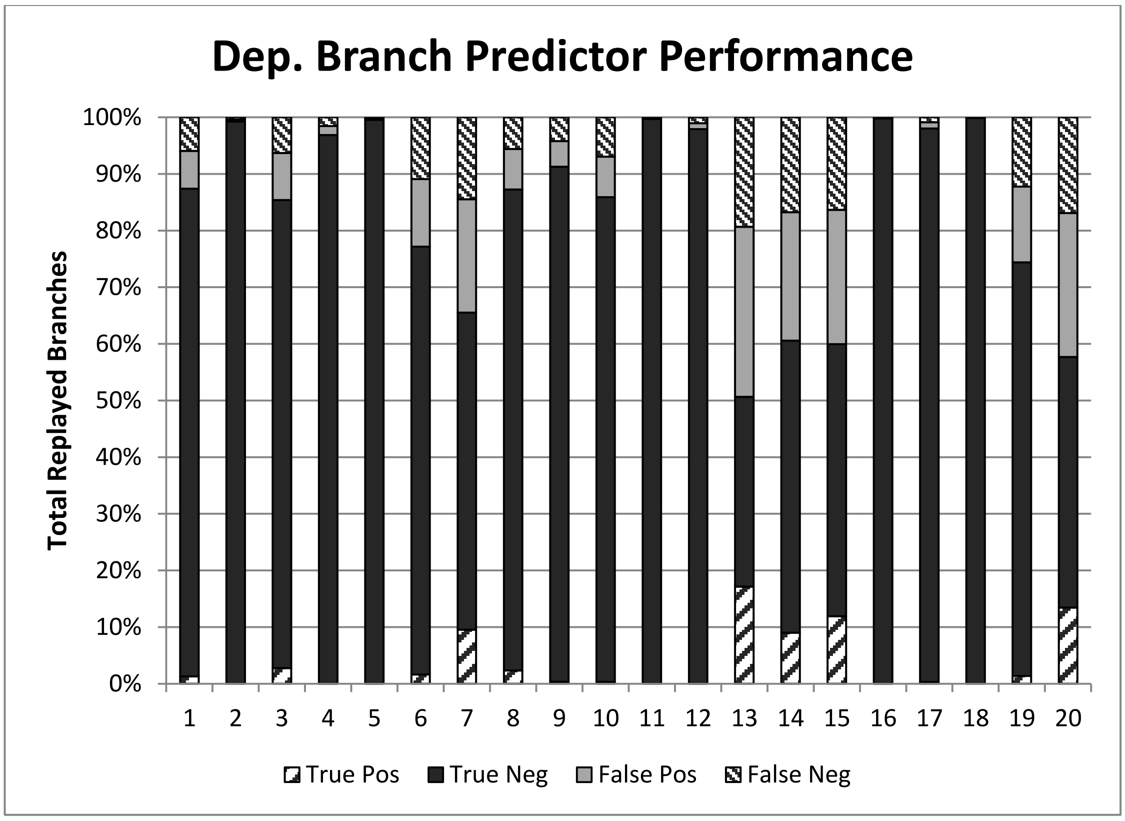


Figure 33. Dependent branch predictor statistics

shaded pattern indicates that the true negatives form the majority, at least for branches that do not mis-predict frequently. For branches that mis-predict, the true positives, false positives and false negatives occur in almost equal measure. A better predictor should attempt to increase the true positives, minimize false positives and completely avoid false negatives.

Future work can focus on building more accurate predictors that look at the path history of other branches. Another option is to stall the front-end based on the size of the speculative state. If the speculative state is reaching a large size and a not-so-confident branch is encountered, it is better to stall the front-end in order to avoid paying a large rollback penalty. On the other hand, if the speculative state is not very large, the risk is not very high, so execution can be allowed to proceed.

7.3 Evaluating Energy per Instruction

This section compares the energy consumed per instruction across the three cores.

7.3.1 Energy per Instruction Comparison of Tuned-CFP and non-CFP cores

Figure 34 shows the total core energy per instruction (EPI) increase of Tuned-CFP over the non-CFP baseline for all three machine configurations of interest. We observe that benchmarks that miss the cache incur an additional increase in EPI that depends on the amount of instructions replayed from the waiting buffer. Compare the increase in EPI to the speedup observed for the same benchmarks. Some benchmarks like gcc, equake and hmmer miss the cache but benefit from CFP speculative execution while the cache miss data is being fetched. This point is validated by the large speedup observed in these benchmarks at the cost of a small increase in power.

Some benchmarks like twolf and sjeng also miss the cache but encounter frequent dependent branch mis-predictions that offset the performance gain while burning power in the process. Finally, benchmark traces like libquantum and milc that do not show any benefit from CFP, also do not consume additional power. This is because the CFP structures, i.e. the waiting buffer and SRL, observe major switching activity only in CFP execution mode.

7.3.2 Energy per Instruction Comparison of S-CFP and Tuned-CFP

Figure 35 shows the percentage increase in energy per instruction (EPI) of S-CFP and Tuned-CFP over the non-CFP baseline for each of our simulated benchmarks.

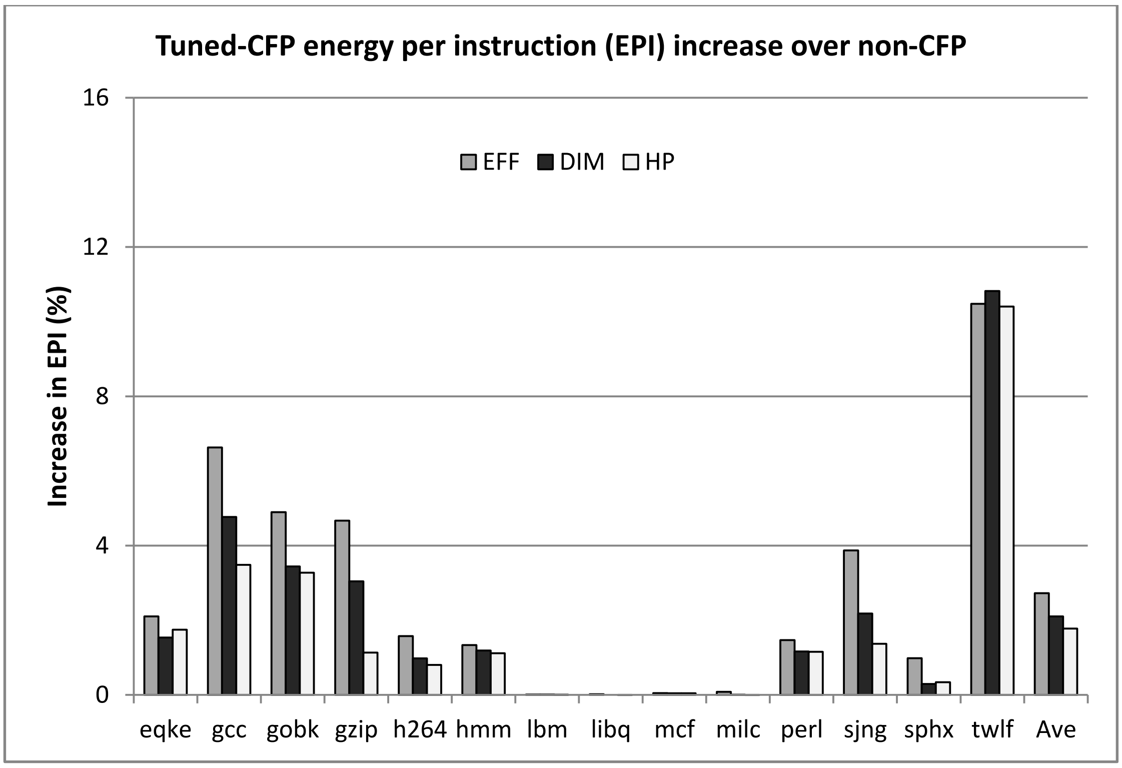


Figure 34. Tuned-CFP percent increase in EPI compared to non-CFP EFF, DIM and HP models

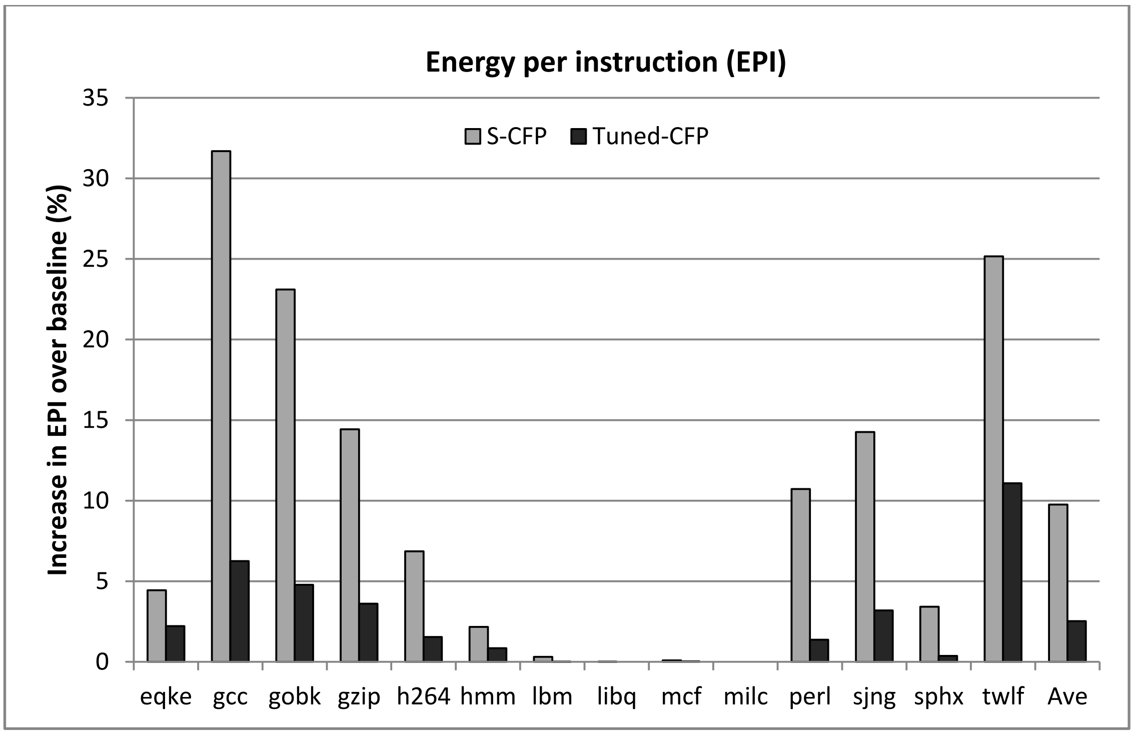


Figure 35. Energy per instruction % increase of S-CFP and Tuned-CFP over ROB baseline

Tuned-CFP is able to achieve this reduction in overall energy because of eliminating excessive replayed instructions, rollback instructions and wrong path instructions, as we will show in the next section.

7.3.2.1 Reduction in Replayed Instructions

Figure 36 shows the percentage of replayed instructions over total instructions in both Simultaneous-CFP and Tuned-CFP. With the optimizations explained in Section 3.2.2 Tuned-CFP is able to reduce the amount of replay by a significant amount. For example, the percentage reduction in replay is ~22% in gcc and ~17% in perl. Also importantly, Tuned-CFP ends up replaying on and average only 3% of overall instructions, which is much lower compared to average 10% replayed instructions in the case of S-CFP. This reduction in replay is significant as it reduces the probability of replaying a potential mis-predicted branch and having to rollback to a checkpoint.

7.3.2.2 Reduction in Rollback Penalty

Figure 37 shows the percentage of rollback penalty over total executed instructions in S-CFP and Tuned-CFP. Eliminating the probability of a rollback to the checkpoint is very important both from performance and energy efficiency perspectives. On a rollback, all instructions between the mis-predicted branch and the checkpoint, referred to as *rollback instructions*, have to be brought into the pipeline again. Rollback instructions not only increase the re-execution overhead, but also consume fetch and decode resources, which are major contributors to the overall core energy consumption. From the figure, it can be seen that Tuned-CFP pays a rollback penalty of 1%, which is considerable reduction from S-CFP (5%). This contributes very well to the overall

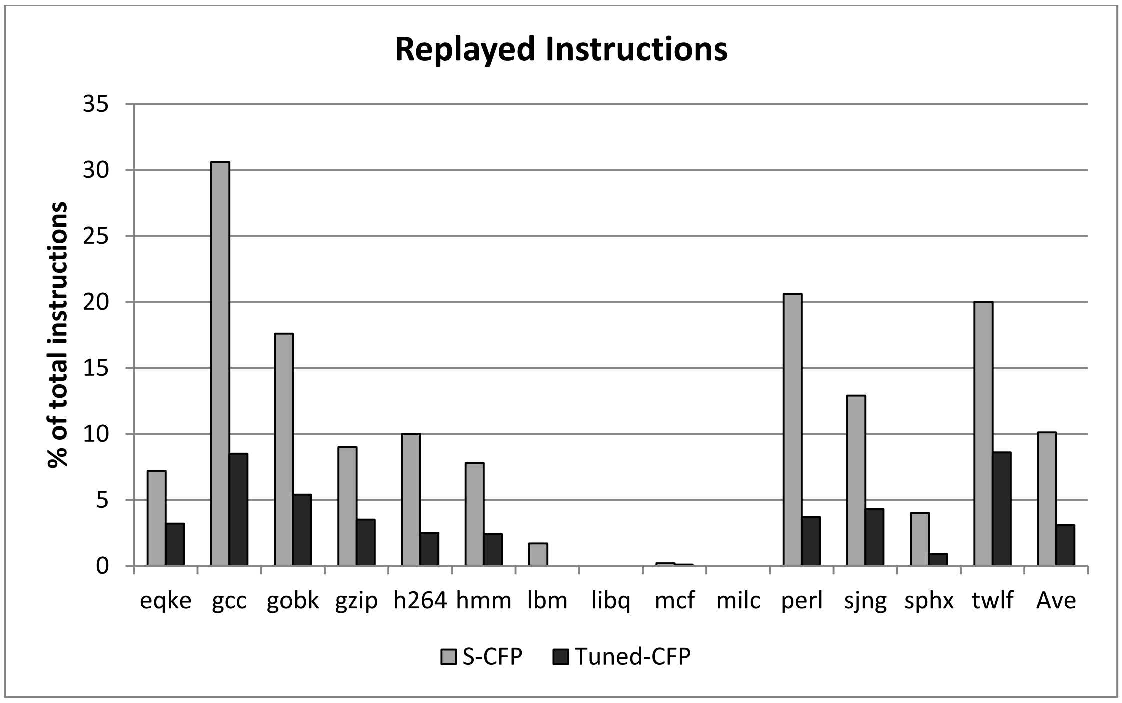


Figure 36. Percentage of replayed instructions over total instructions in S-CFP and Tuned-CFP.

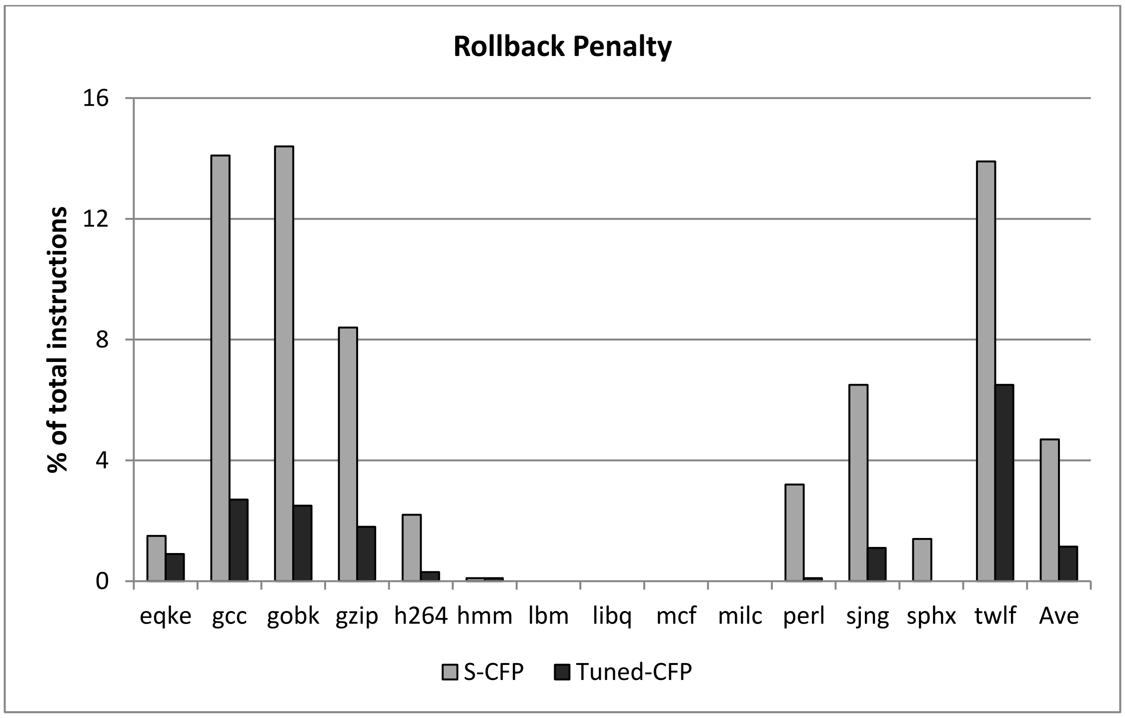


Figure 37. Reduction in rollback penalty in Tuned-CFP compared to S-CFP

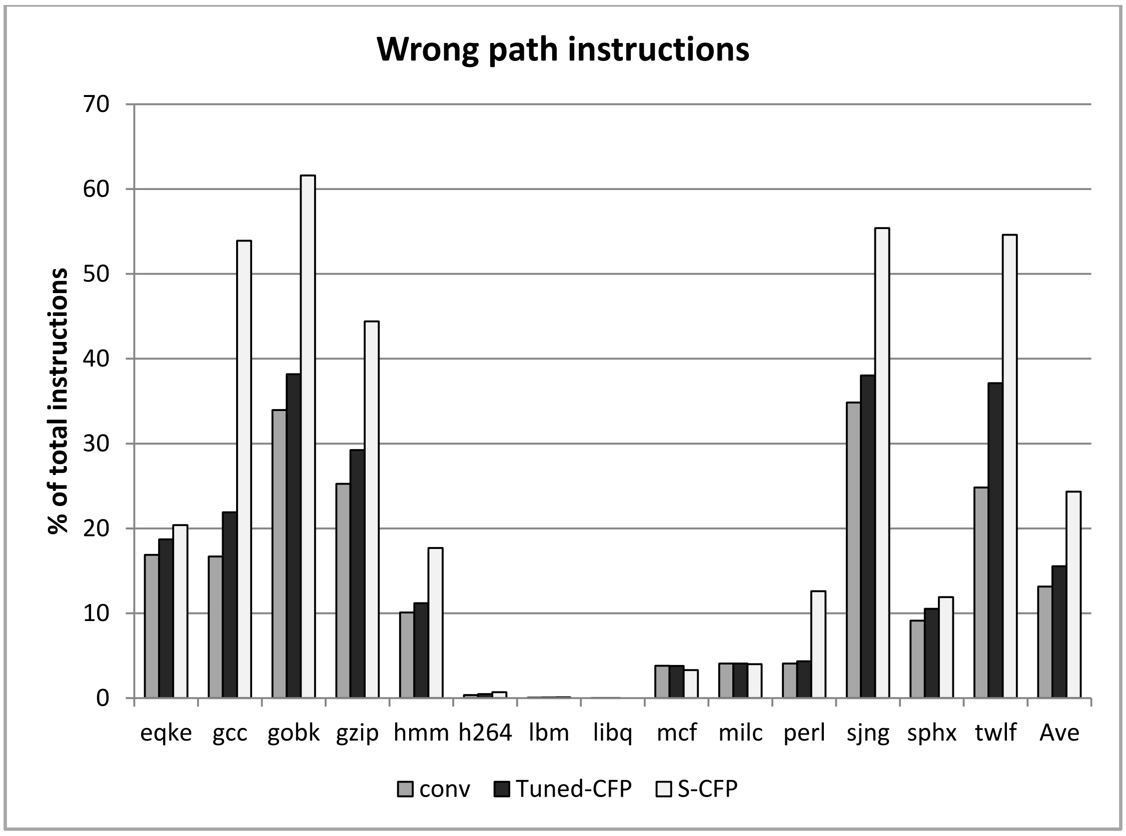


Figure 38. Percentage of wrong path instructions over total instructions in conventional non-CFP, Tuned-CFP and S-CFP cores

Tuned-CFP improvement in performance and energy over S-CFP.

7.3.2.3 Wrong Path Instructions

Figure 38 shows the percentage of instructions from the wrong path over total instructions in conventional non-CFP, Tuned-CFP and S-CFP cores. They represent the instructions following a mis-predicted branch. Note that while a conventional processor stalls the pipeline early in the event of a cache miss, a CFP processor continues processing many instructions in the shadow of the miss. While wrong path instructions do not have any impact on performance, they have a very high impact on energy. All wrong path instructions cost dynamic power because they go through every stage of the

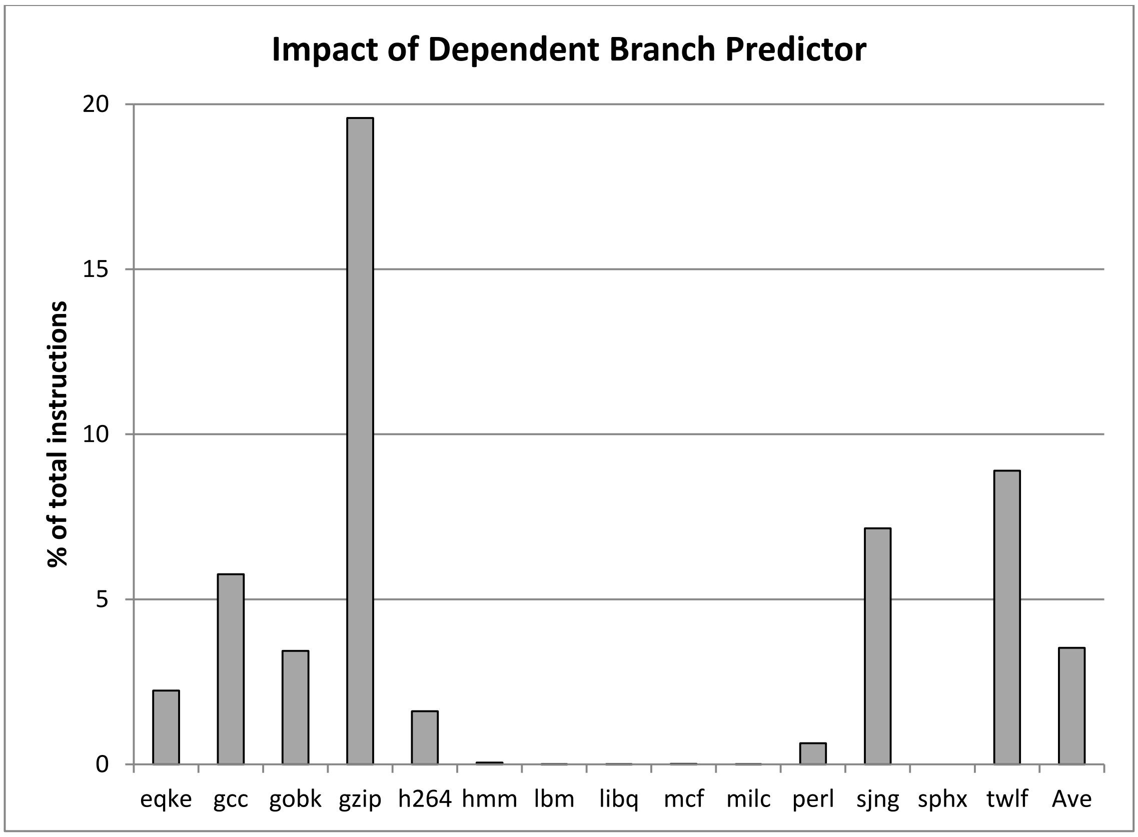


Figure 39. Percentage reduction in wrong path instructions in Tuned-CFP core when dependent branch predictor is used

pipeline wastefully. The figure shows wrong path instructions following both independent and dependent branch mis-predictions. While independent branch mis-predictions do not cost much because the processor state is recovered with the less costly re-order buffer mechanism, dependent branch mis-predictions are a lot costlier, especially in CFP processors, because of their late resolution and run-ahead execution.

From the figure, it can be seen that there is a 10% reduction in wrong path instructions compared to S-CFP. This can be mainly attributed to fact that Tuned-CFP is able to resolve many branches before they move into the waiting buffer. The percentage of wrong path instructions compared to a conventional non-CFP processor of both S-CFP and Tuned-CFP cores is also shown in Figure 38. It can be seen that

Tuned-CFP increases the wrong path instructions only marginally. The role of the dependent branch predictor is to avoid such wrong path instructions impacting the energy consumption of the core. The percentage reduction in wrong path instructions as a result of using the dependent branch predictor is shown in Figure 39.

Table 9 shows average EPI across all simulated benchmarks of various functional blocks as well as the total non-CFP baseline, S-CFP and Tuned-CFP cores. All the numbers in the table are shown relative to the total EPI of the non-CFP baseline core. Tuned-CFP shows ~8% less energy per instruction compared to S-CFP due to reduction in replayed instructions execution, rollback and wrong path instructions. Finally, notice that Tuned-CFP and the non-CFP baseline consume about the same energy per instruction with Tuned-CFP measuring on average just about 2% additional energy per instruction over the non-CFP ROB baseline core.

We summarize this section by saying that it is evident from Figures 24 and 34 that on many benchmarks the Tuned-CFP speedup is greater than the EPI increase indicating that Tuned-CFP gives good performance return on energy invested.

We next conclude compare the ROE or energy efficiency of Tuned-CFP and Simultaneous-CFP to non-CFP superscalar cores for different buffer configurations.

Table 9. S-CFP and Tuned-CFP average increase in energy % relative to non-CFP ROB superscalar

	L1 ICache	Decode	RAT	RS	EU's	ROB/ RRF	LSQ	L1 DCache	SRL/ WB	Total
Base	8.6	18.9	15.5	10.8	13.8	12.9	10.8	8.6	0.00	100.0
S-CFP	9.2	20.2	16.5	12.3	14.7	15.4	11.5	9.2	1.9	110.9
Tuned -CFP	8.7	19.2	15.7	11.2	14.0	13.2	10.9	8.7	1.1	102.8

7.4 Return on Energy Comparison of Tuned-CFP, S-CFP and Non-CFP Cores

Previous works have argued for CFP as energy efficient, scalable, large instruction window architecture. However, there has been no work that quantifies the energy efficiency of CFP and compares it with that of conventional non-CFP superscalar architecture.

In this section, we compare the ROE of Tuned-CFP to that of S-CFP and non-CFP core by sweeping across a range of buffer configurations starting from the best efficiency EFF configuration to the best performance HP configuration. The baseline relative to which the ROE is computed is the 32_16_16_12 non-CFP configuration.

Figure 40 shows the average ROE across all the simulated benchmarks for Tuned-CFP, S-CFP and non-CFP cores. Tuned-CFP shows better ROE than non-CFP and S-CFP machines on all configurations. This clearly demonstrates the importance of virtual register renaming and other optimizations featured in the Tuned-CFP architecture.

Notice that the gap between Tuned-CFP ROE and non-CFP ROE is widest for maximum energy efficiency configuration (EFF) and tends to reduce with increasing buffer sizes as the non-CFP core suffers fewer stalls. With small buffer configurations like 48_24_24_18 and 64_32_32_24, the non-CFP core does not have enough resources to hide even short latency misses that hit the on-chip L2 cache. Hence the majority of cache misses end up stalling the non-CFP core. In this case, Tuned-CFP benefits more over the non-CFP core on many benchmarks, particularly when the speculative execution does not have to be discarded. When the buffer sizes are increased moderately (e.g. 80_40_40_30 configuration), Tuned-CFP still maintains a healthy ROE gap between itself and the non-CFP core. On the other hand, when we move over

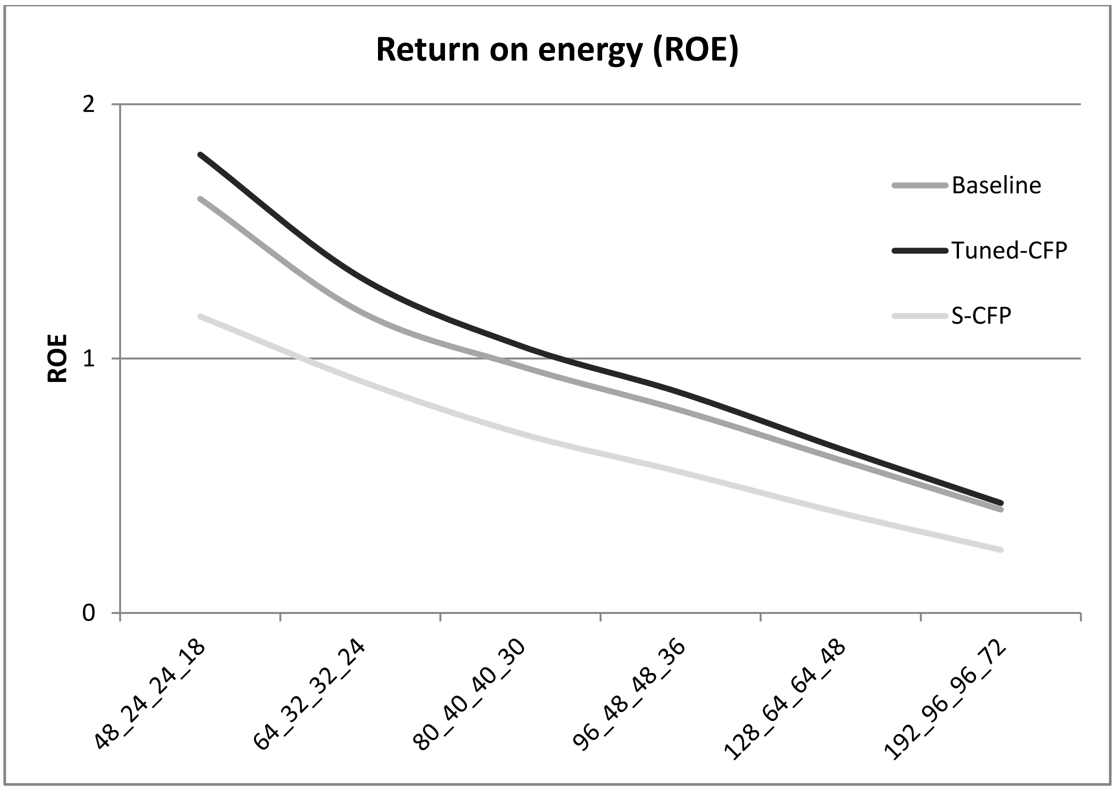


Figure 40. Return on energy of non-CFP baseline, Tuned-CFP and S-CFP cores

to a large machine like 192_96_96_72, the gap between non-CFP and Tuned-CFP reduces because there are enough instruction buffers to handle all short latency misses that hit the on-chip cache. The slight improvement in Tuned-CFP ROE comes because of the inevitable misses that go all the way to DRAM. In this case, even a large machine with 192_96_96_72 configuration is not able to keep the pipeline units busy for the entire time while the miss remains outstanding. It is important to notice the very poor ROE of a large machine for both non-CFP and Tuned-CFP which is at a disappointing 0.4 value. This shows that increasing the instruction buffer sizes to this extent hurts EPI considerably, proving the widely established opinion that traditional methods of increasing buffer sizes to get performance are already on a downward ROE slide.

Figure 40 also shows that the ROE of S-CFP is worse compared to both Tuned-CFP and non-CFP cores. The excessive replay and rollbacks in S-CFP not only bring down its overall performance but also dissipates excessive energy per instruction in the process.

Finally, Table 10 shows the percentage improvement in ROE of Tuned-CFP over the non-CFP core. Tuned-CFP shows up to 11% improvement in ROE for small sized cores. The ROE gap starts decreasing for medium sized cores and reaches a minimum for the largest buffer configuration. It is interesting to note that even on the large machine configuration Tuned-CFP still manages to improve the ROE by 4.8 percentage points.

Table 10. Tuned-CFP percent improvement in ROE over non-CFP. ROE computed relative to 32_16_16_12 non-CFP configuration

Configuration	Non-CFP Core ROE	Tuned-CFP Core ROE	Percent improvement in ROE
48_24_24_18	1.63	1.80	10.4%
64_32_32_24	1.18	1.32	11.8%
80_40_40_30	0.97	1.05	8.2%
96_48_48_36	0.80	0.87	8.7%
128_64_64_48	0.60	0.64	6.6%
192_96_96_72	0.41	0.43	4.8%

CHAPTER 8

QUALITATIVE ANALYSIS

This chapter makes qualitative comparisons between the ideas discussed in this work and previous work on latency tolerance. First a qualitative comparison between SRL and hierarchical store queue mechanisms for memory ordering is presented based on results from previous SRL work [14]. Next the performance of Tuned-CFP and data prefetching is compared. Finally we conclude this Chapter with a qualitative comparison between Tuned-CFP and other latency tolerance proposals in literature.

8.1 Qualitative Energy Efficiency Analysis of SRL Mechanism

The hierarchical store queue design gives almost the same performance as an ideal large store queue design as shown in Figure 30. But the problem with this mechanism is that it consumes considerable energy because of the large number of comparators needed for the CAM match.

The SRL structure is much simpler in comparison which does not indulge in forwarding action and off-loads this responsibility to the small store queue and L1 cache. Earlier work [14] has shown the area and power advantage of SRL over hierarchical STQ and we summarize the results from [14] in Table 11. As can be seen from the table, the energy advantage gained from SRL is much larger compared to the average 3% loss in performance shown in Figure 30. Based on these results, it can be inferred that the SRL mechanism gives better return on energy compared to the hierarchical store queue design.

Table 11. Area and power comparison of Hierarchical STQ design and SRL design

Structure	Area	Leakage	Dynamic
Hierarchical STQ	1.4 mm^2	95 mW	440 mW
SRL	0.35 mm^2	40 mW	30 mW
Forwarding Cache	0.45 mm^2	48 mW	37 mW

8.2 Tuned-CFP Performance with Data Prefetch

A core architect might ask whether CFP would still be useful to performance on cores that implement data prefetching hardware [27], and whether the performance of CFP justifies its hardware and increased energy cost.

The answer to the first question is not complicated. CFP should indeed benefit cores with data prefetch hardware. This is because CFP benefits any cache misses whenever they occur, while data prefetch, being a predictive mechanism, benefits only cache misses that are predictable. Any cache misses that are not anticipated by the data prefetch hardware will be handled more effectively by a CFP core.

The question whether the hardware and energy overhead due to CFP can be justified on cores with data prefetch hardware is a more complex question to answer and requires empirical evaluation. In this work, we have evaluated CFP with an aggressive stream-based data prefetcher with 16 stream buffers. Our empirical results show negligible speedup (< 0.01%) from data prefetch on our benchmarks suite, mainly because the memory access patterns of our benchmark traces do not exhibit frequent streams that can be exploited with the stream buffers hardware. We can therefore conclude from our results that CFP has clear performance and energy efficiency benefits, at least on our benchmark traces, with or without data prefetch streaming buffers hardware. Previous work [46] that was done on industrial core architecture model with data prefetch hardware using an extensive set of application traces, including SpecCPU benchmarks and other server, workstation and productivity traces, reported significant performance benefit from CFP beyond the data prefetch hardware. Our current study reconfirms the previous results on a smaller set of benchmarks.

8.3 Qualitative Comparison of Tuned-CFP with Other Latency Tolerant Architectures

Early proposals for latency tolerant out-of-order cores include the Waiting Instruction Buffer (WIB) [30], Virtual ROB [11], Cherry [32], Checkpoint Processing and Recovery [2][3], Continual Flow Pipelines [46] and Out of Order Commit Processors [9]. None of these however deal with L1 data cache misses or execute miss-dependent and independent instructions simultaneously.

The WIB design [30] releases pressure on the issue queue by making long latency cache misses release their issue queue entries. However, in order to support a large instruction window WIB physically buffers the entire window with a multi-level register file and large instruction buffers. In Tuned-CFP the entire instruction window is virtual, with only the miss-dependents, which are far fewer in number, stored in physical locations. In WIB, each instruction dependent on a load miss sets a bit vector dedicated for each outstanding miss, which allows miss-dependents to be reissued post wakeup without needing the complex broadcast logic of the issue queue. Their waiting buffer is organized as a multi-banked structure which allows miss-dependents to be re-issued in any order as and when the wakeup arrives. Physically buffering the entire instruction window also allows instructions dependent on multiple load misses or miss-dependent misses to be moved in and out of the issue queue multiple times, although at the cost of excessive re-execution and energy. In comparison, Tuned-CFP uses an energy efficient single ported waiting buffer that reissues only when the load miss at the head wakes up. While this idea is simple and energy efficient, it also does not compromise on performance mainly because miss-independents continue to be processed in the SMT core while the load at the waiting buffer head is waiting for the wakeup to arrive.

Runahead execution increases memory level parallelism on in-order cores [12], and on out-of-order cores [34] without having to build large reorder buffers. In Runahead execution, the processor state is checkpointed at a load miss to DRAM. Execution continues speculatively past the miss for data prefetch benefits. When the miss data returns, Runahead execution terminates, the execution pipeline is flushed, and execution rolls back to the checkpoint. Except for the prefetch benefit, all work performed during runahead mode is discarded. In [35], the Runahead overhead is reduced with optimizations targeting short, overlapping and useless runahead periods; yet the Runahead execution results are still discarded. In comparison to these, Tuned-CFP executes ahead of L1 data cache misses and does not waste energy by discarding large number of instructions. Some later proposals have evaluated the benefit of not throwing away the execution results from Runahead execution [36][50].

Flea-Flicker [4][5] and Tuned-CFP differ in their execution and result integration methods and their instruction deferral queues. Flea-flicker executes instructions in an in-order pipeline, saves advanced instructions and results in its queue and merges results sequentially during backup pipeline execution.

CFP on in-order cores proposed in [37], iCFP [17] and Sun Microsystems ROCK [7] is suitable for highly energy constrained computing devices but less suitable for the performance needs of conventional single-thread applications targeted by the Tuned-CFP multicore architecture.

BOLT [18] and Tuned-CFP are similar in that both architectures leverage on existing SMT hardware for better performance. But Tuned-CFP has the advantage of virtual register renaming over BOLT, with which it is able to mitigate excessive replay and rollback activity, providing better performance and energy efficiency.

Gonzalez et al. [15] proposed using virtual registers to shorten the lifetime of physical registers. Kilo instruction processors [10] also used virtual renaming and ephemeral registers to do late allocation of physical registers. In contrast to virtual-physical registers and ephemeral registers, Virtual Register Renaming (VRR) [41] and Tuned-CFP do not require physical registers for any allocation of execution results, and accomplish renaming with virtual IDs and capture reservation stations.

CHAPTER 9

CONCLUSIONS

This dissertation leverages on existing SMT hardware in commercial processors to execute miss-dependent and miss-independent instructions, far away from each other in program order, concurrently. This permits the design of a processor core that not only tolerates last level cache misses to DRAM but also first level cache misses that hit on-chip. Moreover, latency tolerance to first level data cache misses allows designing smaller cores with smaller caches for better energy efficiency. In addition to the above, this work allows a simpler reorder buffer based core design with less implementation overhead compared to conventional CFP architecture [46].

This work also presents a tuned Continual Flow Pipeline architecture that uses virtual register renaming and optimized replay policies to improve performance and reduce replay loop circuit activity and checkpoint rollback execution compared to previous CFP designs.

Our Simultaneous-CFP architecture improves performance over conventional-CFP architecture by ~12% when CFP is applied to L1 misses. Our Tuned-CFP architecture further improves performance and power consumption over Simultaneous-CFP architecture by ~10% and ~8%, respectively. The speculative cache and SRL mechanism proposed in this work is able to perform within ~3% of a hierarchical store queue design while providing better energy efficiency. Finally, Tuned-CFP gives better *Return on Energy* for small, medium and large buffer configurations compared to Simultaneous-CFP and non-CFP superscalar cores.

In summary, the ideas presented in this dissertation show that it is possible to design a processor core that gives good single-thread performance and yet remains energy efficient, thus providing a promising design option for future multi-core processors.

9.1 Future Work

We have attempted to cover most of the aspects of designing a latency tolerant processor by providing mechanisms to manage the register file, issue queue and memory ordering. We have discussed the branch predictor and memory dependence predictor needed for our CFP architecture. With and Architectural Level Power Simulator we have been able to measure the increase in power dissipation from replay activity and runahead execution.

The Tuned-CFP architecture proposed in this dissertation works with a reorder buffer. The advantage of using a reorder buffer is that on an independent branch mis-prediction, recovery action is less costly in terms of performance. However as previously pointed out [2][3], the ROB enforces serial retirement constraints and could be a bottleneck for performance improvement. The effectiveness of Tuned-CFP optimizations on a fully checkpointed architecture that uses virtual register renaming, similar to [41], would be an interesting study.

A Tuned-CFP core in a Speculative Multi-threaded processor like Disjoint Out-of-Order Execution architecture (DOE) [42] would also be an interesting study since a Tuned-CFP core provides latency tolerance for cache misses while DOE removes the sequential instruction fetch constraint of a conventional processor by processing instructions very far ahead in the program, for instance control independent points of branches.

The hardware implementation of Tuned-CFP core on FPGA would provide an opportunity to study the timing and implementation related issues in the design, which are not precisely covered in a high-level simulation model. For example, four conditions are checked before an instruction is moved into the waiting buffer – 1) the instruction is at the head of the order list, 2) the instruction is poisoned, 3) one of the instruction buffers is full and 4) every source operand of the instruction is poisoned or ready. It will be interesting to see if all these conditions can be checked in one clock cycle. It also gives a chance to run a wider range of applications in real-time and for longer durations.

Miss-dependent branch mis-predictions are a major cause for concern in CFP architectures because the processor state is rolled back to the checkpoint, discarding all the speculative work done so far. We attempt to reduce the impact of miss-dependent branch mis-predictions by using a history based scheme to identify potential branches that can lead to a rollback and stall the pipeline when such branches are encountered. This static predictor is not able to capture complex control flow patterns in some benchmarks, which explains the difference between it and an Oracle predictor. Branch prediction is a widely researched area and better techniques to predict the behavior of miss-dependent branches will improve both performance and energy immensely.

REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. 2000. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture.* ACM, 248-259.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. 2003. Checkpoint processing and recovery: towards scalable large instruction window processors. In *Proceedings of the 36th Annual ACM/ IEEE Symposium on Microarchitecture.* ACM/ IEEE, 423-434.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. 2004. An Analysis of a Resource Efficient Checkpoint Architecture. In *ACM Transactions on Architecture and Code Optimization*, Vol. 1, issue 4.

[4] R. D. Barnes, E. M. Nystrom, J. W. Sios, S. J. Patel, N. Navarro, and W.W. Hwu. 2003. Beating in-order stalls with flea flicker two-pass pipelining. In *Proceedings of the 36th Annual ACM/ IEEE Symposium on Microarchitecture.* IEEE, 387-398.

[5] R. D. Barnes, S. Ryoo, and W.W. Hwu. 2005. Flea flicker multi-pass pipelining: an alternative to the high power out-of-order offense. In *Proceedings of the 38th Annual ACM/ IEEE Symposium on Microarchitecture.* IEEE, 319-330.

[6] D. Brooks, V. Tiwari, and M. Martonosi. 2000. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture.* ACM, 83-94.

[7] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer and M. Tremblay. 2009. Simultaneous speculative threading: a novel pipeline architecture implemented in Sun's Rock processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture.* ACM, 484-495.

[8] G. Z. Chrysos and J. Emer. 1998. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture.* ACM, 142-153.

[9] A. Cristal, D. Ortega, J. Llosa, and M. Valero. 2004. Out-of-order commit processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture.* IEEE, 48-59.

[10] A. Cristal, O. J. Santana, M. Valero and J. F. Martinez. 2004. Toward kilo-instruction processors. In *ACM Transactions on Architecture and Code Optimization,* Vol. 1, issue 4, 389-417.

[11] A. Cristal, M. Valero, J. Llosa, and A. Gonzalez. 2002. Large virtual ROBs by processor checkpointing, Tech. Report, UPC-DAC-2002-39, Department of Computer Science, Barcelona, Spain.

[12] J. Dundas and T. Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing.* 68-75.

[13] O. Ergin, D. Balkan, D. Ponomarev and K. Ghose. 2004. Increasing Processor Performance Through Early Register Release. In *Proceedings of ICCD-22.*

[14] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai. 2005. Scalable load and store processing in latency tolerant processors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture.* ACM, 446-457.

[15] A. Gonzalez, J. Gonzalez, and M. Valero. 1998. Virtual-physical registers. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture.* IEEE, 175-184.

[16] M. Hill and M. Marty. 2008. Amdahl’s Law in the Multicore Era. *IEEE Computer,* 41(7):33–38.

[17] A. Hilton, S. Nagarakatte, and A. Roth. 2009. Tolerating all-level cache misses in in-order processors. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture.* IEEE, 431-442.

[18] A. Hilton and A. Roth. 2010. BOLT: energy-efficient out-of-order latency-tolerant execution. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture.* IEEE.

[19] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. 2001. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 5(1).

[20] S. Hsu, B. Chatterjee, M. Sachdev, A. Alvandpour, R. Krishnamurthy, S. Borkar. 2003. A 90nm 6.5 GHz 256x64b Dual Supply Register File with Split Decoder Scheme. In *Symposium of VLSI Circuits Digest of Technical Papers*.

[21] W.W. Hwu and Y. N. Patt. 1987. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture.* ACM, 18-26.

[22] W. Hwu and Y.Patt. 1987. Checkpoint Repair for High-Preformance Out-of-order Execution Machines. In *IEEE Transactions on Computers*, vol C-36, issue 12, pg. 1496 – 1514.

[23] K. Jothi, H. Akkary, and M. Sharafeddine. 2011. Simultaneous continual flow pipeline architecture. In *Proceedings of 29th IEEE International Conference on Computer Design.* IEEE, 127-134.

[24] K. Jothi and H. Akkary. 2013. Tuning the Continual Pipeline Architecture. In *Proceedings of International Conference on Supercomputing.* ACM, 243-252.

[25] K. Jothi and H. Akkary. 2013. Virtual Register Renaming: Energy Efficient Substrate for Continual Flow Pipelines. In *Proceedings of the 23rd Great Lakes Symposium on VLSI.* ACM, 43-48.

[26] K. Jothi and H. Akkary. 2013. Tuning the Continual Pipeline Architecture with Virtual Register Renaming. In *ACM Transactions on Architecture and Code Optimization.* ACM, Accepted November 2013.

[27] N. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*. 364–373.

[28] T. Karkhanis and J. E. Smith. 2002. A Day in the Life of a Data Cache Miss. In *Workshop on Memory Performance Issues*.

[29] M. S. Lam, R. Wilson. 1992. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture.* ACM, 46-57.

[30] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. 2002. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture.* ACM, 59-70.

[31] S. Manne, A. Klauser, and D. Grunwald. 1998. Pipeline Gating: Speculation Control for Energy Reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture.* ACM, 132-141.

[32] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic and J. Torrellas. 2002. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual ACM/ IEEE Symposium on Microarchitecture.* ACM/ IEEE, 3-14.

[33] A.Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. 1997. Dynamic Speculation and Synchronization of Data Dependences. . In *Proceedings of the 24th Annual ACM/ IEEE Symposium on Microarchitecture.* ACM/ IEEE.

[34] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. 2003. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture.* IEEE, 129-140.

[35] O. Mutlu, H. Kim, and Y. N. Patt. 2005. Techniques for Efficient Processing in Runahead Execution Engines. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture.* ACM, 370-381.

[36] O. Mutlu, H. Kim, J. Stark, and Y. N. Patt. 2005. On Reusing the Results of Pre-Executed Instructions in a Runahead Execution Processor. In *Computer Architecture Letters* Vol. 4, issue 1.

[37] S. Nekkalapu, H. Akkary, K. Jothi, R. Retnamma, and X. Song. 2008. A Simple Latency Tolerant Processor. In *Proceedings of the 26th IEEE International Conference on Computer Design.* IEEE, 384-389.

[38] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. 1996. The case for a single-chip multiprocessor. In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems.

[39] D. B. Papworth. 1998. Tuning the Pentium Pro microarchitecture. In *IEEE Micro*, Vol. 16, No. 2, 8-15.

[40] M. Pericas, R. Gonzalez, D. Jimenez, and M. Valero. A Decoupled KILO-Instruction Processor. 2006. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 53–64.

[41] M. Sharafeddine, H. Akkary and D. Carmean. 2013. Virtual register renaming. In *Proceedings of the 26th International Conference on Architecture of Computing Systems*, 86-97.

[42] M. Sharafeddine, K. Jothi and H. Akkary. 2012. Disjoint Out-of-Order Execution Processor. In *ACM Transactions on Computer Architecture and Code Optimization,* Vol. 9, No. 3, 2012.

[43] J. E. Smith and A. R. Pleszkun. 1988. Implementing Precise Interrupts in Pipelined Processors. In *IEEE Transactions on Computers,* Vol. 37, 562-573.

[44] J. E. Smith and G. S. Sohi. 1995. The microarchitecture of superscalar processors. In *Proceedings of the IEEE,* Vol. 83, No. 12, 1609-1624.

[45] G. S. Sohi. 1990. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. In *IEEE Transactions on Computers,* Vol. 39, 349-359.

[46] S.T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. 2004. Continual flow pipelines. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 107-119.

[47] R. M. Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic Units,'' in *IBM Journal of Research and Development*, Vol. 11, 25-33.

[48] D. M. Tullsen, S. J. Eggers, and H. M. Levy. 1995. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture.* ACM, 392-403.

[49] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. 1996. *The 23rd Annual International Symposium on Computer Architecture,* 191-202.

[50] S. R. Wolff and R. D. Barnes. 2011. Reexamining Instruction Reuse in Preexecution Approaches. In *the 9th Annual Workshop on Duplicating, Deconstructing and Debunking*.

[51] www.simplescalar.com