Barcelona School of Informatics (FIB)
Master in Information Technology

Final Master Project Report

Memory Dependence Prediction Methods Study and Improvement Proposals

Otto Fernando Pflücker López

March 2011

Tutor: Gladys Miriam Utrera Iglesias
Supervisor: Adrián Cristal

Table of Contents

Index of Illustrations

Index of Tables

Abstract

Nowadays, most modern high performance processors employ out-of-order (O3) execution. In these processors, instructions are executed as soon as possible increasing in this way the instruction level parallelism (ILP) and, in consequence, the processor performance. However, not all instructions could be executed in O3 way. Memory access instructions sharing the same memory address must be executed in order to keep the original program semantic. For this reason, O3 processors use memory dependence predictors. These are specialized units in charge of reducing, as much as possible, the number of loads and stores executed in-order. Good predictors aid to release all the ILP potential in O3 processors.

This project studies current used (in commercial hardware) and proposed (in academic papers) methods for predicting memory dependencies in an O3 processor. New opportunities to exploit instructions locality and improve predictor's accuracy are proposed and tested. In particular, the concept of extreme locality is introduced and applied in a new method, named MiniCAM. The results using this method are presented and discussed.

Keywords: Memory dependence predictor, out-of-order processors, memory disambiguation, memory dependence speculation.

Paraules clau: predictor de dependències de memòria, processadors fora d'ordre, desambiguació de memòria, l'especulació de dependència de memòria.

Chapter 1. Work Presentation

1.1 Introduction

Even now, when many new computers execute with out-of-order (O3) processors, the programs are generated to be executed in-order. In O3 processors, loads and stores instructions accessing to the same memory addresses must be executed in-order to keep intact the semantic of the program. Force all loads and stores to be executed in-order produces and important reduction in the instruction-level parallelism (ILP) and, in consequence, in the final performance. *Memory dependence prediction* is a technique that reduces the amount of in-order executions of loads and stores, predicting which loads depend on which stores. Only those instructions are executed in-order. When a prediction fails, a recovery mechanism is launched.

This master project studies current methods for predicting memory dependencies in O3 processors and propose new ones. The study includes implementation and testing of some of these methods over an Alpha 21264 processor using M5, a software simulator widely used in computer architecture research. The study covers:

- A summary of the work in memory dependence prediction field and other related techniques.
- Proposing of new methods.
- Testing of previous and proposed methods and results analysis.

This chapter presents the memory dependence problem, the objectives of this project, the time scheduling and a summary of this document organization.

1.2 Problem Description

In an out-of-order (O3) processors, instructions are executed as soon as possible increasing in this way the instruction level parallelism (ILP) and, in consequence, the processor performance. But dependencies between instructions exist and must be kept to ensure original program semantic. In other words, if an instruction A depends on the data produced by other instruction B, then A must be executed after B. Because data is stored in memory, two dependencies occur: Register dependency and memory access dependency.

Register dependency could be detected efficiently (without losing CPU cycles) with the simple analysis of instructions' registers use. The following instruction sequence is an example:

1: add $1, $2, $3

2: add $4, $5, $1

In this example, instruction 2 needs the result stored in register $1 by instruction 1. This dependency determines that instruction 1 must be executed before instruction 2.

Memory access dependencies could not be detected statically as in the previous case, as could be seen in the following sequence:

1: store ($1), ($2)

2: load ($3), ($4)

In this example, we need to know previously the stored values in registers 1 and 3 to decide if both instructions access to the same memory location and, in consequence, there is a dependency between them. Here the problem arises when register 3 is computed before register 1, because the O3 processor behavior. In this case the processor can (a) delay instruction 2 execution until register 1 is resolved or (b) make a prediction and launch instruction 2 as soon as possible. If later the processor detects there was a dependency, it have to roll back and re-execute instruction 2 (and maybe other later instructions). Because missing a prediction uses to be expensive, this alternative only has advantages if the predictor hits most of the time.

The previous instruction sequence is a case of read-after-write dependency (RaW). A RaW dependency is a dependency between a load instruction and a previous store instruction (in program order) that access to the same memory location. Other memory access dependencies cases exist but are resolved efficiently by using other methods. This is the reason why RaW dependencies are also named real dependencies.

Good memory dependency predictors aid to release all the potential ILP in O3 processors. Without prediction, many loads will be blocked until all possible dependencies were resolved, with an appreciable reduction of the ILP and final performance.

This project studies current used (implemented in real commercial hardware) and proposed (in academic papers) methods for predicting memory dependencies between load and store instructions executed in an out-of-order processor, and tries to find new opportunities to exploit instructions locality to improve the prediction accuracy.

1.3 Problem Objectives

The main goals of the project are:

- Build a summary of the state-of-the-art in memory dependence prediction. The studied methods will be compared to each other, remarking its advantages and inherent limitations. Following the current trend in computer architecture, scalability will be the main metric in the comparison (in terms of the number of concurrent instructions and the number of cores used for program execution).
- Characterize a base-system with a standard store-sets implementation memory dependence prediction unit, emphasizing on potential opportunities to exploit instructions locality and improving prediction accuracy.
- Propose and test modifications in the store-sets method.
- Propose and test a new introduced method: MiniCAM. The results will be compared to the standard store-sets implementation. Low resources consumption and low implementation complexity are important goals in this proposal.

1.4 Time scheduling

The following table summarizes the distribution of the time used to develop the different activities in this project.

Table 1 Project time scheduling.

#	Activity	Time (hours)
1	Memory dependence prediction methods researching.	80
2	Characterize the base-system about memory dependence behavior.	240
3	Perfect predictor implementation and testing.	80
4	Load-load dependency testing.	20
5	Store-sets original algorithm improvement implementation and testing	160
6	New proposed methods implementation and testing.	80
7	Results gathering, comparison and final report.	90
Total		750

1.5 Document organization

This document is organized in the following way:

- Chapter 2 makes and overview of the research in the memory dependence field in the last years. Methods used in current processors are also described.
- Chapter 3 details the platform (software and hardware) used to perform the experiments.
- Chapter 4 presents a set of evaluations over store-sets implementation that comes with M5 simulator. This evaluation was important in two aspects: (1) deeply understand the algorithm behavior and figure out opportunities to improve it, and (2) obtain a reference base to compare with the results in the next chapter.
- Chapter 5 describes the experiments performed and their results.
- Chapter 6 states some conclusions and proposes future work in the field.

Chapter 2. Background

2.1 Introduction

In O3 processors, *memory dependence prediction* is a kind of *load prediction*, a more general concept. All load prediction methods target the same objective: improve the system performance by reducing the number of loads and store instructions forced to be executed in-order.

This chapter describes load prediction methods with special focus in memory dependence predictors because the final proposed method, MiniCAM, falls in this category. The sections are organized in the following way. Section 2.2 describes the criteria used to compare the load prediction methods presented. Section 2.3 describes the methods classification used by different authors. Section 2.4 presents two methods used to recovery the system after a miss-prediction. Section 2.5 describes memory dependence prediction methods. Section 2.6 describes other types of *load prediction*. Section 2.7 presents load prediction methods used in commercial architectures. Section 2.8 makes a comparison between the studied methods using the criteria described in 2.2.

2.2 Comparison criteria

To evaluate advantages and disadvantages of the presented methods, we evaluate them with two criteria: Scalability over large instruction windows, and scalability under parallel processing.

Methods with good scalability over larger instruction window use to: (1) store few data per tracked instruction, (2) use low complex algorithms, (3) require data structures that can be stored using low power consumption hardware. When we found these characteristics in a method, we can expect good scalability.

Methods with good scalability under parallel processing use to be not tightly coupled to the fetch and execution phases in the processor pipeline. These methods do not require global knowledge of in-flight stream of stores to synchronize them with load instructions. In consequence, less data must be communicated between cores, and we can expect good scalability.

Most authors describe the scalability of these methods when the instruction window grows, but few of them give results under parallel processing. However, in some cases it is possible to infer them. For example, if a method does not use *fetch phase information*, like *fetch stamps*, it does not need a centralized fetch stream. This method can avoid tracking precise information about store instructions life cycle. In consequence, it is easy to implement this method in architectures with parallel processing with potential good scalability when the number of cores is increased.

2.3 Load prediction methods classification

In this section we describe the classifications found in the literature for *load prediction* methods. This is important because it give a general idea about the research field. This is also useful for putting in context the *memory dependence prediction methods*.

For *load prediction*, two classifications were found. The first one was presented by Chrysos [02]. He uses the term *load speculation* instead of *load prediction*. His classification is:

- **Value speculation**, where the value stored or read is speculated.
- **Dependence speculation**, where load-store dependence is speculated. This category was divided in *synchronized* and *unsynchronized* methods.

The second classification was presented by Calder [01]. He classifies *load speculation* techniques in:

- **Dependence predictors**, where the existence of store-load dependences is predicted. Examples of this category are the methods *load-wait table* and *store-sets.*
- **Address predictors**, where the accessed memory address is predicted. An example of this category is the

stride predictor. This kind of prediction can be used for reducing latency of load instructions via pre-fetching.

- **Value predictor**, where the value stored or read is speculated.
- **Memory renaming**, a special case of *value predictor*.

Other two authors present classifications with focus in *memory dependence prediction*. The first one was proposed by Onder et al. [11]. Their classification is:

- **Independence based**: Prediction of load dependencies is performed without tracking store instructions. These methods are simpler to implement, but produce less precise results because they tend to be very conservative. They are suitable to be implemented for parallel processing because no communication between individual stores and loads are required.
- **Pairing based**: Methods in this category try to find dependencies between individual pairs of load and store instructions. They are more complex and more precise than the *independence* approach, but they need more communication between individual stores and loads. Therefore, they are less suitable for being implemented in parallel processing architectures.
- **Set based**: Methods in this category try to find dependencies between loads and *set of stores*. They have a complexity in the middle between *independence based* methods and *pairing based* methods.

The second classification was presented by Moshovos [09]. His classification was based on the types of *memory dependence locality*:

- **Status locality based**: When a load-store dependency is found, subsequent instances of the same load will likely experience a similar dependency. These methods do not make statements about *which particular stores* a load may or may not be dependent on.
- **Set locality based**: When a load is dependent on a set of stores, future instances of that load will likely be dependent on the same set of stores.

2.4 Miss-prediction recovery

Because any speculation technique has effectiveness less than 100%, when a miss-prediction happens, the system must launch a recovery process. Two main variants of miss-prediction recovery are widely used [01]:

- Squash: All instructions issued after the miss-predicted one must be re-issued.
- Re-execution: Only the instructions issued after the miss-predicted one that depend on this are re-issued. This is a more precise and less costly technique.

2.5 Memory dependence predictors

During the literature research phase of this project, we found documentation about the following methods. For each method we will describe: the key concepts, the main implementation characteristics, and the scalability behavior. We include a comparison of these methods at the end of this section.

2.5.1 Store-load pair dependence predictor (1997)

Proposed by Andreas Moshovos in [10], this method uses two fully associative tables to predict dependencies in load-store pairs. The first table, a *memory dependence prediction table* (MDPT), identifies pairs that must be synchronized. When a violation occurs, a new entry in MDPT is allocated, storing the pair PCs and its *dependence distance* (the distance, in cycles, between the fetch-times of the instructions in the pair). When a MDPT matching entry is found for an instruction, this instruction must be synchronized; otherwise it can be executed immediately. In the former case, an entry in the second table, the *memory dependence synchronization table* (MDST), is allocated or found. MDST contains *condition variables* used for synchronization between pairs. When synchronization is

needed, store instructions signal their conditional variables on fetching and un-signal them on issuing. Load instructions check their conditional variables state on fetching. In this way, loads wait their predicted dependant store before execution. Entries in MDPT are never released and some special politics are used to release entries in MDST on data and control miss-predictions. Some ideas are suggested for an implementation in distribution architectures, but not tested.

This proposal arguing that wider instruction windows reduce significantly the net performance due to erroneous speculations. However, the suggested implementation has poor scalability because the use of two full associative tables.

2.5.2 Store barrier cache (1997)

It was proposed by James Henry Hesson in [06]. In this method, each store, that caused an ordering violation, increments a saturation counter in the barrier cache. At fetch time of a store, the barrier cache is queried and if the counter is set all following loads are delayed until the store is executed. If the store does not produce an ordering violation, the counter is decremented.

This method needs few resources and the implementation complexity is low. Therefore, good scalability is expected when instruction window grows. It is also expected good scalability under parallel processing, because few communication data is needed.

2.5.3 Store-sets predictor (1998)

It was proposed by George Z. Chrysos and Joel S. Emer in [02]. This is the most studied and referenced method until these days, and it is used for comparing other methods. A *store set* is a group of stores a load (or loads) has ever become dependent. A set is created when a violation occurs between a store and a load. The store becomes part of the new set and the load will be synchronized against it.

The method uses two tables to keep the information about the active sets: a *store set identification table* (SSIT) used to identify the active sets, and the *last fetched store table* (LFST) used to store the ID of the last store introduced in the set.

Because many loads could produce violations against the same store, and one load could have dependencies with many stores, a merge mechanism is defined between store-sets. In this way, any store could be in at most one store set and any load could be synchronized against at most one store set. When a store is fetched, it is introduced in its related store set, if one is found. When a store is issued, it is retired from its related store set. When a load is ready to be executed, if it has a related store set and at least one store is inside, the load is done dependent of the last introduced store in the store set. Additionally, all stores in the same set are forced to execute in-order for semantic correctness.

The described merge mechanism allows a very simple and fast implementation using (apparently) only two direct-mapped tables. With very few data per table entry and direct-mapped tables, one could expect good instruction window size scalability, but this is not the case. As described by Onder et als. [10], increasing the window size produce significant performance loss because, with more stores on fly at the same time, the average store set size tends to grow and more stores instructions are forced to execute in order. If stores execute in order, dependent loads also will execute in order producing more performance loss. Additionally, with bigger store-sets, more false dependencies will be produced between a load and stores in its related store set. Finally, implementation of store-sets requires not only two direct-mapped tables, but also the addition of one associative table attached to the load-store queues to track the dependencies and to allow loads wakeup by the dependent stores [16]. This reduces more the scalability of this method.

2.5.4 Inclusive and exclusive collision predictor (1999)

It was proposed by Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan in [20]. The central idea is to predict that a load instruction will produce a collision (memory dependence violation) with any of the stores in the execution window. The exact load-store collision pair is not predicted.

The *inclusive collision predictor* is a binary predictor. If a load is predicted to conflict with a store in its instruction window, it is delayed until all current stores have been executed. If not, it is advanced. Because all stores must be waited, this method is named *inclusive*.

The *exclusive collision predictor* enhances the inclusive collision predictor. It works annotating the minimal distance between *fetched stamps* of a load and the stores against it collide. In this way, the set of stores in the instruction window could be divided in two, one set further of the minimal distance, and one set within the minimal distance. Then, the load must wait only the stores in the second set.

The proposed implementation works with a *collision history table* (CHT). Many alternative policies for control CHT are presented; some use full associative tables, some not. About parallel processing, the use of *fetch stamps* for instructions age calculation and the use of store completion tracking for launching delayed loads introduce serious difficulties for using these methods in such cases.

2.5.5 Enhanced store-sets predictor (1999)

It was proposed by Soner Onder and Rajiv Gupta in [12]. The method focus on increase out-of-order issuing of store instructions in the original store-sets method, avoiding the necessity of force in-order execution of all stores in the same set. For this purpose, it determines correctness of speculated instructions by matching actual data values, not actual accessed addresses.

The method introduce the following modifications to the original store set algorithm [02]: (1) memory order violations are detected in the retire phase (not in the issue phase), and must be done considering the value recovered not the address acceded, (2) no order is imposed to store instructions that belongs to the same store set, (3) a mechanism for *fixing* the *forwarded value* to a load is described, because not imposing any order to stores could lead to forward incorrect values from a store to a load.

These modifications do not produce reduce the memory dependence order violations, but increase the final IPC (instructions per cycle).

This method has better window size scalability than the original. However, it has the same problems when applied to parallel processing.

2.5.6 Color sets (2002)

It was proposed by Soner Onder and Rajiv Gupta in [11]. The key in this approach is to use multiple speculation levels within the processor, termed as *speculation colors*, or simply *colors*. Each color in the spectrum represents increasing levels of aggressiveness in load speculation. These colors divide load instructions into distinct sets. Initially all loads belong to the *base color*, the set of loads which have never collided with unready store instructions in the past. The method also assigns a color to the processor state, based on the memory port utilization and the presence of store instructions in the instruction window. These store instructions must have collided with speculative load instructions in the past. A load is allowed to issue only if its color is less than or equal to the current processor color. If later a speculated load produces a collision with a store, the load's color is increased.

The method was proposed as having a performance close to *store-sets* but with a hardware cost close to *store barrier cache*. In consequence, its implementation only requires a direct mapped table with 2 bits per-entry. These bits are used to represent the four possible colors (four speculation levels). An instruction's color is recovered or modified accessing to the direct mapped table using the instruction's PC. Additionally, minor instruction window modifications are needed. This leads to good window size scalability. This method has also good potential for being implemented for parallel processing because no communication is needed between individual load and store instructions.

2.5.7 Store distance (2006)

It was proposed by Changpeng Fang, Steve Carr, Soner Onder, and Zhenlin Wang in [03]. It is a compiler/micro-architecture cooperative scheme. The compiler receives feedback from previous executions generating code with annotations used at runtime for improving memory dependence prediction. The method defines and uses the *store*

distance concept. The *store distance* is the number of stores between a load and the nearest previous store that access to the same memory location. The method profile a program with one small input and analyze the instruction-based store distance distribution generating a representative store distance for each static load instruction (a static instruction is one in a given PC). Then, a cost effective micro-architecture mechanism is developed for the processor to determine accurately on which specific store instruction a load depends according to its store distance annotation.

This method's implementation relies on the processor instruction decode mechanism, therefore no additional structures are needed, leading to a great implementation efficiency. However, it needs compiler modifications, programs recompilation, and micro-architecture changes, so great commitments are needed.

This method does not need additional data structures other than those mentioned. In consequence, even with no empirical results, we can expect good scalability when the instruction window grows. It is expected low potential to use this method under parallel processing because it uses store distance to speculate exact load-store pairs' dependency.

2.5.8 Store vector (2006)

It was proposed by Samantika Subramaniam in [16]. It was designed as a method that mimics store-sets behavior but replacing the use of full associative tables with direct-mapped ones, spending less space per table entry and using a faster operation algorithm. The objective is reducing the obstacles for industrial adoption of techniques like store-sets.

The key idea is replacing the use of full associative tables for keeping instruction's dependencies, with the use of a dependency matrix called *load scheduling matrix* (LSM). For N instructions we only need a LSM of N by N bits. A dependency between instruction *"i"* and instruction *"j"* in a LSM is represented with just one bit set in row *"i"* and column *"j"*. An instruction with non-resolved dependencies have at least one bit set in its corresponding row in LSM, and that check could be implemented efficiently using AND operations. When instruction *"j"* issues, it just clears all bits in column *"j"*, which could be implemented very efficiently with a single bit latch-clear signal. This method use only one bit per dependency because it tracks the relative age between loads and stores, not the stores' PC, like in store-sets. The relative age corresponds to the *fetch order* distance between memory access instructions, and it is represented in LSM as the horizontal distance between bits in a row. Then, the LSMs width will be equal to the capacity of the store queue.

To keep track of the dependency history of a load instruction, a second table is used, the *store vector table* (SVT). When a load is decoded, its PC is used to recover the corresponding entry in SVT and it is copied to the LSM. While any of the bits in the LSMs row is set, the corresponding load instruction will not be considered ready to issue. When a load-store order violation occurs, the corresponding SVT entry for the load instruction is updated setting the bit corresponding to the relative age distance between the load and the store. Periodically, all bits in SVT are reset to clear out predicted dependencies that may no longer exist.

This method presents a better prediction ratio than store-sets mainly because two important behavior differences: (i) The stores in a set can be issued in any order; (ii) a store can be a predicted input dependence for any number of stores, in other words, stores could belong to more than one set, where it is forbidden in store-sets implementation.

It is expected this method has good window size scalability, but no evidence is given by the author. Because no exact load-store pair dependency is tracking, good potential for parallel processing is expected, but again, not tested by the author.

2.5.9 Synchronizing store-sets (2006)

It was proposed by Stone SS, Woley KM, Malik K, Agarwal M, Dhar V, Frank MI in [15]. This is a special implementation of store-sets algorithm adapted for architectures with speculative parallelization. The parallelism occurs when the processor extracts multiple threads from a single sequential thread and executes them in parallel. This method allows using store-sets for parallel processing. There are not enough implementation details in the method description to infer the potential window size scalability.

2.5.10 Counting dependence predictors (2008)

It was proposed by Franziska Roesner, Doug Burger, and Stephen W. Keckler in [13]. This method was designed to be implemented in systems with parallel processing capabilities (multi-core or many-core), without a centralized fetch system and without global tracking of in-flight stores processing. The method predicts a *learned* number of arbitrary *stores events* before mark as *ready to issue* a load instruction predicted to be dependent on that *learned* number. No specific load-store dependency is predicted. Here, the definition of what store events consider depends on the chosen politic. For example, the store address calculation, store issue and store completion could be valid events.

The implementation presented uses a PC-indexed table, with 2 bits per entry. These bits are used to represent the *prediction state* of a load instruction. A *state machine* is used for updating loads' prediction state. These states go from aggressive behavior (where the load must not wait any store address calculation to be issued) to conservative behavior (where the load must wait all older stores to be completed before issue). All loads are initialized in aggressive state, and the state is updated according to subsequent load behavior.

About window size scalability, good results are presented by the author as expected for a method with no associative table use and little information kept for each speculated load.

2.6 Other load dependence methods

Memory dependency prediction is used to speculate about the right moment to issue a load instruction without complete knowledge about previous not yet issued store instructions. But other methods have been proposed for load speculation: *value* and *address* speculation methods. The following sections describe some of them.

2.6.1 Last value predictor (1996)

Methods in this category (proposed by Mikko H. Lipasti, Christopher B. Wilkerson and John Paul Shen in [07]) preserve the last value of a particular load instruction and speculate that it will re-use the same value during the next execution. It is interesting to note that the same idea could be used to predict the accessed address.

These methods use a direct mapped, tagged table with 4K entries. Each entry contains the tag and the predicted value. Because not all load instructions behaves in the same way, the author classifies them in three groups based on their dynamic behavior: (1) loads with unpredictable values, (2) loads always predictable, and (3) loads almost always predictable. For this purpose, a separate *load classification table* is used. Finally, for always predictable loads, a *constant verification unit* is used to keep the last used value and avoid accessing the conventional memory system completely.

2.6.2 Memory renaming (1997)

It was proposed by Gary S. Tyson and Todd M. Austin in [19]. The key idea is avoiding memory access by a load instruction. This method keeps track of store/load dependencies in order to directly communicate predicted values from stores to loads, bypassing memory.

An important difference between this method and store-sets is that memory renaming keeps track of dependencies over a larger window of instructions. The *store cache* used in this method can include information of store instructions no longer in the current window instruction.

It is interesting to note that this idea could be easily combined with other kinds of methods that predict dependences between loads and stores.

2.6.3 Stride predictor (1997)

It was proposed by José Gonzalez and Antonio Gonzalez in [05]. This predictor keeps, for each load instruction, the last address accessed and the difference between this address and the previous one. This difference is called *stride*. The predictor speculates that the new accessed address will be the sum of the previous address and the stride.

This method uses a direct mapped table. Each table entry contains a tag, the predicted value and the predicted stride.

2.6.4 Context predictors (2000)

It was proposed by Brad Calder in [01]. In general, a context predictor bases its prediction on the last several values seen. This could be seen as a generalization of *last value* and *stride* methods. Calder et al. propose a context predictor that looks at the last 4 values (addresses) seen by a load.

The implementation uses two direct mapped tables: (1) a *value history table* that contains the last four values per load, and (2) a *value pattern table* that contains actual values to be predicted.

2.7 Predictors used in manufactured architectures

2.7.1 The Alpha load-wait table

The Alpha 21264 uses a *load wait table* to predict memory dependence of loads instructions [08]. Each table entry only has one bit. When a violation occurs, the bit in the corresponding load entry (mapped using the load PC) is set. If the bit of a load is set, its execution must be delayed until all prior stores have issued. To avoid miss predictions due to forever set bits, every amount of ticks, all the bits in the *load wait table* are unset.

Table 2 Load prediction methods.

Method name	**Kind of prediction**	**Has good scalability when grows? (YES, NO)**	
		Inst. window size	**Parallel processing**
Store-load pair dependence predictor	Dependence	NO	NO
Store-barrier cache	Dependence	YES	YES
Store-sets predictor	Dependence	NO	NO
Inclusive and exclusive collision predictors	Dependence	YES	NO
Enhanced store set predictor	Dependence	NO	NO
Color sets	Dependence	YES	YES (*)
Store distance	Dependence	YES	NO (*)
Store vector	Dependence	YES	NO (*)
Synchronized store-sets	Dependence	NO	YES
Counting dependence predictor	Dependence	YES	YES
Last value predictor	Value	YES (*)	(-)
Memory renaming	Value	YES (*)	NO (*)
Stride predictor	Address	YES (*)	(-)
Context predictors	Value	YES (*)	(-)
Load-wait table	Dependence	YES	NO
Distributed load-wait table	Dependence	YES	YES
P6 processor ordering model	Dependence	(-)	(-)

(*) Not enough empirical evidence, but could be inferred. (-) Not focus by author and could not be inferred.

This method has great window size scalability because a direct-mapped table is used (not associative one) with only one bit per table entry. A distributed version is described later.

2.7.2 The P6 processor pessimistic predictor

The Intel P6 processors family uses a memory order model defined as *processor ordering* to maintain consistency in the order data is read and written in a program and the order the processor carries out the reads and the writes [21]. Reads can pass buffered writes, but the processor guaranties program correctness if one *passed write* targets to the same memory location. However, if the last case happens, all instructions that depend on the speculated read data are blocked until the prediction is confirmed. This is considered a *pessimistic policy* predictor.

2.8 Summary

Table 2 summarizes described methods. Each method is followed by its classification, and its scalability properties for large instruction windows and parallel processing enable architectures.

As shown in the comparison above, scalability for parallel processing is clearly less supported than instruction window size scalability. The research reported in this document focuses only in the last one.

Chapter 3. Evaluation framework

In this chapter we will describe the hardware and software used to perform the experiments. The main software tool is the simulator M5. The configuration used for this simulator is also detailed.

3.1 The M5 simulator

This simulator is widely used in computer architecture research. It is a modular platform, encompassing system-level architecture as well as processor micro-architecture [22]. Its key features are:

- Pervasive object orientation: Major simulated structures (like CPUs, buses, caches, etc.) are represented as C++ objects. The system could be deeply configured using python scripts. This means that many systems could by represented and tested without coding in C++. It also improves the reutilization of simulated structures.
- Multiple interchangeable CPU models: simple, functional and a one-CPI out-of-order SMT-capable CPU.
- Event-driven memory system: It makes easier understand, modify, and trace changes in memory at execution time. This system includes non-blocking caches and split-transaction busses.
- Multiple ISA support: M5 decouples ISA semantics from its timing CPU models, enabling effective support of multiple ISAs. M5 currently supports the Alpha, SPARC, MIPS, and ARM ISAs, with x86 support in progress.
- Two simulation modes: System-call emulation and full-system. The first simulates the execution of a statically compiled binary by functionally emulating any system call it makes. It is useful for executing individual tests in isolation. The second simulates a full system, including a kernel, I/O devices, etc., and it allows executing a complete operating system. Some full system supported architectures are Alpha, ARM, and SPARC.
- Multiprocessor, multi-system capability: Thanks to M5's object orientation, instantiation of multiple CPU objects within a system is trivial. Combined with the snooping bus-based coherence protocol supported by the caches, M5 can model symmetric multiprocessor systems. Because a complete system is just a collection of objects (CPUs, caches, memory, etc.), multiple systems can be instantiated within a single simulation process. In conjunction with full-system modeling, this feature allows simulation of entire client-server networks.

The object orientation and the use of simulation modes were especially useful in this project.

3.2 Framework configuration

The tests have been performed using the M5 simulator running in a Dell server with an quad-core Intel Xeon Processor 5160, 4M cache, 3.00 GHz, 1333 MHz fsb, 16GB RAM, running Ubuntu in 64 bits. The CPU model configured with M5 corresponds to the Alpha 21264, a pipeline processor with 6 phases: (1) fetch, (2) decode and rename, (3) issue, (4) registers read, (5) execute, (6) and memory access (write-back) and instruction retire. The model details are described in Table 3.

The tests were running against SPEC CPU2000 benchmarks. Integer and floating point benchs were used. Because simulator M5 is in continuous developing and some Alpha processor characteristics are not yet supported, not all benchs could be run. Therefore, a subset from SPEC CPU2000 was chosen: *gzip*, *gcc*, *mesa*, *art*, *mcf*, *equake*, *crafty*, *ammp*, *parser*, *bzip2*, and *twolf*. All benchs were compiled using *alpha-unknown-linux-gnu* cross compiler, version 3.4.3, with optimization flag O2.

Table 3 CPU Model

Instruction queue capacity (instruction window size)	**128 instructions**
Fetch capacity	8 instructions per cycle
Dispatch capacity (renamed insts. pass to inst. queue)	8 instructions per cycle
Issue capacity	8 instructions per cycle
Instruction cache	64 KB
Data cache	64 KB
Cache block size	64 bytes
Load queue entries	32 instructions
Store queue entries	32 instructions
Simultaneous threads	1

Chapter 4. Base CPU model characterization

4.1 Introduction

Before making any modification, we studied the base CPU model behavior related to memory dependence prediction. It is important to remark that it is expected to found some differences between the Alpha CPU model implemented in M5 and the real machine. These differences affect the performance measures, but as all tests are executed in the same simulated machine, the results and conclusions derived are valid.

This section summarizes the tests performed to get this goal, and it is organized in the following way: section 4.2 describes the parameters used to evaluate the base system; section 4.3 focuses on the M5 implementation of the store-sets algorithm; sections 4.4 states some conclusions.

4.2 Evaluation parameters

A set of parameters were considered to expose potential locality characteristics of the base CPU model: program counter distance (PC-distance), sequential number distance (SN-distance), try to execute distance (TE-distance), and registers reuse degree. All parameters are gathered in *memory dependence violation events*, when the tested system detects a *real dependence violation* (read-after-write, or RaW).

In a RaW violation, the event occurs when the system tries to execute a *store* instruction and finds a younger[1] executing *load* which access to the same memory address. Therefore, this *load* is considered a *violator instruction*, its execution is stopped and its effects are discarded; the execution of the involved *store* instruction is not affected. Even though this is the usual implementation, a RaW violation, we will discuss later an alternative implementation that also considers load-load violations (Read after Read, or RaR) for the sake of produce artificial contention.

The base CPU model was instrumented for gathering these parameters:

- PC-distance: It corresponds to the difference between the program counter (PC) addresses of the *load* and *store* instructions pair that produces a RaW violation. A PC-distance value is positive if the *violator load* goes after the *store* in file-order[2], and negative otherwise. This parameter has *bytes* as measure units.
- SN-distance: It corresponds to the difference between the sequence number (SN) of the *load* and *store* instructions involved in a RaW violation. A sequential number is the sequential ordinal given to every instruction when fetched. This parameters is always positive, otherwise no violation could happen. A small *sn-distance* indicates a RaW violation between *near instructions* in program execution order. This parameter has *cycles* as measure units.
- TE-distance: It corresponds to the difference between the system *cycle-counter* of the *load* and *store* instructions, involved in a RaW violation, when they are *tried to be executed*. We define the term *try-execution cycle* as the cycle when the processor tries to execute and instruction, successfully or not. In this case, the difference is computed when the store instruction, involved in the violation, is issued. In this moment, the violation is detected and TE distance is computed as the difference between the current cycle and the cycle when the detected *violator load* was issued. The idea behind this parameter is try to find a system locality behavior not related to the *program execution order*, but to the *real system execution order*. This is a *time-distance parameter*, and depends on the system capacity to resolve all instructions memory dependencies, using exact detection or prediction. This parameter has *cycles* as measure units.
- Registers reuse: The main idea is detecting the reuse percentage degree of the registers in the store and load instructions involved in memory dependence violations. To avoid registers utilization affected directly by

[1] An instruction A is younger than other instruction B if B was fetched before A, following program-execution order.

[2] Considering a program file, file-order refers to the relative position between two instructions inside.

the compiler politics, the instrumentation used works with the physical registers, not the logical ones.

To simplify value reading, for any *XX* distance, *XX_distance =XX_load –XX_store*, where the load instruction is the *violator*, and the store instruction was issued when the violation was detected.

4.3 The M5 store-sets implementation: behavior study

The project first phase was dedicated to study the actual store set algorithm behavior implemented in M5. This algorithm was chosen because it is the *de facto* referenced algorithm. All studied memory dependence prediction algorithms are compared against this one [02].

Store-sets algorithm not only avoids most of the memory dependence violations produced in an O3 program execution, but also produce false violation detections (a *false positive*) creating artificial sequential execution and reducing ILP. To get a perspective of the real performance improvement introduced by this algorithm, we test the CPU model replacing the *store-sets predictor unit* with a *naïve predictor unit*, a predictor that always assumes no memory dependences exist.

Table 4 compares the store-sets algorithm performance, under different sizes for tables SSIT and LFST, with the naïve predictor. Values corresponds to *violations per million of executed instructions*. Around 99% of the naïve case violations are avoided using store-sets.

Table 4 Store-sets performance with different tables' sizes. The values corresponds to memory dependence violations per million of executed instructions.

Test	Gzip	vpr	gcc	Mcf	crafty	parser	eon	vortex	bzip2	mesa	art	equake	ammp
Naïve	6851.31	3376.63	3737.35	622.21	1788.61	889.74	6694.37	4107.23	17427.47	9952.73	12894.47	4312.38	9343.87
size=1	40.62	2.90	56.66	3.45	0.03	5.06	15.05	1.85	163.35	0.25	0.00	30.53	4.40
size=8	40.62	3.05	56.75	3.45	0.03	5.04	14.88	1.85	162.68	0.23	0.00	30.52	4.41
size=64	40.63	2.84	56.60	3.89	0.05	5.05	16.02	1.85	161.99	0.27	0.01	30.37	4.39
size=128	40.55	3.47	56.57	110.76	0.07	5.03	16.94	1.86	150.25	0.31	0.02	21.03	4.19
size=256	46.45	1.70	56.56	115.57	0.10	4.83	5.04	1.88	84.09	0.26	0.01	21.24	5.27
size=512	1.04	2.09	55.67	90.19	0.17	13.49	5.29	1.17	89.69	0.20	0.01	11.82	2.29
size=1024	1.26	4.73	54.24	87.28	0.33	13.91	4.73	1.19	50.10	0.07	0.01	11.85	2.32

As expected, values tend to decrease when tables' sizes tend to grow, but the tendency *is not a soft curve*. In most of cases there are not differences between sizes 1, 8, 64 and 128. This gives us a clue about how high locality is handled by store-sets algorithm. With just one entry, the store-sets tables can handle very well most of the memory dependencies. With an issue width of 8 instructions, it is clear that mostly no more than one *ambiguous memory dependency* coexist at the same time in the instruction window. Maybe store-sets is too expensive to handle most of the cases, and a simpler algorithm could be designed for the 90% of cases.

Even with little difference about dependence detection, a significant difference in overall performance is observed. Figure 1 shows the IPC of the different benchs using store-sets with different tables' sizes. This is because a positive detection is only part of the complete predictor result. Two more cases must be taken into account: wrong positive predictions and wrong negative predictions.

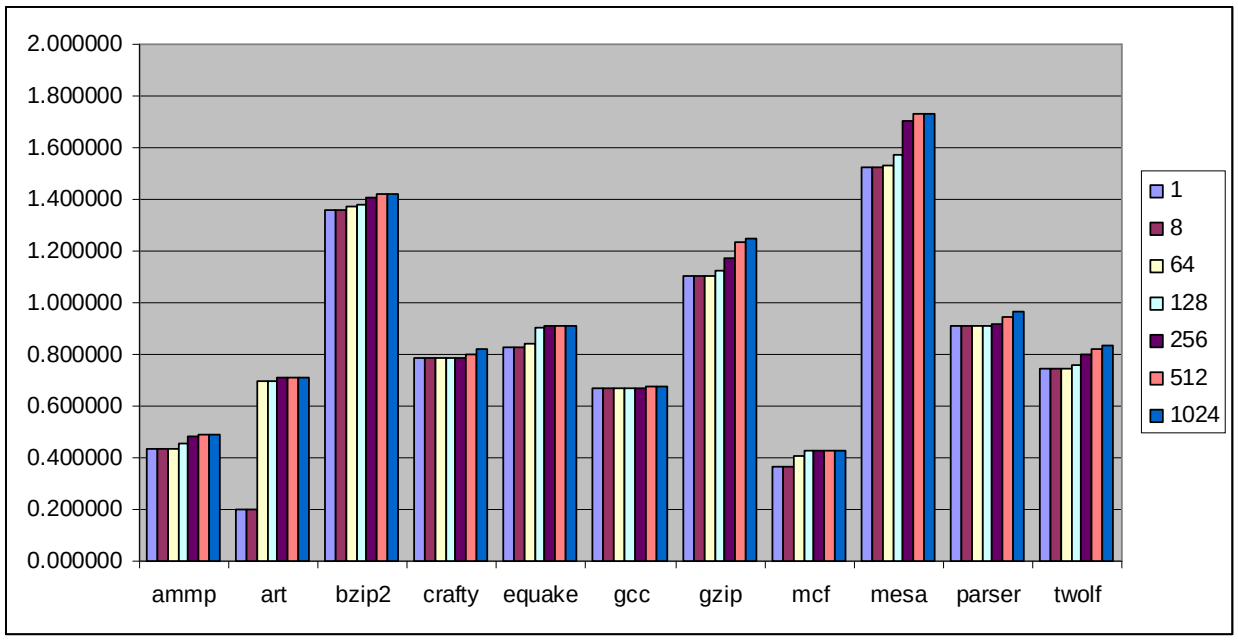

Figure 1 IPC using store-sets with different tables' sizes.

Wrong positive predictions (when the predictor decides a load instruction is safe to be issued but produce a violation later and must be re-scheduled and re-executed) force rollbacks in the system pipeline. Wrong negative predictions (when the predictor assume a load has dependence and later it is confirmed that it has not) create artificial contention. Figure 2 shows how fast the number of wrong predictions decreases when the store-sets internal tables grows. The values are normalized against the miss-predictions of the store-sets with 1024 entries in its internal tables.

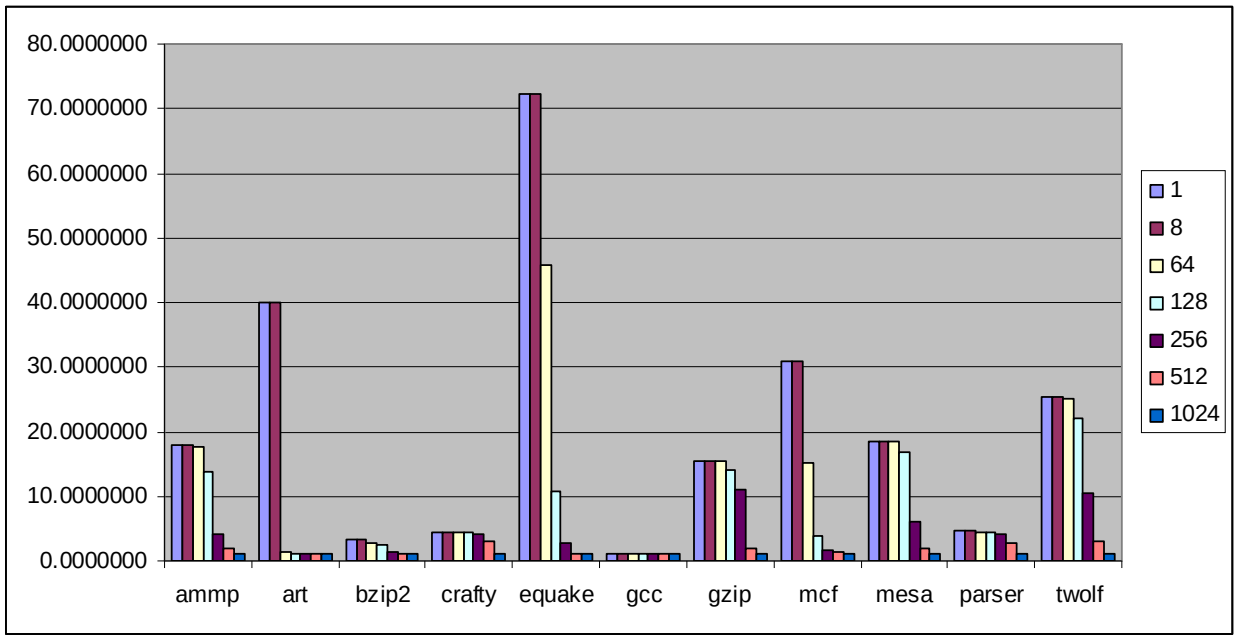

Figure 2 Normalized wrong predictions using store-sets with different tables' sizes.

In summary, avoiding wrong predictions is as important as predicting them correctly. This information was used to improve the original store-sets algorithm, as it will be described later in section 5.6.

4.3.1 PC-distance study

This parameter corresponds to a static perspective of the memory dependence violations, because it focuses on relative position, in program binary code, of instructions in a violation. The intuitive case, when the violator load instruction goes after the store instruction, corresponds to values at the right of 0 in the X axis in Figure 3 and Figure 4. Values at the left correspond to violations between instructions inside a loop or between function calls. In all cases, a strong locality is observed. This result could be used, for example, to understand the impact of increasing or decreasing the instruction window size in the issue phase of the processor pipeline. Figure 3 corresponds to the executions with a naïve predictor, and Figure 4 to executions with the store-sets predictor with tables (SSIT and LFST) with 1K entries.

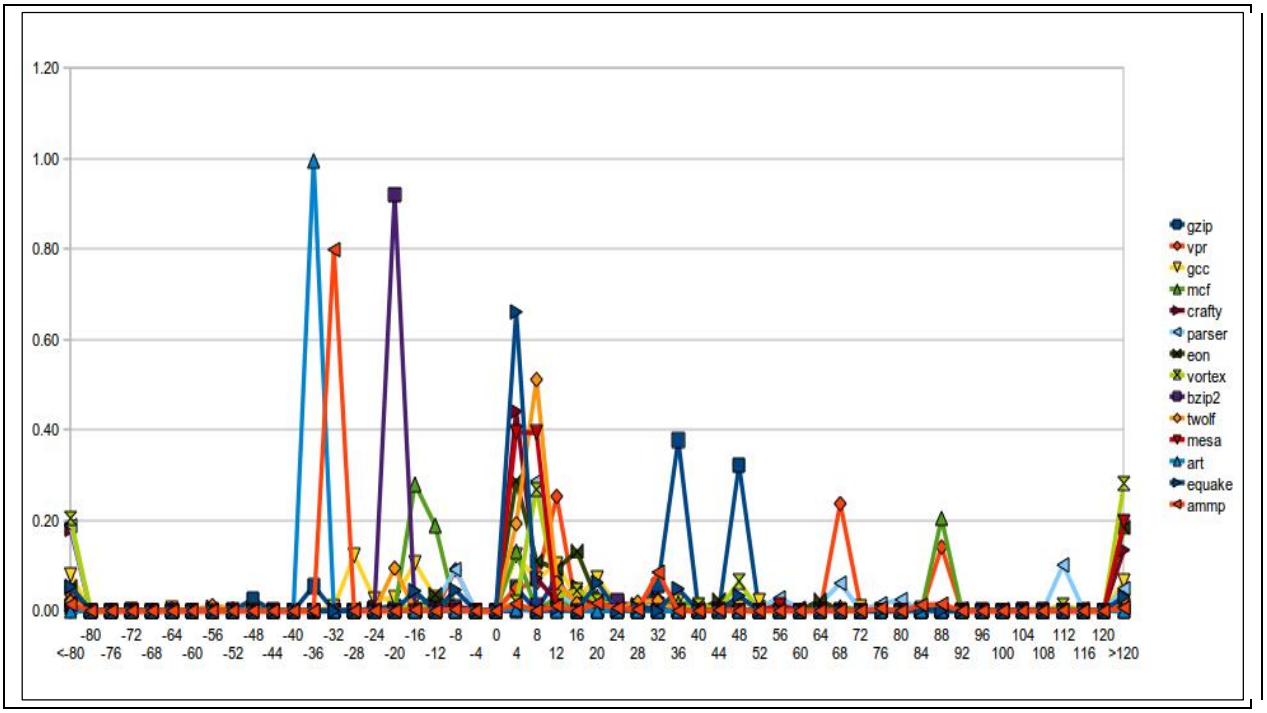

Figure 3 PC-distance for violations under naive predictor.

After eliminating most of the violations, in Figure 4 we observe that benchs have clear violation distribution patterns. For example, in bench *bzip2*, most of the violations are produced in a PC-distance of 92 bytes (23 instructions). Others benchs, as *parser*, have a more wide spread distribution. It is interesting to note that most not-cached violations still have very low PC-distance. We will return to this problem this problem later in this document.

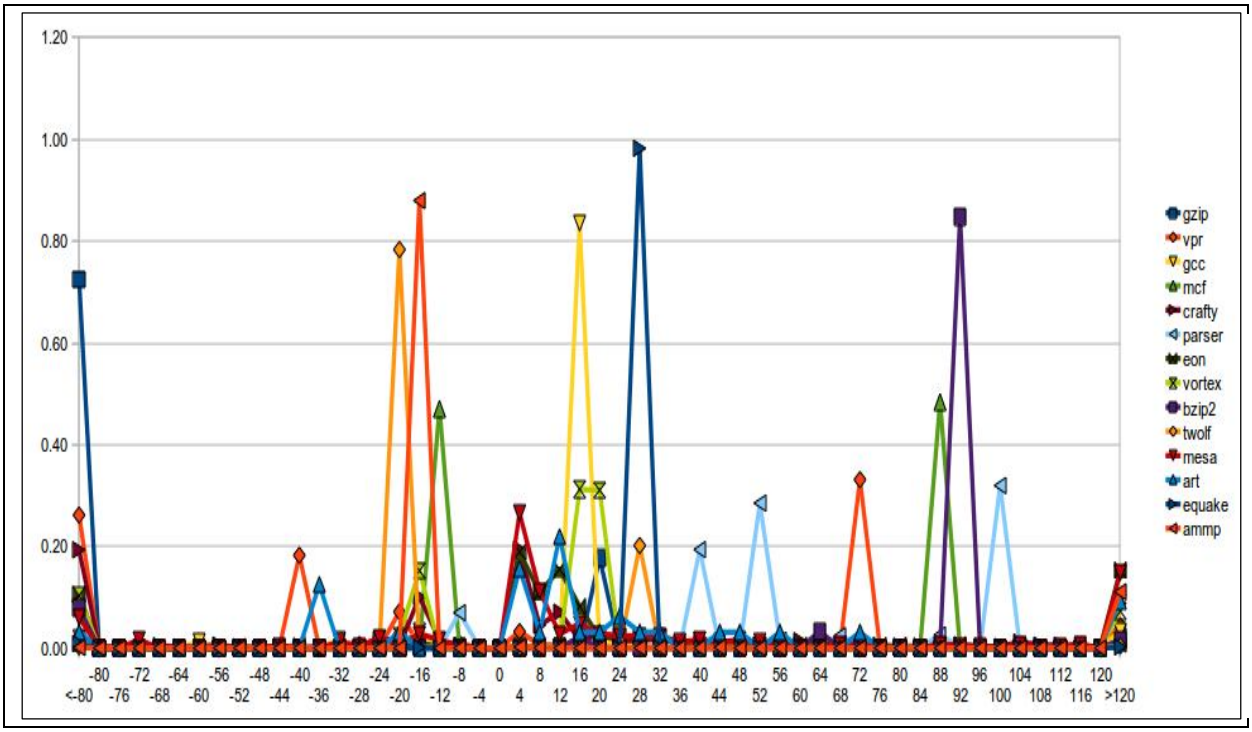

Figure 4 PC-distance for violations under store-sets predictor (1K table size).

4.3.2 SN-distance study

This parameter corresponds to a dynamic perspective of the memory dependence violations, because it focuses on the order instructions are fetched (the named *program order*) and that order varies along program execution as a consequence of branches and function calls. All SN-distance values are positive, because the violator instructions *must be* fetched later than the involved store in order to violations could happen.

Figure 5 shows the normalized SN-distance parameter computed for system with *naïve predictor*. A stronger locality than PC-distance case is observed here. More than 90% violations fall inside a 12 instructions window. This information will be used later in a proposed memory dependence detector.

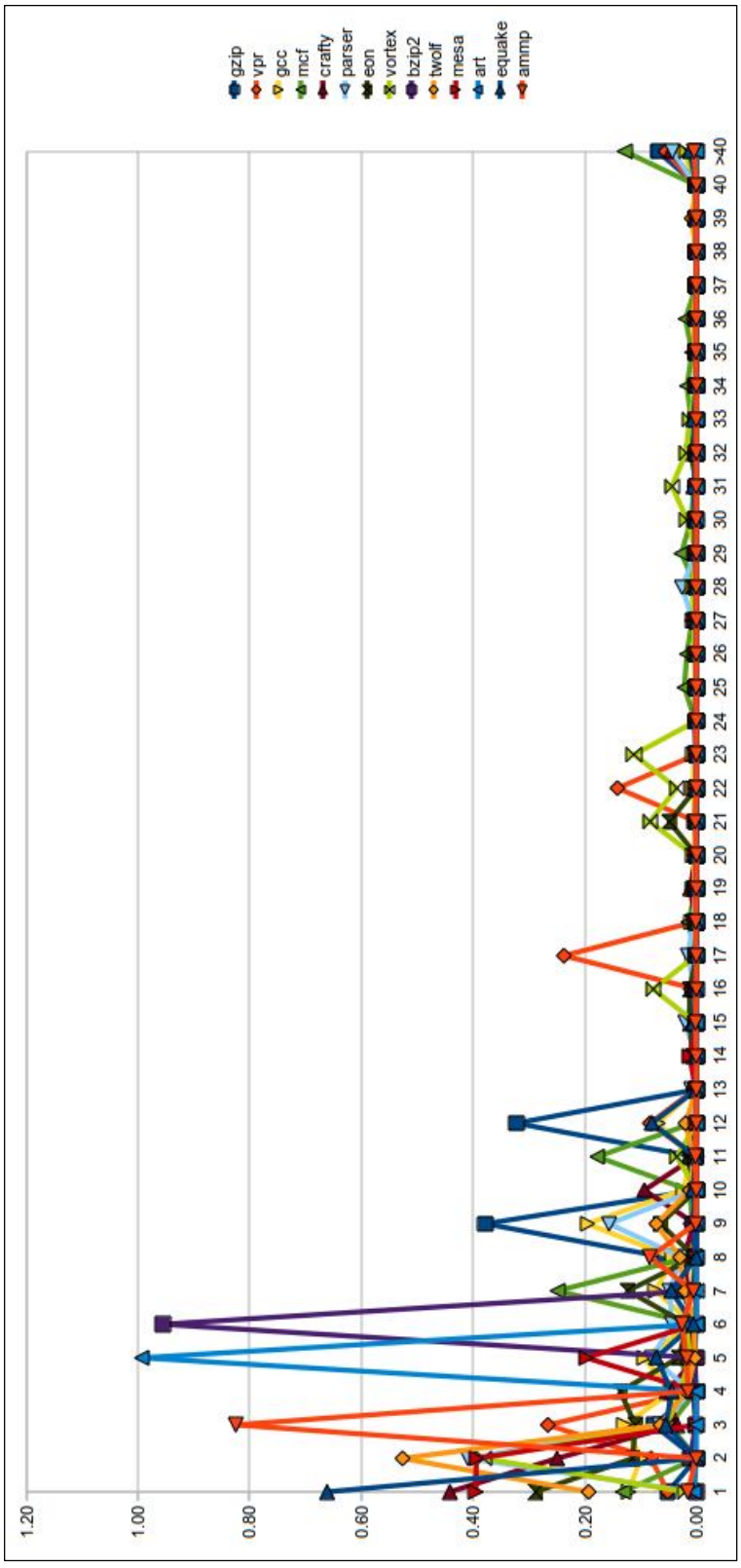

Figure 5 SN-distance for violations under naïve predictor.

Figure 6 shows the SN-distance parameter computed for system with *store-set predictor*. The figure shows that store-sets works well when violations are produced between instructions fetched very close. However, it is interesting to see a significant amount of non-detected violations with SN-distance between 1 and 4 cycles. This problem relates with the way original store-sets algorithm updates its tables. We will analyze deeply this case later, in section 5.6.1.

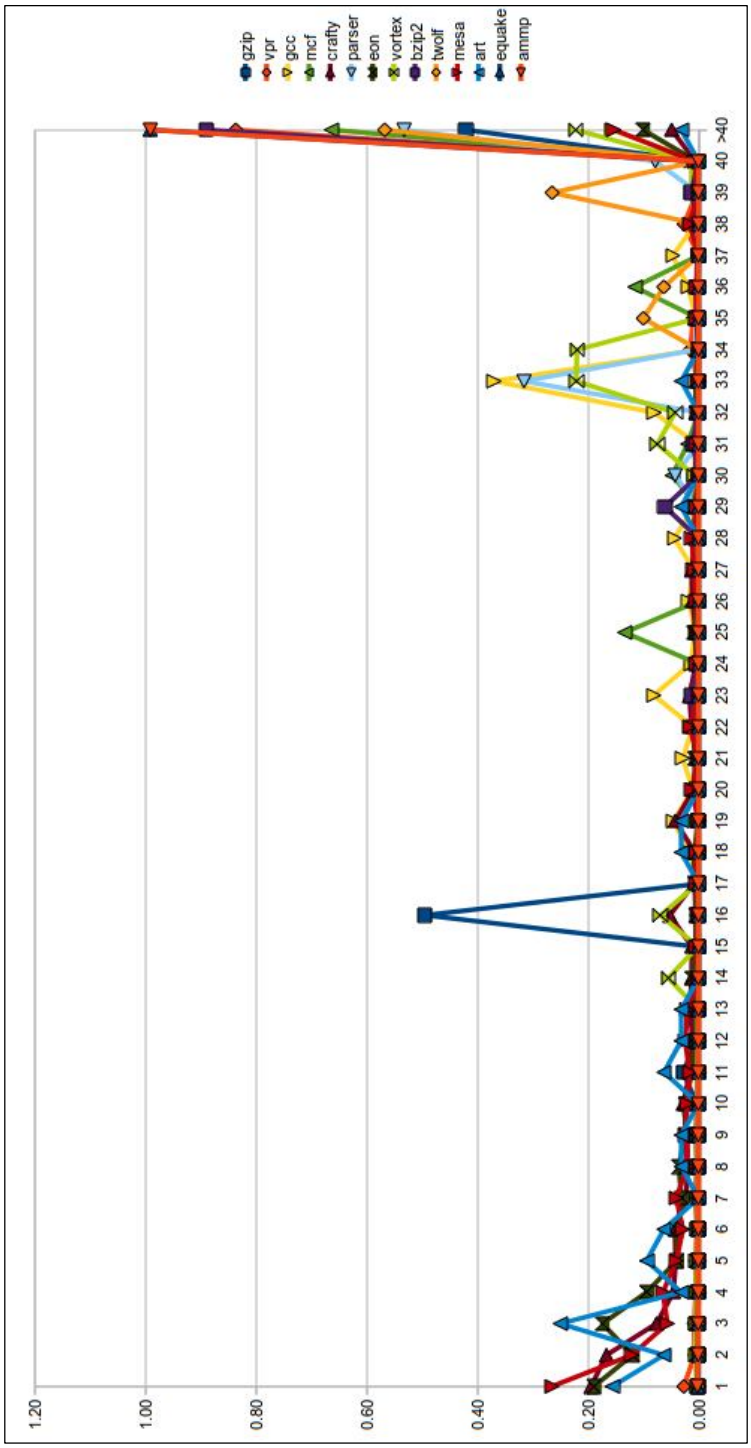

Figure 6 SN-distance for violations under store-sets predictor (1K table size).

4.3.3 TE-distance study

This parameter is another dynamic distance measure, but it focuses in the order instructions are tried to be issued, independently of the fetch order. Differences between Figure 5 and Figure 7 show that violations are more related with *this* parameter than SN-distance, because higher agglutination of small distance violations is observed here.

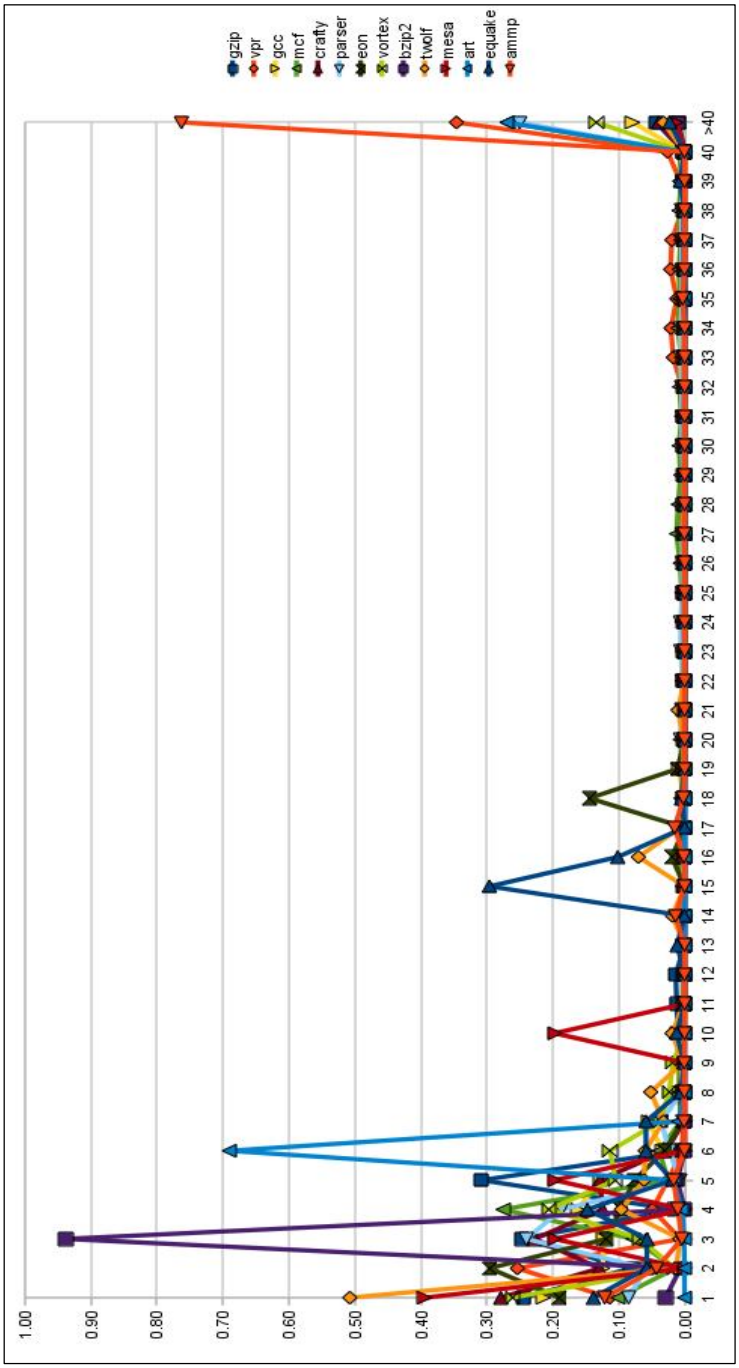

Figure 7 TE-distance for violations under naïve predictor.

Figure 7 correspond to the naïve case. In a similar way to Figure 6, Figure 8 shows the store-sets algorithm keep a significant amount of violations with small TE-distance.

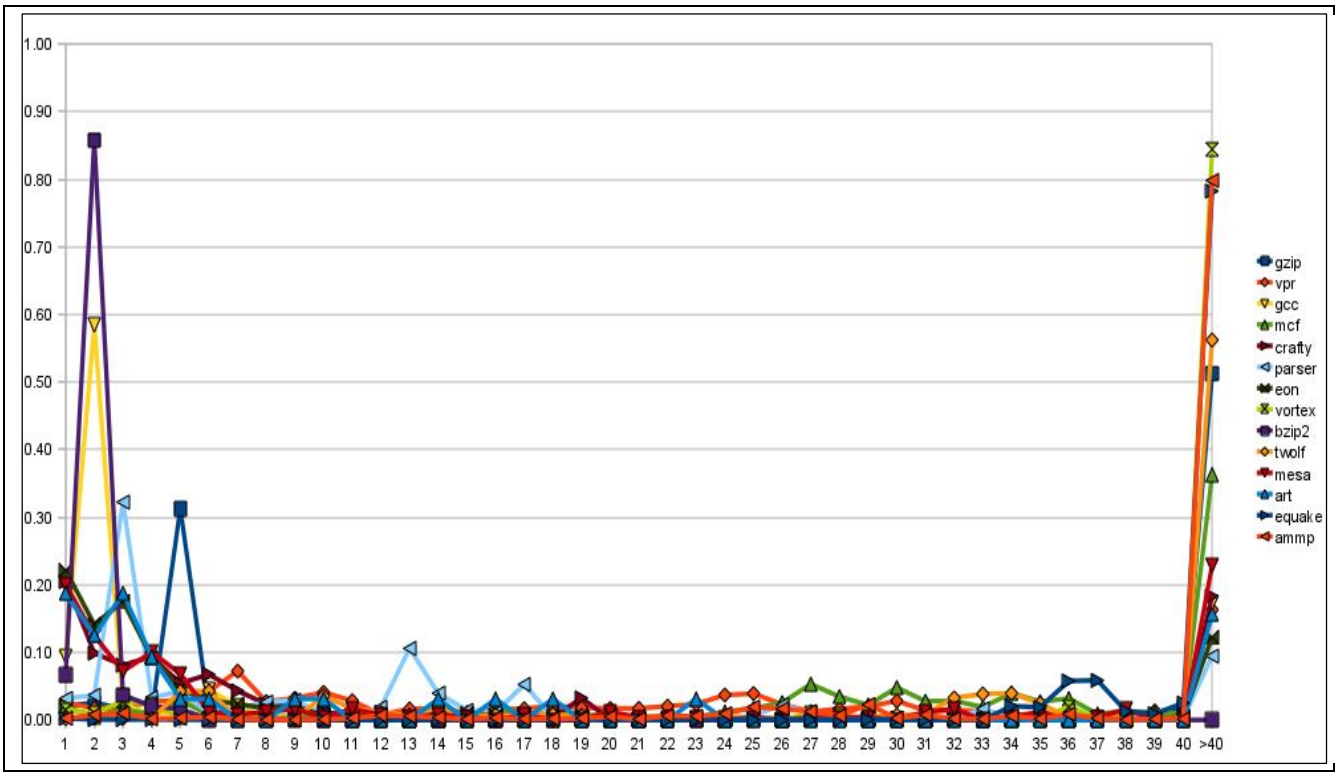

Figure 8 TE-distance for violations under store-sets predictor (1K tables' sizes).

4.3.4 Registers reuse study

We studied also the way the logical registers are used by pairs of instructions that produce violations. Figure 9 shows a comparison between pairs of instructions that produce memory dependence violations. The meaning of the series is:

- Reg1, off, reg2: Registers and offset are identical.
- Off + reg2: Only registers used to compute the accessed memory address and offsets are identical.
- Reg2: Only registers used to compute the accessed memory address are identical.
- Only off: Only offsets used to compute the accessed memory address are identical.

We can observe that, on average, only 19.12% of instructions pairs use the same logical registers and the same offset value, but the value grows to 74.74% when we consider only the second register and the offset, both used to compute the target memory address to be accessed. Considering only the second register, the coincidence scales to 80.56%, and only considering offset, the value is 86.63%.

It is interesting to note that in one case, the bench *art*, the 100% of memory dependence violations occurs between instructions with the same offset and second register, and 99.3% corresponds to instructions pairs with the same three elements. It is also interesting the results in the bench *ammp*, because the difference between *only off* case and the other cases is large. Cases like *vpr* and *crafty* return with *only off* values close to 50%. We think there is not

enough regularity here to use this information as part of a memory dependence predictor.

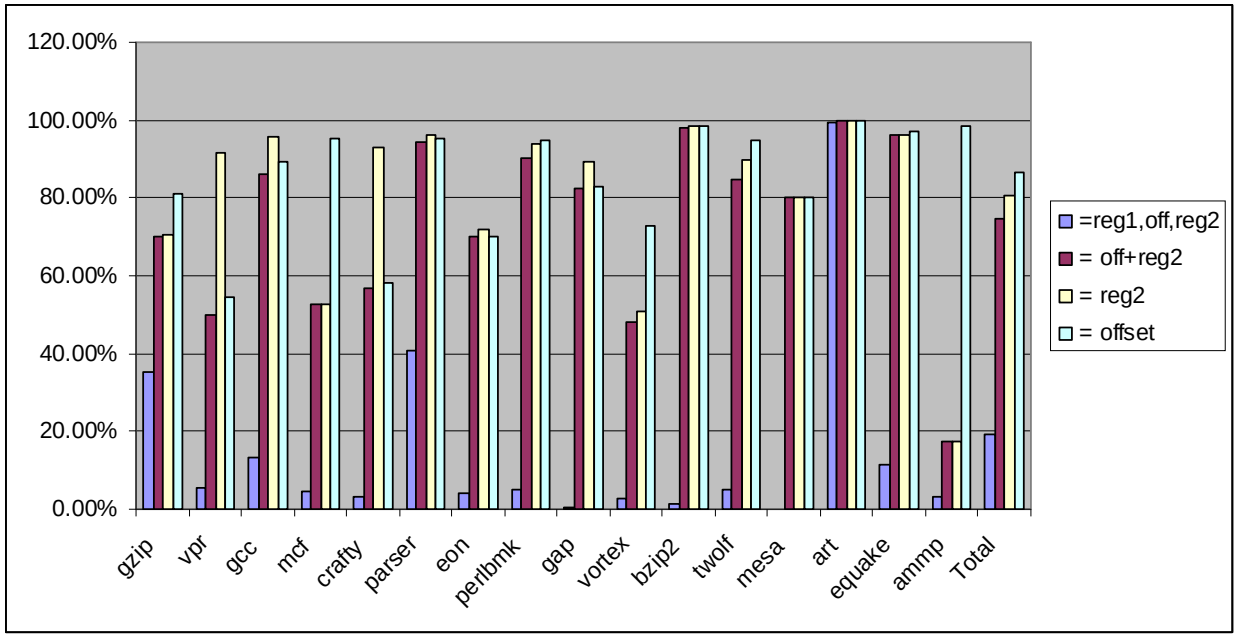

Figure 9 Comparison of logical registers and offset values between load and store instructions that produce memory dependence violations.

4.3.5 Violation frequency

We have studied the frequency of memory dependence violations along a program execution path. We named *load-store violation pair* (or just *a violation pair*) to any pair of instructions load and store that uses to produce violations when the store is issued. Because the execution path can execute many times the same fragment of binary code, an individual *violation pair* could be repeated many times. If the predictor *was not able to catch them the first time*, they produce a violation or just *forget it* because the time between two violation instances from the same pair is too large to keep it in the predictor historical data.

We were interested in the way the violation frequency is distributed, in other words, home many *violations pairs* have low frequency and how many have high one. Figure 10 shows the maximum fraction of the total violations produced by the any *violation pair*. The test was executed using the normal store-sets implementation, with internal tables' sizes of 1024 entries. When this fraction is small, like in *mesa*, we can say that violations are distributed homogeneously along the program. When this fraction is high, like in *equake*, there is a small set of violation pairs that produce most of the violations.

As an example, Figure 11 shows the frequency of violations in bench *gcc*. We chose this bench because it shows an average violation distribution in Figure 10. The X axis corresponds to the number of times a same violation pair produce a violation. The Y axis is the percentage of the total number of violations produced. In the graph, only the last quarter of the values in axis X (with frequency 1100 and above) have the 77.44% of the participation in the total number of violations. The violation pairs with frequencies below 1100 have a very homogenous frequency distribution.

On average, 44.24% of the total violations in a program are attributable to *less than 2% of the violation pairs*. This information was very useful to focus the research work, because we can give special attention to those high frequent violations pairs for any new modification to the original store-sets predictor, or any new prediction method

proposed.

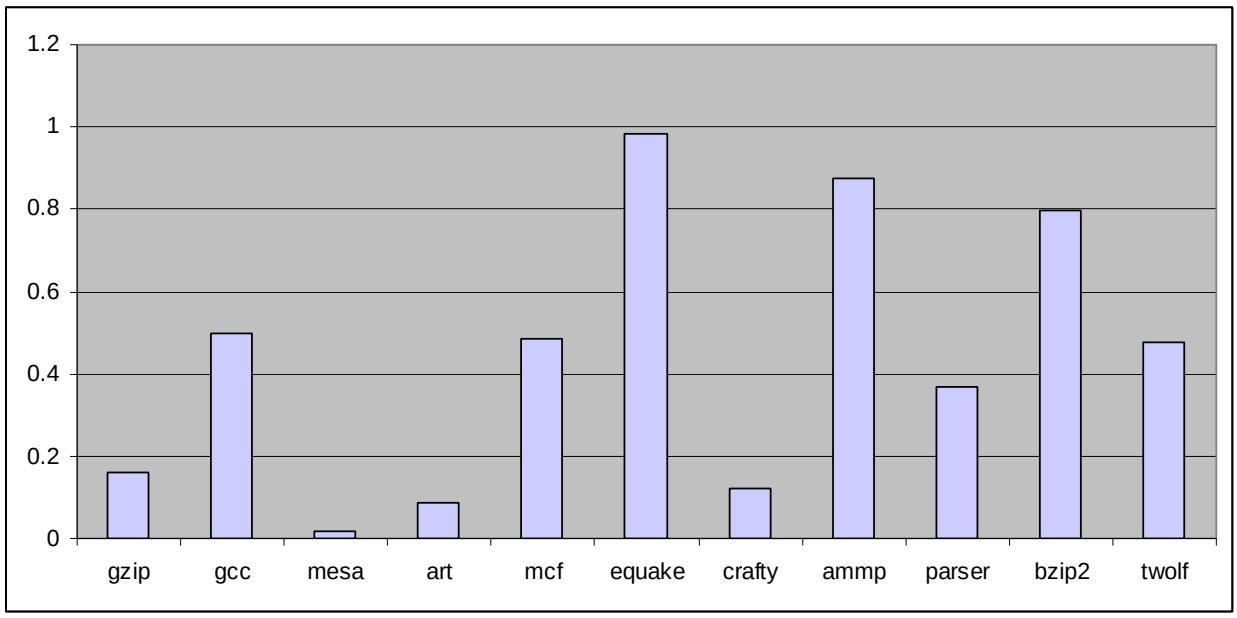

Figure 10 Maximum weights of any load-store pair relative to the total number of violations.

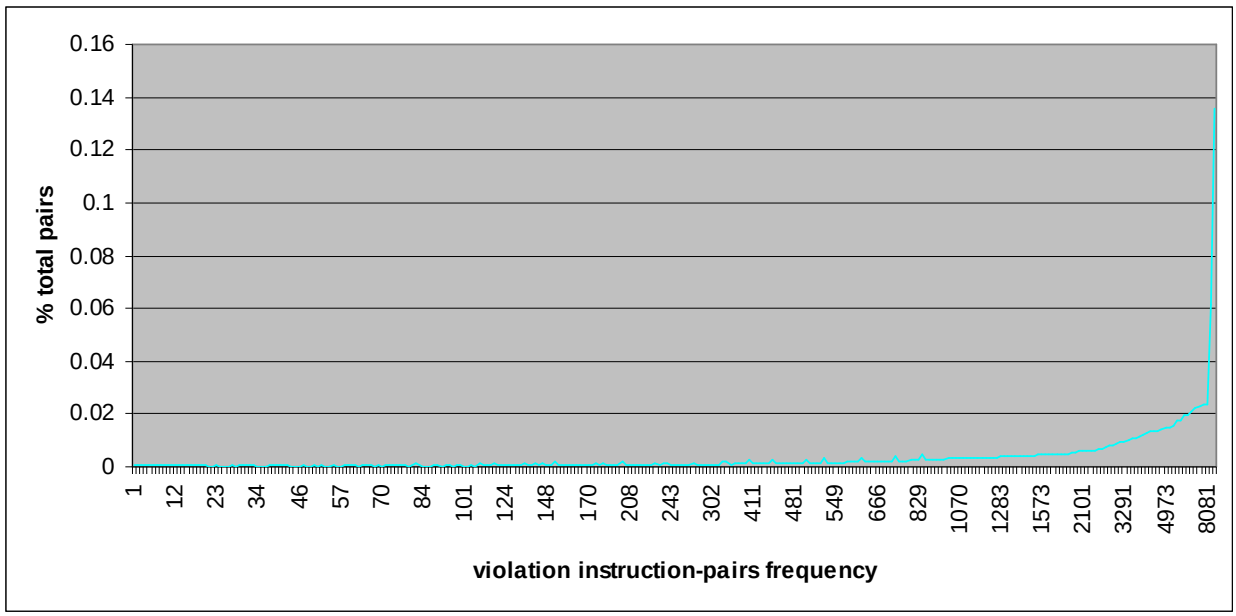

Figure 11 Frequency distribution of violations per pair.

4.4 Summary (identified opportunities)

The above study expose clearly that instructions involved in a memory dependence violation are more closely related with the time of issue (parameter TE-distance) than with fetch time (SN-distance), and this last one have more relevance than the binary code arrangement (PC-distance). Therefore, the best opportunities to use this locality are working with instructions already inside the instruction window (issue pipeline stage). We have use this information to propose a simpler, more efficient and less resource consume solution to deal with violations. We call this method *MiniCAM*. The implementation details and the experimental results are described in the next section.

Chapter 5. Experiments

5.1 Introduction

Now that we understand well the behavior and performance of the default store-sets implementation in our CPU model using the M5 simulator, we can modify this model and perform experiments. We have arranged the experiments in the following groups of predictors:

- Naïve behavior: Two versions were implemented: A full-naïve and a half-naïve. The full-naïve version issues instructions as soon as they are ready, and half-naïve waits until effective address is computed in load and store instructions to issue them.
- Perfect predictor: It is obtained recording the branch decisions and memory accesses performed along the critical execution path in a bench program. Using this information, the bench can be executed avoiding all the memory accesses violations.
- Store-sets predictor under different internal tables' sizes: This is a sanity check test, confirming previous results in the original store-sets paper publication [02].
- Store-sets predictor under artificial *read after read* (RaR) dependence contention: Load accesses to the same memory locations are forced to be executed in order. The objective is to study the store-sets algorithm behavior in this context.

Two additional groups correspond to proposals:

- Store-sets predictor with modifications: Some flaws in the original store set implementation are modified and fixed in order to improve it.
- MiniCAM predictor: This corresponds to the proposed method for reducing, in a more efficient way, the memory dependence violations.

Sections 5.2 to 5.7 describe in detail these experiments: their characteristics and results. It is important to keep in mind that our final objective is improving the program execution, achieving, when it is possible, other important goals as resource utilization and implementation complexity. Section 5.8 describes some additional aspects in the processor architecture that affects the performance of the memory dependence predictor unit. Finally, section 5.9 makes a summary of the chapter.

5.2 Naïve predictor

Two versions are implemented. The first one is a full naïve predictor. This is an optimistic predictor, because it assumes all memory access instruction will not produce violations, therefore it issue them as soon as their used registers are ready. This test gives us a base for measuring the total improvement for the rest of modifications.

The half naïve predictor makes a less optimistic assumption. It waits memory access instructions until the effective memory address is computed. When this occurs, the predictor checks the address with all the stores waiting in the instruction window, hoping to catch potential violations. If no conflict is detected, the predictor issues the instruction. This behavior reduces the violations, as we can see in Figure 12 where the IPC is compared against a perfect predictor. In this figure, the difference between the "naïve bar" and the "perfect" bar corresponds to the performance improvement that it is possible to reach using any memory dependence predictor in the corresponding bench. This difference depends on the ratio between the amount of violations and the amount of executed instructions. Figure 13 shows this ratio.

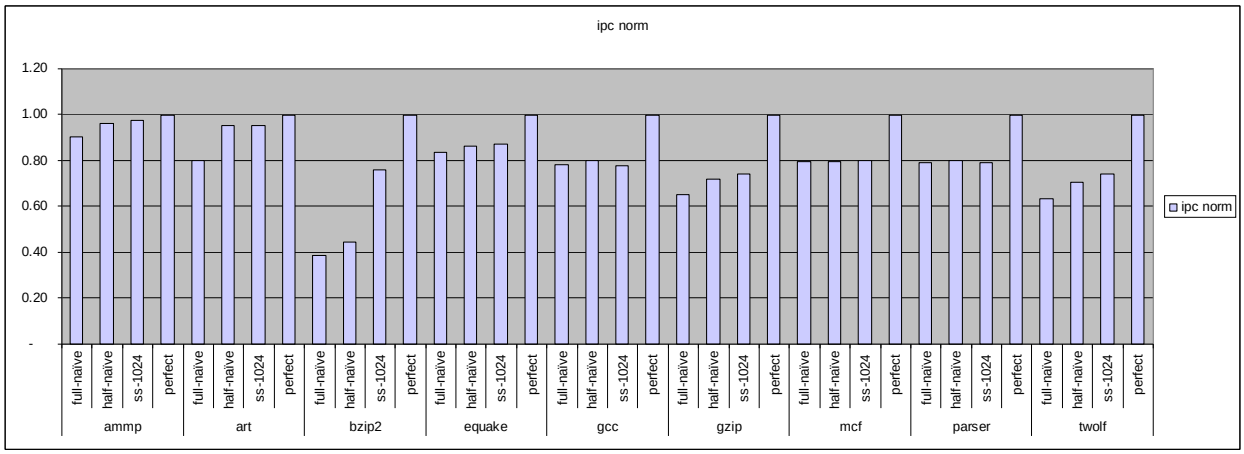

Figure 12 Normalized IPC comparing naïve predictors with perfect predictor.

Comparing results in Figure 12 and Figure 13, it is clear that as higher the percentage of violations per executed instruction, as higher the opportunity to improve the performance using a predictor. For example, bzip2 shows a big improvement in Figure 12 because it produces a high number of violations, as we can corroborate in Figure 13.

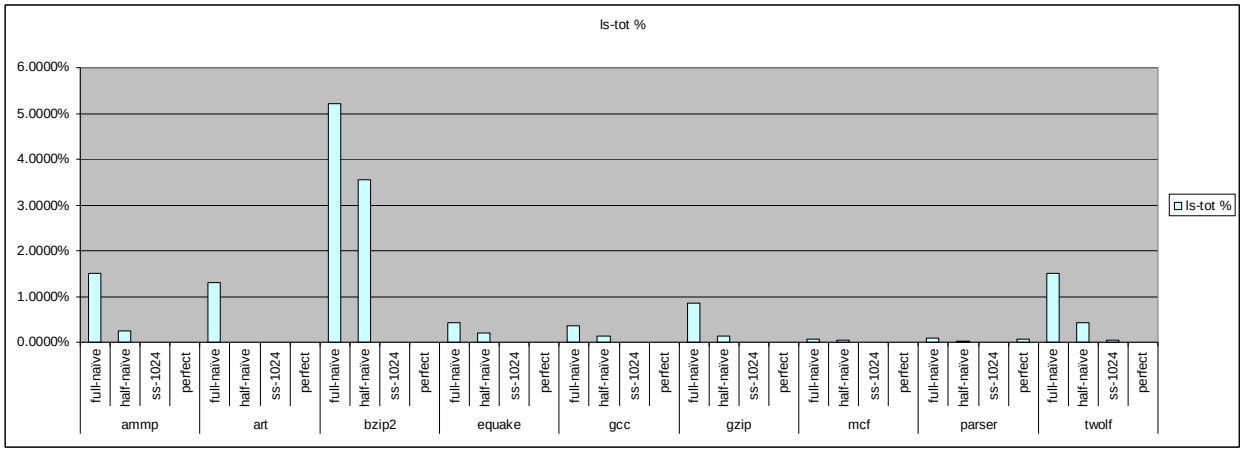

Figure 13 Load-store violations normalized against total executed instructions.

5.3 Perfect predictor

A perfect predictor was implemented in three steps. The first one executes the bench program recording all branch valid decisions made by the branch predictor (*branch history*). The second step executes again the same bench using a modified branch predictor unit that uses the previous information to always follow the right critical path. Under this circumstance, all valid store memory accesses are recorded (effective address accesses history, or *EA history*). Here, a valid store instruction is one successfully executed. The last step executes one more time the same bench using the previous perfect branch predictor and a perfect memory dependence predictor unit. This unit uses the previous recorded *histories* to make always right predictions and only contain load instructions that will not produce violations.

This predictor gives us the upper limit for any memory dependence prediction algorithm. Combined with the naïve predictor information, we could measure all the rest of proposals.

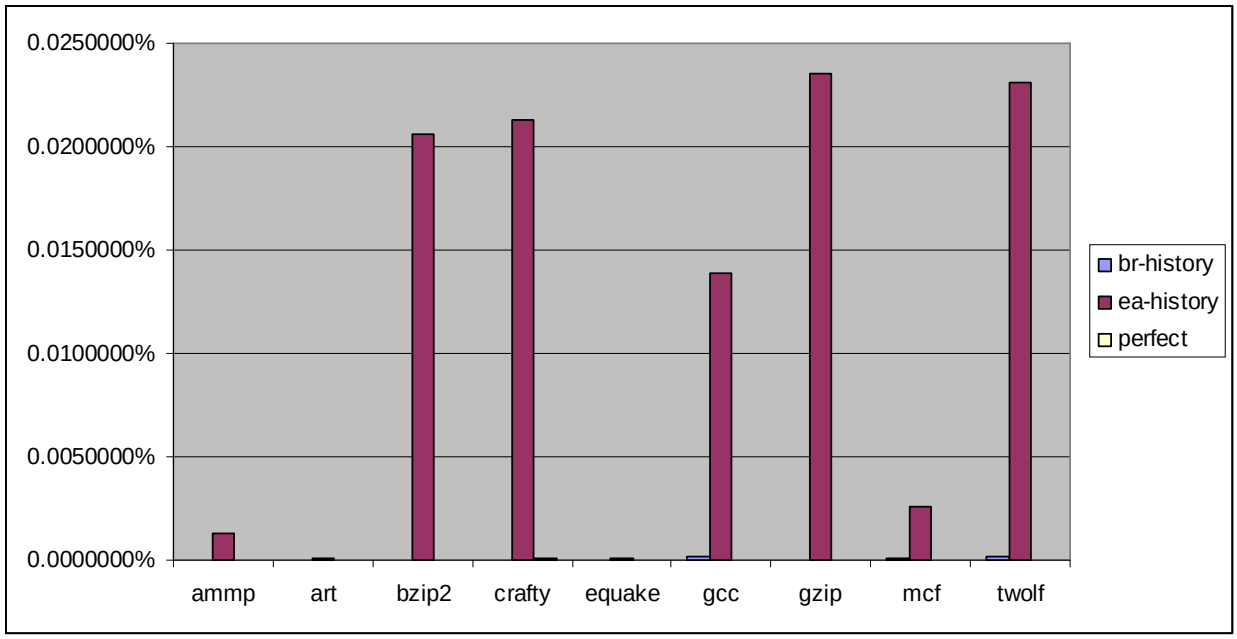

Figure 14 Memory dependence violations (normalized against total executed instructions) comparison between perfect predictor and store-sets predictor (1024 entries tables) using branch history and EA history.

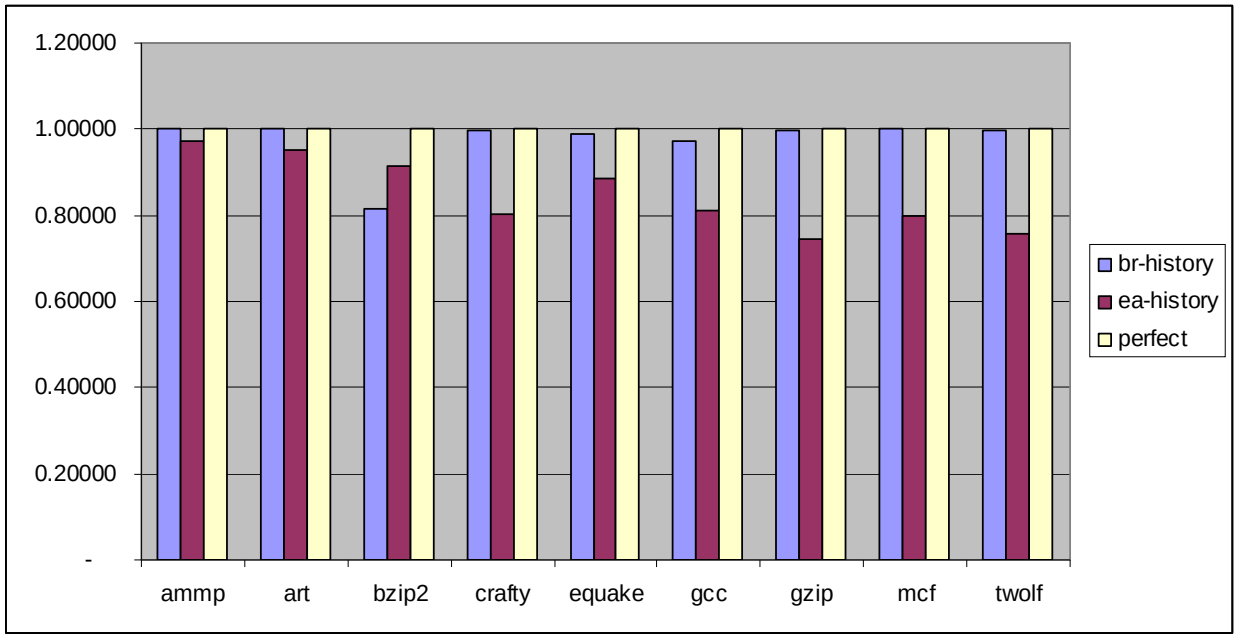

Figure 15 Normalized IPC (against perfect predictor) between perfect predictor and store-sets predictor (1024 entries tables) using branch history and EA history.

Figure 12 shows the perfect predictor results for some benchs, but we were curious also to check the effect when using the EA history and only using the branch history. Figure 14 shows the amount of memory dependence

violations in the three cases. As we can observe, the branch predictor have a significant influence in the final effect of the predictor, at the point that, using a perfect branch predictor (br-history case) leads to a near perfect behavior for the store-sets algorithm.

These results correspond with Figure 15 results. In some cases, the lost of performance is larger than 10%, as in *gzip* and *twolf* benchs.

5.4 Store-sets original implementation

We perform a sanity check, testing the built-in implementation algorithm that comes with M5 simulator, changing internal tables sizes to observe how the violation detection level changes. The idea behind is reproducing store-sets author results presented in [02], verifying the validity of the M5 implementation. The results are already described in section 5.2, as part of the comparison with a naïve predictor. Those results corresponds

5.5 Store-sets with read after read contention

Dependences between stores and loads (*read after write*, or *RaW*) are named *real* because the other two possibilities, between stores (named *write after write*, or *WaW*) and between loads (called *read after read*, or *RaR*) are resolved efficiently with other mechanisms, not predictors, without a false contention penalty. But RaR dependencies, treated as real ones, could positively change the behavior of a program execution. Let's put an example:

```
(1) load r1, 10(r2)
...
(2) store r3, 20(r4)
...
(3) load r5, 30(r6)
```

In the previous sequence of instructions, if registers r1, r2, r5 and r6 are resolved before registers r3 and r4, then load instructions 1 and 3 could be executed before instruction 2. But if the three instructions access to the same memory location and we consider RaR dependencies as real ones, instruction 3 must be delayed until instruction 1 is issued. This gives the possibility to resolve r3 and r4 on time to catch the RaW dependency between instructions 2 and 3.

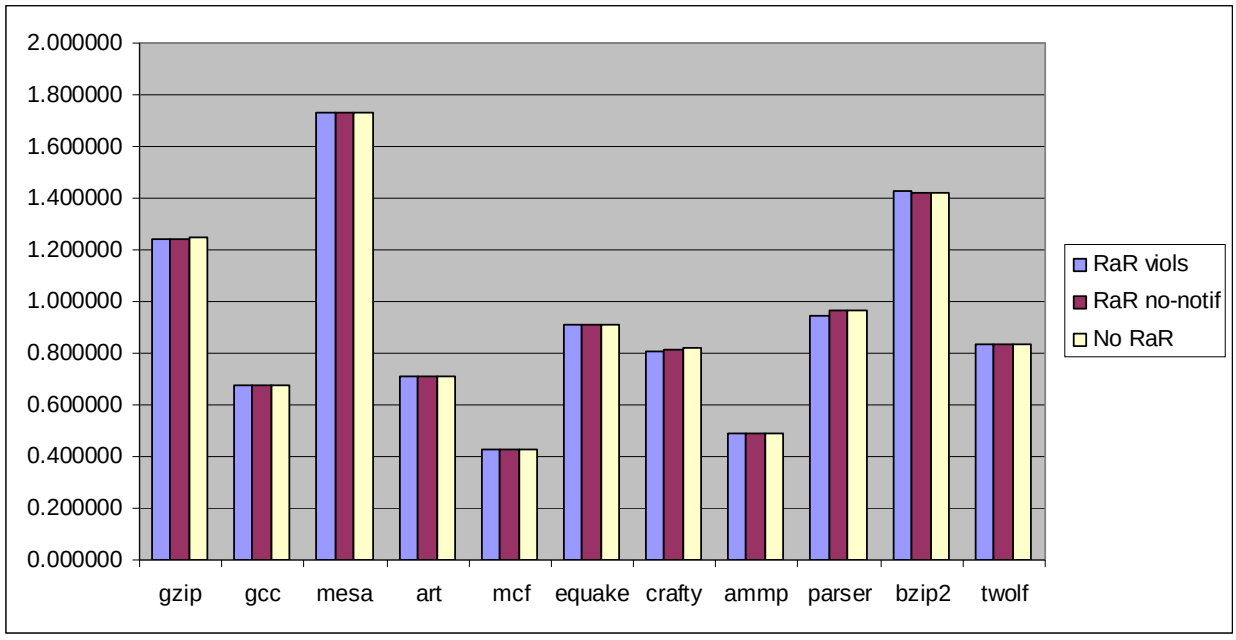

Figure 16 IPC for store-sets considering load-load dependences as real dependences.

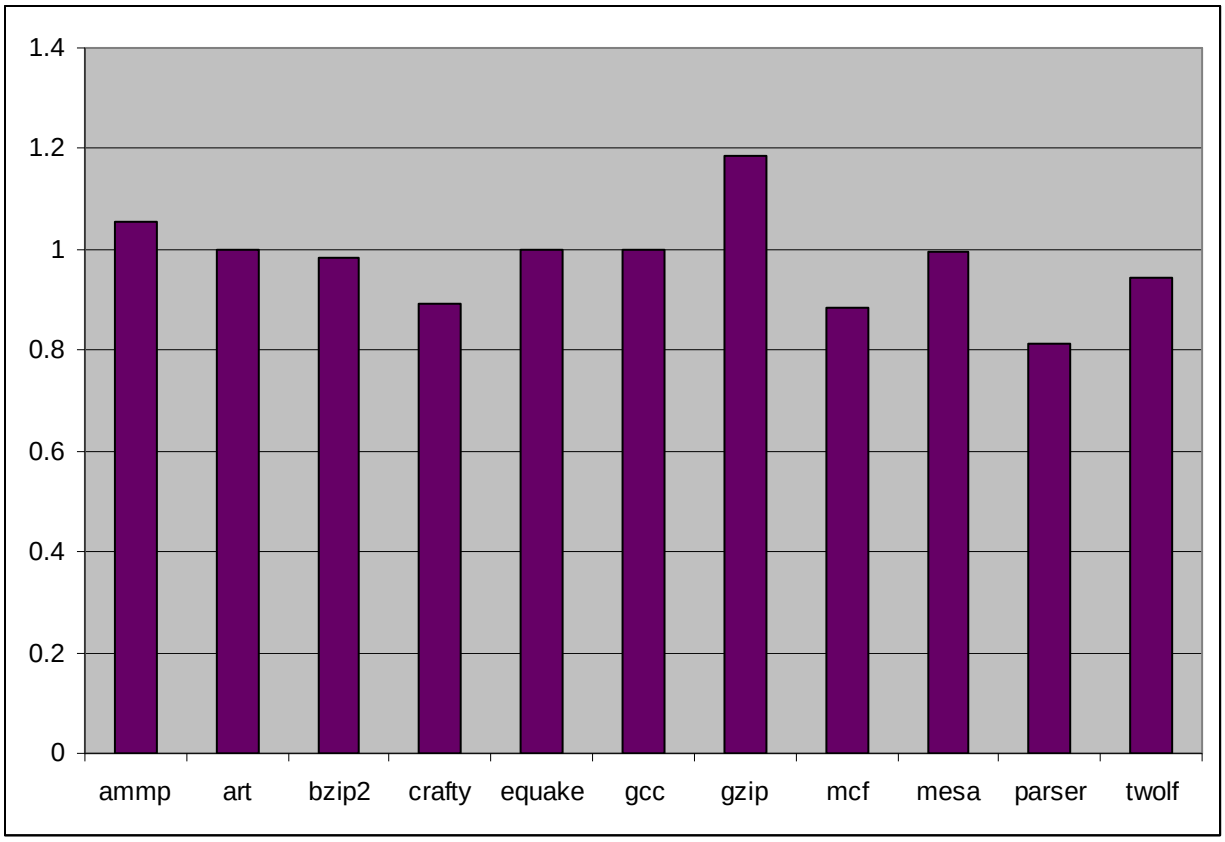

Figure 17 Miss-predictions of RaR-no-notif, normalized with miss-predictions of original store-sets.

Another situation, specific for store-sets, occurs when a RaW violation is detected. In this case or a new *set* is created, or one of the loads is put in the *set* of the other. In the first case, if the internals tables (SSIT and LFST) are small, creating new *sets* increase the probability of previous well executed stores and loads fall inside those new sets, creating artificial contention. In the second case, introducing new instructions in the already created *sets* could also create false dependences and more contention.

Nevertheless, it is worth testing this configuration. We test two flavors for RaR artificial contention:

- Consider RaR dependences as real violations and notifying them to the store-sets predictor unit. This corresponds to the *RaW viols* series in Figure 16.
- Consider RaR dependences as real violations but not notifying them to the store-sets predictor unit. This corresponds to the *RaW no-notif* series in Figure 17.

In Figure 16, the No RaW series correspond to the normal case, when load-load dependences are not considered as real violations. The results show no significant difference in performance between the three cases.

It is interesting to note that processing load-load dependence violations but not notify them to store set unit produces a small reduction of the amount of miss predictions because less sets are created and more load instructions could be speculative executed. Figure 17 shows the miss-predictions found in this case. The values are normalized using the amounts of miss-predictions per bench of the original store-sets implementation.

5.6 Improving store-sets

The original store-sets algorithm has some weak points. In the following sections we describe the modifications introduced and its results.

5.6.1 Restoring LFST

The first one is relative to the LFST table. When the branch predictor unit makes a miss prediction, information about squashed stores remains in the LFST table, leaving it dirty. The original implementation mark as invalid the entries in LFST affected. But this is not enough in many cases. Let's put an example:

```
(1) store r1, 10(r2)
(2) bnz r1, XYZ
(3) load r3, 20(r4)
...
XYZ:
(4) store r5, 30(r6)
```

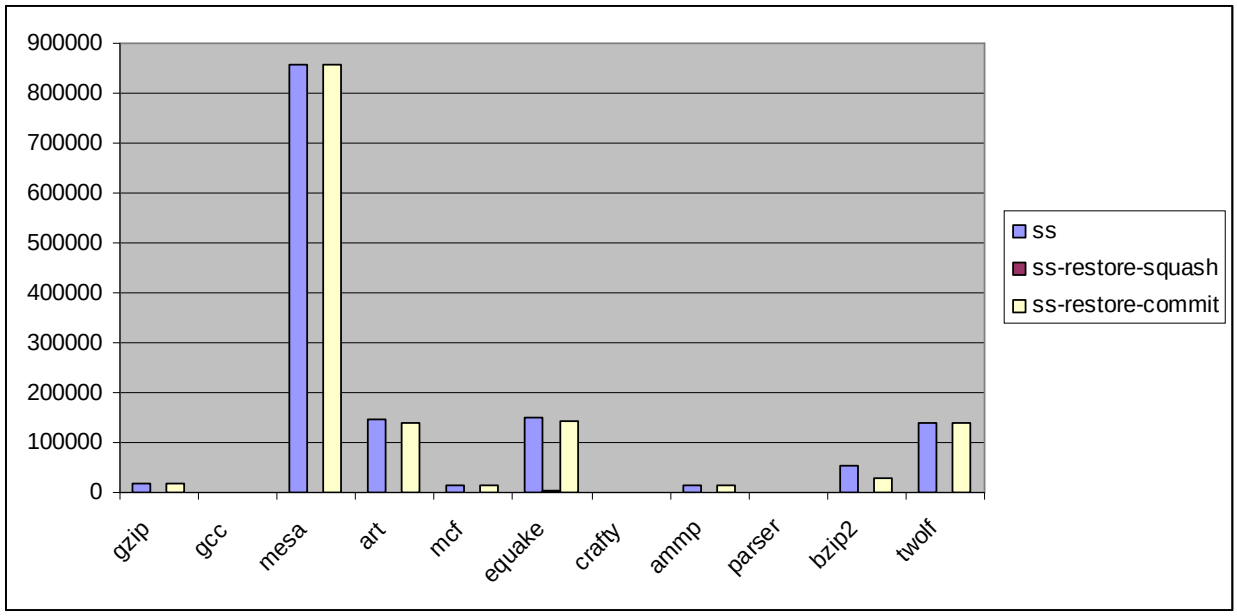

Figure 18 Number of memory dependence violations using store-sets with LFST restored after squash.

Suppose that instruction 1, 3 and 4 access to the same memory location and branch instruction 2 miss predicts a jump to address XYZ. In this situation, stores 1 and 4 will end in the same set. Before the branch was speculatively executed, the corresponding entry in LFST will mark store 1 as the last store in its set. After branching, store 4 replaces store 1 in the same LFST entry. When store 4 is squashed, the LFST entry is marked as invalid. Later, if the load 3 is issued O3 before store 1, the store-sets predictor unit will not realize that a real dependence exist between instructions 1 and 3 and a violation will be detected when finally store 1 is issued. This is a good example where an independent predictor affects others, later in the pipeline.

One possible solution, implemented and tested in this work, is to restore the LFST affected entries with the last previous store in the same *set*. A more sophisticated version could restore the table entry with the last not-speculated store in the same *set*. Two possibilities were tested for the first case: Restore LFST on squash time, and restore it on commit time.

Even when restoring LFST at squash time is the obvious solution, it is common to have time limitation when the processor deals with a squash. That is why we have also considered the alternative of delaying this restoring to the commit phase.

Figure 18 shows that waiting to commit to restore the LFST table is too late. The reduction of violations is not significant because the effect of the dirty entry in the table is already produced in most of the cases. Restoring the

table during the squash event increases the precision of the algorithm in a significant way.

5.6.2 Using feedback: Deactivating old sets

The second improvement consists in avoiding *sets* remains active eternally. Because it is common that a program execution focus over different program parts at different times, sets created for one program part will not be useful in another. Leaving unnecessary *sets* always active increase false contention because load and store instructions, in the new program section, could fall into *sets* in the old program section.

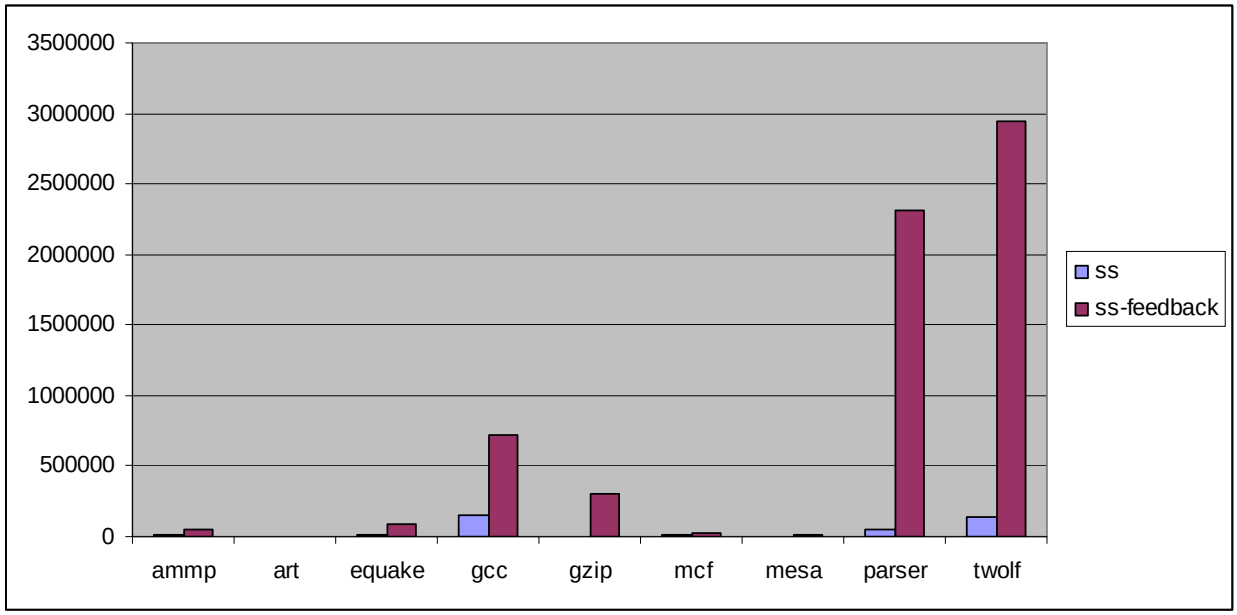

Figure 19 Memory dependence violations using store-sets predictor and feedback for sets deactivation.

One way to avoid this is including a mechanism that removes sets not used. We have used a 2-bits saturation counter associated with each active set for this purpose. The counter works in this way: The counter always starts with the maximum value. When a predicted dependant load is issue, the prediction is checked. If this prediction was correct, the counter associated to its *set* receives again the maximum value (aggressive politic). If this prediction was incorrect, the counter is decreased. When the counter reaches zero, the set is considered old and marked as inactive.

Figure 19 shows worst precision using feedback compared to normal store-sets implementation. This is expected because fewer entries in SSIT table are active, on average, at any execution time. The real benefit must be looked in the number of miss-prediction produced. Figure 20 shows that, on average, miss-predictions are reduced.

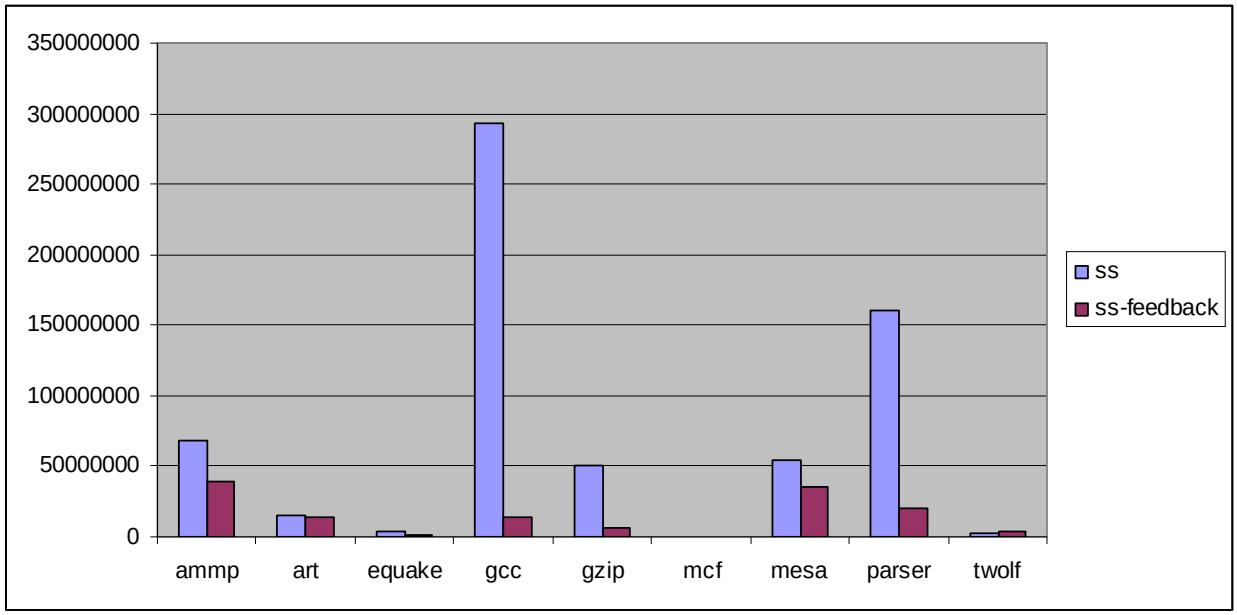

Figure 20 Miss-predictions produced using store-sets predictor and feedback for sets deactivation.

5.6.3 Mixing restoring LFST with old sets deactivation

We have tested a mixed solution combining the two previous presented. As we can see in Figure 21, the improvement in performance is not significant.

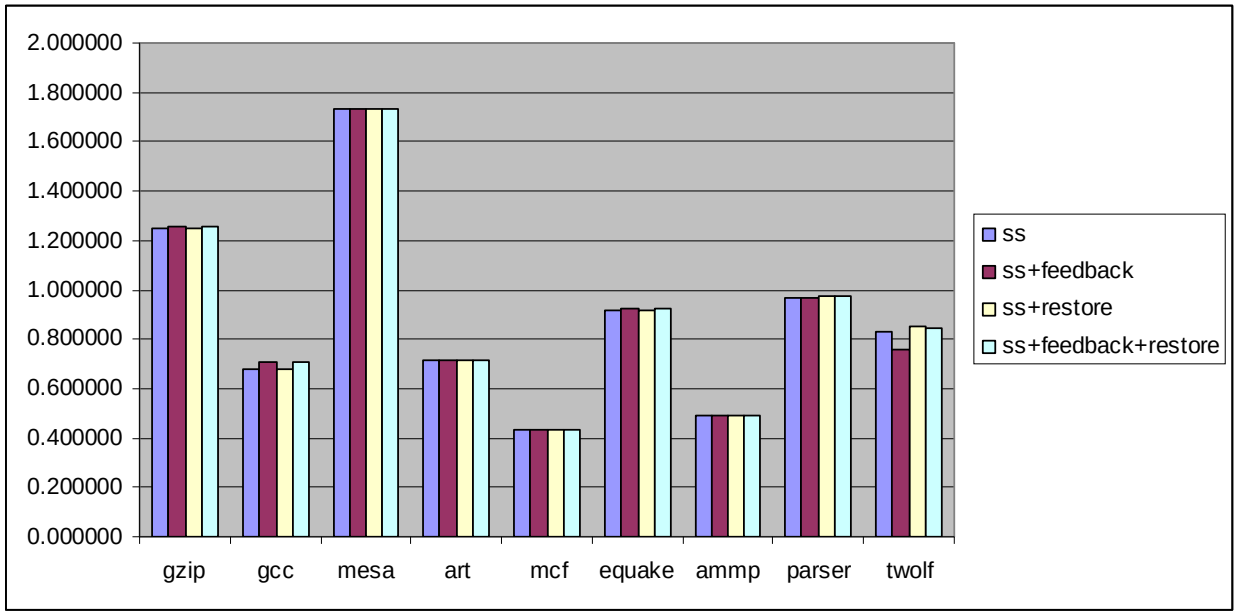

Figure 21 IPC compared between normal store-sets and modified versions that include feedback and restoring LFST at squash.

5.7 Non speculative solutions

Two non speculative solutions are presented in the following sections. A mixed version, combining a speculative solution and a non speculative one, is also presented.

5.7.1 Ordering dependence checking

In the original store-sets implementation, load instructions are checked against the active sets as soon as they are introduced in the instruction window. This corresponds to a full O3 execution, avoiding as long as possible any contention, but at the risk of creating violations. On the other side, waiting to resolve all registers used for each memory access instruction, before issuing it, introduces unnecessary contention.

An alternative halfway solution is to delay dependence checking until register related with target memory address is available. This introduces less contention and could improve store-sets predictor results. The IPC comparison between the original store-sets algorithm and the new one (with ordering dependence checking) is showed in Figure 22.

We have observed some new violations in this solution. Reviewing the logs, we found that delay dependence checking with stores sets produce some checking errors. Let's put an example. The following fragment corresponds to bench 164 (gzip). It is a loop where there is a dependency between instructions 120011f4c y 120011f5c:

```
120011f40: 00 00 29 2c    ldq_u   t0,0(s0)
120011f44: c2 00 29 48    extbl   t0,s0,t1
120011f48: 01 00 29 21    lda     s0,1(s0)
...
120011f4c: 00 00 23 2c    ldq_u   t0,0(t2)
120011f50: 41 00 23 48    mskbl   t0,t2,t0
120011f54: 62 01 43 48    insbl   t1,t2,t1
120011f58: 02 04 41 44    or      t1,t0,t1
...
120011f5c: 00 00 43 3c    stq_u   t1,0(t2)
120011f60: 01 00 63 20    lda     t2,1(t2)
120011f64: 24 31 80 40    subl    t3,0x1,t3
120011f68: f5 ff 9f f8    bge     t3,120011f40
```

When the loop executes, it mostly happens that source registers of load 120011f4c are ready before registers of store 120011f5c, therefore the system tries to issue load before. Suppose the execution is in the iteration N, the store 120011f5c is inserted in the LFST table, updating the corresponding entry. Because source registers are not available and dependences are not checking yet, system does not issue it. Later, in the iteration N+1, load 120011f4c is processed by the dependence predictor unit. Again, source registers are not ready, then dependencies are not checked and instruction is not issue. Next in iteration N+1, a new store 120011f5c is inserted in LFST, in the same entry as before, updating it. A moment later, load 120011f4c passes to have it source registers available, and dependencies are checked against the same LFST entry, but it find a younger sequence number. In consequence, it is assumed that load instruction has no dependencies (clearly a mistake), and it is issued. When store 120011f5c in iteration N is issue, the system finds a violation against load 120011f4c in iteration N+1. In other words, because store-sets depends, for a correct operation, that dependences were verified *in order*, including a delay in the dependence checking process breaks this requirement.

The previous problem could be solved keeping information about all the stores in every store set, not only the last store inserted. In this way, when a load instructions is checked against an active set, the list of stores inside are iterated looking for the youngest one, older than the load. If all stores in a set are younger than the checked load, then we confirm that there are no dependencies. But this modification adds complexity to the predictor implementation, and requires a longer execution time. We have not tested it because this will exceed the project time limitations.

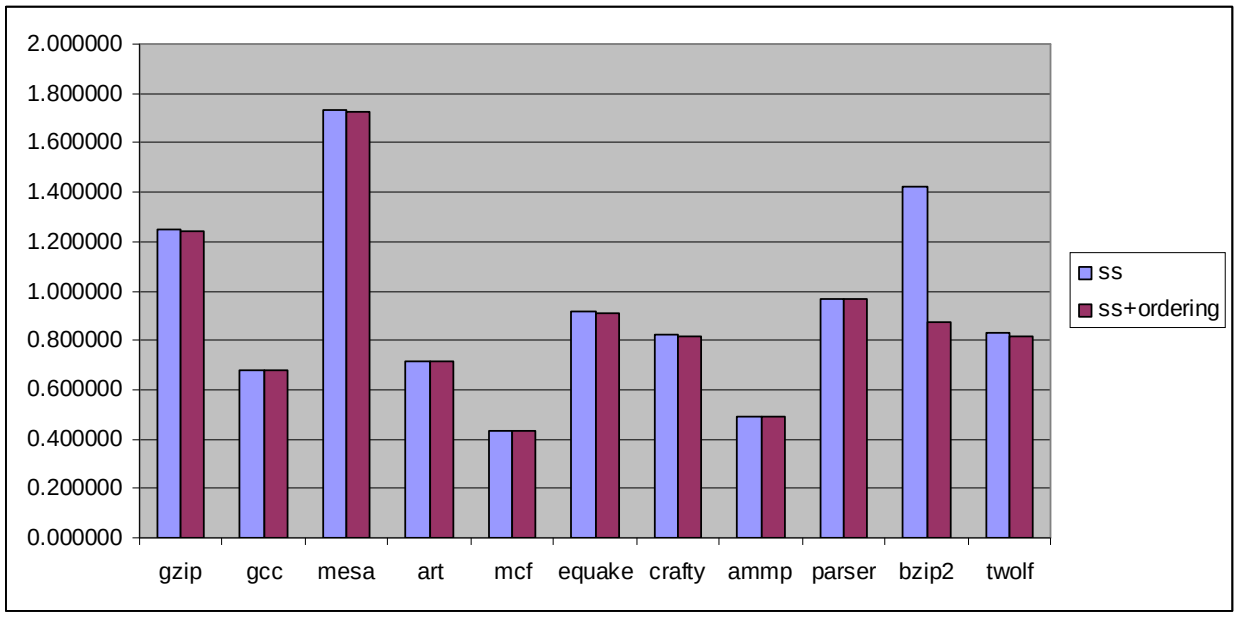

Figure 22 IPC comparison between original store-sets and a modification using ordered memory dependence checking.

5.7.2 Exploiting extreme locality: The MiniCAM solution

The key idea is exploit the fact (as will be presented in test results below) that many memory dependencies could be resolved inside a very short space and time locality. Two cases are studied:

- Mini CAM by cycles: Time locality is exploited. A very small CAM is used to find (not speculate) memory dependencies between the *registers-ready load instruction* and the instructions fetched in the previous N cycles.
- Mini CAM by history: Space locality is exploited. A very small CAM is used to find (not speculate) memory dependencies between the *registers-ready load instruction* and the N previous fetched instructions.

We also consider that it is possible to catch most of the memory dependencies with a simpler approach than store-sets using the strong locality we have found in the previous results. We propose the use of a very small content addressable memory (or MiniCAM) for detecting only the dependencies under strong locality conditions. We have study the following variants:

- MiniCAM with the instructions fetched in the last N cycles, with a limit in the CAM size. We name it MiniCAM-Cycles (mc-cycles).
- MiniCAM with the last N instructions fetched. We name it MiniCAM-History (mc-history).

In mc-cycles, we focus only in instructions extremely close at fetch time. When a new fetched store is inserted in the mc-cycles CAM, if there is no space, the oldest instruction in the CAM is removed and the free entry is filled with the new store instruction data. Additionally, instructions in the CAM that becomes old (pass the maximum N cycles configured) are removed to not be considered in next memory dependence checking. The case of mc-history is similar to mc-cycles but not considering the stores fetch time.

To produce useful memory dependence checking, MiniCAM delays it until the registers involved in computing a memory address become available. It is important to note that only those registers are waited. This produces less contention than waiting all registers to be ready. It is also important to note that this method does not perform any

prediction. The memory dependence is always checked against instructions with its accessed memory address already computed.

Figure 23 shows the IPC obtained when the mc-cycles predictor is used with CAM sizes 16 and cycle limits varying from 1 to 16. The performance difference with respect to original store-sets implementation (with internal tables' sizes of 1024 entries) is very short, as we can see in Table 5. Results with cycle's limit 16, 8, and 4 are near identical, and a small degradation is perceived with cycle's limit 2 and 1.

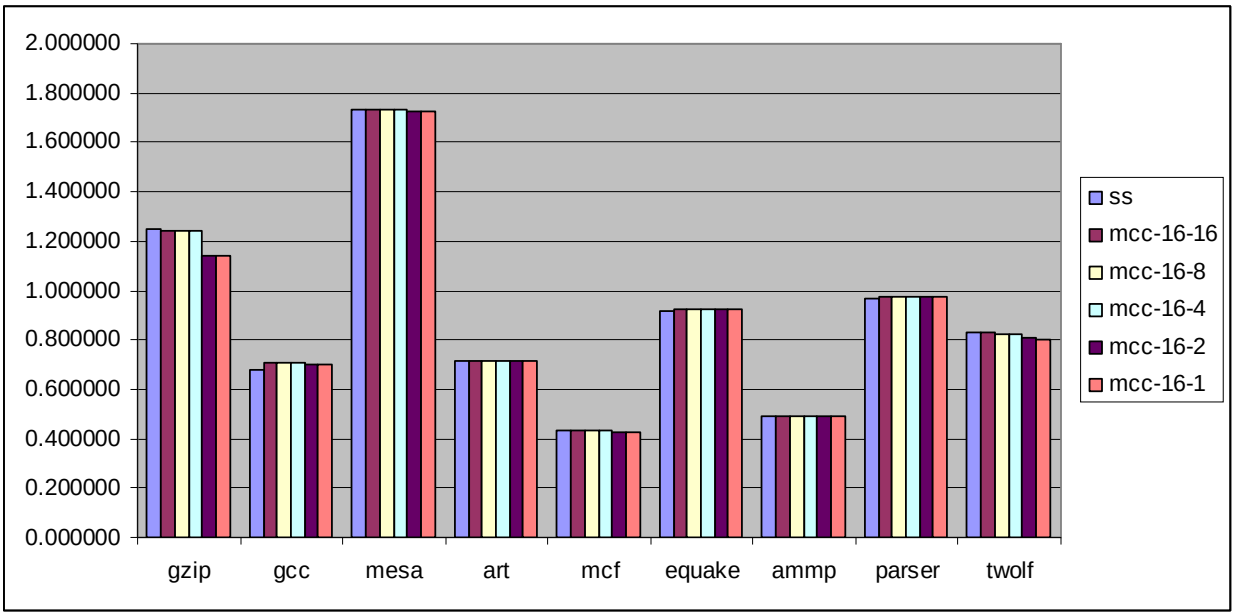

Figure 23 IPC of the original store-sets algorithm compared to MiniCAM-Cycles proposal.

Table 5 summarizes the performance difference, on average, between store-sets alternative and MiniCAM-Cycles algorithm. We corroborate that, with a threshold of 4 cycles proximity, the final performance is comparable with the original store-sets proposal.

Table 5 Performance difference between store-sets and MiniCAM-Cycles.

	mcc-16-16	**mcc-16-8**	**mcc-16-4**	**mcc-16-2**	**mcc-16-1**
Performance difference with store-sets	1.0054	1.0044	1.0034	0.9876	0.9862

Figure 24 shows the IPC obtained when the mc-history predictor is used with CAM sizes varying from 1 to 16. Because this alternative is less restrictive than mc-cycles, a small improve is observed.

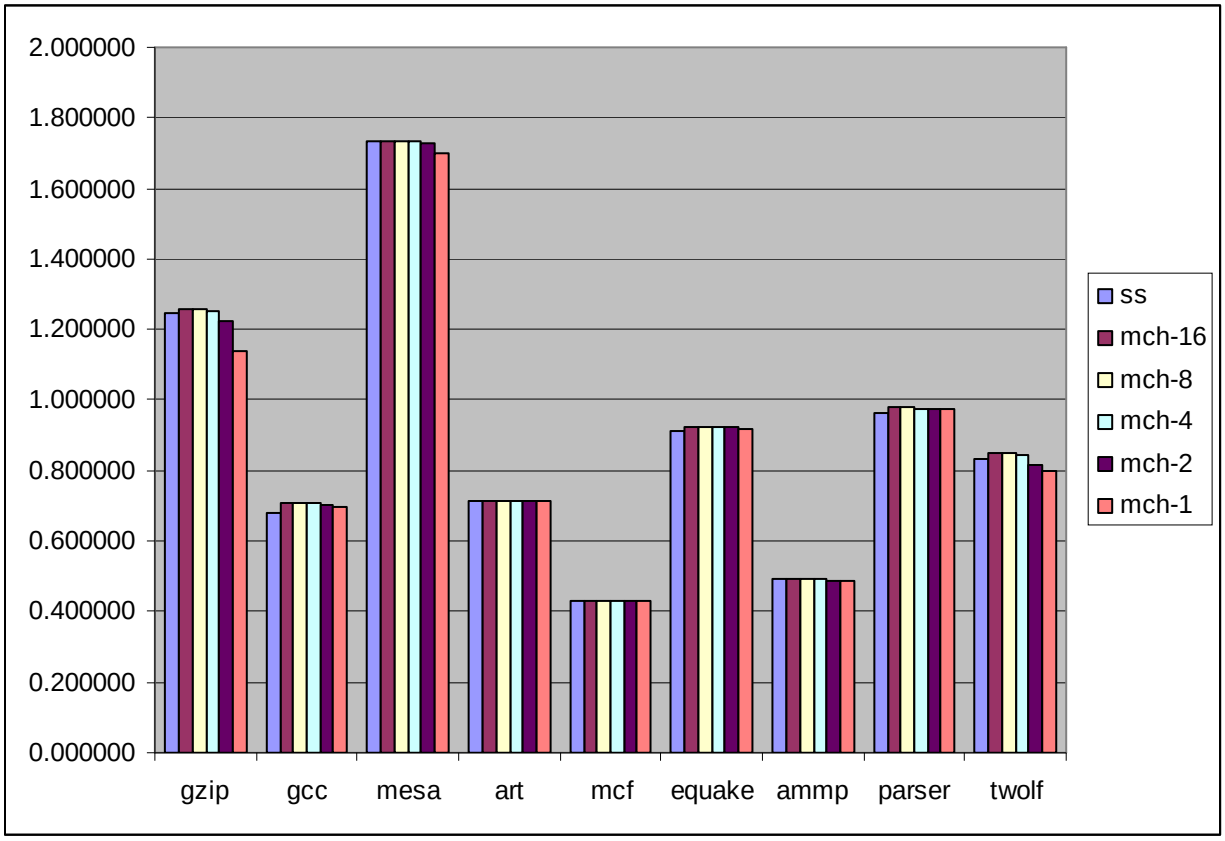

Figure 24 IPC of the original store-sets algorithm compared to MiniCAM-History proposal.

The following table summarizes the performance difference, on average, between store-sets alternative and MiniCAM-History algorithm.

Table 6 Performance difference between store-sets and MiniCAM-History.

	mch-16	**mch-8**	**mch-4**	**mch-2**	**mch-1**
Performance difference with store-sets	1.0105	1.0104	1.0083	0.9989	0.9808

These results is consistent with the results in the study of the SN-distance and TE-distance (sections 4.3.2 and 4.3.3 respectively), where we observe an extremely strong locality with SN-distances less or equal than 13 (Figure 5), and TE-distances less or equal than 10 (Figure 7).

5.8 Indirect issues affecting prediction behavior

We have observed an issue related to the way a violation is identified by the system. This identification is done immediately after a memory access instruction is issued. In this event, the system recovers the index of the load at the tail of the load-queue when the store was introduced in the store-queue. Then, the index is used to travel the load-queue (in the direction to the tail of the load-queue, meaning younger loads) trying to find one executed load accessing the same memory address. If a load is found, a memory dependence violation is declared.

There is a problem with this method. Let's put an example code to clarify the problem. Suppose the following sequence of instructions in a program:

#	Instructions	Order
1	ST @1	3
2	ST @1	1
3	LD @1	2

The first column is the instruction sequence number, the second is the instruction (with it accessed memory address) and the third column is the order the processor will try to issue them. It is clear that instruction 3 depends on instruction 2, not instruction 1. Nevertheless, when instruction 3 is issued, the processor will recognize a memory dependence violation against instruction 1. Even when this situation is very infrequent, this could affect negatively to any memory dependence predictor.

A possible solution is checking the stores younger that the current issued and older than the potential violator load, against that load. Using Figure 25, suppose that when store 1 was inserted in store-queue, load 1 was on the tail of the load-queue.

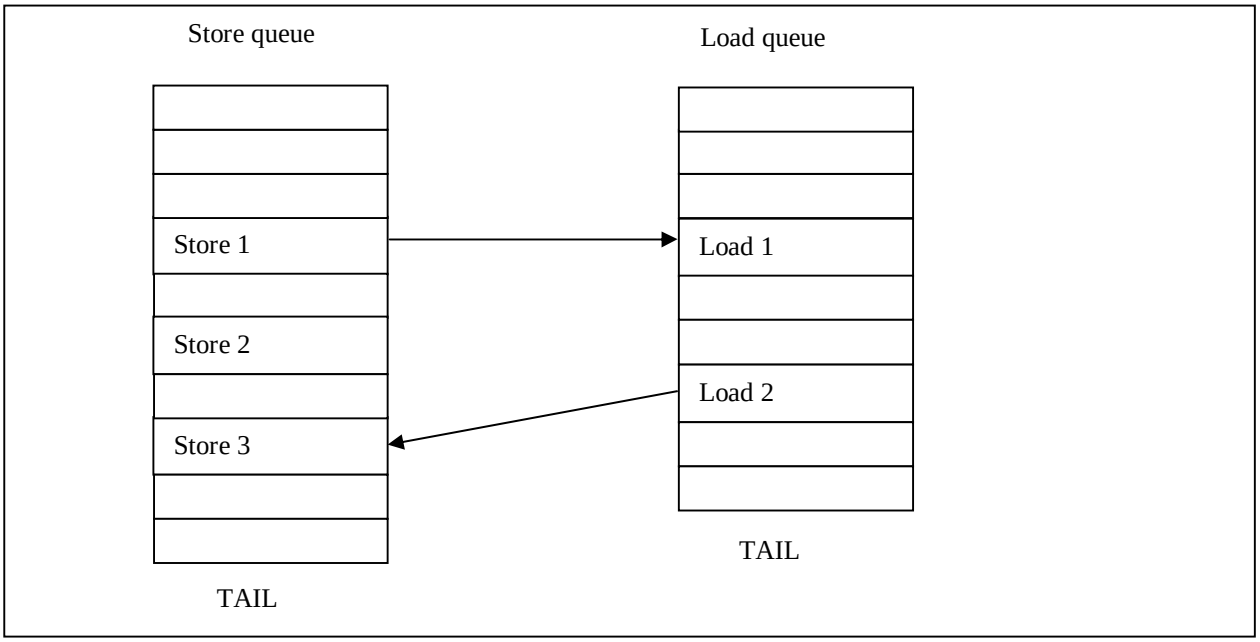

Figure 25 IPC of the original store-sets algorithm compared to MiniCAM-History proposal.

Later, after store 1 was issued, the system checks the list of loads in the load-queue, starting from the next load after load 1, and in tail direction. Suppose that load 2 matches the same accessed memory address than store 1. When load 2 was inserted in the load-queue, store 3 was in the tail of the store-queue. Then, the system must check all the stores between store 1 and store 3 trying to find one store that matches the same accessed memory address. For example, if store 2 matches, then no violation exist between store 1 and load 2. If no store matches, it exist a violation.

The modification for this new behavior is simple to implement in the simulator, and it is also expected a simple hardware implementation, but also a penalty in the *pipeline issue phase* time, because a *search* is executed.

This modification was tested against original store-sets implementation, but no important difference in the number of violations was observed. Figure 26 shows a comparison between this modification and the original. On average, the improvement is only 0.84%.

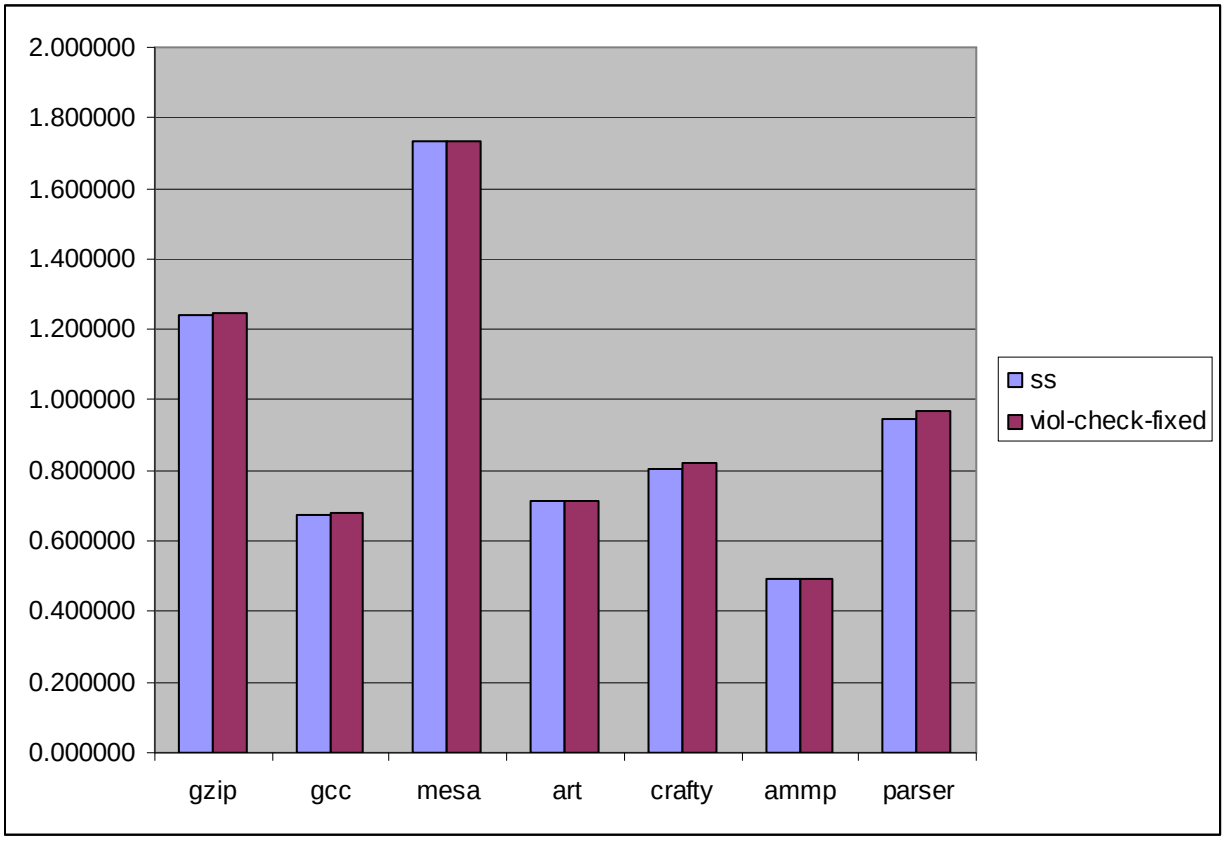

Figure 26 IPC of fixed check violations with original implementation.

5.9 Summary

In this chapter we have analyzed the behavior of the original store-sets algorithm. Some modifications have been tested, and a new non-speculative method was introduced, the MiniCAM. It was tested and compared, with performance very close to store-sets (see Table 6), but with less complexity and using fewer resources.

Between the store-sets modifications, deactivating old store-sets entries (using feedback) increase the number of violations, but reduce the number of false dependences created by the predictor. Because the false dependences reduction is bigger than the increase in violations, the final result is a small increase in the system performance. On the other hand, restoring the LFST after a squash reduces the negative effect of the branch predictor mistakes. In some benchs, feedback produces better results than LFST restore, and the opposite in others.

Finally, MiniCAM proved to be an interesting solution, with cheap implementation cost in complexity and expected very low penalty in the time increase in the issue pipeline.

Chapter 6. Conclusions

6.1 M5

The M5 simulator proves to be an excellent tool for researching. The internal organization is clear and it's easy to execute and trace an executed program. Modifications, when the original structure is not modified, are relatively easy to design and implement. However, it has some important drawback: (1) The poor documentation about its internal implementation, and (2) some internal functional disorganization, for example, some responsibilities to clean up internal data structures are spread along many stages in the pipeline, with clear comments of the authors declaring *not to be sure* if that is correct or not.

6.2 Experiment results

In memory dependence prediction, a good solution must produce good detection of real dependences and avoid false dependences. In the extreme case, a non-speculative solution will serialize all memory access instructions. This avoids completely violations and false dependencies but creates unnecessary contention and reduces the final IPC. On the other side, a naïve solution will issue instructions as soon as possible. This will produce many violations and a corresponding reduction of IPC, but no false dependencies. All prediction methods are between those extremes.

If a prediction algorithm takes too many risks, it tends to increase violations. If a prediction algorithm is too conservative, it tends to reduce violations, but it usually increasing false predictions at the same time (like store-sets without feedback). If the method tries to be more precise, it has to maintain more historic information and to use more complex algorithms, increasing the implementation cost. All presented methods try to find a good balance between these parameters.

We have study deeply the store-sets algorithm. Even when the number of violations not caught by this algorithm is very low, a high number of miss predictions reduce the potential IPC. The experiments performed show that including feed-back for disabling useless store-sets reduce significantly the miss prediction rate but increase the number of violations. This happens because each store-set is created by violations in a specific program section. When the execution passes to other sections, this never-disabled store-set tends to produce invalid predictions.

We found that table LFST use to become dirty after squashing a store instruction. Restoring the LFST improves store-sets to nearly predict all the memory dependencies, and also reduce some of the miss predictions. We test this technique in two different moments: when the processor notifies that a store will be squashed (squash-time), and when the squashed store is finally committed (commit-time). On average, restoring LFST on squash-time produces better results than restoring it on commit-time. This happens because, when a store activates a store-set, a younger store could disable it. A load that depends on the former store could be issued before the commit of the younger store. In consequence, if the younger store is squashed and we restore LFST at squash-time, the referred dependency could not be detected. In this case, a violation will arise.

Between the modifications implemented to the original store-sets algorithm, restoring LFST give the best results (see Figure 21). The additional complexity is very low and the right moment for applying it is squash time. However, this solution implies a penalty to the time of the pipeline state where the LFST cleaning is executed. Combining feedback with LFST restoring does not produce significant benefits (see Figure 21).

The proposed method, MiniCAM, exploits locality between very close instructions. The results obtained shows that this method gives a solution to memory dependence violations as good as the original store-sets algorithm, the most referenced method now a days. Focusing on very close instructions has the advantage to reduce the amount of resources needed to maintain historic data. MiniCAM has the additional advantage to have a very low implementation complexity and an expected very low execution time penalty because just one CAM, with a very small size, is enough.

Finally, the effect of bad branch predictions in the memory dependence prediction is too high to ignore it (see Figure 14). We think that coordination between both predictors is necessary.

6.3 Future work

Many ways of coordinating branch and memory dependence predictors are possible. One way is marking instructions as predicted or not predicted, at fetch time, depending on branch prediction unit state. The memory dependence predictor unit could ignore branch-speculated instructions, or consider them in an independent way. For example, in the store-sets algorithm, speculated instructions could create speculated sets, and could be used only for checking dependence against other speculated instructions. Another possibility with store-sets is managing separate tables, SSIT and LFST, for speculated and non-speculated instructions. Once a speculated branch is confirmed, tables could be merged. Another possibility is completely ignoring branch-speculated instructions for creating sets and check dependence with store-sets. To avoid *speculated violations*, memory access branch-speculated instructions could be forced to execute in-order.

On other side, testing benchmarks is a long time process, and in many cases, they are well known patterns of store-load layout in a binary code fragment that are reproduced in all benchmarks, producing the same effect. In consequence, there is a possibility to create a synthesized bench that reproduce those patterns, using independent and configurable parameters that enable us to select with patterns reproduce, and in which intensity. A good example of this idea is Eigenbench, a synthesized bench with focus on transactional memory [23].

Chapter 7. References

[01] Brad Calder and Glenn Reinman. 2000. **A Comparative Survey of Load Speculation Architectures**. Journal of Instruction-Level Parallelism 2000; 2:2000.

[02] George Z. Chrysos and Joel S. Emer. 1998. **Memory dependence prediction using store-sets**. In Proceedings of the 25th annual international symposium on Computer architecture (ISCA '98), Doug DeGroot (Ed.). IEEE Computer Society, Washington, DC, USA, 142-153.

[03] Changpeng Fang, Steve Carr, Soner Onder, and Zhenlin Wang. 2006. **Feedback-directed memory disambiguation through store distance analysis**. In Proceedings of the 20th annual international conference on Supercomputing (ICS '06). ACM, New York, NY, USA, 278-287.

[04] Freddy Gabbay and Avi Mendelson. 1998. **Using value prediction to increase the power of speculative execution hardware**. ACM Trans. Comput. Syst. 16, 3 (August 1998), 234-270.

[05] José González and Antonio González. 1997. **Speculative execution via address prediction and data prefetching**. In Proceedings of the 11th international conference on Supercomputing (ICS '97). ACM, New York, NY, USA.

[06] James Henry Hesson. 1997. **Apparatus to dynamically control the out-of-order execution of load/store instructions in a processor capable of dispatchng, issuing and executing multiple instructions in a single processor cycle**. United States Patent 5666506.

[07] Lipasti, Mikko H. and Wilkerson, Christopher B. and Shen, John Paul. 1996. **Value locality and load value prediction**. In Proceedings of the seventh international conference on Architectural support for programming languages and operating systems (ASPLOS-VII). ACM, New York, NY, USA.

[08] R. E. Kessler, E. J. McLellan, and D. A. Webb. 1998. **The Alpha 21264 Microprocessor Architecture**. In Proceedings of the International Conference on Computer Design (ICCD '98). IEEE Computer Society, Washington, DC, USA.

[09] Andreas Ioannis Moshovos. 1998. **Memory dependence prediction**. Doctoral dissertation. University of Wisconsin - Madison.

[10] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. 1997. **Dynamic speculation and synchronization of data dependences**. In Proceedings of the 24th annual international symposium on Computer architecture (ISCA '97). ACM, New York, NY, USA, 181-193.

[11] Soner Onder. 2002. **Cost Effective Memory Dependence Prediction using Speculation Levels and Color Sets**. In Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02). IEEE Computer Society, Washington, DC, USA, 232-.

[12] Soner Onder and Rajiv Gupta. 1999. **Dynamic memory disambiguation in the presence of out-of-order store issuing**. In Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture (MICRO 32). IEEE Computer Society, Washington, DC, USA, 170-176.

[13] Franziska Roesner, Doug Burger, and Stephen W. Keckler. 2008. **Counting Dependence Predictors**. In Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08). IEEE Computer Society, Washington, DC, USA, 215-226.

[14] Tingting Sha, Milo M. K. Martin, and Amir Roth. 2005. **Scalable Store-Load Forwarding via Store Queue Index Prediction**. In Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38). IEEE Computer Society, Washington, DC, USA, 159-170.

[15] Stone SS, Woley KM, Malik K, Agarwal M, Dhar V, Frank MI. **Synchronizing store-sets (SSS): Balancing the benefits and risks of inter-thread load speculation**. 2006.

[16] Samantika Subramaniam and et al. **Store Vectors for Scalable Memory Dependence Prediction and Scheduling**. 2006.

[17] Samantika Subramaniam and Gabriel H. Loh. 2009. **Design and optimization of the store vectors memory dependence predictor**. ACM Trans. Archit. Code Optim. 6, 4, Article 16 (October 2009), 33 pages.

[18] Samantika Subramaniam and Gabriel H. Loh. 2006. **Fire-and-Forget: Load/Store Scheduling with No Store Queue at All**. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39). IEEE Computer Society, Washington, DC, USA, 273-284.

[19] Gary S. Tyson and Todd M. Austin. 1997. **Improving the accuracy and performance of memory communication through renaming**. In Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture (MICRO 30). IEEE Computer Society, Washington, DC, USA, 218-227.

[20] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. 1999. **Speculation techniques for improving load related instruction scheduling**. In Proceedings of the 26th annual international symposium on Computer architecture (ISCA '99). IEEE Computer Society, Washington, DC, USA, 42-53.

[21] Intel Corporation. 1996. **Pentium Pro Family Developers Manual**. http://www.intel.com/design/archives/processors/pro/docs/242690.htm

[22] M5 Simulator System. http://www.m5sim.org/wiki/index.php/Main_Page

[23] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. 2010. **Eigenbench: A simple exploration tool for orthogonal TM characteristics**. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10) (IISWC '10). IEEE Computer Society, Washington, DC, USA, 1-11.