UPPSALA
UNIVERSITET

Institutionen för informationsteknologi

Besöksadress:
ITC, Polacksbacken
Lägerhyddsvägen 2

Postadress:
Box 337
751 05 Uppsala

Hemsida:
http:/www.it.uu.se

Abstract

Store prefetch policies

Analysis and new proposals

Boström, Carl

This work will be focusing on how to gain performance when executing programs on a CPU. More specifically the store instructions will be studied. The memory instructions often causes huge delays while waiting on data to be brought in from main memory to the L1 cache. If an out-of-order (OoO) CPU cannot be fed with useful instructions, cycles will be wasted. To try to over overcome this issue, my supervisor, Alberto Ros Bardisa, have come up with the idea to use predictors to try to predict which data to be needed in the feature and brought it in to the L1 cache in advance. This is similar to the branch predictor that has been around for a while, where the CPU is fed with instructions to start working in advance based on the prediction whether or not a branch is to be taken. In this case we guess on which data to be brought in to the L1 case instead of which instructions to be brought in, in advance. Of course you want the prediction to be correct, but even if we assume that it always prefetches useful data there might still be a problem: When to bring in the data? Too late an you still need to wait since the need of it occur before it is in place. Too early and it will be evicted from the L1 cache due to space issues and there will be a waiting time to brought it in again when the need occur. The latest will also be a waste of energy since we brought in something to be evicted before use. The question I will try to answer with this master thesis is when to prefetch data to gain optimal performance.

Handledare: Alberto Ros Bardisa
Ämnesgranskare: Stefanos Kaxiras
Examinator: ??

Sammanfattning

Detta arbete kommer att fokusera på hur prestandan vid körning av program på en CPU kan ökas. Mer specifikt kommer store instruktionerna att studeras. Minnes instruktioner osakar ofta stora förseningar i samband med väntan på att data ska överföras från RAM-minnet till L1 cachen. Om en out-of-order CPU inte kan hitta andre instruktioner att jobba med i väntan på datan så kommer dessa cyklar att slösas bort. För att försöka överkomma denna problematik så har min handledare, Alberto Ros Bardisa, kommit med iden att använda predictors för att förutspå vilken data som kommer att användas inom snar framtid och överföra den till L1 cachen i förväg. Detta liknar branch predictors som används i flera år, där CPUn matas med instruktioner att börjar jobba med baserat på en gissning om branchen kommer att tas eller inte. Här gissar vi vilken data som ska överföras till L1 i förväg istället för vilken instruktioner som kommer efter en branch. Såklart är det önskvärt att gissningen är korrekt, men även om den är det så kan vi ha problem: När ska datan överföras? För sent och vi måste fortfarande vänta eftersom behovet av data uppkommer ni den är på plats. För tidigt och data kan bli avlägsnad från L1 cachen på grund av platsbrist och det blir en väntetid för att överföra datan på nytt när behovet väl uppstår. De senare är också ett slöseri av energi då vi överför något som kommer att avlägsnas innan användning. Frågan i detta masterarbete är när data ska överföras för optimal prestanda.

Acknowledgment

I want to thank Alberto Ros from Murcia University and Stefanos Kaxiras from Uppsala University for presenting the interesting topic of this thesis to me. Alberto and Stefanos have give me the required information to carry out this work. They have also shared their versions of Sniper 2.2.2 and Gems 2.2.3 with me and even manipulated it with this work in mind. Ideas about new store policies have also been shared by Alberto.

Contents

List of Figures

List of Tables

Glossary

ALU Arithmetic logic unit, the computation part of a CPU... 6, 7

Branch predictor Predicts if a branch (i.e. one more leap in a loop) is to be take or not.. 6

cache A small memory unit with small access times, which holds data that are likely to be needed in the near feature.. 7

Commit Last stage, here are the result of the out of order executed instructions putted back into order.. 6

Decode Second stage, the CPU figures out what type of instruction it is and what to done with it.. 6

Dispatch Third stage, dependences is here figured out i.e if some other instructions needs to be finish before the computation with this one. begins. 6

Execute Fourth stage, here takes the actual computation place (in one of the ALUs).. 6

Fetch The first stage in the pipeline where instructions are brought in to the CPU.. 6

FQ Fetch queue, fetch instructions waits here to be decoded.. 7

Icache A small memory unit with small access times, which holds instructions that are likely to be executed in the near feature.. 6

IQ Instruction queue, where the instruction waits for suitable ALU and any dependent instructions to be finished.. 6, 7

NoPrefetch No prefetch will be issued, the data will be brought in when the store is at the the least place in SB.. 8

OnCommit The prefetch is issued then the store leaves the ROB.. 8

OnExecute The earliest and must speculative prefetch policy.. 8

OnExecuteAndOnCommit The prefetch will be issued twice, both as for OnExecute and OnCommit.. iii, 14

OnNonBSPeculative This prefetch stores that are to be executed i.e. it can not turn out to be infected be a wrong branch prediction.. iii, 14

PCPredictor When to issue the prefetch will be based on earlier experiences for stores with PC:s in the same range.. iii, 15

ROB ReOrderingBuffer, the result of an instruction will wait here until the on before it (in the original order) is done.

. 6, 7

SB Store buffer, here waits all the stores for its data to be ready in the L1 cache.. 7

StoreBufferLoad The prefetch will be earlier if the load of SB is low.. iii, 15

1 Introduction

1.1 Motivation

We are always in the need of faster and smaller computers that burns less energy. In the second part of the twentieth century the speed of our CPUs (center process unit) were increased by letting them run faster and faster. The problem is that a linear frequency speed up causes an exponential growth in energy and heat. A rule of thumb is that you will be spending as much energy as needed for the computation to cool the circuit. Since we want to make smaller and smaller units, its gets harder and harder to cool down. Less materials that are getting warmer will be almost impossible to cool down after some limit. But we are on the other hand still developing software that requires more and more of the hardware. When we cannot speed up our CPUs in the same way as before, we need to see if we can be more effective. Thats when CPUs introduced pipelining. Lets say that an instruction take six seconds to compute the we and up with two finished instructions after twelve seconds. If we then divided the work into three stages where one instruction spends two second each, meaning that we can begin working on a new instruction every two seconds. Given this, four instruction (twice the amount) will be done in twelve seconds, this results in increasing the work that can be done during a period of time without increasing the speed of the CPU and therefore the energy consumption roughly stays the same.

Another thing to take advantage of is that computing some instructions especially memory instructions (loads and stores) are involving long waiting times. If we then instead of just waiting, goes ahead and computes other work, with the next instructions. We are not effected by the waiting time in the long run because we are starting the computation of the instruction next in line while waiting. This is called out of order execution and it is up to the CPU to decide if the work can be done in another order and still produce the same outcome.

We have sad that the computation of memory instructions include waiting on data and that we can do useful tasks while waiting. Still it is good if we can decrease the waiting time, since we might not always have useful tasks to work on that cover the entire waiting time. In this thesis will look if we can shorter the waiting time by prefetching the data needed for store instructions in advance. The earlier data is prefetch the earlier it will be ready to use. This little pieces in the question of making a CPU do more useful work per unit of energy is what this thesis will be about.

1.2 Scope

This master thesis will introduce and evaluate three given policies for prefetching data for load instruction. There are not much research to be found that are focusing on the acceleration of load instructions in particular. The three basic policies will be combined in different ways to try to come out with a more optimal policy in terms of

speed up (number of cycles) and power consumption. A bit of the preparation work in terms of manipulating and understanding traces from benchmarks programs will be covered as well.

1.3 Related work

Below you can read about four related works that have in common that they have all used gems 2.2.3.

No need to squash and re-execute for reordering Alberto Ros, Trevor E. Carlson, Mehdi Alipour and Stefanos Kaxiras, have written the report "Non-Speculative Load-Load Reordering in TSO" [12]. For decades it has been a fact that if a CPU (in a multi core architecture) reorders a load to gain performance then the other core must upon observation squashed and re-execute speculative instructions. This paper can, for the first time, show that the action is not necessary for total store ordering (TSO). The reordering can instead be hidden for the other cores. This allow us to commit reordered loads out-of-order without having to wait for the loads to become non-speculative or without having to checkpoint committed sate and rollback if needed.There solution is cost-effective and increase the performance of out-of-order commit by a sizable margin in comparison with the base case where memory operations where not allowed to commit if the consistency model cold be violated.

Transactional Memory Systems. Lee Baught, Naveen Neelakantam and Craig Zilles, has written a paper called "Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory" [5]. In this paper they compare and construct new policies for fine-grained memory protection, hardware support for atomic memory instructions are to be compared with software transactional memory system (STM). They demonstrate new hybrid TM (transactional memory system) that avoids common over heads that was associated with the old once.

Write Update Optimizations. Liqun Cheng and John B. Charter has written a paper called "Extending CC-NUMA Systems to Support Write Update Optimizations" [7]. Modern multi core processors employ write-invalidate coherence protocol to ensure coherence between multiple cores that shears data. The problem is that this introduces read misses witch slows down the execution. In some cases performance can be gained by using an update protocol instead of an invalidate one. Previous research have pointed out the the updating policy causes a lot of network traffic in between the cores. Given this the paper introduced a speculative update policy that only update the data for the cores that are likely to need it and invalidate the data for the other cores. This speculative update requires no changes to the cores, bus interface or memory consistency model. It was find to reduce the miss rate by 30 %, network traffic by 15 % and on the same time improve the performance by 10%.

Distributed Cooperative Caching Enric Herrero, José González and Ramon Canal, has written a paper named "Distributed Cooperative Caching" [9]. They presents the Distributed Cooperative Caching, a scalable and energy-efficient scheme to manage chip multiprocessor (CMP) cache resources. They have redesigned the Coherence Engine to allow its partitioning and there by eliminate the size constraint imposed by the duplication of all tags. Distributed Coherence Engines spread across all nodes improve stability and load balancing of the network traffic over the entire chip by avoiding bottlenecks. This results in an performance increase for a 32-core CMP by 21 % or a traditional shared memory configuration and 57 % over the Cooperative Caching scheme.

1.4 Structure of the report

This report consist of seven chapter and one appendix chapter.

Chapter 1 - Introduction It gives a motivation of the work along with its scope and related work.

Chapter 2 - Background This chapter is divided into two parts. The first covers the architecture and related terms that will be used throughout the report. The first part will also cover the three given polices. The second part covers the given softwares used and benchmarks.

Chapter 3 - Setting Up The Testbed This can be seen as a continuing of the second part in chapter 2, here we go through the manipulation of existing tools and the creating of new ones. The trace which has manipulated to support the simulation is covered in this chapter.

Chapter 4 - Proposed Prefetch Policies This will introduce all prefetch polices that is proposed within the work of this thesis. They are all combinations of the basic three introduced in the background.

Chapter 5 - Results Here you will find graphs comparing the different polices with different settings in terms of power, L0 accesses, useful prefetch and energy.

Chapter 6 - Conclusions This chapter will try to explain the results and draw some conclusions of it.

Chapter 7 - Discussion mandatory??

Chapter A - Implementation details In this appendix we will cover the key parts of the implementation of the new policies in some pseudo code examples.

2 Background

This chapter is divided into two sections. The first one, 2.1 Theoretical Background, will provide a description of a simple CPU architecture which focus on the parts used for executing memory instruction stores. This will be followed by a introduction of the four already implemented prefetching strategies. Section 2.2 will go thought the work flow of this thesis and introduce the tools that are used, modified or created, as well as how they are combined.

2.1 Theoretical Background

2.1.1 The simulated Architecture

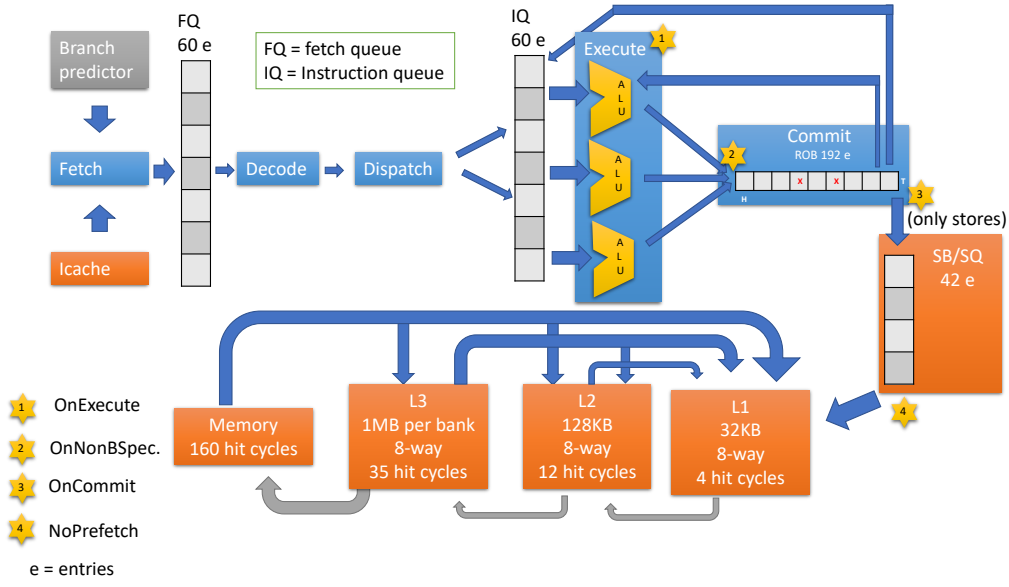

Figure 2.1: Our architecture

In the image 2.1 you see the architecture in use. This images symbolizes almost all the features and parts of the CPU that will be discussed and analyzed from different perspectives. If there is a picture that you should keep in mind when reading this report then this would be it. The stars symbolizes whre a prefetch is issued for the three simplest and already implemented store prefetch policies in 2.1.2 as well as one of the proposed once namely NonBSpeculative. The numbers is the one used for the simulated CPU (see section 3.3). In modern CPU's there are often more then one core which all have a process pipeline with several stages. The pipeline is split in stages to increase the throughput of executed instructions. One can compare it with

a car factory where the car travels on a conveyor belt through different stations and on each station something is done with the car, i.e. seats, wheels or the engine can be installed.

The CPU to be simulated have a five stage pipeline. As going through the stages, some key parts of the CPU will be introduced. One thing to mention before moving on is that the CPU will execute instructions out of order but it promise that the effect will always be like if the instruction where executed in order.

Fetch

A new instruction (see 3.4) will be added to the processor. In the basic case the next instruction will be brought in from the **Icache** (Instruction cache). Icache is a L1 cache the holds the instructions to be executed in the near feature **trace_ref**. The CPU has a **PC** (program counter) which keeps track of the address of the current instruction that is fetched, when fetching the next instruction the PC will be incremented by the length of the current instruction. Caches will covered in 2.1.1. Different loops are widely used in programming and a loop tells us if we should go back to the beginning of the body of the loop or if the loop should be terminated and the lines below should be executed. A CPU cannot for example execute C code, it needs to be compiled first. Compilation is a translation from code to instructions within the instructions set that are supported by the CPU in question. A loop will be compiled to a branch instruction which will have the target instruction address and the condition to be satisfied if the branch is taking (take one more lap in the loop). The problem is that we want to continue to bring in new instructions before the condition have been computed, but which instructions will be next. Here comes the **Branch predictor** to use, it predicts the outcome of a branch and make the CPU able to load instructions based on that prediction. If a branch is taken the PC is set to the branch target address. If a prediction turn out to be wrong the CPU will remove the work and all the side effects produced by the wrongly executed instructions. Fetched instructions will be placed in the fetch queue (our has 60 entries).

Decode

The instruction (from the instruction queue) will here be interpreted.

Dispatch

In this stage we take care of dependencies. If we want to add two numbers and then multiply the result with a third number, then the addition has to be finished before the multiplication can be executed, thus the multiplication has a dependency to the addition. If we also have to store the result of the multiplication then the multiplication needs to be computed before we can store the result of it. This gives a dependency between the store and the multiplication. When the dependencies have been worked out, the instructions will be put in the **Instruction queue (IQ)** (who has 60 entries as well).

Execute

A CPU has several **ALUs** (Arithmetic Logic Units) which does the computations i.e. the brain of the CPU. Different ALUs can preform different operations for example; all can do integer addition but only one can preform floating point division. The instructions will wait in the instruction queue for a suitable ALU to be free and for the instruction it are dependent on to be finished.

Commit

Commit is the last step of the pipeline and it contains a **ROB** (ReOrdering Buffer) with 192 entries. This buffer puts the out of order executed instructions back into

order. This means it is possible, given just the ROB, to execute 191 instructions ahead while waiting on an instruction to be finish. When all instructions in a sequence are done they can leave the pipeline. The x in figure 2.1 illustrates a squashed instruction, if a branch prediction turns out to be wrong then we need to squash all the instruction that should never have been executed but have been based on that faulty prediction. Going back to the example with the sum of two number multiplied by a third number. When the addition is done and the result is added to the ROB, the multiplication instruction in the **IQ** (Instruction Queue) is informed that its dependent instruction is done (the arrow from ROB to IQ in figure 2.1). Finally the arrow from ROB to ALU will provide the result from the addition to the multiplication.

SB

The path that have been described until now is taken by every instruction but now we will continue the way that store instructions are taken. When a store instruction leaves the ROB it comes to the **store buffer** (which can also be called store queue): The store instruction is here represented in terms of the memory address to be fetched and the data to be stored. The buffer is FIFO-ordered (first in first out), the first store in the buffer sits and waits for its data to be ready in the L1 cache (see 2.1.1) and blocks all operations behind in the buffer. This is the means that the buffer only interact with the L1 cache.

Memory and caches

The memory structure (in the center of figure 2.1) is of course a key part when talking about store instructions. We begin with the main memory that have a storage capacity of some gigabytes, a latency of 160 cycles if a hit, and sits on the mother board. Then we have the cachees, which are placed on the CPU chip. A cache is a quick and small memory unit in which the data that are likely to be used in the near feature is placed. In this case we have three which are named L1 (32kiB 8-way 4 hit cycles), L2 (128kiB 8-way 12 hit cycles) and L3 (1MB per bank 8-way 35 hit cycles). Where L stands for level and the greater the digit, the bigger the cache is and the further away from the core it is placed. A bigger cache means that it takes longer time to find certain data in it. The time is measured in **cycles**, in every cycle something can occur in the CPU i.e. a instruction can be placed in FQ and/or another one can be placed in the ROB. If our CPU runs in a frequency of 2.2 GHz, there are:

$$2.2 \times 1000^3 = 2.2 \times 10^9 \text{cycles per second}$$

Given that one cycles takes:

$$\frac{1}{2.2 \times 10^9} = 2.2 \times 10^{-9} s = 2.2 ns$$

This give us that it takes: $4 \times 2.2 = 8.8 ns$ to retrieve data from the L1 cache. A cache keeps copies of needed data and the data will be saved in different regions in the cache depending on the address of the data. The number of ways a cache have is the number of data shanks in every address range that can be kept simultaneously in the cache. A larger number of ways means that it takes longer time to determent if a certain data is present in the case. A smaller number of ways increases the risk of conflict misses, that is when the cache runs out of places for data from a specific address range. Our caches are all 8-ways which means that a piece of data can be find in one of eight places (if it is in the cache) and that a cache can hold no more then eight data chunks from a given address range. The Icache is like another L1 cache that only holds instructions, the L2 and L3 cache holds both data and instructions. When data is loaded into the L1 cache from the main memory it will often be loaded to all the other cachees at the same time.

2.1.2 State-of-the-art of prefetch policies

The question for this master thesis is when to prefetch data into the L1 cache to minimize the time a store instruction has to wait in the first place of the store buffer. Three approaches are already implement in the Gems simulator 2.2.3 and these will be described and their pros and cons while be mentioned below, based on the architecture introduced in section 2.1.1. The starts mentioned in the paragraph headings are the once in figure 2.1

OnExecute (star 1) This is the earliest and most speculative one, where the operation to brought memory into the L1 cache is issued in the execute stage. This ensure that the data arrives in L1 before it is needed, i.e. its instruction is first in the buffer. The downside is that several things with impact on the data to be prefetch can occur. First, if the store instruction is affected by a branch, that branch can turn to be misspredicted which means that we waste energy and space in the small L1 cache by bringing in unneeded data. Bringing in data to a cache can cause an eviction of other data. If a store instruction arrives to the first place in the buffer and found that its data have been evicted from the L1 cache it have to wait for the data to be brought to the L1 again. This will hopefully take less time since the data can be left in the L2 cache and be brought from it instead of the main memory. Given that the prediction is right we might still be in trouble, since when we issue the prefetch, the instruction have to finish the pipeline and passes through the queue in the buffer. This might take a long time. During the time the data can have arrived to the L1 buffer and been evicted due to lack of space since more data have been prefetched from the point in time it arrives in L1 until the instruction is first in the buffer. A too early prefetch can also be a vulnerability since malicious software can cause a prefetch of illegal data before the core figures out that it is illegal, and when it does the data have already been exposed. To conclude: one can say that this alternative is the best if nothing goes wrong but it is a lot of things that can go wrong. OnExecute has proposed in a paper by Kourosh Gharachorloo and Anoop Gupta and John Hennessy. [8]

OnCommit (star 3) Here the prefetch is issued in the commit stage (when passing the instruction to the store buffer). This means that unneeded data will not be prefetched. There is still a possibility that the data while be brought in and evicted due to lack of space, while waiting, as describe in the previous paragraph. Prefetching data on commit might still mean that the instruction may be waiting in the first line of the buffer. This can, in the worst case, fill up the entire buffer which can stall the processor i.e. block it from executing any other instruction. OnCommit is used by Intel 64 and IA-32 Architecture [10]. (They do not use the name OnCommit but they quickly describe a load prefetch with the following sentence: "Reading for ownership and storing the data happens after instruction retirement and follows the order of store instruction retirement." "Reading for ownership" is the same as prefetch with write permissions and "instruction retirement" is another name for "instruction commit".) To conclude, OnCommit is the man in the middle between the two extremes, OnExecute 2.1.2 and NoPrefetch 2.1.2.

NoPrefetch (star 4) Here we have no prefetch, the data will be brought into the L1 cache then the instruction is first in the buffer. The data we brought will be needed and not evicted before use, this waste no energy. The downside is that every instruction will be blocking the buffer for a long time waiting on its data to become available in the L1 cache. The risk for filling up the buffer and stall the processor (see 2.1.2) is high. To conclude: this is the most energy efficient but the most time consuming trade off. Upon analyzing the source code of Gem5 [6] it was discovered that NoPreftch was the default implemented policy (during the work of this thesis all other prefetch policies was implemented as well).

2.2 Simulation infrastructure

Below the used tools which are provided by the supervisor, Albert Ros Bardisa, are covered. The one that are modified or created within the work of this master thesis are to be found in chapter 3.

2.2.1 Benchmarks

To examine how a CPU behaves and how a memory prefetching approach will work one have to run, or in this case simulate the run of a program. This is often called benchmark. For this work the SPEC CPU®2016 industry-standardized benchmark suite [3] was used. It is designed to stress both the CPU and the memory subsystem and consists of 55 different benchmark programs that have been used to evaluate the different prefetch policies.

2.2.2 Sniper

Sniper is a parallel, high-speed and accurate x86 simulator with support for multicore according their web page [2]. Sniper takes two things as input, a bunch of configurations that describes the architecture to simulate, and a command line with the call to the application which we want to simulate, for example one of the programs in 2.2.1. Sniper can then produce graphs and other document that describes and measures the simulation of the chosen program on the chose CPU configuration. Two examples of plots that can be generated is CPI stacks [1] and energy stack which are bar diagrams, with one bar per thread in the program. The bars are divided in different regions based on the percentages of time or energy spent in that region. The region can be: Ifetch, mem-l1, mem-l2 or branch.

2.2.3 GEMS

The Wisconsin Multifacet Project at the university of Wisconsin have released a General Execution-driven Multiprocessor Simulator (GEMS). The simulator is written in C/C++ and is an open-source software. The version used in this work was provided by my supervisor (Albert Ros Bardisa) who had implemented the three basic prefetch policies (NoPrefetch, OnCommit and OnExecute) along with one that he have come up with but not publicized (OnNonBSpeculative) as well as support for the changes to the traces in Sniper that was done during this thesis (see 2.2.2). Support for the policies to be proposed in this work was implemented based on the received source code.

GEMS takes a path to a folder containing traces (like in 3.4), an array of configuration settings (see 3.2 and 2.1) and a path for writing the output stats-file. A stats-file consist of around 2000 lines and are divided into two part, a configuration part were you find the information given to GEMS in the configuration array and a Stats part with a lot of measurements like number of cycles and the number of accesses to the caches, to just mention tWo of them.

3 Setting Up The Testbed

3.1 Changes to Sniper

In this work we will focus more on the traces which also can be generated by Sniper. Traces are simply a list of the CPU instructions that was simulated, we come back to this in 3.4. The trace is what we bring from Sniper to the next step in this work flow. Some minor updates has been done to parts of Sniper that prints the trace:

- Writing the start PC (program counter) first in the trace.
- Writing the length of the previous instruction to gather with every instruction.
- Prints the target PC of a branch together with a "*" if the branch is taken.
- Implementing a new flag, *–insert-clear-stat-by-icount=n*, where n is a positive integer, which leaves a line with just a "C" after n lines of instructions.

3.2 Configuration

Configurations is a list of keys with assigned values that sets out the characteristics for the CPU to be simulated. Some examples can be the number and sizes of cashes, the sizes of buffers, number of cores, frequency and a lot more. The PROCESSOR_STORE_PREFETCH key is the one used to set the prefetch policy. Another key to keep track of is SIMULATION_BENCHMARK which holds the name of the simulation benchmark program so it can be added to the stats-file (see 2.2.3). Between the different simulations only the values of the two described keys is changed, the rest stays the same and sets out the CPU architecture which will be described in the following subsection.

3.3 CPU architecture

The CPU design simulated for this master thesis (the numbers introduced in figure 2.1) was the one used in the paper "Non-Speculative Load-Load Reordering in TSO" [**?**]. You can read more about it under related work 1.3. they claim that the architecture follows the Bell-Lipasti design.

3.4 Trace: Interface Sniper-GEMS

A CPU cannot read program languages like C or Java directly. Instead the CPU has a set of instructions that it can compute, called instruction set. The instruction set can differ from one type of CPU to another, but three types of instructions that can been found in some way in any instruction set are; arithmetical operations such as, addition, subtraction, multiplication and division, memory instructions such as,

loads and stores, and branches. Branches tells the CPU if it should jump to another instruction or not. Consider a for-loop, that loop will be represented by a branch instruction telling the CPU if it should jump back to the beginning of the loop or continue on below it. When compiling a C or Java file you end up with a binary file that contains the translation from the source code to a particular instruction set, this is the file you use to run the application in question. A trace file is a human readable "binary file".

PC stands for program counter and it keeps track of which instruction in a program to be executed next, then taking a branch the CPU changes the PC to the one of the target instructions from a branch and carries on from that.

The trace files used here will cover more details regarding memory instructions, that is loads and stores. Thats why a trace file has a linebreak after each memory instruction. Below follows a number of examples on lines from traces which will be explained and translated to English.

The trace starts with the value of the starting PC in hexadecimal-format from now

```
4030fc
```

Figure 3.1: The first line of a trace is the starting PC

on the differentiation of the PC from on instruction to the next is write in decimal-form along with the instruction. Note that the second instruction shows the length (the change to the PC) of the first one, and so on.

```
0 4 L4 ee7e220 4
```

Figure 3.2: Two anonymous functions, both with length four followed by a load of 4 bytes from memory address ee7e220

The 0 and the 4 denotes two anonymous functions (we care more about loads, stores and branches). Since the first digit denotes the size of the previous instruction we know that the two anonymous function both have the size of four (the "4" after the first space and the "4" after the "L"). The "L" tells us that the instruction is a load instruction and we load from the hexadecimal memory address ee7e220. The number of bytes to be loaded is written after the last space, four in this case.

```
0d1d3 b4d1t99* S99 7fff8368acd8 8
```

Figure 3.3: A anonymous function of size 4, that are dependent of the the first and the third instruction prior to this. Then a taken branch to PC+99 ahead with a dependency to the prior instruction where we a store of 8 bytes to the address 7fff8368acd8.

Given the previous examples this line start with one anonymous instruction of size four. In this case the first instruction are also dependent of the result from the two previous instructions, this is denoted by the two "d"s before the first space. The digit after each "d" points out how many instructions before the dependent one are, here one and three. The next instruction starts with a "b" which tells us that it is a branch. The branch are dependent of the instruction before, the anonymous one that

was discussed in the beginning of this example. The "t" denotes the branch target follow by the difference between the current PC and the target address. Here 99 should be added to the PC if taken. The "*" denotes that the branch is taken. After that there is an "S", namely a store instruction. Here one can once again see that the branch is taken since the difference from the branch instruction is 99 (directly after the "S") as the branch target. A store interaction follows the same pattern as a load, here it stores 8 bytes to the memory address 7fff8368acd8.

```
0m3 b4t-166* L-166 7fff8368acd8 8
```

Figure 3.4: A anonymous instruction of size 4 that are depend of data from memory that are retrieved in the third prior instruction. After that there is a branch with a negative PC difference (-166) as the target which leads to a load of 8 bytes to the address 7fff8368acd8.

In this example we only have two new things to cover, the rest have been covered in the previous examples. The "m" denotes a memory dependency, some data that are to be loaded in the third previous instruction. The "d"'s are dependencies on data to be computed by the instruction in question and the "m"'s are data that is needed to be loaded into the CPU. The next thing to cover is that we here have a negative PC difference for the target address. We now have all knowledge to directly translate the line to English: A anonymous instruction of size 4 that are dependent of data from memory that are retrieved in the third prior instruction. After that there is a branch with a negative PC difference (-166) as the target which leads to a load of 8 bytes to the address 7fff8368acd8.

3.5 Metrics for evaluation

During this thesis a python script was written to automatizes the run of the simulation on every sub folder in a predefined folder (all sub folders have to contain traces from a benchmark program). In the source code there is a predefined array with the names of the prefetch policies you want to run. The script will then run GEMS on every trace with every store prefetch policy and name the stat-file in the following way: [name on benchmark]_[name on store prefetch policy].stats

The script will also generate the results by reading some values from every stats-file, the values to be read are hard coded into an array. After collecting all these values from every file, the values from all benchmarks for every store policy will be summed. The values for NoPrefetch will be set to 100% and the other will get their percentages based on that. This percentages will the be plotted into bar charts (which can be find in the results).

3.5.1 Energy graphs

Table 3.1 lists the energy consumption for accessing our three caches and the network flits which are the energy needed for transmission of a data unit between caches and the pipeline. One flits for transmission of control messages and five for data [12] In order to retrieve the numbers for the caches we use the CACTI-P tool [11] to estimate the power consumption of the different cache structures assuming a 22 nm technology node. To estimate the dynamic energy consumption of the interconnection network, we assume that it is proportional to the data transferred [4] and that each flit transmitted through the network consumes the same amount of energy as reading one word from an L1 cache each time that it crosses a link (link and router energy). These number is used by the python script 3.5 to generate the power graphs that can

CPU part	Energy [nJ]
Accessing L0	0.013343
Accessing L1	0.0214929
Accessing L2	0.0454353
Network flit	0.0033357

Table 3.1: Energy measurements for different CPU parts.

be seen in the results 5.

4 Proposed Prefetch Policies

In this chapter the store prefetch policies proposed in this master thesis will be described. This polices are based on the three basic ones (OnExecute, OnCommit, NoPrefetch) see 2.1.2. The polices introduced below will be a combination of two or more of the basic ones.

4.1 OnNonBSPeculative (star 2 in figure 2.1)

This alternative is proposed by Albert Ros Bardisa (the supervisor of this master thesis) and has not been implemented nor tested in a broader setup, it is only implemented in the GEMS simulator 2.2.3. The alternative can be seen of as taken the best of OnExecute 2.1.2 and OnCommit 2.1.2. If you have a load instructive that are effected by a branch we will wait until the outcome of the branch is known, to make sure that all data we bring in to the L1 cache will be needed. We act as OnCommit. If the load is not effected by a branch we will then benefit on the performance by acting like OnExecute 2.1.2. The difference between OnNonBSpeculative and OnCommit is that the prefetch will, if not effected by a branch, be issued upon arriving to the ROB, we earn the time the instruction is waiting in the ROB if we compare with OnCommit.

4.2 OnExecuteAndOnCommit

OnExecute 2.1.2 is the most speculative one, and one issue that was pointed out was that the data can be brought in to the L0 cache and evicted from it during the time it takes for the instruction to pass through the rest of the pipeline and the store buffer. To lower the risk of eviction OnExecuteAndOnCommit will issue a prefetch in the execute stage and once again, in the end of the commit stage, to make that data less vulnerable to eviction. The second prefetch is issued in the end of the commit stage, were it is known that the instruction will be executed i.e. it will not be blocked by a branch. Given this, this policy might be named OnExecuteAndOnNonBSpeculative. The difference between OnCommit and OnNonBSpeculative is that OnNonBSpeculative will wait with the prefetching of data, for a store instruction, that are effected by a branch until that branch have been computed, so it does not prefetch data for an instruction that will not be executed. This policy sets the instruction it currently work with first, it will not bother about the power consumption (prefetch twice is a waist of power) or if the two prefetching attempts will lead to an eviction of some other data. If the prefetch for one store instruction cause an eviction of the data for a instruction that are ahead of it in the pipeline and store buffer then our instruction will be blocked while the one before is waiting on its data to be brought into L0 again. When the instruction ahead gets it data back in it can cause on eviction of our data and you end up waiting on our data after waiting on the instruction that was ahead of you. In this way you can shoot ourself in the foot.

4.3 StoreBufferLoad

StoreBufferLoad takes the current load of the store buffer in to account when deciding when to prefetch. The idea is the if the load of the store buffer (how full it is) are low then the instruction will be ready earlier since it will pass the buffer in a shorter amount of time, and there is a higher need of an early prefetch as well. In that case the prefetch will be like for OnCommit. On the other hand if the load is high there is not really a need of an early prefetching since it will take a while for the instruction to pass through the buffer. A early prefetch here will also cause a high risk of eviction since there must be a lot of needed data in and on its way to the L0 cache. The action here is like OnNonBSpeculative.

The next question to consider is what the threshold should be. To answer this several versions of the StoreBufferLoad policy was implemented and named by the following convention StoreBufferLoad=x where x is a number of filed entries in the store buffer. If the less the x entries are filed then the load is considered to be low. The supported values of x are: 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024.

4.4 PCPredictor

Predictor has been used in CPUs for a while to predict if a branch is to be taken or not by keeping a history of the previous branches with the same target address and if those branches has been taken or not. PCPredictor is a predictor that based on the PC for a store instruction will speculate if the prefetch will hit, be to early (miss) or be to late (miss) and then decide on when to prefetch based on the prediction. The prediction is an integer between two and zero, here follows a list over what they mean and what prefetch policy to be used:

(0) means that it predicts to hit in the L0 cache and therefore no prefetch is needed, acting like NoPrefetch.

(1) means that it predicts an early miss, in this case it acts like OnNonBSpeculative to try to prefetch the data later so it will not be evicted before use.

(2) means that it predicts a late miss, it now needs to prefetch earlier and it acts like OnCommit. **OnExecute?**

A predictor can be seen as an array with severe predictors. This array needs to have a static length so it can be implemented in hardware. It will be impossible to have one slot in the array mapped to every single possible PC. The solution to this is to have a slot in the prediction array mapped to a range of PCs. The trade off here is how many indexes the array should have. A long array (with many indexes) will be slow and expensive in terms of power, place and materials. On the other hand fewer indexes means that one index will match to a bigger range of PCs, and if you end up with a two or more stores in the same PC range which behave differently, their prediction will interfere and end up with some kind of mean between these stores behaviors. To get to know which size that will be best different bit length was implemented. A bit length of two means the there are four indexes were you keep track of the PCs (in binary form) that begins with 00, 01, 10 and 11. The implemented bit sizes are: 1, 2, 4, 10, 16.

4.4.1 Predictor Implementation

When it comes to the implementation of a predictor for a given PC range. There are some approaches to take in to account, how to update a predictor and how to retrieve the prediction. In both cases two different versions have been implemented and combined into four versions.

Update prediction A predictor which has an integer array with three integers which can take values between zero and two, the higher value the higher prediction for the alternative with that index. Upon updating the prediction the value for that prediction will be incremented by one and the other two decremented by one. That was the original version of the prediction update. The other one excepts prediction values (the values in the array) up to seven and will increase the counter for the index by two instead (or one if it happens to be six already). This will keep the counters more stable since the total increment and decrement will be the same (if possible).

Get prediction The getPrediction function also has two versions, one that simply returns the counter of their prediction which can be in the range between zero and two or zero and seven depending on the implementation of update prediction in 4.4.1. In the other case the return will be binary 0 if predicted to be false and 1 if predicted to be true. For the 0-2 range a value greater or equal to one will return one and for zero to seven the value needs to be greater or equal to three. Having a prediction with 0-3 range makes it more adaptive to changes it has seen i.e. many late misses and are entering a section with hits, all the time. If you have a larger range the prediction can be more confident and needs a greater number of miss predictions before chasing the prediction.

4.4.2 Implemented versions

Below follows a list over the implement versions of PCPredictor.

- PCPredictor=1
- PCPredictor=2
- PCPredictor=4=0
- PCPredictor=4=1
- PCPredictor=4=2
- PCPredictor=4=3
- PCPredictor=10
- PCPredictor=16=0
- PCPredictor=16=1
- PCPredictor=16=2
- PCPredictor=16=3

The number after the first equal sign is the buffer size i.e PCPredictor=2 has a buffer size for $2^2 = 4$ entries. The second number describes the implementation as in table below, the one missing the last number are all acting as if it was a zero.

Last digit	prediction range (Update prediction 4.4.1)	return range (Get prediction 4.4.1)
0	0-3	0-3
1	0-7	0-1
2	0-7	0-7
3	0-3	0-1

Table 4.1: Naming convention for PCPredictor

4.5 ReExececution

This is not a fully prefetch policy, it is more of a question on how to handle ReExecution of instruction. Here we compare between issue a prefetch on ReExececution or not. If the ReExecution happens close in time to the first execution then the data for that store is likely to be in L0 and/or in another cache hence prefetching again will not be necessary and thereby just a waste of energy. Reasons for reExecution can either be that a load follows a store with unknown address that is later computed to the same as the store. In this case the load has to be reExecuted to get the data that is updated by the previous store. Another reason can be that the load in question has a previously not resolved load. If the load is invalidated or evicted from the cache, then the load (and all the next instructions) have to be ReExecuted.

5 Results

Below you find the results in graphs the results are based on simulated executions of 55 benchmarks. All graphs shows percentages were NoPreftch are set to 100 % as a reference. The number in the heading of every graph is the difference between the highest and the lowest value in the table.

5.1 State-of-the-art of prefetch policies

In this section we compare the for basic prefetch policies: NoPrefetch, OnExecute, OnCommit and OnNonBSpecultive.

The number of cycles i.e. the speedup. A lower percentages gives more speed up. One can see that the difference between NoPrefetch and the other three is the biggest.

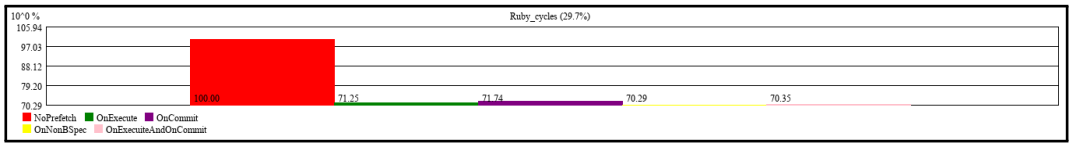

Figure 5.1: Run times for the four given policies

We have OnCommit at 71.74 % of the number of cycles used by NoPrefetch, followed by 71.25 % for OnExecute, and finally the fastest OnNonBSpeculative with 70.29 %.

Power consumption, a higher percentages means higher power consumption.

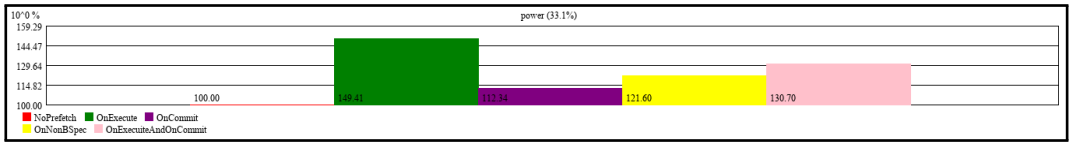

Figure 5.2: Power consumption for the four given policies

OnExecute burns most energy 149.41 %, followed by OnNonBSpecutative 121.60 % and OnCommit 112.34 %. NoPretetch has the smallest consumption (100%)

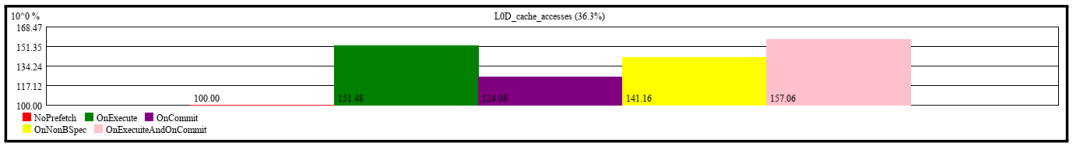

Figure 5.3: Number of L1 accesses for the four given policies

Figure 5.4: Number of useful prefetch for the four given policies

5.2 StoreBufferLoad

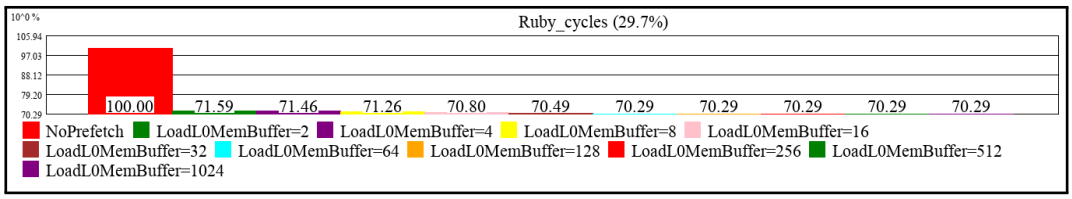

Figure 5.5: Run times for the different implementations of StoreBufferLoad

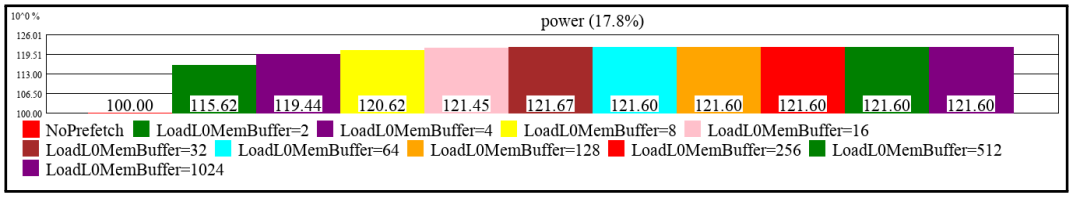

Figure 5.6: Power consumption for the different implementations of StoreBufferLoad

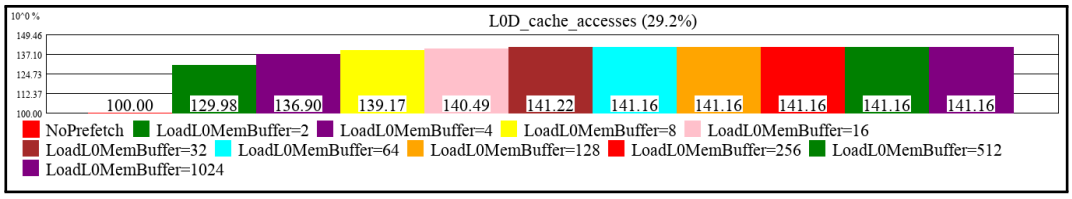

Figure 5.7: Number of L1 accesses for the different implementations of StoreBufferLoad

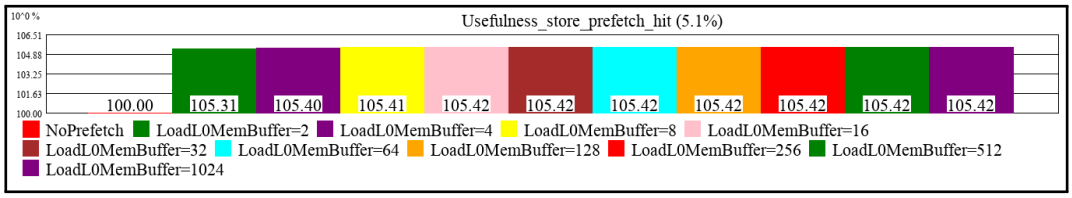

Figure 5.8: Number of useful prefetch for the different implementations of StoreBufferLoad

5.3 PCPredictor

5.3.1 Buffer size

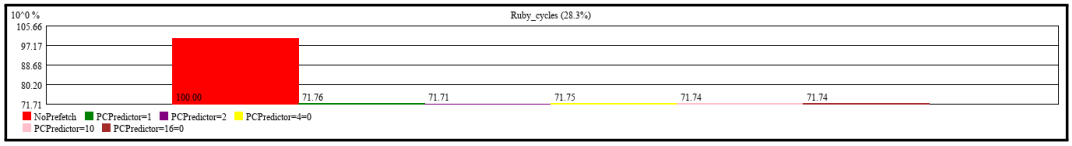

Figure 5.9: Run times for the different buffer sizes for PCPredictor

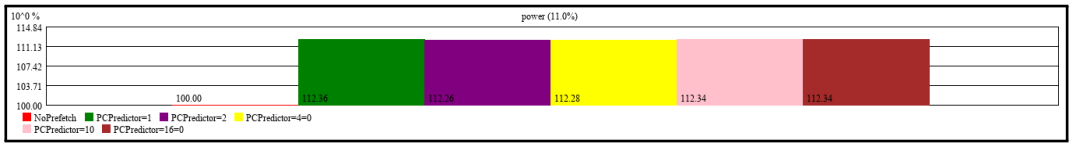

Figure 5.10: Power consumption for the different buffer sizes for PCPredictor

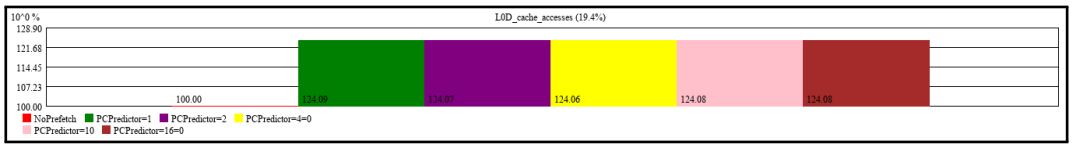

Figure 5.11: Number of L1 accesses for the different buffer sizes for PCPredictor

Figure 5.12: Number of useful prefetch for the different buffer sizes for PCPredictor

5.3.2 Implementation

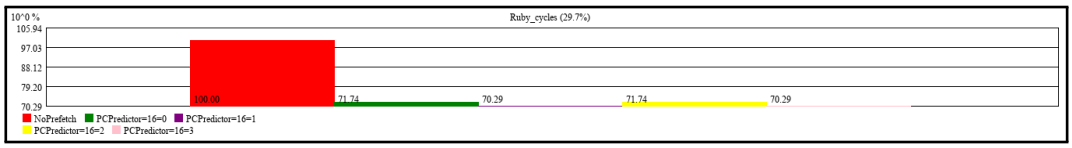

Figure 5.13: Run times for the different implementations for PCPredictor

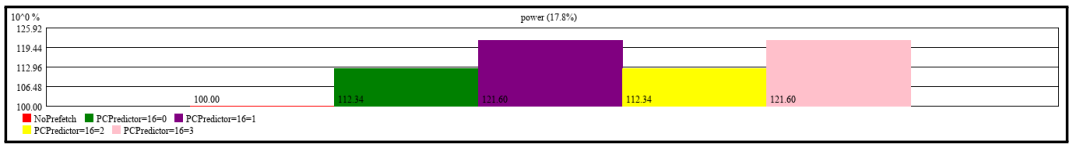

Figure 5.14: Power consumption for the different implementations for PCPredictor

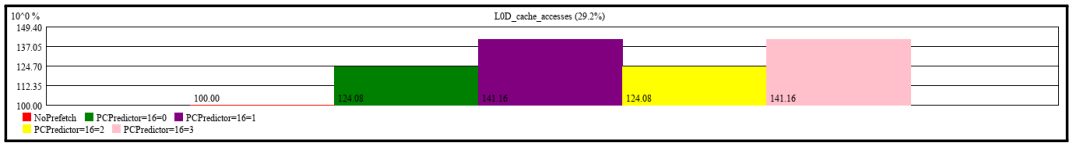

Figure 5.15: Number of L1 accesses for the different implementations for PCPredictor

Figure 5.16: Number of useful prefetch for the different implementations for PCPredictor

5.4 ReExcecution

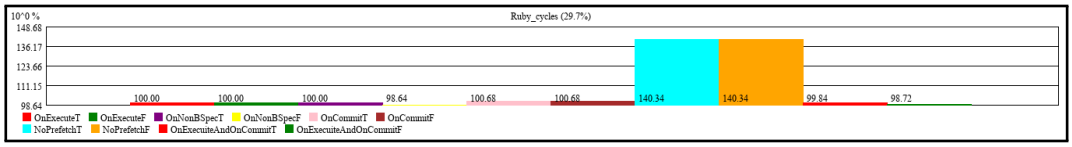

Figure 5.17: Run times for the four basic once with or without prefetching on reExecution

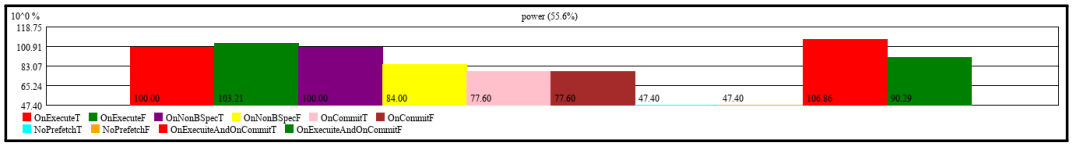

Figure 5.18: Power consumption for the four basic once with or without prefetching on reExecution

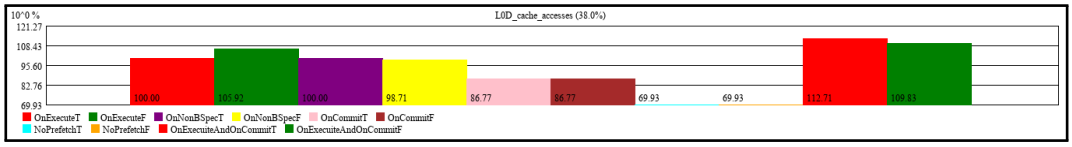

Figure 5.19: Number of L1 accesses for the four basic once with or without prefetching on reExecution

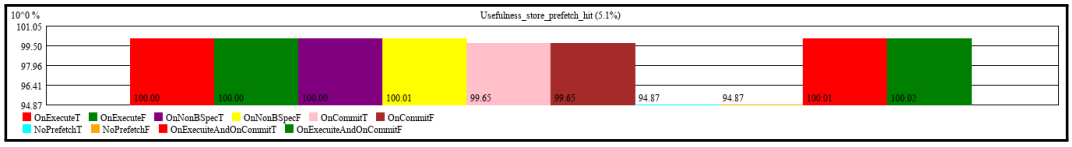

Figure 5.20: Number of useful prefetch for the four basic once with or without prefetching on reExecution

5.5 The best

6 Conclusions

Here you can just mention general conclusions. Like "this thesis explores different prefetch policies and we obtain this improvements in energy or time. And the important findings in the thesis. The particular conclusions about each policy can be discussed along with the evaluation.

6.1 State-of-the-art of prefetch policies

6.2 StoreBufferLoad

6.3 PCPredictor

6.3.1 Buffer size

6.3.2 Implementation

6.4 ReExcecution

6.5 The best

7 Discussion

mandatory??

Bibliography

[1] Cpi stacks. `http://snipersim.org/w/CPI_Stacks`. Updated: 2011-11-28 taken: 2018-01-31.

[2] The sniper multi-core simulator. `http://snipersim.org/w/The_Sniper_Multi-Core_Simulator`. Updated: 2017-08-07 taken: 2018-01-31.

[3] Spec cpu ®2006. `https://www.spec.org/cpu2006/`. Updated: 2018-01-09 taken: 2018-04-04.

[4] A. Banerjee, P. T. Wolkotte, R. D. Mullins, S. W. Moore, and G. J. M. Smit. An energy an performance exploration of network-on-chip architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(17):319–329, 2009.

[5] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *2008 International Symposium on Computer Architecture*, pages 115–126, June 2008.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News.*, 2011.

[7] L. Cheng and J. B. Carter. Extending cc-numa systems to support write update optimizations. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2008.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *20th*, pages 355–364, Aug. 1991.

[9] E. Herrero, J. González, and R. Canal. Distributed cooperative caching. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 134–143, New York, NY, USA, 2008. ACM.

[10] Intel. Intel 64 and ia-32 architectures optimization reference manual.

[11] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *ICCAD: International Conference on Computer-Aided Design*, pages 694–701, 2011.

[12] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras. Non-speculative load-load reordering in tso. *ISCA '17, June 24-28, 2017, Toronto, ON, Canada*, 7 2017. Not publicized jet.

Appendices

A

Implementation details

The algorithm describes the implementation of the policies in the execution stage.

Algorithm 1 When to prefetch according to the different policies

```
1: if not i.prefetch_issued ▷ Regarding of policy prefetch will not occur if it is already issued
2: and (m_processor_store_prefetch = ONEXECUTE ▷ Prefetch here for OnExecute
3: or (m_processor_store_prefetch = ONNONBSPEC and not unresolved_branch) ▷ Prefetch here for OnNonBSpecultive if an eventual branch is resolved
4: or (m_processor_store_prefetch = ONEXEXCUTEANDONCOMMIT and not unresolved_branch) ▷ Prefetch here for OnExecuteAndOnCommit if an eventual branch is resolved
5: or (m_processor_store_prefetch = PCPREDICTOR and getPrediction(instructionPC)= 1 and not unresolved_branch) ▷ Prefetch here for PCPredictor if the prediction is one and an eventual branch is resolved.
6: or (m_processor_store_prefetch = LOADL0MEMBUFFER and (m_store_buffer_ptr.numUsedEntries()+previous_loads_performed_count) <BufferLoadLimit and not unresolved_branch)) ▷ Prefetch here for LOADL0MEMBUFFER if the buffer load are less then the limit and if an eventual branch is resolved
7: then
8:     executeMemoryInstruction(i, PrefetchBitYes)
9: end if
```