AN ABSTRACT OF THE DISSERTATION OF

Shih-Chang Lai for the degree of Doctor of Philosophy in Electrical and Computer Engineering presented on October 31, 2002.
Title: Tolerating Processor-Memory Performance Gap

Abstract approved:

Redacted for Privacy

Shih-Lien Lu

While the performance gap between microprocessors and main memory is ever increasing each year, cache memory has been a bridge to alleviate this discrepancy. In this thesis proposal, we introduce three techniques to tolerate this processor and memory speed imbalance. First, we propose the bloom filter scheme to identify which load operant could cause cache miss. Second, we explore a new fault-tolerant microarchitecture to detect transient error occurs. Third, we proposed a novel hardware-only mechanism to solve pointer-chasing problem in Link-list Data Structure application. The simulation shows that the bloom filter may filter out 99% of cache miss. The new fault-tolerant microarchitecture reduce the penalty caused by detecting instruction error about 1.8~13%. The hardware-only data prefetch mechanism accurate predict over 80% of irregular address pattern and improve the performance by 7%.

Tolerating Processor-Memory Performance Gap

By

Shih-Chang Lai

A DISSERTATION

Submitted to

Oregon State University

in partial fulfillment of
the requirements for the
Degree of

Doctor of Philosophy

Presented October 31, 2002
Commencement June 2003

Doctor of Philosophy dissertation of Shih-Chang Lai presented on October 31, 2002.

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head of the Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorized release of my thesis to any reader upon request.

Redacted for Privacy

Shih-Chang Lai, Author

ACKNOWLEDGEMENTS

The author expresses sincere appreciation to Professor Shih-Lien Lu and his wife for their unconditional support during this PhD program. Thanks to Professor Jih-Kwon Peir from University of Florida. For his continuous effort on guiding author in advance research technology and kindness during his stay at Beaverton, Oregon. In addition, thanks to all the members of Microprocessor Research Lab, Intel Corporation, Hillsboro, Oregon, especially Konrad Lai, Stark Jared and Krishnamurthy Ram. For their friendly and generous contribution to many published papers.

Most important of all, author would like to special thanks to his wife, Yang Ling Katherine Cheng. Without her, author could not possibly earn Doctoral degree.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF TABLES

Dedicated to my wife, my mother in heaven, father and brother, whose support was invaluable.

TOLERATING PROCESSOR-MEMORY PERFORMANCE GAP

CHAPTER 1 INTRODUCTION

As shown in Figure 1, while the performance gap between microprocessors and main memory is ever increasing 50% each year [1][2], cache memory has been a bridge to alleviate this discrepancy. In general, cache is made of 6-transistors SRAM cell[3][4] and two factors: cost and speed, determine the characteristic of cache in a system design. Hence, compared to main memory, cache has to be limited in a certain size. Because of this constrain, memory operations has become the critical path in the current pipeline processors.

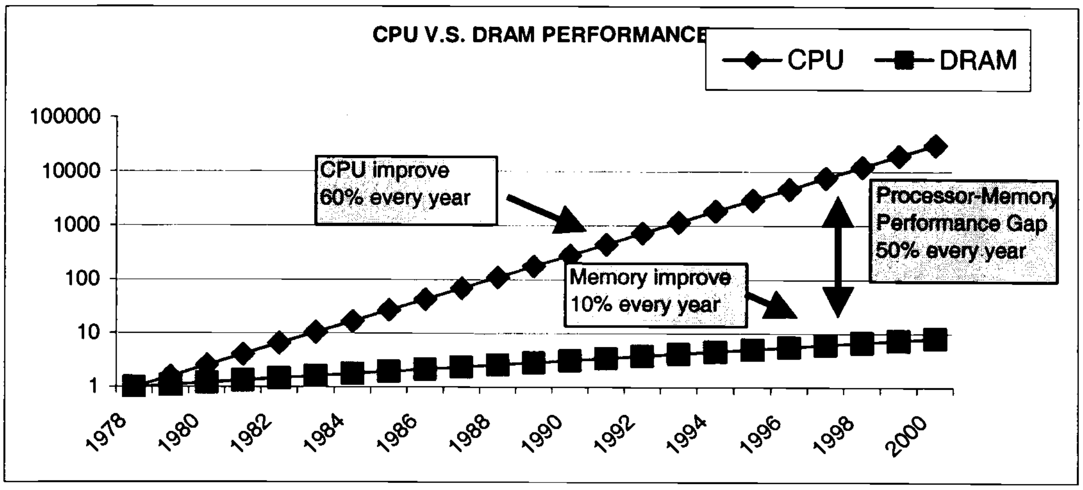


Figure 1 CPU vs. DRAM performance

In this thesis proposal, we introduce three techniques to tolerate this processor and memory speed imbalance. First of all, scheduling the dependent of load instructions is a major challenge because the memory access latency is not known until the address tag comparison is done. We propose the bloom filter scheme to identify which load operant could cause cache miss and schedule the dependent instructions accordingly. Second, as memory latency getting longer and longer, we propose a new fault-tolerant architecture design, named "ditto processor", to verify the correctness of previous committed instructions. Third, linked data structures (LDS) are increasing in importance due to the widespread use of object-oriented programming and application domains that involve large dynamic data structures. Hiding the memory latency incurred in traversals of such data structures, however, is notoriously difficult. We proposal the new scheme to identify these linked data structures as early as possible and prefetch the data from cache into a small buffer before it is needed.

1.1 BLOOM FILTER SCHEME

To achieve the highest performance, a processor must execute a pair of dependent instructions with no intervening pipeline bubbles. It must arrange for---or schedule---the dependent instruction to begin execution immediately after the instruction it depends on (i.e., the parent instruction) completes execution. Accomplishing this requires knowing the latency of the parent.

Unfortunately, a modern processor schedules an instruction well before it executes, and the latency of some instructions can only be determined by their execution. For example, the latency of a load depends on where in the cache/memory hierarchy the load data exists, and can only be determined by executing the load and querying the caches. At the time the load is scheduled, its latency is unknown. At the time its *dependents* should be scheduled, its latency may *still* be unknown. Hence, the timely scheduling of the instructions that are dependent on a load is a problem in modern processors.

The Intel Pentium 4 illustrates this problem. On an Intel Pentium 4[5][6], a load is scheduled 7 cycles before it begins execution. Its execution (load-use) latency is 2 cycles. At the time a load is scheduled, its execution will not begin for another 7 cycles. Two cycles after the load is scheduled, if the load will hit the (first-level) cache, its dependent instructions must be scheduled to avoid pipeline bubbles. However, two cycles after the load is scheduled, the load has not yet even started executing, so its cache hit/miss status is unknown. A similar situation exists in the Compaq Alpha 21264[7]. A load is scheduled 2 cycles before it begins execution, and its execution latency is 3 cycles. If the load will hit the (first-level) cache, its dependents must be scheduled 3 cycles after it has been scheduled to avoid pipeline bubbles. However, the load's cache hit/miss status is still unknown 3 cycles after it has been scheduled.

One possible solution to this problem is to schedule the dependents of a load only after the latency of the load is known. The processor delays the scheduling of the dependents until it knows the load hit the cache. This effectively increases the load's latency to the amount of time between when the load is scheduled and when its cache hit/miss status is known. This solution introduces unnecessary bubbles into the pipeline, and can devastate processor performance. Our simulations show that a processor using this solution drop 17% of its performance (in Instructions Per Cycle [IPC]) compared to an ideal processor that uses an oracle to perfectly predict load latencies and perfectly schedule their dependents.

A better solution---and the solution that is the focus of this work---is to use *data speculation.* The processor speculates that a load will hit the cache (a good assumption given cache hits rates are generally over 90%), and schedules its dependents accordingly. If the load hits, all is well. If the load misses, any dependents that have been scheduled will not receive the load's result before they begin execution. All these instructions have been erroneously scheduled, and will need to be rescheduled.

Recovery must occur whenever instructions are erroneously scheduled due to data (mis)speculation. Although misspeculation is rare, the overall penalty for all misspeculations may be high, as the cost of each recovery can be high. If the processor only rescheduled those instructions that are (directly or indirectly) dependent on the load, the cost would be low. However, such a recovery mechanism is expensive to implement. The recovery mechanism for the Compaq Alpha 21264 simply reschedules all instructions scheduled since the offending load was scheduled, whether they are dependent or not. Although it's cheaper to implement, the recovery cost can be high with this mechanism due to the rescheduling and re-execution of the independent instructions. Regardless of which recovery mechanism is implemented, as processor pipelines grow deeper and issue widths widen, the number of erroneously scheduled instructions will increase, and recovery costs will climb.

To reduce the penalty due to data misspeculations, the processor can predict whether the load will hit the cache, instead of just speculating that the load will always hit. The load's dependents are then scheduled according to the prediction. As an example of a cache hit/miss predictor, the Compaq Alpha 21264 uses the most significant bit of a 4-bit saturating counter as the load's hit/miss prediction. The counter is incremented by one every time a load hits, and decremented by two every time a load misses. Unfortunately, even with 2-level predictors[8], only about 50% of the cache misses can be correctly predicted.

In this study, we describe a new approach to hit/miss prediction that is very accurate and space (and hence power) efficient compared to existing approaches. This approach uses a *Bloom Filter* (BF), which is a probabilistic algorithm to quickly test membership in a large set using hash functions into an array of bits[9]. We investigate two variants of this approach: the first is based on *partitioned-address* matching, and the second is based on *partial-address* matching. Experimental results show that, for modest-sized predictors, Bloom Filters outperform predictors that used a table of saturating counters indexed by load PC. These *table-based* predictors operate just like the predictor for the Compaq Alpha 21264, except they have multiple counters instead of just one. As an example, for an 8K-bit predictor, the Bloom Filter mispredicts 0.4% of all loads, whereas the table-based predictor mispredicts 8% of all loads. This translates to a 6%

improvement in IPC over the table-based predictor. Compared to a machine with a perfect predictor, a machine with a Bloom Filters has 99.7% of its IPC.

1.2 FAULT TOLERANT MICROARCHITECTURE

Transient errors, also called soft errors, can be introduced by alpha or neutron particles strikes. They can also be introduced by power supply disturbances or other environmental variations. As supply voltage scales to accommodate technology scaling and to lower power consumption, transient errors are more likely to be introduced[19][20][21][22][23]. Transient errors may affect microprocessors in many ways[24][25]. One possible manifestation of soft errors in the modern processor is undetected data corruption. Experiments done by injecting faults into unprotected microprocessors resulted in the observation of non-negligible risk of data corruption[26]. Soft errors cannot be detected by manufacturing testing nor by periodic testing. With widespread usage of microprocessors in critical financial data processing, it is desirable to have microprocessors capable of transparent recovery and protection from data corruption in the face of soft errors.

The basic idea behind any error tolerance schemes involves some type of redundancy. Redundancy techniques can be categorized in three general categories[27][28]: (1) hardware redundancy[29], (2) information redundancy[30][31] and (3) time redundancy[32][33]. Hardware redundancy employs physical duplication and achieves redundancy spatially. Information redundancy with error detection and correction coding is effective in protecting memory elements against transient faults. Transient faults that occur in the logic blocks have no easy way to increase immunity besides utilizing either hardware redundancy or time redundancy. Time redundancy re-executes operations with the same hardware and obtain redundancy temporally. Time redundancy can be performed at different levels of the microprocessor. Work done by Nicolaidis[34] proposed a way to duplicate in time at the circuit level. This method introduces a delay element between the combination logic and the pipeline register allowing the data to be latched twice at different time. At the microarchitecture level, time redundancy can be achieved by instruction re-execution or by check pointing and rollback[35]. At the software level it can be accomplished by statically duplicating the program in multiple versions[36]. It assumes that if one version fails, other versions will produce correct results. In this study, we focus on the microarchitecture level of time redundancy technique.

Existing microarchitecture level time redundancy mechanisms lose performance due to blindly duplicating the execution of instructions at either decode stage[37][38][39][40] or at commit stage[41][42][43][44]. Both schemes verify the result at the point when the original copy is ready to retire and redundant copy has completed execution.

1. Duplicating the instructions at decode stage generates many unnecessary instructions to consume hardware resource when branch mis-prediction occurs.

2. The second scheme stored the committed instructions to a buffer in program order. This buffer provides the information of retired instructions to the fetch units. The instructions would then be re-fetched, re-decoded, re-renamed and re-executed.

The main drawback of the first scheme is that it does not cover faults that may occur at the frond-end of the pipeline. The second approach is commonly used in Simultaneous multithreading (SMT) based fault tolerant processors. They have better fault coverage compare to the first approach. However, if they only have limited resources, the performance degradation of the second scheme is worse than the first. The main reason is that the second scheme reduced the instruction bandwidth available to the original instruction stream[38]. Since long latency operations tend to stay in reorder buffer longer than short latency operation, our study reveals that the long latency operations are important factor to the performance loss of both schemes. Here we categorize memory reference micro-operation, multiply and divide operations as long latency instructions and the rest, including data effective address calculation, are short latency operations.

In this study we proposed a *Ditto Processor* to combine the advantages of two previous schemes and still be able to reduce the performance loss needed for reliable computing. It achieves the goal by handling short and long latency instructions in slightly different ways. After the instructions are decoded, long latency instructions would speculatively execute twice and the results are compared before instructions committed. All instructions are cloned when they are ready to retire. The duplicated instructions are held in a buffer and send back to the beginning of the pipeline. Since results of long latency instructions are checked, the clones of these instructions would not pass execution stage again after renaming operation are verified. For the clones of short latency operations, once they completed the re-execution, results are compared with the results of original instructions. If the results of any types instructions do not match with their clones, processor rollbacks to the point prior to the execution of these instructions.

This approach is unique in several ways:

1. It does not require SMT support and the operation system needs not to be aware of the duplicated instructions.

2. The entire pipeline except the commit stage is covered instead of just functional units. Commit stage must be duplicated in order to have full coverage.

3. Detecting the transient fault of short and long latency instructions in different ways and having fewer penalty cycles for fault recovery help to reduce performance loss. Our simulation result shows 1.8~13.3% performance degradation.

This study attempts to quantify and compare performance degradation of various time redundant schemes using a microarchitecture simulator when faults are present. It assumes transient faults are few and occur only as isolated single event. When a fault occurs during the simulation, it is always detected.

Performance lost due to re-execution and accounting is logged. This study does not guarantee schemes used will detect all faults.

1.3 POINTER-CHASING PROBLEM

As performance gap between processor and memory wider each year, data prefetching technique has proven to be the best way to tolerant this gap. A well-designed data prefetching mechanism will correctly predict the load address and bring the data into small memory structure, such as buffer or cache. How to accurately predict the location of data and have it ready in time at Level One Cache are two major concerns of data prefetching methodology.

If an application present regular address reference characteristic, a larger cache size may easily solve the cache miss problem without data prefetching. However, a larger cache would increase the access latency and it degrades the processor core performance. Rather than increasing the size of cache, a stride address predictor keep track of the history stride patterns, predict the future address and prefetch the data into cache.

For some other applications, such as C++, JAVA or other object-oriented program, the data structure may be dynamically, or statically, allocated and deallocated. These types of program usually define an object as a node and use pointer variable to link different objects or nodes together to build the complex data structure, named Link-list Data Structure (LDS). LDS may be single link-list, double link-list, tree or even multi-way tree. The most advantage of using LDS is that the structure may be dynamically expended or shrink based on the run-time program execution requirement. However, its advantage turns out to be a challenge job for microarchitecture researcher. First of all, in order to access a specific node, the program has to go through the entire predecessor nodes until it reach the target. For instruction, without the knowledge of every node's location, if one try to find a particular information in a tree structure, the only way is to travel from root and then goes down to each level until it find the desired information. Second, when an object is allocated at run-time, the Operation System may, or may not, provide an available virtual memory space, which is near its predecessors. It means that for a LDS, all nodes may spread all over the entire virtual memory. The stride predictor will not be able to catch this kind of abnormal address pattern. "Pointer Chasing" is usually a term to represent the problem of LDS application.

In this study, we propose a novel hardware-only data prefetch mechanism to solve pointer-chasing problem. Our motivation is that when a LDS nodes is allocated, it location will not change throughout its lifetime until it became trash. Therefore, we use a small cache to dynamically profile all these nodes location. This mechanism first identifies which load operation would producer the node's address and only these loads, named Pointer load, and store operation may access the small cache. The output of small cache will be the potential base address of next nodes. After address computation, the data is prefetched in prefetch cache to not pollute Level One data cache (L1-D cache). Since, the size of L1-D cache is not sensitive to the LDS application, we choose to downsize the L1-D cache and append one small cache to store address and one cache to store prefetched data. Later on, our study will reveal that the performance of these three small caches together are 7% better than a single larger L1-D cache.

CHAPTER 2 BACKGROUND STUDY

2.1 DATA SPECULATIONS

2.1.1 The Fundamentals

To facilitate the presentation and discussion, we consider a baseline pipeline model that is similar to the Compaq Alpha 21264[7]. We modified the SimpleScalar out-of-order pipeline to match our baseline model for the performance evaluations. In the baseline model, the front-end pipeline stages are: instruction fetch and decode/rename. After decode/rename, the ALU instructions go through the back-end stages: schedule, register read, execute, writeback, and commit. Additional stages are required for executing a load. After decode/rename, loads go through schedule, register read, address generation, two cache access cycles, and an additional cycle for hit/miss determination (data access before hit/miss using way prediction[11]), writeback, and commit. Thus, there are a total of 7 and 10 cycles for ALU and load instructions, respectively.

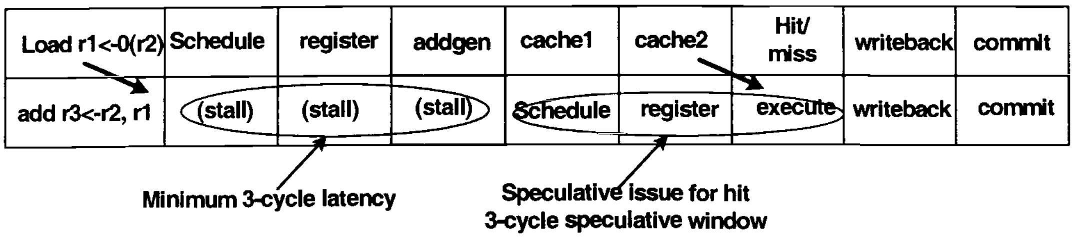


Figure 2 Example of Data Speculation for a load

Figure 2 illustrates the fundamental problem in scheduling the instructions that are dependent on a load. For simplicity, the front-end stages are omitted. In this example, the *add* instruction consumes the data produced by the *load* instruction. After the *load* is scheduled, it takes 5 cycles to resolve the hit/miss. However, the dependent *add* must be scheduled the third cycle after the load is scheduled to achieve the minimum 3-cycle load-use latency and allow back-to-back execution of these two dependent instructions. If the processor speculatively schedules the *add* assuming the *load* will hit the cache, the *add* will get incorrect data if *load* actually misses the cache. In this case, the *add* along with any other dependent instructions scheduled within the illustrated 3-cycle speculative window must be canceled and rescheduled.

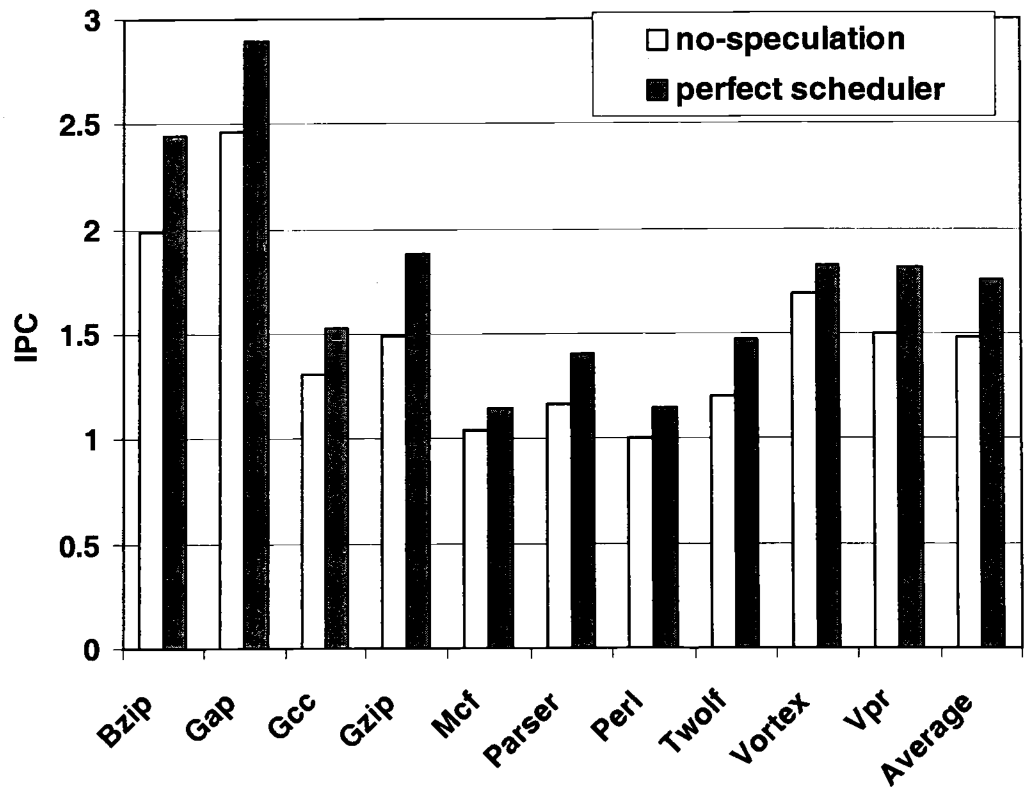


Figure 3 No-Speculation vs. Perfect Scheduling

To demonstrate the performance potential of using data speculation for scheduling instructions that are dependent on loads, we simulated the SPECint2000 benchmarks on the modified SimpleScalar model. We compare two scheduling techniques. The first is a *no-speculation* scheme: the dependents are delayed until the hit/miss of the parent load is determined. The second uses a *perfect* hit/miss predictor that knows the hit/miss of a load in time to schedule its dependents to achieve minimum load latency. The performance gap (in IPC) between these two extremes demonstrates the performance potential of speculatively scheduling the dependents of loads. **Figure 3** shows the results. In these simulations, we double the default SimpleScalar issue width to 8 and adjust the other parameters accordingly. A more detailed description of the simulation model will be given in Section X. On average, the IPC for perfect scheduling is17% higher than the IPC for the no-speculation scheme. Thus, the main focus of this study is to recover this 17% performance gap, by using mechanisms for efficient load data speculation.

2.1.2 Related Work

The Compaq Alpha 21264 uses a mini-restart mechanism to cancel and reschedule all instructions scheduled since a misspeculated load was scheduled[7]. While this mini-restart is less costly than restarting the entire processor pipeline, it is still expensive to reschedule (and re-execute) both the dependent and the independent instructions. To alleviate this problem, the Compaq Alpha 21264 uses the most significant bit of a 4-bit saturating counter as the load's hit/miss prediction. The counter is incremented by one every time a load hits, and decremented by two every time a load misses. The load's dependents are scheduled according to the prediction. If the prediction is wrong, either the load was predicted to miss and it hit, in which case the execution of the dependents will be unnecessarily delayed; or the load was predicted to hit and it missed, in which case dependents may have been erroneously scheduled and will need to be rescheduled.

Yoaz et al.[8] used 2-level local predictors, 2-level global predictors, and hybrid predictors for cache hit/miss prediction. Their results show that these predictors only correctly identify half of the misses (for SPECint95), laving the other half predicted as hits. Furthermore, they incorrectly identify a small percentage of the hits as being misses.

The MIPS R10000 speculatively issues instructions that are dependent on a load and reschedules them if the load misses the cache[12].

The Intel Pentium 4 achieves minimum 2-cycle load-use latency by leveraging the fact that most accesses hit the first-level L_1 cache. The scheduler issues the dependent micro-operations (called *uops*) before the parent load has finished executing[5][6].
In most cases, the scheduler assumes the load will hit the L_1 cache. A `*replay*` mechanism is used to handle the case where the load misses the L_1 cache. The replay logic keeps track of the dependent *uops* of each speculative load. When a load misses, all its dependent *uops* are re-executed with the correct data when that data becomes available.

Morancho, Llaberia, and Olive discuss a recovery mechanism for load latency misprediction[13]. A recovery buffer retains all speculatively scheduled instructions. After a latency misprediction, the load's dependent instructions can be re-scheduled directly from the recovery buffer as soon as the data becomes available. The recovery buffer allows the processor to remove instruction from the scheduler earlier, providing more space for other instructions.

2.2 BACKGROUND AND PREVIOUS WORKS ON FAULT TOLERANT MICROARCHITECTURE

Present single-chip Commercial-Off-The-Shelf (COTS) microprocessors have concentrated the design effort on performance. Reliability has not been the primary focus. However, some fault tolerant features have been added into COTS microprocessors [45][46].

Hardware redundancy is one possible approach to cover logic errors. The Pentium® Pro processor family has built-in mechanism to connect two processors into the master/checker duplexing configuration for functional redundancy checking. It allows duplicated chips to compare their outputs and detect errors. However, this technique required 100% or more logic overhead. Other hardware redundancy approaches adopted involve duplicating selected logic within the chip and include error-checking logic in all functional elements. IBM's G5 processor is a good example of this approach[45][47]. G5 duplicates its I-unit and E-unit. It incurs no delay penalty with the duplication because it is able to hide the compare-and-detect cycle completely. Therefore G5 achieves improved checking without any performance penalties. However, there is a 35% circuit overhead.

Recently there has been a resurgence of interest in utilizing time redundancy at the microarchitecture level to recover transient faults. We may classify these related works into two categories. The first category utilizes SMT mechanism to execute two redundant threads in a processor with SMT support. The second type focuses on modifying superscalar processor. We group several existing designs into these two time-redundant schemes.

2.2.1 Utilizing SMT mechanism in a SMT processor

Active-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) proposed by Rotenberg[41] exploits several recent microarchitectural trends to protect computation from transient faults and some restricted permanent faults. In this approach, a SMT processor executes an instruction stream called active stream (A-stream) first. Results committed from this instruction stream are stored in a delay buffer. A second stream (R-stream) of instructions tails behind the A-stream with a distance equals to the length of the delay buffer. Results from the R-stream execution are compared with results stored in a delay buffer and committed if they match. Since there are two threads being executed, there are two memory images maintained.

Recently, the same research group has proposed a new paradigm for increasing both performance and fault tolerance coverage called "slipstream". Instead of executing two exact instruction streams as in AR-SMT, slipstream processors' A-stream is shortened by the removal of ineffectual instructions. This approach[42] allows the A-stream to run ahead of the R-stream and thus provides not only fault-tolerant coverage but also performance improvement.

Work done by Reinhardt and Mukerjee on the Simultaneous and Redundantly Threaded (SRT) processor[44] also utilizes redundant thread in a SMT processor to detect faults. The SRT dynamically schedules the redundant thread to hardware resources to have higher performance. Their work introduces the abstraction called sphere of replication to identify the fault coverage. .

Rashid et. al. proposed fault tolerant mechanism in the Multiscalar Architecture[43]. Multiscalar processor usually has many processing units to exploit the instruction level parallelism (ILP). This technique utilizes a minor part of the processing units for re-executing the committed instructions. Both permanent and transient faults in the processor units can be detected.

2.2.2 Modified superscalar processor

Work done by Franklin[37] utilizes spare resources in a superscalar processor to implement time-redundancy. This approach duplicates all instructions at either the dispatch or the issue stage. Duplicated instructions occupy the otherwise under-utilized functional units to produce checking results for verification.

Nickel et. al.[48] extended Franklin's work and tried to improve performance of time-redundant processors by adding spare capacity. After an instruction completes execution but before it is retired, a duplicated copy is placed in a FIFO queue. This duplicated instruction is re-scheduled and re-executed. In order to minimize the performance loss, this method also strategically adds extra functional units to the pipeline.

Ray et. al.[39] proposed a similar scheme to what Franklin has done. A single instructions stream creates multiple redundant threads at decode stage and results from duplicated threads are verified at commit stage.

Mendelson et. al.[38] mentioned that if the decoding logic is not implemented by table lookup (memory structure) one needs to employ some methods to protect it from transient errors also. However, their approach focused on re-executing the operations twice at execution stage and verifying results before instruction commit. This scheme has minimum hardware requirement to perform error checking and has less performance impact due to error detection. However, compare to previous studies, this scheme has less fault coverage in that it only verified the correctness of functional units.

Austin et. al.[40][49] introduced the concept of using a less complex checker named DIVA to verify faults. The DIVA checker can verify not only transient faults but also design faults. Moreover, the performance impact of this extra checking mechanism is less than 3%.

In summary, we found works on using SMT to detect fault have better fault coverage but suffer higher percentage performance loss. While works on using existing superscalar processor, they do not cover the fault that may occur at the frond-end of the pipeline. In this study, our goal is to provide a fault tolerant processor, which has low cost, low performance degradation and high fault coverage. We use a microarchitecture simulator to quantify the performance loss of several schemes.

2.3 PREVIOUS STUDY ON POINTER-CHASING PROBLEM

Generally, data prefetching technique may divide into two groups: sequential and irregular memory address reference. Sequential reference may sub-divide into regular [53][54][55] and irregular stride addressing patterns [56] [57] [58] [59] [60] [61] [62] [63] [70]. The proposed solution to pointer-chasing problem may classify into three directions: hardware [65][66][67][68][69], software [72][73][74] and hybrid scheme [75][76][77][78]. Hardware-only mechanisms have two major advantages. First of all, with advance branch predictor, it may early resolve run-time memory reference address more accurately. Second, by monitoring memory port utilization, it may prevent normal data access hinder from incorrect data prefetching. Software mechanisms analyze the program code statistically and schedule data-prefetching operation carefully to match relative memory reference[58]. Hybrid mechanism combine hardware and software technique together to minimize the hardware requirement and still be able to gain high address prediction.

Our study is focus on processor-side hardware prediction mechanism to resolve the address of pointer load earlier on superscalar machine. For memory-side and multithreading pointer prefetl3ch study, interested reader may reference to [66][70][71][79]. We begin with describing several hardware-only approaches and then point out the fundamental problem of these solutions.

Amir Roth, et.al. [65] proposed to use Correlation Table (CT) to establish the link between producer and consumer load operation at run-time. The Potential Producer Window (PPW) records the possible producer that its result would be the consumer's base address. Whenever a load is committed, its base address lookups the PPW. If it matches the previous load operation's result, this producer-consumer link is saved into CT indexed by producer's Program Counter (PC). A completed load also probes the CT for existent link. For a match link, the predict address is generated and prefetched from the cache. The results shows 1~25% improvement

L. Ramos, et. al. [67] adopt Dependence Table (DT) and Link Table (LT) to find the possible linear link. DT contains producer load's tag and current time stamp to implement the address-computing tree structure. LT use 2-bits confidence counters to correctly identify the linear link. The result shows most of the benchmark may predict over 75% of address.

M. Bekerman, et. al. [68] use context-based predictor to track the possible LDS. This predictor contains Load Buffer (LB) and LT. LB records the history of recently used address per load and LT provide the predict address. The author also suggests that LB and LT should index by base address. Since base address may reduce the number of link required to perform address prediction and reduce the misprediction ratio. This study also use Gshare-based address predictor to identify control-based load. The result shows 67% of addresses are correctly predicted.

B.K. Chung, et. al. [69] suggest to use Register Update Table (RUT) and Update Link Table (ULT) to build the dynamic link of producer and consumer. RUT is index by prior-renamed register id. It records the least instruction that updates the register. ULT, similar to LT, records the link from the instructions that updates the base register to the consumer load instructions. This study constructs Cache-coordinate Resolution Table (CRT) to early predict the load address. The prefetched data is used to trigger load's dependent instructions. Combined with stride-based predictor, this mechanism accurately predicts 97% of load address and potential 18% performance improvement.

There are two fundamental problems for existent mechanisms:

1. Most of the schemes required two large tables to identify potential pointer load and predict its address. One table records on-the-fly load operation and another table keeps track of the link between producer and consumer load.
2. As deeper and deeper pipeline stages of modern microarchitecture are defined to accommodate higher circuit operation frequency, those proposed scheme would be lost its prediction accuracy. Since in some tight loop application, many iterations of loop operations are already fetched, decoded and renamed at Reorder Buffer (ROB). Those methods would not be able to prefetch the data in time before a load request it.

To overcome these two problems, we propose a new mechanism, named Pointer-Element-Prefetch-Unit (PEPU), to alleviate pointer-chasing performance impact. PEPU is composed of four-subunits, Target Register Bitmap (TRB), Bitmap Stack (BS), Prefetch Cache (PFC) and Address Cache (AC). These 4 subunits are controlled by finite state machine to perform predict and prefetch next data from L1 data cache. TRB is an n-bit table, where n is equal to the total number of architecture integer registers in a system. It serves to identify Pointer Loads. When branch miss prediction occur, BS rollback the TRB to correct state. The prefetched data is placed into PFC to not pollute the data cache. AC records the history of target address of pointer load. The size of PFC and AC combined together are smaller than L1 data cache size to gain faster access latency.

CHAPTER 3 BLOOM FILTERS

A *Bloom Filter (BF)* is a probabilistic algorithm to quickly test membership in a large set using multiple hash functions into an array of bits[9]. A BF quickly filters (i.e., identifies) non-members without querying the large set by exploiting the fact that a small percentage of erroneous classifications can be tolerated. When a BF identifies a non-member, it is *guaranteed* to not belong to the large set. When a BF identifies a member, however, it is *not guaranteed* to belong to the large set. To put it more simply, the result of the membership test is either: it is definitely not a member, or, it is probably a member. In this study, we consider two variants of the BF for filtering cache misses: one based on *partitioned-address* matching, and the other based on *partial-address* matching. To simplify our discussion, we first assume both the BF and the cache use physical addresses. Afterwards, we will describe using virtual addresses.

3.1 PARTITIONED-ADDRESS BLOOM FILTER

Consider a cache line address with n bits (ignoring the offset bits). A large, direct-mapped array of 2^n bits is required to precisely record whether each cache line address is in the cache. To reduce the space and allow a quick access, a *partitioned-address* BF can be constructed. Instead of using the entire line address, the address can be split into m partitions, with each partition using its own array of bits. The result is m sub-arrays with $2^{n/m}$ bits, each of which records the membership of the respective address partitions of lines stored in the cache. A cache miss is identified when one or more of the address partitions for the address of a requested line does not belong to the respective address partition of any line in the cache. A *filter error* is encountered when a cache miss cannot be identified.

This situation happens when the line is not in the cache, but all *m* partitions of the line's address match address partitions of other cache lines. The *filter rate* represents the percentage of cache misses that can be identified.

Figure 4 illustrates how the *partitioned-address* BF works. A load address is partitioned, in this example, into 4 equally divided groups, *A1, A2, A3*, and *A4*. Each of the four address partitions is used to index separate BF arrays, *BF1, BF2, BF3*, and *BF4*, respectively. Each entry in the BF arrays contains the information of whether the address partition belongs to the corresponding address partition of any line in the cache. If any of the 4 BF arrays indicates one of the address partitions is absent from the cache, the requested line is not in the cache. Otherwise, the requested line is probably in the cache, but it's not guaranteed to be.

Given the fact that a single address partition can exist for multiple lines in the cache, the primary difficulty of the *partitioned-address* BF is to maintain the correct membership information. When a line is removed from the cache, an exhaustive search is necessary to check if the address partitions for the address of the removed line still exist for any of the remaining lines. To avoid such a search, each entry in the BF array contains a reference counter that keeps track of the number of cache lines with the entry's corresponding address partition. When a cache miss occurs, each counter for the address partitions for the address of the newly-requested line is incremented, while the counters for the address partitions for the address of the replaced line are decremented. A zero count indicates the corresponding address partition does not belong to any line in the cache. Although accurate, this counter technique requires extra space in the BF arrays for the counters along with adders to handle the updates. A similar idea has been considered to reduce the number of comparators for a set-associative cache[14] and to filter cache-coherence traffic in a multiprocessor environment[15].

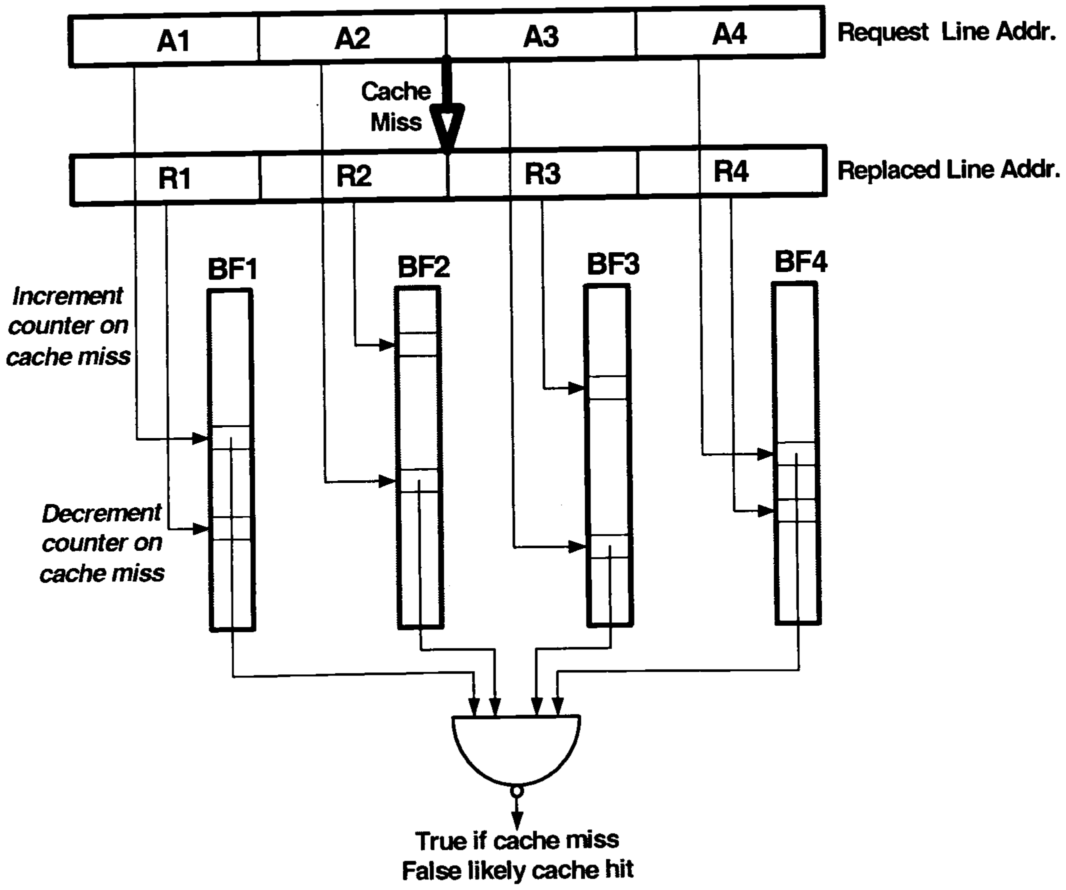


Figure 4 Partitioned-Address Bloom Filter for Cache Miss Detection

3.2 PARTIAL-ADDRESS BLOOM FILTER

The *partial-address* BF uses the least-significant bits of the line address to index a small array of bits. Each bit indicates whether the partial address matches any corresponding partial address of a line in the cache. The array size is reduced to 2^p bits, where p is the number of partial address bits. A *filter error* occurs when the partial address of the requested line matches the partial address of an existing cache line, but the other portion of the line address does not match. We call such cases *collisions*. The least-significant bits are selected rather than more-significant bits to reduce the chance of collisions. Due to memory reference locality, the more-significant line address bits tend to change less frequently. With a sufficient number of low-order partial address bits to represent cache line addresses, collisions are rare[16].

The design of a *partial-address* BF is illustrated in Figure 5. A *BF array* with 2^p bits indicates whether the corresponding partial address matches that of any cache line. The BF array is updated to reflect any cache content change. When a cache misses occurs, except for the caveat described in the paragraph below, the entry in the BF array for the replaced line is *reset* to indicate that the line with that partial address is no longer in the cache. Then, the entry for the requested line is *set* to indicate that a line with that partial address now exists in the cache.

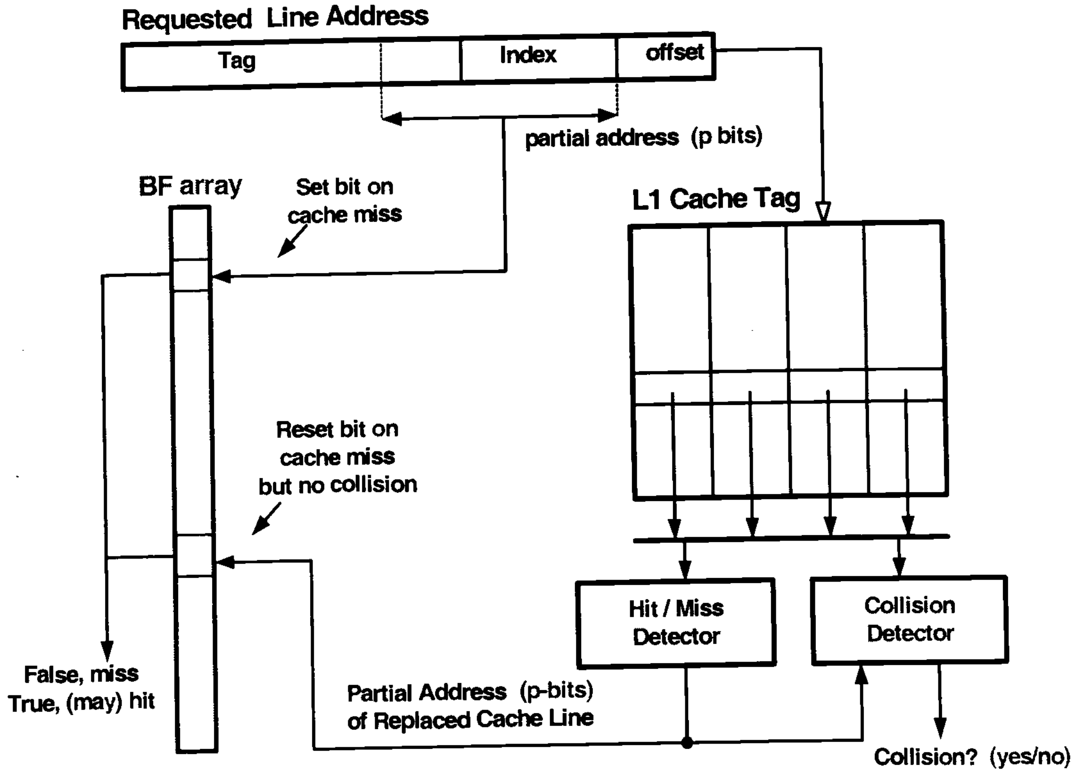


Figure 5 Partial-Address Bloom Filter for Cache Miss Detection

If the partial address is wider than the cache index, when two cache lines share the same partial address, they must be in the same set in a set-associative cache. The BF array indicates which partial addresses exist in the cache, so if one of these lines is replaced, the BF entry for the replaced line should *not* be reset, since the partial address still exists for the line that was not replaced. When a cache line is replaced, the *collision detector* checks the remaining cache lines in the same set as the replaced line to see if any of them have the same partial address as the replaced line. If any do have the same partial address, the BF entry is not reset. Otherwise, the entry is reset. The collision detection is done in parallel with the cache hit/miss detection. The BF array is updated on the detection of a cache miss.

3.3 BLOOM FILTERS USING VIRTUAL ADDRESSES

The hit/miss prediction for a load must be done before the scheduling of its dependents. If the physical address is not available in time to perform the prediction, the virtual address must be used. When a virtual address is used to access a BF, it is called a *virtual-address BF*. If the cache is virtually indexed and tagged, the virtual-address BF operates analogously to the BF and cache that both use only physical addresses. However, if the cache is either virtually-indexed physically-tagged or physically-indexed physically-tagged, the BF array update for the virtual-address BF must be modified. In this section, we describe these modifications.

With virtual addresses, two virtual addresses can map to the same physical address, causing an *address synonym*. With a virtual-address BF, the BF might identify the first address as missing the cache, even though the line is in the cache set identified by the second address. That is, the BF identifies a load as missing the cache even though it hits. This situation can arise regardless of whether the cache is physically or virtually indexed. In this situation, the processor simply delays scheduling the load's dependent instructions. Since cache hits by synonyms are rare, the performance loss caused by the delayed scheduling is minimal. In fact, for some virtually-indexed caches, the load-use latency for a synonym hit is longer than for a non-synonym hit. For scheduling, the processor may initially treat the synonym hit as a cache miss, in which case the BF should identify the synonym hit as a cache miss anyway.

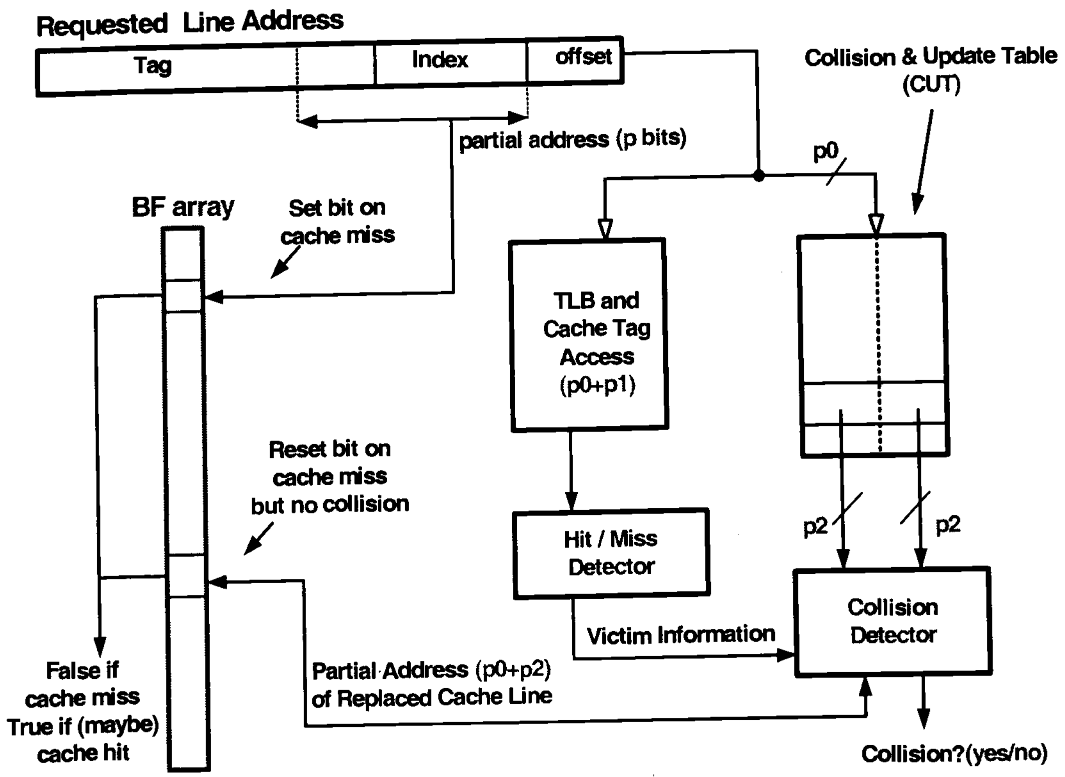

Requested Line Address
Tag
Index
offset
partial address (p bits)
Collision & Update Table (CUT)
p0
BF array
Set bit on cache miss
TLB and Cache Tag Access (p0+p1)
Reset bit on cache miss but no collision
Hit / Miss Detector
p2
p2
Collision Detector
Victim Information
False if cache miss True if (maybe) cache hit
Partial Address (p0+p2) of Replaced Cache Line
Collision?(yes/no)

Figure 6 Partial-Virtual-Address Bloom Filter for Cache Miss Detection

A more essential issue is correctly updating the BF array on cache misses. Let's first focus on the *partial-address* BF shown in Figure 6. To simplify our discussion, assume the cache is physically indexed and tagged with *p0+p1* index bits, where *p0* bits are within the page offset and *p1* bits are beyond the offset. During a cache access, *p1* bits are translated. Also assume *p0+p2* partial virtual address bits are used to access the BF, where *p2* bits are beyond the page offset. To correctly update the BF array, the *p2* bits of each cache line are stored in a

Collision and Update Table (CUT). When a line is replaced, its *p2* bits are read from the CUT. These *p2* bits are then combined with the requested line's *p0* bits to update the BF array.

The CUT is organized as a two-dimensional array and indexed by the *p0* bits. During each cache access, the set of *p2* bits indexed by *p0* are read from the CUT. If a cache miss is detected, the *p2* bits of the victim (e.g., LRU) line in the accessed cache set are compared to the *p2* bits for the other lines in that CUT set. If the victim's *p2* bits don't match any other line's *p2* bits, there is no collision, and the victim's *p2* bits are used along with the *p0* bits to reset the BF array to indicate that the line with the *p0+p2* partial address is no longer in the cache. If the victim's *p2* bits do match another line's *p2* bits, the victim and the other line share the same partial address, and there is a collision. In this case, the BF entry for the victim line is left alone. Then, the BF entry for the requested line is set using the partial virtual address of the requested line. Note that when the cache is virtually-indexed physically-tagged, all the cache index bits are used to access the CUT. In this case, only the partial address bits beyond the virtual cache index bits need to be saved in the CUT and compared for collision detection.

Handling a virtual *partitioned-address* BF is straightforward. Virtual address tags must be stored in the cache tag array along with the physical tags. When a line is replaced, the replaced line's virtual address tag is used to update the counter in each partitioned BF.

For the remainder of the study, we will assume virtual-address BFs. The virtual address needed to access the BF is available after the address generation cycle. Due to its rarity, we will omit discussions of synonym hits. If fact, for our benchmarks there are no synonyms.

3.4 THE MICROARCHITECTURE

In our baseline model, ALU instructions require a minimum of 7 cycles: instruction fetch (IFE), decode/rename (DEC), schedule (SCH), register read (REG), execute (EXE), writeback (WRB), and commit (CMT). Loads extend the execute stage to 4 cycles: address generation (AGN), two cache access cycles (CA1, CA2), and hit/miss determination (H/M). Assuming a load hits the L_1 cache, there is a 3-cycle speculative window in which the load's dependents and their children are scheduled. When a miss occurs, all of the dependent instructions and their children scheduled in these 3 cycles must be canceled and re-executed using the correct data when it becomes available.

3.4.1 Predictor Timing and Mini-Restart

If data cache misses can be predicted early enough and accurately enough, the processor's scheduler can avoid inserting pipeline bubbles between a load and its dependent instructions. To be effective, the load's cache hit/miss prediction must be done before its dependents must be scheduled. Thus, there are two basic issues: (1) when, and (2) how fast the hit/miss prediction can be performed. Hit/miss predictors that use saturating counters, like the one used by the Compaq Alpha 21264, can access the counter at the beginning of the pipeline. Since our pipeline has a minimum 3-cycle load latency, the prediction is available before any of the load's dependents need to be scheduled. If a miss is predicted, the dependents are blocked from scheduling until either the data comes back from the outer levels of the memory hierarchy or the prediction is found to be incorrect.

The proposed Bloom Filter approach, on the other hand, requires the load address to accurately identify (filter) misses. This filtering can only be performed after the load address is calculated in the address generation cycle. As shown in **Figure 2**, the load's dependent instructions must be scheduled the cycle after the load's address generation to avoid pipeline bubbles. By using a small BF, cache misses can be filtered in the cycle after the address generation, which is two cycles before the hit/miss determination. However, it is still one cycle too late to prevent the dependent instructions from scheduling.

To reap the prediction accuracy benefit provided by the BF, the load's dependents are always aggressively scheduled assuming a cache hit. At the end of the cycle the dependents are scheduled, the parent load has finished accessing the BF. If a miss is identified, the dependents are canceled and recovered in the next cycle. Since there is only a single-cycle speculative window, a precise recovery of the load's dependents may be feasible without excessive hardware complexity. This could be achieved by preventing the load's scheduled dependents from broadcasting their tags to their dependents, inhibiting the wakeup of their dependents. All independent instructions scheduled during this single-cycle window would be allowed to continue.

The Compaq Alpha 21264 has a similar precise recovery scheme to handle the dependents of floating-point loads. It also has a 3-cycle minimum load-use latency (1 for address generation and 2 for cache access). The cache hit/miss detection is done in the second cache access cycle, so the speculative window is only 2 cycles. The dependents of floating-point loads are always delayed from scheduling by one cycle. Consequently, the two-cycle speculative window for integer loads is reduced to a one-cycle window for floating-point loads. When the dependents of a floating-point load are being scheduled, the hit/miss detection is being performed in the same cycle. If a miss is detected, the dependents in this one-cycle window are precisely recovered in the next cycle[17]. This recovery should incur minimum penalty, as these dependents have to wait for the load data to return from the outer levels of the memory hierarchy anyway. The only potential adverse impact is that these dependents unnecessarily occupy functional units.

If a load is predicted to hit the cache, and it is later identified by the normal cache access as a miss, all dependent instructions scheduled during the entire 3-cycle speculative window have been or will be incorrectly executed. It is not sufficient to only re-schedule those instructions that directly depend on the load. Descendants of those dependent instructions may have been scheduled, and also need to be canceled and re-scheduled. A simple and workable scheme is to squash all instructions scheduled during the 3-cycle speculative window, as is done by the Compaq Alpha 21264. This simple recovery scheme reduces the hardware complexity needed to track all the dependencies and speculative states. However, both dependent and independent instructions scheduled in this 3-cycle window are canceled and re-scheduled. Independent instructions are rescheduled the cycle after the misprediction is detected. Dependent instructions are rescheduled according to the correct completion time of the load, which in most cases is determined by the level-2 cache access time.

Figure 7 illustrates the recovery mechanism for data mis-speculation. Again, the first two pipeline stages are omitted to simplify the figure. When the BF identifies a load as missing the cache, only those dependent instructions scheduled in the same cycle are canceled. The cancellation does not affect any independent instructions scheduled in this cycle, as shown in Part (a) of the figure. When a miss cannot be correctly filtered by the BF, and the miss is detected during the regular cache access, all of the instructions that were scheduled during the 3-cycle speculative window are canceled. Cancellation and re-execution involves resetting the canceled instructions' processor state. We assume it takes a separate *flush* cycle before the canceled instructions can be re-scheduled. Although independent instructions can be re-scheduled right away, they encounter a minimum 2--4 cycle penalty depending on where they reside in the speculative window. For example, a 4-cycle penalty occurs for those instructions that were scheduled in the first of the three speculative cycles as marked in Part (b) of the figure. Other factors such as data and resource dependencies may further increase the number of penalty cycles.

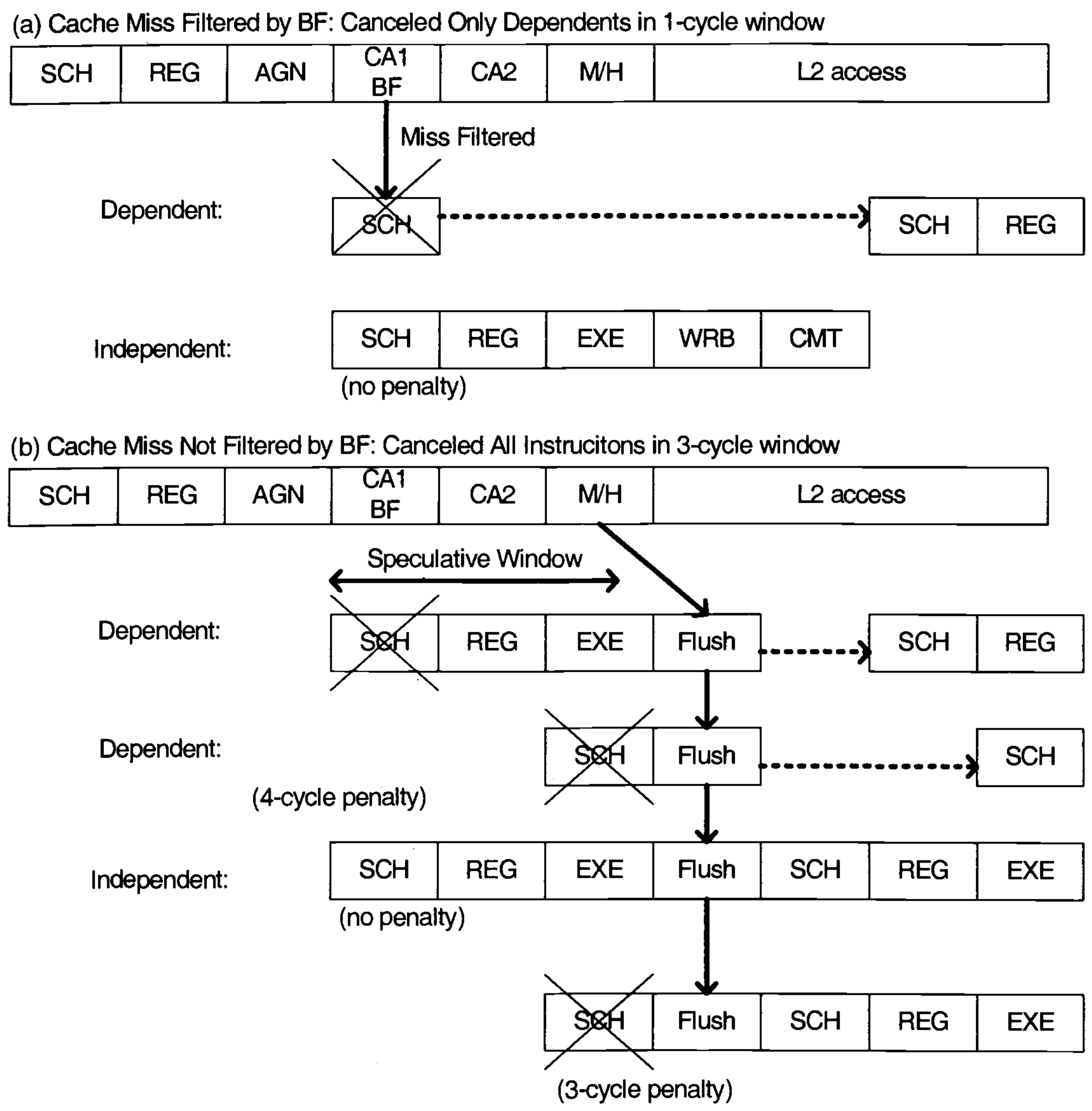


Figure 7 Recovery and Re-execution for:(a) Cache Miss Filtered by BF, and (b) Cache Miss not Filtered by BF

3.4.2 Prefetching and Memory Dependencies

Compared with other cache hit/miss predictors, the BF is unique in that misses that are identified must not exist in the L_1 cache. Therefore, once a miss is identified, it is safe to issue a miss request to the second-level cache L_2. In our pipeline model, this effectively reduces the L_2 cache and memory latencies by two cycles. Although other predictors also allow early L_2 cache access, they may incorrectly identify some L_1 cache hits as being misses, introducing extra penalties and complexity into the processor.

In our simulator, a Load-Store Queue (LSQ) is used to detect and enforce memory dependencies. It also allows loads to fetch data directly from an aliasing store in the LSQ without accessing the cache. The memory dependence is detected after the address of the load is generated (AGN). The load is forced to wait if the address of any potentially aliasing store in the LSQ is unknown. In our processor model, we assume this memory dependence detection is done early and accurately in the pipeline, which we model with a perfect memory dependence predictor. This allows the scheduling of the cache access to be inhibited if the load depends on a store in the LSQ. For the SPECint2000 benchmarks we tested, half of them have a very low percentage (1--3%) of loads which fetch data from the LSQ. However, the other half have higher percentages, indicating the importance of knowing memory dependencies before scheduling a cache access. If memory dependence detection can not be done early enough to avoid pipeline bubbles, cache accesses can be speculatively scheduled before memory dependencies are known. The speculative cache access---and any instructions dependent on the load that were scheduled/executed---are canceled and potentially re-scheduled and re-executed if a memory dependence is later detected. The BF may also be used in conjunction with a memory dependence predictor[18] to provide more accurate scheduling of loads and their dependents. Further discussion in this direction is out of the scope of this study.

3.5 PERFORMANCE EVALUATION

To evaluate the potential performance benefit of using a BF as a cache hit/miss predictor, we modified SimpleScalar to support BFs and other hit/miss predictors, and then ran most of the SPECint2000 benchmarks through the simulator. Our evaluation will compare the proposed BF technique to the other hit/miss predictors. Our simulated machine is a general-purpose out-of-order processor capable of issuing 8 instructions per cycle. The branch predictor consists of an 8K entry 4-way set-associative BTB and a 16 bit Gshare predictor. As described in Section 3.4, the pipeline is a minimum of 7 stages for ALU instructions and 10 stages for loads. A small 64-entry reorder buffer (called the RUU in SimpleScalar) was used for our studies as a larger instruction window may affect the cycle time. We modeled a detailed memory hierarchy, with the size and latency at each level reflecting current trends. We slightly modified the original SimpleScalar L_1 cache: instead of updating the cache tag array when a miss is detected, the tag array is updated when the missed data comes back from the outer levels of the memory hierarchy. This modification more accurately simulates cache misses, since the LRU line is not removed until the new data comes in.

Table 1 summarizes the simulation parameters. We simulated 10 of the SPECint2000 benchmarks using the reference input file. For each benchmark, we skip the first 500 million instructions and collect result statistics on the next 500 million instructions. We collect both prediction accuracy and IPC for the BFs and other cache hit/miss predictors.

Fetch/Decode/Issue Width	8
Branch Predictor	8K-entry 4-way BTB 16-bit Gshare
RUU/LSQ Size	64
L1 Inst/Data	16KB 4-way
L2 Cache	4MB 8-way
Access Latency: L1/L2/Mem	2/7/100
Memory Ports	4
Integer Add/Mult ALU	4/2
Floating-P Add/Mult ALU	4/2

Table 1 Bloom Filter simulation parameters

We simulated different sizes of the two BF variants. For the *partitioned-address* BF, we simulated three (*Partition-3*) and four (*Partition-4*) equal partitions of the line address (27 bits). Each entry in the BF array maintains a counter capable of counting the entire number of L_1 cache lines. To avoid overflow in our simulations, each counter was 10 bits. For the *partial-address* BF, the BF array size ranges from having only one entry per L_1 cache line (*Partial-1x*) all the way up to having 64 entries per L_1 cache line (*Partial-64x*). In our baseline model, the L_1 data cache is 16KB with a 32-byte line size. We also perform sensitivity studies on cache size.

We also evaluate two previously proposed hit/miss predictors and some simple extensions to them. The first is to always predict cache hit (*Always-hit*). This method does not require any prediction table. The second is the predictor in the Compaq Alpha 21264, which uses a single 4-bit saturating counter (*Counter-1*). We also evaluate using an untagged table of 4-bit saturating counters, indexed by the PC of the load. We vary the size of the table from 128 counters (*Counter-128*) to 8192 counters (*Counter-8192*). Since each counter is 4 bits, the total size of the

Counter-128 predictor matches the size of the *Partial-1x* predictor. For the counter-based predictors, the prediction is performed in the instruction fetch cycle, and the counters are updated after the cache hit/miss status is known. Table 2 summarizes the predictors we simulated and the amount of storage they require. Note that besides the predictor array tables, other logic such as adders and comparators are required to perform predictions.

Prediction Method	Array Size (in bits)
Partition-3	15360
Partition-4	4480
Partial-1x	512
Partial-4x	2048
Partial-16x	8192
Partial-64x	32768
Always-hit	0
Counter-1	4
Counter-128	512
Counter-512	2048
Counter-2048	8192
Counter-8192	32768

Table 2 Cache hit/miss predictors and their required storage

3.5.1 Prediction Accuracy

Figure 8 plots the filtering rates of the BFs. Recall the filtering rate is the percentage of misses identified (filtered) by the BF. In general, *partitioned-address* BFs performs poorly. The average filtering rate of *Partition-3* is only about 45%. *Partition-4* (not shown) has a dismal 5% average rate. Due to memory reference locality, the lowest partition of an address provides the most information, with the upper partitions providing almost no information. This is evident when comparing *Partition-3* to *Partial-1x*. The lowest partition of *Partition-3* uses the same 9 bits as *Partial-1x*. Yet the upper two partitions used by *Partition-3* only help to identify an additional 3% of misses. For *partial-address* BFs, the filtering rate improves dramatically as the size of the BF array increases. For a modestly sized 8K-bit BF array, the average filtering rate of *Partial-16x* is 97%.

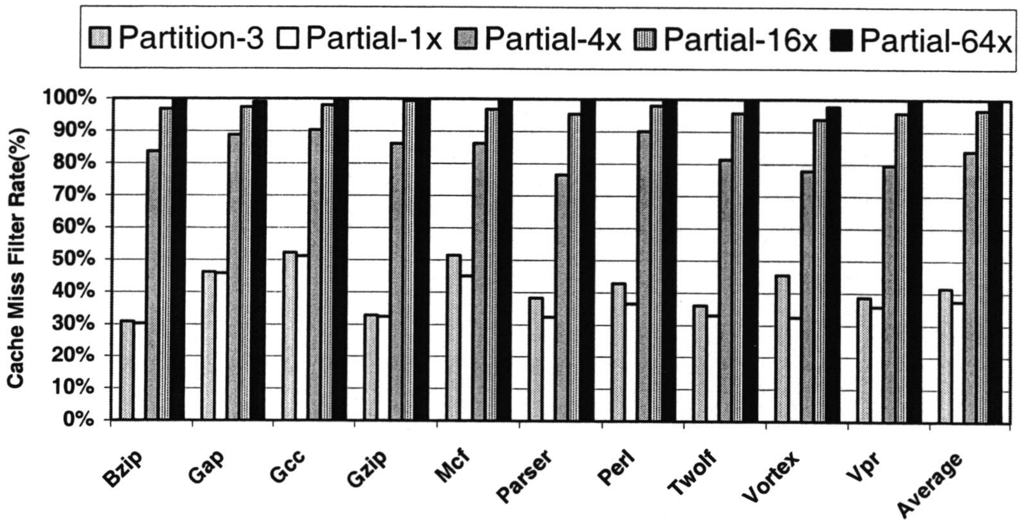


Figure 8 Cache Miss Filter Rate Using Partitioned and Partial Address Bloom Filters

Figure 9 shows the average (over all the benchmarks) correct and incorrect cache hit/miss prediction rates. It shows the prediction accuracy for BFs as well as other predictors. Correct predictions include both predict-hit-actual-hit and predict-miss-actual-miss cases. Incorrect predictions are separated into two groups. *Incorrect-cancel* is the case where a hit is predicted, but the load actually misses the cache. All speculatively scheduled dependents of the load must be canceled and rescheduled. *Incorrect-delay* is the case where a miss is predicted, but the load actually hits the cache. This misprediction unnecessarily delays the scheduling of the load's dependents and hence injects bubbles into the pipeline.

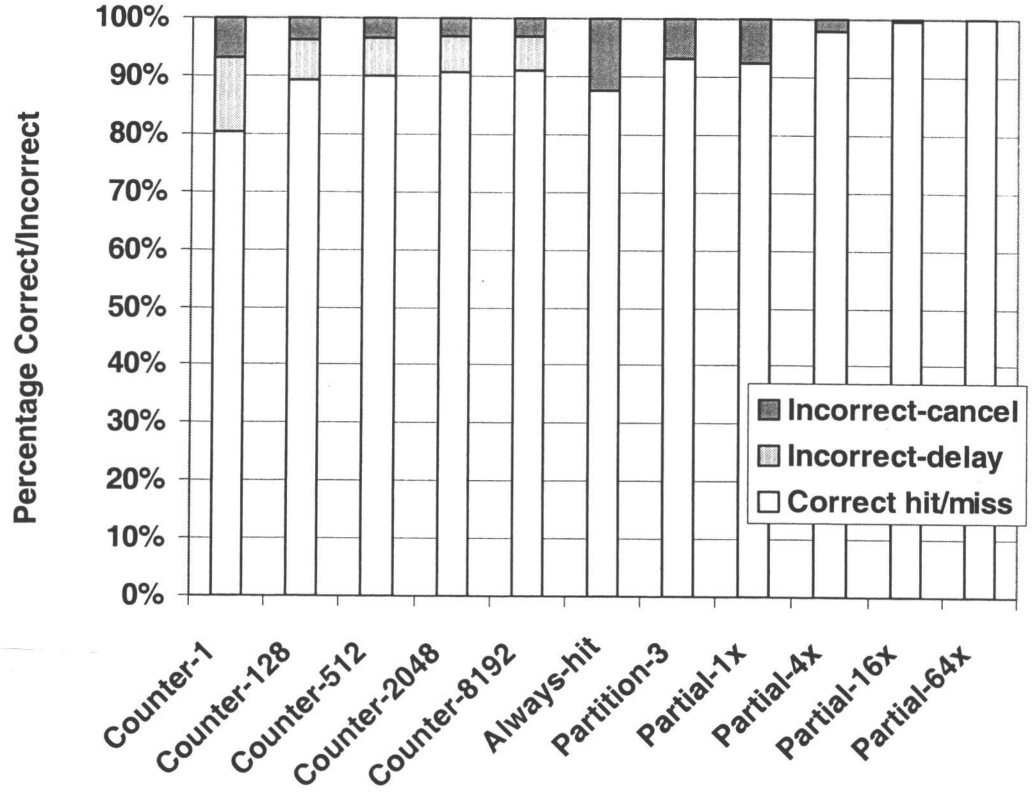


Figure 9 Prediction Accuracies for Different Cache hit/Miss Predictors

The predictors using saturating counters have a significant percentage of predictions in the *Incorrect-delay* group, and this percentage is insensitive to the predictor size. For *Counter-2048*, 5.2% of predictions are in the *Incorrect-delay* group and 2.7% are in the *Incorrect-cancel* group. The BFs, on the other hand, don't have any predictions in the *Incorrect-delay* group. In addition, the percentage in *Incorrect-cancel* decreases dramatically with larger BFs. The total misprediction rate is only 0.4% for *Partial-16x* using a moderately sized 8K-bit BF array. As expected, the simple *Counter-1* and *Always-hit* predictors have the two highest average misprediction rates.

3.5.2 IPC improvement

Figure 10 compares the IPCs for several data speculation methods. In addition to the different types of hit/miss predictors, we include the IPC of a machine that doesn't use any data speculation and of a machine that uses a perfect hit/miss predictor (*Perfect-sch*). Also, the benefit of data prefetching using *Partial-16x* and *Perfect-sch* are shown (labeled with *-DP* in the legend of the figure). We show results for all the individual benchmarks since the IPC improvements are very different among them. *Partial-16x* without data prefetching shows a 17\% improvement over *No-speculation* and a 4% improvement over *Always-hit*. Compared to *Counter-1* and *Counter-2048*, the improvements are 9% and 6%. With data prefetching, the improvements rise to 19%, 6%, 11% and 8%, respectively. It is important to point out that *Partial-16x* reaches 99.7% of the IPC of *Perfect-sch*, and *Partial-16x-DP* reaches 99.7\% of the IPC of *Perfect-sch-DP*.

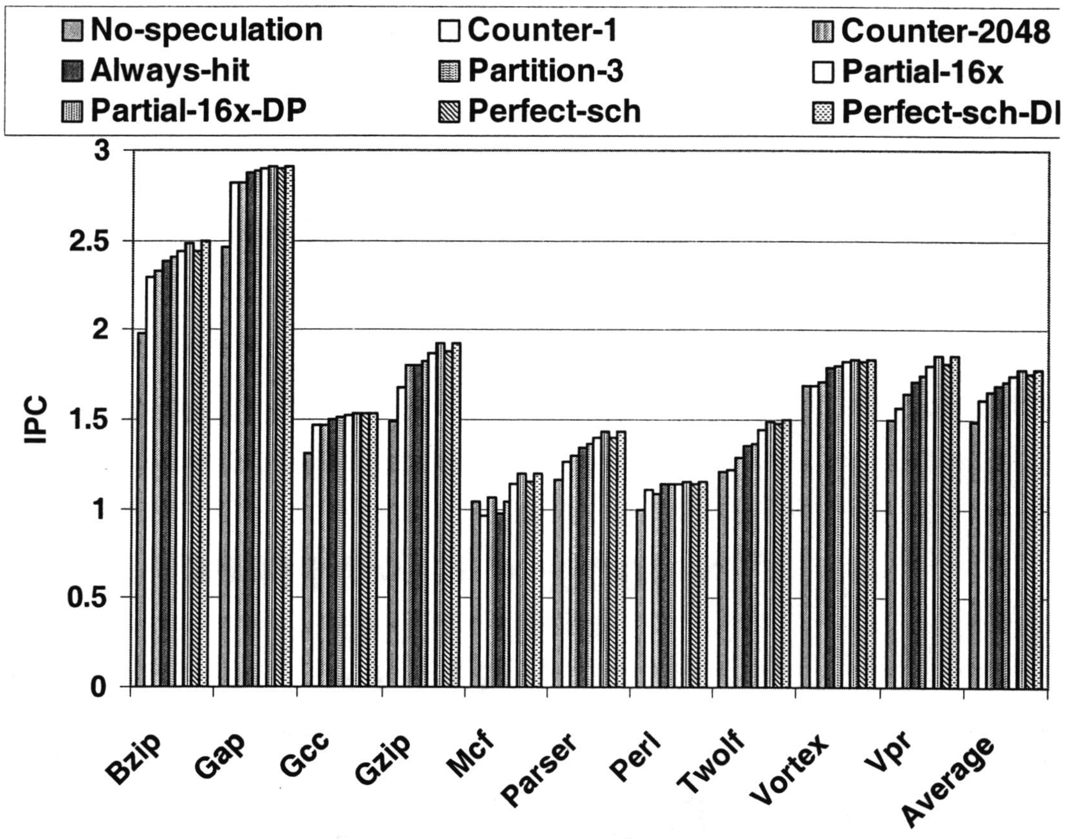


Figure 10 IPC comparisons for Different Data Speculation Methods

Among the benchmarks, *Gcc*, *Perl*, and *Vortex* show little difference between the different data speculation methods. Analysis reveals that these three programs have a large number of L_1 instruction cache (I-cache) misses. The high I-cache miss rate prevents instructions from entering the pipeline, reducing the benefit of data speculation. The lower instruction fetch rate greatly reduces the RUU occupancy, which is measured as the average number of RUU entries occupied. Since the RUU occupancy is much lower, loads and their dependents can stay in the RUU longer without blocking other instructions, so there is less of a difference between *No-speculation* and aggressive speculation such as *Partial-16x.*

I-cache	Gcc			Perl			Vortex		
	IPC%	I-miss%	Ruu-ocu	IPC%	I-miss%	Ruu-ocu	IPC%	I-miss%	Ruu-ocu
8k	7.7	6.3	15.5	5.3	8.5	14.3	1.5	10.5	14.7
16x	10.7	4.2	19.8	7.0	6.0	17.6	4.6	6.9	22.2
32x	16.3	2.0	27.6	13.6	2.9	26.4	7.9	4.3	30.5

Table 3 Percent IPC improvement, I-cache miss rate, and RUU occupancy for 3 I-cache sizes

Table 3 summarizes the performance improvement of *Partial-16x* over *No-speculation* for 3 different I-cache sizes. Note the IPC improvement grows as the I-cache size increases. The IPC improvement for *Always-hit* also grows as I-cache size increases (not shown in the table), but not as quickly as it does for *Partial-16x*.

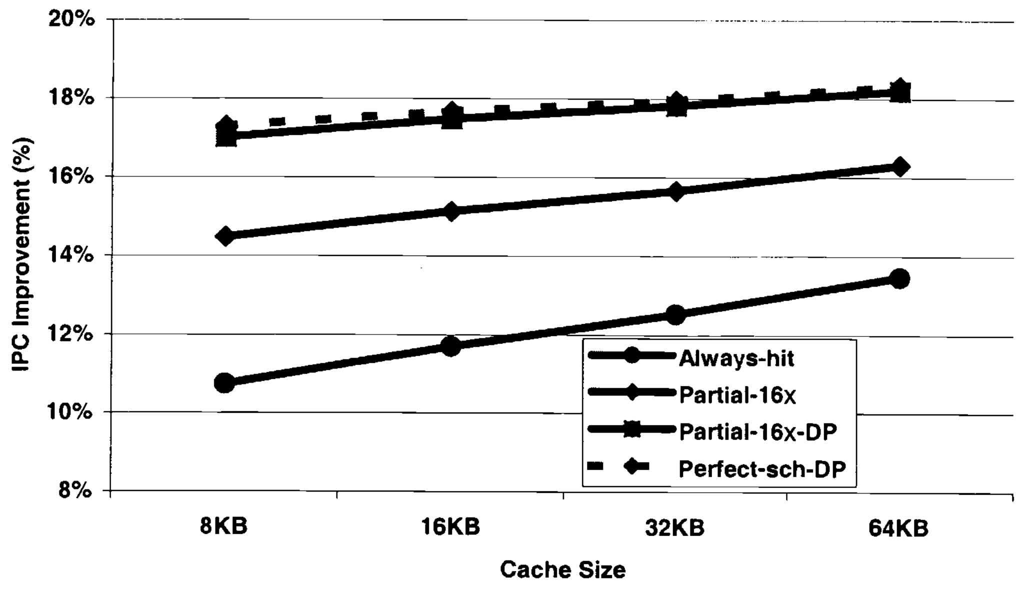


Figure 11 IPC improvements Over No-speculation for Different Data Cache Sizes

3.5.3 Sensitivity Studies

In this section we examine the effect of BFs on processor performance for various data cache sizes, RUU sizes, and different branch predictors.

Figure 11 plots the IPC improvement of *Always-hit*, *Partial-16x*, *Partial-16x-DP*, and *Perfect-sch-DP* over *No-speculation* for four different data cache sizes. We make three observations. First, the bigger the cache, the better the IPC improvement for all 4 data speculation methods. With bigger caches, scheduling becomes more important, because the performance bottleneck caused by data cache misses is reduced. Thus, delaying the scheduling of a load's dependents until after its cache hit/miss status has been determined (as is done by the *No-speculation* method) is a bigger loss of opportunity. Second, the IPC of *Always-hit* improves faster than the other methods as cache size increases. This is because its prediction accuracy is directly tied to the cache hit rate, so it sees the biggest improvement in prediction accuracy as the cache size increases. The IPC improvement of *Partial-16x-DP* over *Always-hit* reduces from 5.9% to 5.4% to 4.9% to 4.3% as the cache size is increased from 8KB to 64KB. Nevertheless, we expect future high-performance processors will use smaller first-level caches to enable higher clock frequencies. Third, due to high accuracy, *Partial-16x-DP* achieves
99.9% of the IPC of the machine with a perfect scheduler for large caches.

Figure 12 shows the IPC improvement of *Partial-16x-DP*, *Partial-16x*, and *Always-hit* over *No-prediction* for three RUU sizes: 32, 64, and 128. We make several observations.

First, the IPC improvement is the greatest for the small RUU for all three methods. To achieve high performance with a small RUU, instructions need to flow through the RUU freely. Without data speculation, instructions that are dependent on loads block the flow. Thus, data speculation---even with all the rescheduling of dependent instructions due to mis-speculations---is essential for high performance when the RUU size is small.

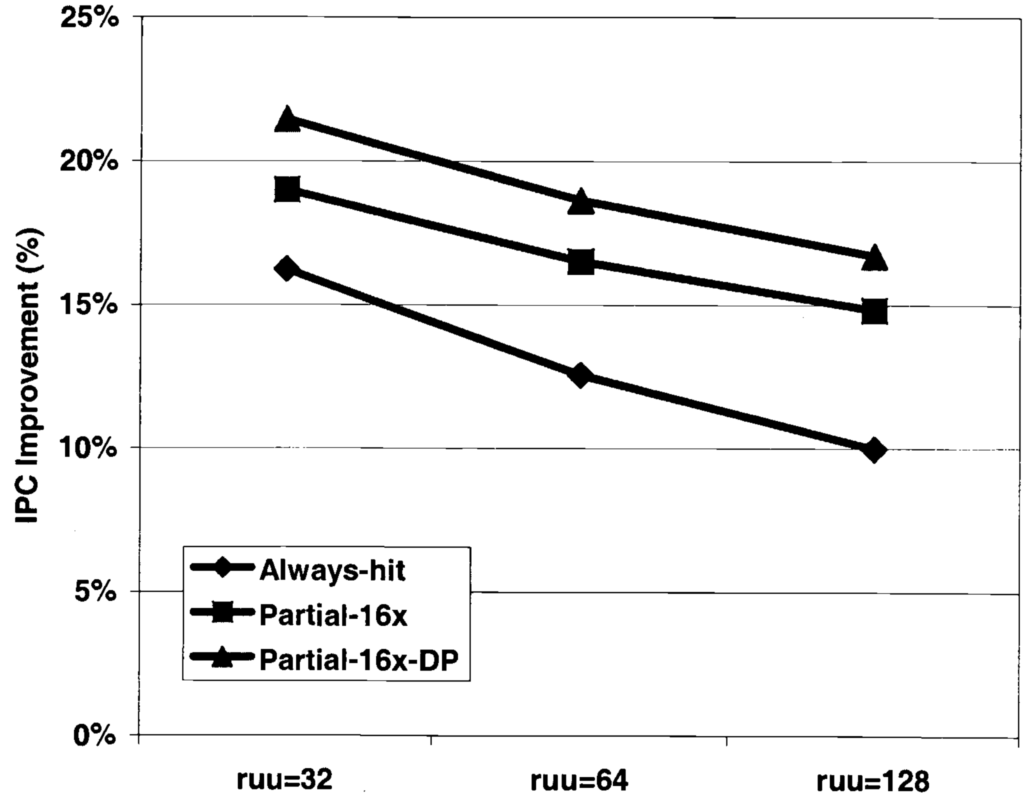


Figure 12 IPC improvement Over Always-hit for Different RUU Sizes

Second, immediately prefetching the data when the BF identifies a miss improves IPCs by an additional 2-3%.

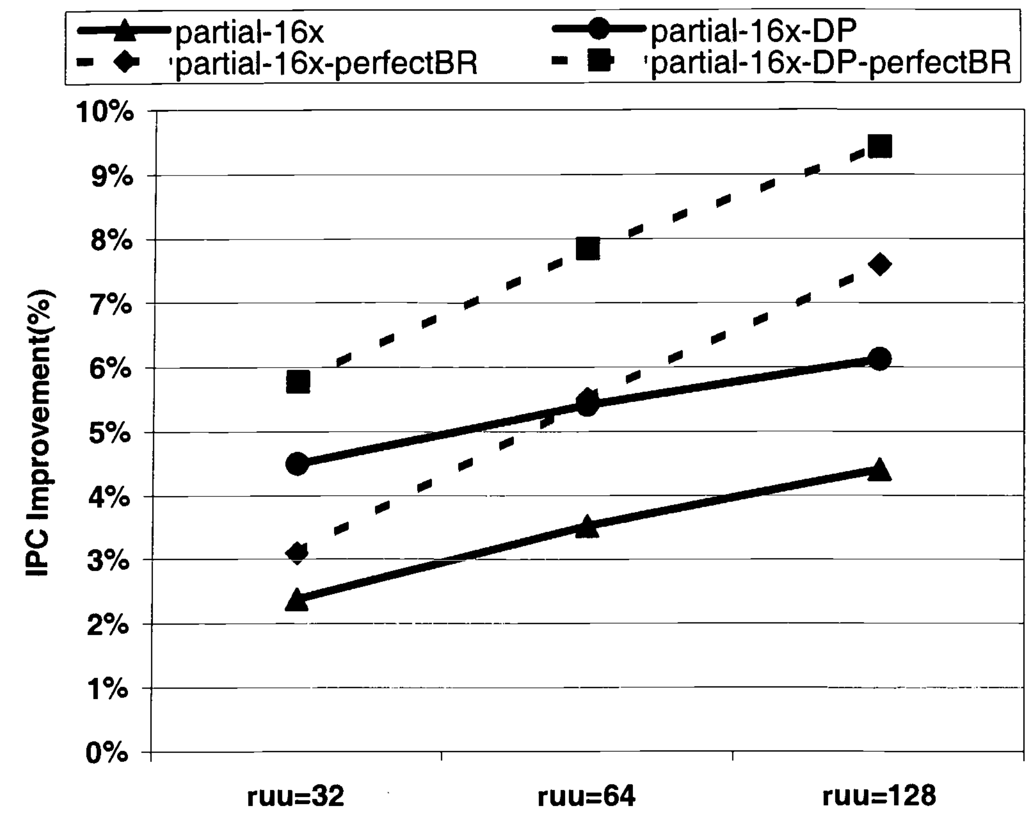


Figure 13 IPC improvement Over Always-hit for Different RUU Size

Third, the IPC improvement for *Always-hit* drops faster with increasing RUU size than the other two methods. And the performance gap between *Partial-16x* and *Always-hit* widens with bigger RUUs. To better illustrate this behavior, Figure 13 plots the IPC improvement of *Partial-16x-DP* and *Partial-16x* over *Always-hit*. In addition to the default Gshare predictor, the figure plots the performance of the two methods using a perfect branch predictor (labeled with -*perfectBR* in the legend of the figure). The results clearly show that the IPC improvement over *Always-hit* increases for bigger RUUs. Our simulation results show that with a small RUU, *Partial-16x* and *Always-hit* have similar RUU occupancies even though *Always-hit* produces more mispredictions. With a larger

RUU, *Partial-16x* produces fewer RUU-full stalls than *Always-hit*. Effectively, *Partial-16x* has a larger instruction window in which to find instruction level parallelism.

Lastly, in Figure 13, the IPC improvement of *Partial-16x* grows faster with increasing RUU size for the perfect branch predictor than for the default Gshare predictor. With a perfect branch predictor, the performance bottleneck due to branch mispredictions is eliminated, and instruction scheduling becomes more important. In addition, RUU occupancy is very high, since there are never any branch mispredictions that flush the RUU. A critical scheduling resource---RUU entries---becomes incredibly
scarce. *Partial-16x* makes better use of this critical resource than *Always-hit*, as it cancels and reschedules fewer instructions. For a 128 entry RUU and a perfect branch predictor, the proposed *partial-address* BF improves IPC by more than 9% over the *Always-hit* method. Note that as branch prediction technology improves, the performance characteristics of real processors approach the performance characteristics of processors with perfect branch predictors.

CHAPTER 4 DITTO PROCESSOR

4.1 DESIGN OF DITTO PROCESSOR

Ditto Processor differs from previous approaches in that it splits long latency operations and short latency operations into different verification path. After the instructions are decoded, long latency instructions are identified and speculatively executed twice. Results of these long latency instructions are compared but they are not committed. All non-speculative instructions including those non-speculative long latency instructions are cloned before retirement. These duplicated instructions are held in a buffer and send back to the beginning of the pipeline. Since the result of long latency instructions are executed twice and checked, the clones of these instructions would not pass execution stage again after renaming operation are verified.

For the clones of short latency operations, once they completed the re-execution, results are compared with the results of original instructions. Any transient fault can potentially be discovered when the result of the re-execution differs from that of the original execution and simple recovery scheme is used. If faults occurs at the decode stage, results will differ also and be detected.

Prior to our main study, we observe, in an average, only 12% of the resources are utilized for integer and floating-point applications on a baseline 8-issue superscalar processor. This means that there are plenty of opportunities to take advantage of these unused resources to hide the overhead of program re-execution, verification and transient fault recovery. However execution in a superscalar tends to be busty at times. Without careful organization, time-redundancy through cloning still degrades its performance.

In the following sections we will describe in more details the design of Ditto processor. After reading through the design details, interested reader may find an example of pipeline flow for a small piece of sample code in the appendix.

4.1.1 What Hardware is added to support fault tolerant mechanism

Figure 14 illustrates the basic microarchitecture diagram of *Ditto processor.* It has two additional blocks - a "delay buffer" and a "verify logic". Several existing blocks in a superscalar processor also need to be modified. These include the re-order buffer (ROB), the commit logic, the fetch unit and the decode unit. We describe the changes needed for each of these blocks.

Delay Buffer: Instructions are executed normally the first time. Results of committed instructions are queued in the delay buffer similar to other schemes [41][42][48]. However, each entry not only stores the result but also includes the associated instruction code and its instruction address. For long latency operations, we also allocate the immediate entry that follows to store source operands' values. We called these instructions stored in the delay buffer *cloned instructions*. These cloned instructions are removed from the delay buffer when they are scheduled and passed the registered read stage.

Fetch and decode units: Since the gap between processor cycle time and memory access time will likely grow wider each year, most likely fetch and decode units are not the bottleneck. We choose to split the fetch and decode units into two equal parts. Half of the fetch and decode unit is reserved for cloned instructions stream. In order to simplify the maintenance of normal instructions and cloned instructions stream, an extra program counter is added for the cloned instructions stream.

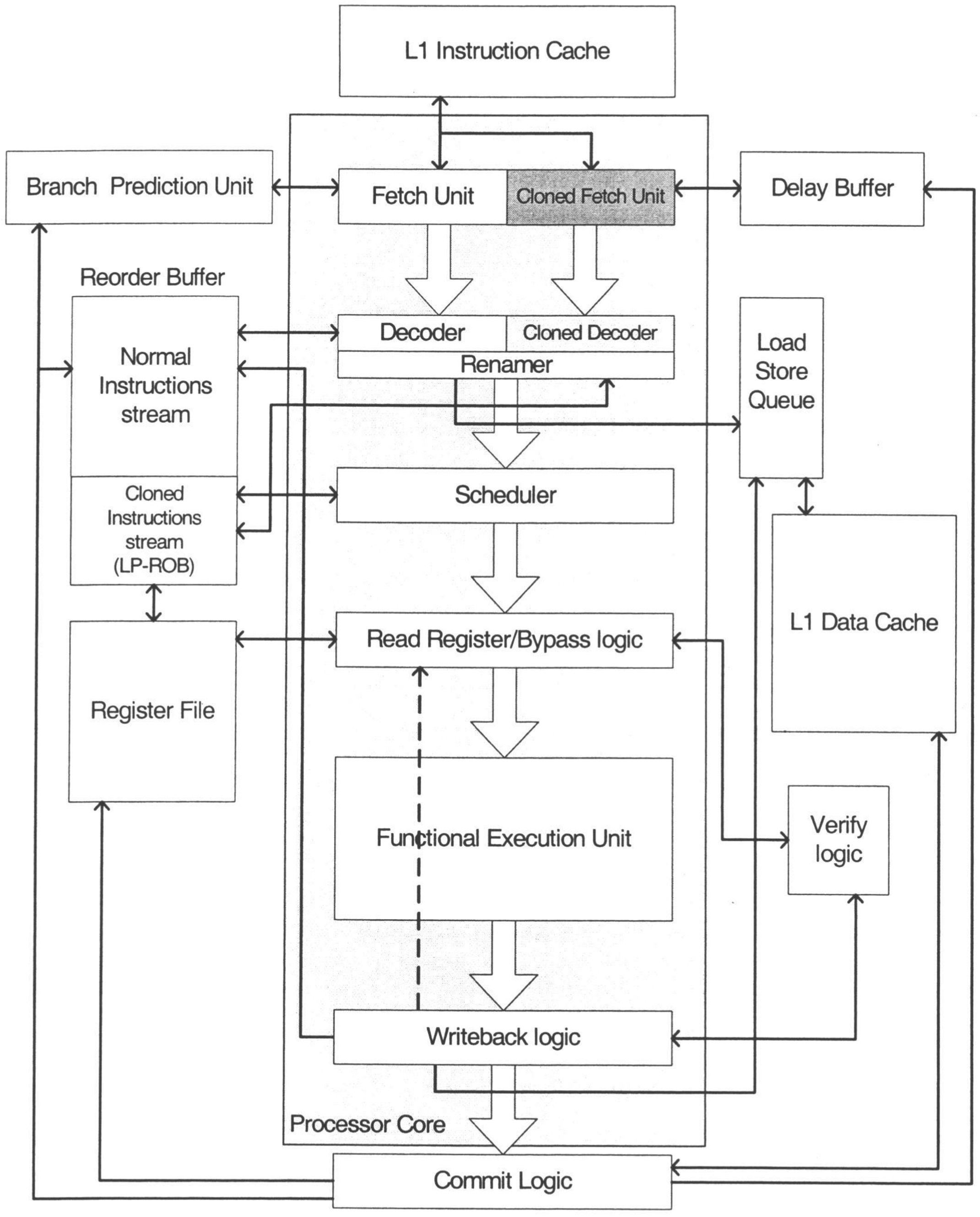


Figure 14 Basic Architecture of Ditto Processor

Reorder Buffer: We also found that the average reorder buffer (ROB) occupancy in the baseline non-fault-tolerant system with 128-entries ROB is about 50% for integer benchmark and 90% for floating point benchmark. By allocating the redundant part of ROB to cloned instructions stream, we may reduce the performance degradation without extra hardware overhead. After the cloned instructions are decoded, they are placed at the lower part of ROB (LP-ROB) as illustrated in Figure 1. Results of normal instructions are copied from the delay buffer to the result field of LP-ROB. Error Correction Code (ECC) checking mechanism protects this copy operation. In order to differentiate long latency instruction and short latency instruction, extra bit is added to each ROB entry. We will describe how to handle cloned instructions stream renaming in section 4.1.3. Furthermore, the size of LP-ROB should be small enough to minimize the effect of normal instructions stream's throughput. Our study reveals that long latency operations would have severe impact on LP-ROB pressure and degrade the performance accordingly. Hence, we suggest that short and long latency instructions should go through different verification path.

Status bit to handle duplicate execution: Since all long latency instructions are executed twice including those that are speculative, we adopt the idea from[38] to handle these duplicate computations. This approach requires the fewest hardware overhead. An extra status bit is added to each of the ROB entries indicating the long latency operation is ready to be executed the second time. Since memory reference micro-ops belong to long latency operations, this extra status bit is also appended to entries of the load store queue (LSQ). Furthermore, the verify bit is used to confirm that the computation of duplicated long latency operations were completed and verified. After these results are confirmed, results from duplicated copies are discarded. Since results from the original instruction and the duplicate copy may be ready at different cycles, we also need to address the scheduling of their dependent instructions. We schedule dependent instructions according to the data ready time of the original copy, since faults are not as frequent This cause no further complication because a mismatch of results will bring back execution prior to the faulty instruction.

Verify logic: Once these cloned instructions complete their execution, cloned instructions' results are compared to the original instructions' results saved in the result field of ROB. Verify logic, next to the write-back stage, is used to handle this error detection and recovery. We will present this mechanism in the following section.

4.1.2 Error detection and Fault recovery mechanism

Ditto Processor employs two checking mechanisms to detect potential transient faults. The first mechanism is placed after the register-read stage. After a cloned instruction's source operands are ready, we compare the decoded instruction with the correspond entry in the delay buffer. It detects two places where transient faults may occur.

1. If this re-fetched instruction does not match the correspond entry in delay buffer, it indicates the occurrences of a transient error in the fetch unit or in the decoder. For conditional and unconditional jump instructions, the decoded target address is also verified.

2. For long latency operations if clones source operands values does not match the correspond values in the delay buffer, it indicates the occurrences of a transient error in renaming logic.

This mechanism allows us to detect faults occurs at earlier stages of the pipeline. The verification process is overlapped with the execution stage and poses no extra delay.

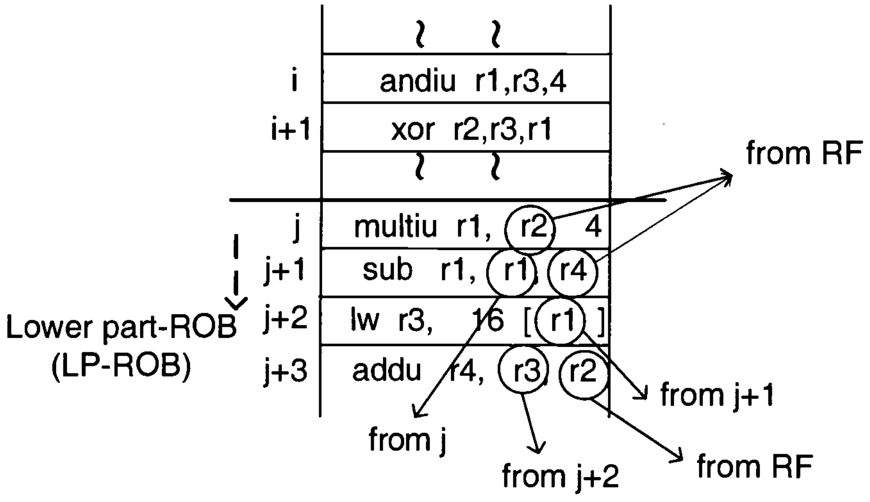


Figure 15 Instruction renaming example

The second checking mechanism occurs when the cloned instructions complete their computation. Results of cloned instructions are compared to the original results stored in the result field of ROB entries. If the results are the same, cloned instructions are removed from reorder buffer. If results do not match, then we have detected a transient fault in functional units. Since long latency instructions already verified computation results while in normal instructions stream, these instructions would not go though this second mechanism.

In both mechanisms, we recover the system back to the known correct state similar to branch mis-prediction recovery. Hence, there is no other extra hardware needed beside what we have mentioned to handle this error recovery on register file rollback. We will present this rollback mechanism in the following section. Since, in this study, we assume the mean time between faults (MTBF) is about 10 million cycles, after several cycles of error recovery, the second try[1] should have a valid result and program may continue to execute.

[1] The second try means the instructions will be fetch, decode and execute twice as mentioned and the result will be verify again.

4.1.3 Cloned instruction renaming and register file rollback mechanism

Since the decoder of normal instructions stream and cloned instructions stream come from different paths, the renamer should not mix these two streams together. Figure 15 shows a snapshot of the ROB during execution. *Ditto processor*'s ROB is divided into two regions – the normal ROB entries region and the LP-ROB entries region. The LP-ROB maintains the program order of cloned instructions stream while the rest of the ROB is used for normal instructions stream.

We present an example to describe how *Ditto Processor* handles instructions renaming. Let's assume the LP-ROB starts with entry *j*. Since "multiu" is at the head of LP-ROB, all previous cloned instructions have been verified. The source operand (r2) of "multiu" is mapped to architecture register file, so is the source operand (r4) of "sub" and (r2) of "addu". The source operand "r1" of instruction "sub" is depending on the previous result of entry *j*. Since the previous result has been copied from delay buffer to entry *j* as described in section 3.1, the source operand (r1) of "sub" may use this value and schedule immediately after renaming. This is true for instructions "lw" and "addu" also. This scenario contains no data hazard and allows cloned instructions to fly through pipeline stages faster then normal instructions. It also reduces possible performance loss due to re-execution come with the time-redundant technique.

For long latency operations, if transient error occurs in this renaming operation, the verify logic will detect the source operands' values are different from values produced by the original instruction and will signal the recovery mechanism. For short latency operations, the verify logic would detect this renaming error if the clone's computation result is different from the original result since clone instruction stream and normal instruction stream handle renaming operation independently.

In a redundant processor using simultaneous multithreading technique such as AR-SMT, each thread must maintain its own register status and values by register map [22], it requires some additional hardware when compared with *Ditto Processor*. In *Ditto Processor* we only need to augment the state bits in architecture register file. Whenever a normal instruction is ready to commit, it writes the result to register file and transits the status bits from "invalid" to "transient". Once the cloned instruction is verified, the status is changed from "transient" to "verified". This approach requires only one extra bit added to each register. From the re-namer and scheduler's point of view, they treat "transient" and "verified" value in the same way as data ready. If a transient error is detected, all "transient" values are flushed from the architecture register file. Moreover, all in-fly instructions are squashed similar to miss-branch prediction recovery.

4.1.4 What types of operation are protected?

In the *Ditto Processor* design, we cover every type of instructions for possible transient error. However, we do assume that there is no self-modifying instruction in our system.

Short latency Arithmetic/logic instructions: After these instructions are ready to retire, they store the result and other information to the delay buffer and ROB entry is free for other normal instructions. The cloned instruction is then fetched, decoded/renamed, scheduled and executed. After the result is verified, the LP-ROB is free for other cloned instructions. Since we assume the Branch Prediction Unit is protected by the ECC mechanism, our scheme may verify the correctness of decoded target address and the outcome of branch.

Multiply/Division instructions: Since these instructions have long execution latency, they are duplicated after decode and speculatively execute twice and result are compared and verified. Result of these instructions and other information are stored in delay buffer for verification later. These instructions are also cloned and re-fetched. However, after it is decoded/renamed, scheduled and read from register, they would not go through computation again. As mentioned before they are checked by the first checking mechanism. After passing the first checking mechanism, these instructions are free from LP-ROB.

LOAD/STORE type instructions: After this type instruction was decoded, it generated two micro-ops: one for data address calculation and the other one for memory reference. Since memory micro-op belongs to long latency operation, it would access cache memory twice based on the normal instruction's calculated data address. When this type instruction is ready to commit, it would store the result and other information into delay buffer. After the clone instruction is decoded, it would discard the memory micro-op since we only need to verify the correctness of data address.

4.1.5 What are protected units?

From Figure 14 we see that processor core is inside the shaded area. In other words, we assume any units outside of this area are protected by ECC logic. Furthermore, any wires and control signals that communicate between processor core and other units, such as data cache or ROB, are also protected by other fault-tolerant techniques [10][11] [21][22][23][24][31]. Whenever a system interrupt or exception occurs, protection logic will guard the transient fault to make sure these requested are being served correctly. Since the correctness of commit logic is imperatively important on placing the result into delay buffer and this logic is very small, we duplicated the commit logic to enforce its correctness.

4.2 SIMULATION CONFIGURATION

We modified the SimpleScalar simulator[51] in order to evaluate the performance degradation of different redundant schemes when transient faults are present. We randomly generate faults with MTTF of 10 millions cycles. When each fault occurs, it could occur at any point of the pipeline. In our study we randomly assign the fault to a particular pipeline stage. 14 SPEC2000 benchmarks (8 integers, 6 floating points) [52] are used for our simulation study. All benchmarks are executed for 500 million committed instructions after skipping the first 500 million instructions.

4.2.1 Baseline Model

In our baseline model, we extend the existing SimpleScalar pipeline model into seven stages: fetch, decode/rename, schedule, register read, execution, writeback and commit. Each stage takes one cycle. In order to eliminate the effect of data speculation, we schedule the dependent instruction at the data ready cycles. For example, in a cache-hit case, load operation takes 3 cycles to access data (2 cycles to access the tag array to determine hit/miss and 1 cycle to access data array). The load dependent instructions will be scheduled 2 cycles later after data effective address is calculated. Table 4 shows the overall baseline system parameters.

Fetch, decode, issue, commit width	8
Branch Predictor Branch Target Buffer	Gshare, 64-entry, 8 way, 8k-entry, 8 way
ROB / LSQ size	128/128 entries
L1 I/D cache	16KB/16KB 4-way, 32B line size
L1 I/D cache hit latency	1/3 cycles
L2 cache	1MB size 8-way, 32B line size
L2 / Memory latency	10/100 cycles
# of pipelined integer ALU/MULT/DIV	4/1/1
Integer ALU/MULT/DIV latency	1/3/20
# of pipelined floating point Adder/MULT/DIV	4/1/1
Floating point Adder/MULT/DIV latency	2/4/24
Read/Write port	4

Table 4 Baseline model system parameters

4.2.2 Different Simulation Machine Model

In this study, we compare five different machine models. The baseline model (Base) is described in section 4.2.1 In order to compare *Ditto Processor* with AR-SMT [41], we adapt AR-SMT into superscalar model, named AR model. There are two differences between AR-SMT and AR model. First, R-stream in AR model does not perform memory reference micro-ops because this operation would require operation system to be aware of A-stream and R-stream. Second, AR model does not contain trace cache. We further define that 30% of ROB entry allocated to R-stream would be AR-30 model (96 entries for A-stream and 32 entries for R-stream). Similarly, R-stream of AR-10 model would utilize 10% of ROB entry (Similarly, 112 and 16 entries).

In our experiment, *Ditto Processor* model's (Ditto) cloned instructions utilize 10% of ROB (16 entries for LP-ROB). Our study reveals that this allocation strategy would have the least performance effect on normal instructions stream. Both AR and Ditto model use 128-entry delay buffer to store the committed instructions.

We also model the 2-way redundant scheme (Dual) by Ray et. al. [39]. The Dual model has the same system parameters as our Baseline. There are two differences between Ray's original architecture and ours Dual model. First of all, the original design has 64KB I-cache, 32KB D-cache, 512KB L2 cache and 2 read/write ports while our memory subsystem modeling is slighted different and summarized in Table 1. Second, ours model has longer pipeline stages. Despite these differences, our dual model matches their result closely.

Out-of-order Reliable Superscalar (O3rs) [38] is also implemented in this study for comparison. System parameters of O3rs are the same as the Baseline model. The O3rs model should have the best result in terms of Instructions Per Cycle (IPC) degradation, since it only verifies the functional units and it does not take away ROB entries from normal instructions like other schemes for re-computation.

4.2.3 Fault Injection Mechanism

In our study, we inject faults randomly at different stages every 10 million cycles for all different schemes described above. In other words, the fault could be at fetch unit, decoder, scheduler, register read operation, execution, bypass logic or others. As instructions with faults pass through our checking mechanism they will be detected as described in section 4.1.2. Machine will be reset back to the known state. In this study, we assume two cycles of error detection and recovery penalty.

4.3 Performance result

In this section, we present the simulation result of our study. Section 4.3.1 shows the IPC degradation of each model and section 4.3.2 shows the functional units resource utilizations of each model. Section 4.3.3 presents the characteristic of *Ditto Processor.*

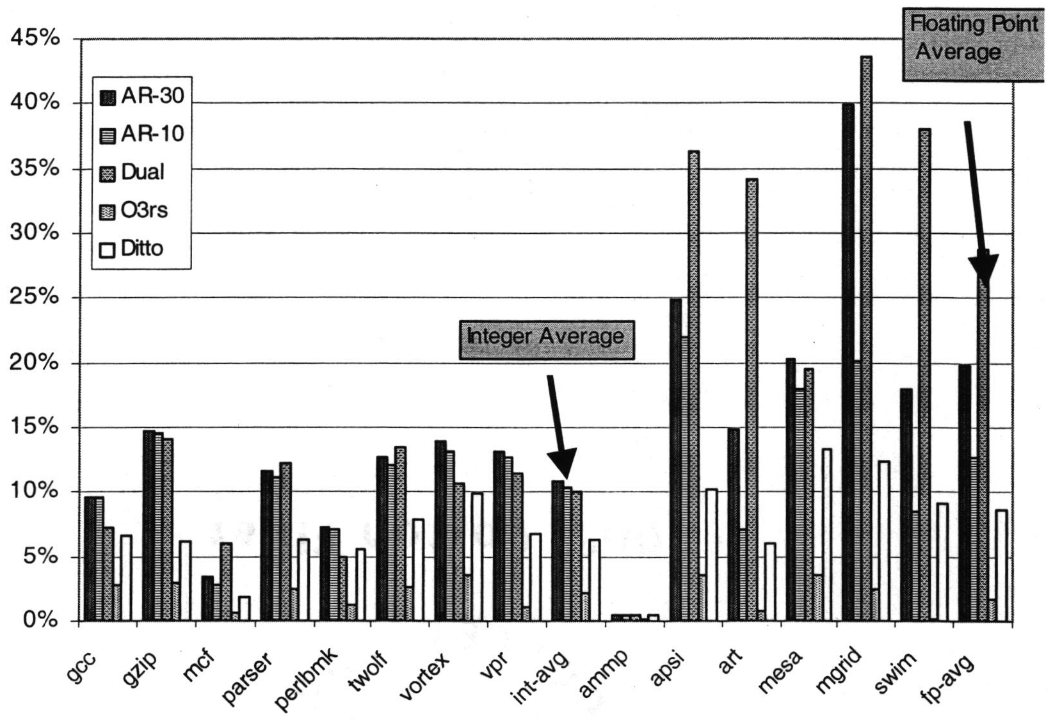


Figure 16 Instruction per Cycle (IPC) degradation

4.3.1 Performance degradation

Figure 16 illustrates the percent performance degradation of several time-redundant fault-tolerant designs. AR-10 has slight performance improvement over AR-30 in that the former utilizes less LP-ROB. However, for one of the floating-point benchmark - "mgrid", it shows a large difference in performance. Further study reveals that "mgrid" has over 65% of long latency instructions. As LP-ROB size reduces, it leaves more space in upper ROB to occupy long latency instructions and, in turns, reduce the performance loss. The average floating-point benchmark result also shows the same behavior that AR-10 outperforms AR-30 by about 7%. Since integer benchmarks have over 70% of short latency operations* and these operations enter and leave LP-ROB within a very short time, they give AR-10 only slight advantage over AR-30.

In the Dual model, after the instructions are decoded, it created another copy of all instructions. A duplicated instruction also occupies a ROB entry as described in [39]. This mechanism reduces the effective size of ROB by half. Therefore, this scheme suffers severe performance loss in floating-point benchmarks and "mcf". In these cases, compared to AR-30, Dual model degrades the performance by about 9% in floating point benchmarks and 3% in "mcf".. The O3rs model has the least performance loss among five models because it does not take away ROB entries for duplicated instructions. O3rs loses 1.7% and 2% performance for integer and floating-point benchmarks respectively. As mentioned O3rs does not cover front-end part of the pipeline nor memory instructions. Since in our Ditto model the cloned long latency instructions do not pass through execution stage again and reduce the pressure on LP-ROB, this further reduces the performance loss. Ditto suffers about 1.8~13.3% performance degradation.

We also observe that "ammp" benchmark has very little performance loss, only about 0.4%, on all models. Further study reveals that "ammp" has very high

Ll and L2 data cache local miss ratio, about 50% and 90% respectively, most of the operations are hinder by lengthy memory reference. In this case all our simulated fault-tolerant models may be able to benefit from normal instructions stream's low throughput and low functional units utilization.

In summary, AR-30, AR-10 and Dual model has an average of 10% performance degradation on integer benchmarks. Ditto model outperforms these three models and reduces the performance loss by 40% to about 6% on integer benchmarks. For floating-point benchmarks, the performance loss of AR-30, AR-10 and Dual models are about 19%, 12% and 28%, respectively. Ditto reduces the degradation by 30% and 70% respectively to 8.6% when it is compared with AR-10 and Dual models on floating-point benchmarks.

4.3.2 Functional units resource utilization

Since different models have different effects on functional unit's resource utilization rate, Figure 17 presents each model's utilization ratio in more detail. Compared to the Base model, all other models have better functional unites utilization, especially Ditto. Since Ditto model verifies all types of instructions, it utilizes resource more efficiently. On average, Ditto utilizes integer ALU units about 15% more than Base model.

Dual model has the similar ratio as Ditto in integer benchmarks, but since it duplicates all instructions including instructions that are speculative, the performance loss is higher. For floating-points benchmarks, there are more long latency instructions putting more pressure on the ROB. This further reduces the effective instructions windows size. For example, Dual model only uses half of the ROB to explore instructions level parallelism (ILP). Hence, Dual has the worse integer ALU utilization rate for floating-point benchmarks. Because both Ditto and

Dual models verify cache access operations, memory ports are used more efficiently on these two models.

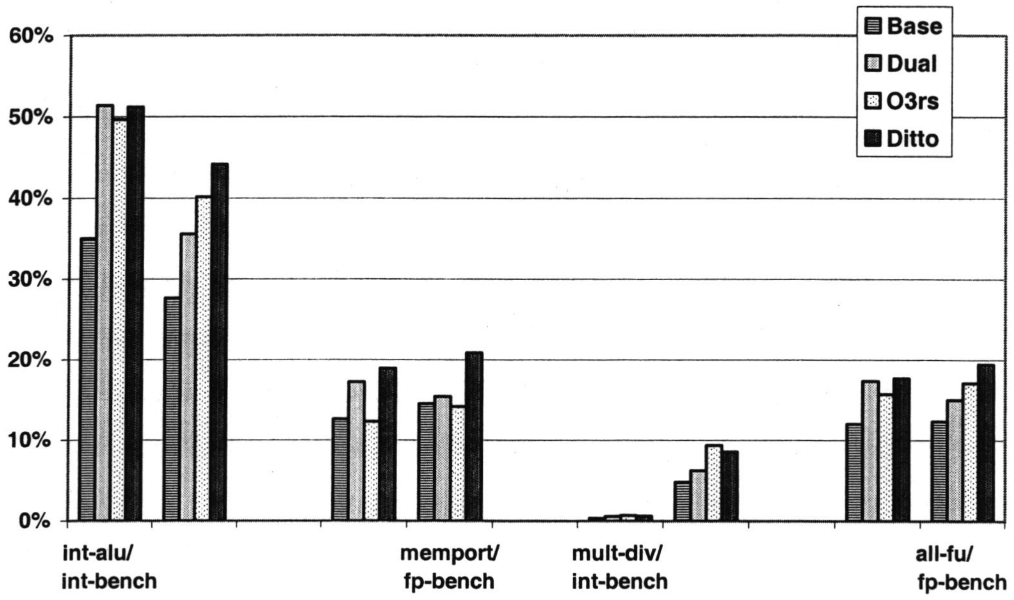


Figure 17 Average functional units resource utilization[2]

O3rs posts no effect on cache ports since O3rs does not verify memory reference micro-ops. Since integer benchmarks rarely use multiplier and divider, all models has very little utilization rate for these modules. O3rs model has slight better MULT/DIV unit utilization for floating-point benchmarks in that it has more ROB entry to explore ILP than Ditto[3]. In summary, we observe that, when compared to Base model, Dual model has about 5% more on functional utilization for integer benchmarks and 3% more for floating-point benchmarks. O3rs model is 3.6% and 4.7%, respectively. For Ditto model, it is 5.7% and 7.4% better on average. Hence, Ditto model has full transient fault coverage with less performance degradation.

[2] There are four groups in this figure and each group contains integer and floating-point benchmarks result. The most left group is integer ALU unit utilization ratio. The second group is memory port utilization ratio and the third group is the combination of integer and floating-point multiplier/Divider unit utilization ratio. The most right side group is overall functional unit utilization ratio.

[3] Ditto model has 112 ROB entries for normal instructions stream and 16 entries for cloned instructions stream.

4.3.3 The characteristic of Ditto Processor

Figure 18 depicts the percentage of IPC degradation when we compare the Ditto to the Base model with different L1 cache hit latency. We observe that as L1 cache hit latency increases, the Ditto model gradually reduces the performance loss on both integer and floating-point benchmarks. One factor that affects the percentage of performance degradation is the amount of idle time available in the processor for time redundancy to perform transient fault checking. The basic motivation for our approach is to utilizing these stalled processor cycles to verify computation through re-execution. As memory latency increases in terms of cycle time, we have more stalled cycles in the processor and more resources available. This gives more opportunity to perform cloned instructions execution and reduce the effect of performance degradation.

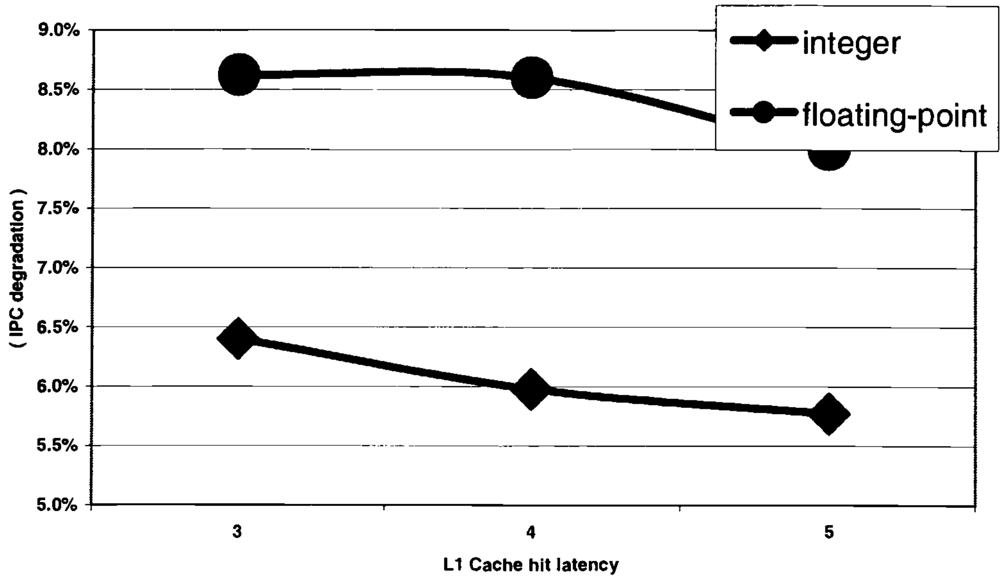


Figure 18 Effect of L1 cache hit latency on Ditto

CHAPTER 5 LDS PREFETCH MECHANISM

5.1 POINTER ELEMENT PREFETCH UNIT (PEPU)

PEPU is composed of four subunits to perform predicting address and prefetching data from Level One data cache (L1 D-cache). Section 5.1 describes the definition of pointer load. Section 5.2 explains how Target Register Bitmap (TRB) can identify pointer load and BS may rollback to normal TRB. Section 5.3 shows that data saved in Address Cache (AC) are used to predict next pointer element's location in memory. Section 5.4 describes prefetched data is saved into Prefetch Cache (PFC).

5.1.1 What is pointer load?

Pointer load is a load that its base address is depended upon the target value of previous load operation [65]. Such load's address pattern is irregular and no stride predictor can capture this pattern correctly. In this study, we choose to classify pointer load into three different types depending upon its producer-consumer relationship. Since branch and store operation do not update register value, these two types instructions would not involve pointer-consumer relationship.

a. Address load: If both of a load's producer and consumer are load operations, this load is called Address load.
b. Data load: If the producer of this data load is also a load operation, its consumer could be any operation except load operation.

c. Data-Address load: Similar to data load, if this load's producer is load operations, Its consumers are both load operation and non-load operation. Figure 19 is a loop-unfold example extracted from "tsp" benchmark. Figure 19(a) shows that instruction 1 and 7 are Address loads and instructions 3 and 9 are Data loads. Figure 19(b) presents that instruction 2 is Data-Address loads since its consumers are instruction 4 and 5.

(a) Example of Address load and Data load

```
 1 : Loop:lw    r4, 4(r4)
 2 :      be    r4, r0, [Exit]
 3 :      lw    r3, 8(r4)
 4 :      addi  r3, r3, 4
 5 :      sw    r3, 8(r4)
 6 :      jmp   Loop
 7 : Loop:lw    r4, 4(r4)
 8 :      be    r4, r0, [Exit]
 9 :      lw    r3, 8(r4)
10 :      addi  r3, r3, 4
11:       sw    r3, 8(r4)
........
```

(b) Example of Data-Address load

```
1:lw     r28, 0x34
         .........
2:lw     r2, -32304(r28)
3:be     r2, r0, [Exit]
4:lw     r3, 0(r2)
5:addi   r2, r2, 4
6:srl    r5, r3, 1
7:sw     r2, -32304(r28)
.........
```

Figure 19 Pointer load example from "tsp" olden benchmark

Figure 20 illustrate that, from 15 benchmarks simulation result, 18%-79% of committed loads are Pointer loads and different benchmark has difference distributions of Address loads, Data loads and Data-Address loads. Within these benchmarks, over 60% of the Pointer loads are from Data loads and Data-Address loads. Except for "voronoi", its Address loads are 57% of Pointer loads. These data indicate that most of the target value of Pointer loads not only used to reference the next pointer element but also been a input value to many arithmetic operations. Especially, "health", "mst" and "perimeter" have less than 1% of Address loads. "tsp" result shows that over 80% of Pointer loads are Data-Address loads since most of the Pointer loads in "tsp" are similar to the example given in Figure 19(b).

5.1.2 Identify Pointer load

In general, a program dependency is established after decode stage. This includes renaming the source operations' registers id to the producer's target register id. It means that no matter if a load's base address is ready, its producer is known after decode stage. Based on this knowledge, we proposed an n-bit bitmap, named Target Register Bitmap (TRB), to keep track of the target prior-renamed register id. The n is equal to the total number of architecture integer register file in the system. As shown in Figure 22, the TRB works as follows:

1. Every load operation set the TRB indexed by its target prior-renamed register id.
2. Every Register-written operation reset the TRB indexed by its target prior-renamed register id. Since branch and store operation have no effect on register value, these two types instructions would not update TRB.
3. Since move operation perform data migration from one register to another register, whether the bit of source register id of the move operation in TRB is set or reset, this bit value would copy to the bit of destination register id in TRB. For example, if a instruction, *MOVE R1, R4*, is decoded, TRB also perform *TRB[4]-> TRB[3]* operation. The existent schemes recognize register-moving operations is normal ALU types operations. From our study, this register-moving operation plays an important role in tree data structure, especially multi-way tree, since different child nodes may move back to the same parent nodes. In other words, register-moving operation would link the parent-child node together.
4. Whenever a load is decoded, its prior-renamed source register id is indexed to TRB. If the indexed value is "1", it is Pointer loads.

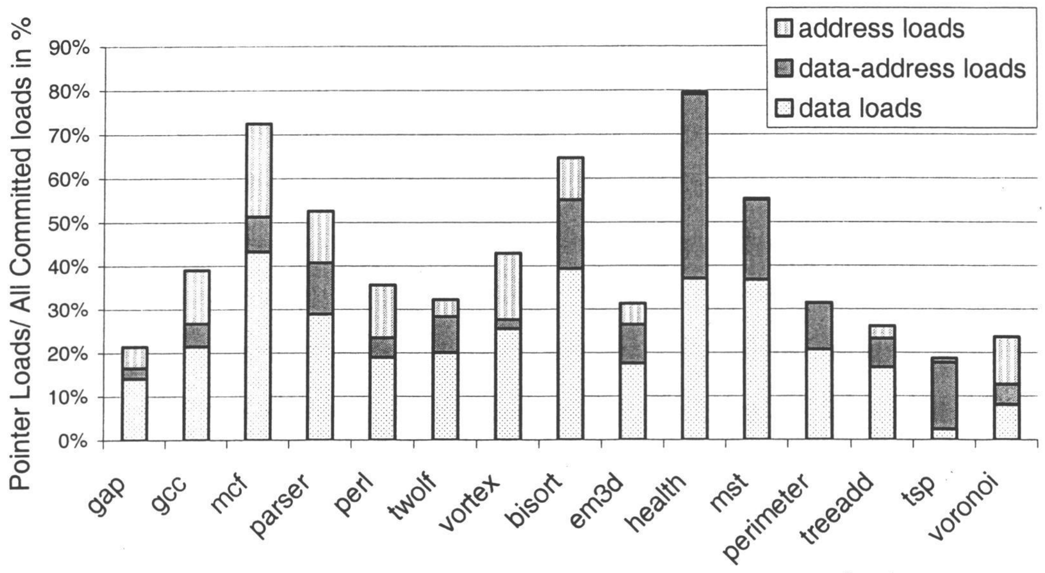


Figure 20 Percentage of Pointer loads for different benchmark

Figure 23 follows the previous example in Figure 19(a). Here assume that before the first instruction is decoded, the value of TRB[4] is set to "1". As the first instruction is decoded, this load operation would be identified as Pointer loads. It implied that each ROB entry would maintain a bit to represent it. As instruction 3 is decoded, it would set the TRB[3] to "1" and reset to "0" after instruction 4 is decoded.

Since miss branch prediction would erroneously set/rest the TRB and affect the accurate identification of Pointer load, Figure 22(b) shows that for each decoded branch, it would push the TRB into a stack, named Bitmap Stack (BS). After the direction of branch is determined, it would pop out the bitmap from the stack. If the branch is miss-predicted, the popped value would update TRB to rollback to the correct status of TRB prior this branch instruction. This stack may be placed at Branch Prediction Unit to associate the bitmap with each on-the-fly branch instructions. For our study, the BS size of half of Reorder Buffer (ROB) is enough to keep all on-the-fly branch instruction's bitmap. If the ROB has 128 entries, total entries of BS would be 64 with 4Byte each entry (assume the system has 32 integer registers file). The size of BS would be 256Bytes in above example.

After TRB identify a Pointer load, this load would update the Address Cache (described in section 5.1.3). The later load may use this AC information to prefetch data into Prefetch Cache (PFC). The main proposed of this Pointer Load identification will be given at section 5.1.3.

(a) Pointer load classification

```
If [TRB identify a Pointer Load]
then    ROB.pload = 1;
if [ consumer of this Load is arithmetic operation]
then    ROB.dload = 1;  # set data load field
if [ consumer of this Load is load operation]
then    ROB.aload = 1;  # set address load field
```

(b) Pointer load truth table

pload	dload	aload	Pointer load types
0	X	X	Not Pointer load
1	1	0	Data load
1	0	1	Address load
1	1	1	Data-Address load

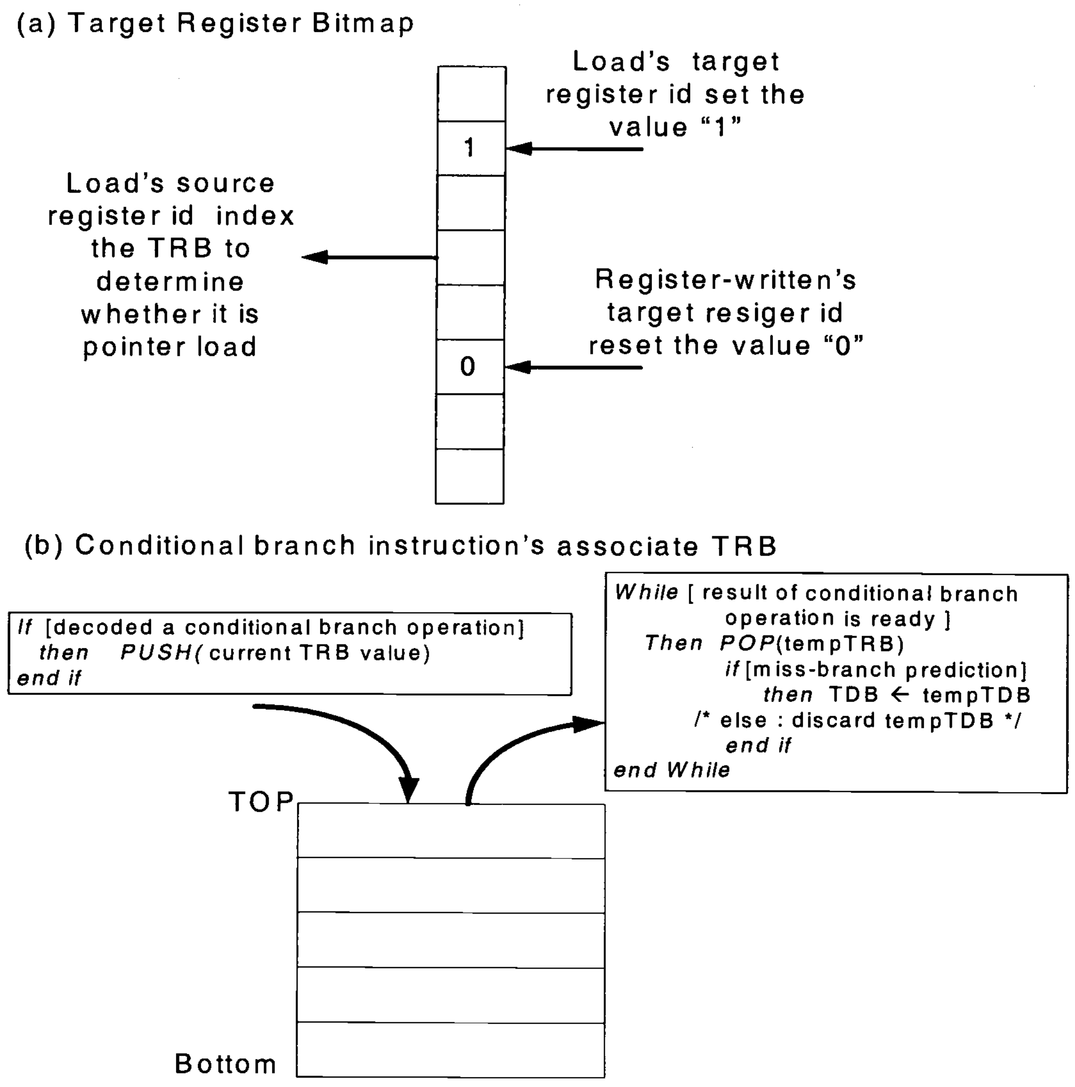
X: means don't care

Figure 21 Pointer load definition

The Reorder Buffer (ROB) is added three bits information to differentiate the Pointer Loads type. Figure 21(a) illustrates each bit's definition and Figure 21(b) pointer out the truth table of identifying different types of Pointer load.

5.1.3 Predict Pointer load base address

Even though for each pointer element's location is not known until its previous element is known, most of the time these element's location are fixed during run-time.


(a) Target Register Bitmap
Load's target register id set the value "1"
1
Load's source register id index the TRB to determine whether it is pointer load
Register-written's target resiger id reset the value "0"
0
(b) Conditional branch instruction's associate TRB
TOP
Bottom

```
If [decoded a conditional branch operation]
  then    PUSH( current TRB value)
end if
```

```
While [ result of conditional branch
          operation is ready ]
   Then  POP(tempTRB)
          if [miss-branch prediction]
             then  TDB ← tempTDB
        /* else : discard tempTDB */
          end if
end While
```

Figure 22 Target Register Bitmap

We proposed to save these locations into an Address Cache. Whenever a producer Pointer load's virtual address is known, not only does this load begin fetch target value from L1-D cache but this address is also indexed to AC to search for the next possible element's location. After the target value of Pointer load is ready, the value in AC, indexed by virtual address, is updated as well. Since Data load's target value would not be a valid address value, only Address load and Data-Address load may update AC. AC has to be small as to meet the one cycle hit latency. Therefore, AC may be direct-map or 2-way associative cache design to minimize the access latency. The line size of AC is 4Bytes line size in 32-bits addressing system or 8Bytes in 64-bits.

Not only did the Address load and Data-Address load would update AC but also store operation may do it as well. For instance, in Figure 19(b), the next iteration of instruction 2 and current iteration of instruction 7 cause the memory dependency case. This is not detectable by TRB. Therefore, in such situation, once the store's virtual address is computed, it would lock the entry in AC (an lock bit is added into each address line) until the store data is ready. If the following load would index to the locked entry in AC, it would cancel the prediction.

The idea of AC is similar to context-based predictor [68] since it records the history of loads' past addresses. The two differences between them are:

1. The former only record the identified Pointer load's target value without pollute the AC and the rear predictor attempted to record all the loads' past addresses. Therefore, context-based predictor need large table to maintain reasonable accurate address prediction ratio.
2. Since context-based predictor use base address to update address buffer, in the multi-way tree application, 4-way or larger associative cache structure would be needed to record all possible locations of child nodes. An extra way-predictor is also necessary to predict the possible child that have the same base address value. AC avoid these extra hardware by using virtual address as index value to correctly reference the possible child.

	TRB[3]	TRB[4]
1: Loop: lw r4, 0(r4)	0	1
2: be r4, r0, [exit]	0	1
3: lw r3, 8(r4)	1	1
4: addi r3, r3, 4	0	1
5: sw r3, 8(r4)	0	1
6: jmp Loop	0	1
7: Loop: lw r4, 0(r4)	0	1
8: be r4, r0, [exit]	0	1
9: lw r3, 8(r4)	1	1
10: addi r3, r3, 4	0	1
11: sw r3, 8(r4)	0	1

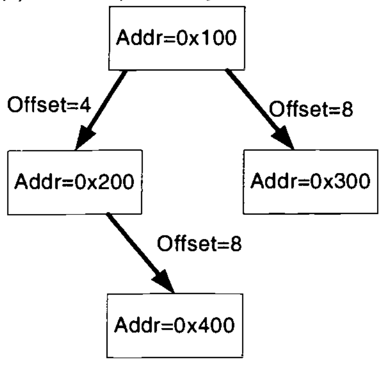
Figure 23 TRB example

Figure 24 illustrates a sample tree data structure. Here we assume the context of AC is valid before Figure 24(d)'s code is decoded. Figure 24(e) shows that as the instruction 5's base address is ready, this value is index to context-based predictor. If way-predictor made the wrong prediction about which child to reference next, the context-based predictor may prefetch down to the wrong path of a tree. Since the penalty of wrong-path prediction may be high, especially for the tight loop application, many schemes [65][68] limits the maximum level of prefetching to reduce the penalty caused by wrong path prediction. Figure 24 Address Cache example(f) demonstrates that even though our scheme take extra cycles waiting for instruction 5's virtual address computation, this address may accurately predict the child path and eliminate the wrong-path penalty. One thing need to remember is that the output of AC will not be used to speculatively index the next possible address. From past study [65] and our experiment revel that conservative fetching next element only for every address prediction in a LDS is fair enough.

5.1.4 Data Prefetch

Because of the natural of LDS's irregular address pattern, two characteristics of cache: spatial locality and temporal locality, may collapse. Increasing the cache size does not a good solution in this pointer-chasing problem any more.

(a) The sample binary tree structure



(b) The C-program code

```
While(src) {
  if(src->data != threshold)
      src = src->right;
  else
      src = src->left;
}
```

(C) The Assembly code

```
Loop:  lw  r3, 0(r4)
       be  r3, r0, Exit
       sub  r10, r3, r9   # r9=threshold
       bne  r10, r0, Right
       lw   r4, 4(r4)     #go to left node
       jmp  Loop
Right: lw   r4, 8(r4)     #go to right node
       jmp  Loop
```

(d) The fetched program sequence

```
1.  Loop: lw    r3, 0(r4)     # Data load
2.        be    r3, r0, Exit
3.        sub   r10, r3, r9
4.        bne   r10, r0, Right
5.        lw    r4, 4(r4)     #Address load
6.        jmp   Loop
7.  Loop: lw    r3, 0(r4)     #Data load
8.        be    r3, r0, Exit
9.        sub   r10, r3, r9
10.       bne   r10, r0, Right
11. Right: lw   r4, 8(r4)     #Address load
12.       jmp   Loop
13. Loop: lw    r3, 0(r4)     #Data load
14.       be    r3, r0, Exit
```

(e) The Context-based predictor

index		
100	200	300
200	400	500

(f) Address Cache
(direct-map in this example)

index	
104	200
108	300
	~
208	400

Figure 24 Address Cache example

We verify this scenario by simulating the different L1-D cache sizes range from 16K to 1M on 8-way RISC-like architecture. Here we assume L1-D cache takes three cycles hit latency. 4MB unified L2 cache has ten cycles hit latency. Only "health" benchmark may begin to takes advantage of the large cache size when the size is larger than 128KB. "parser" benchmark has steady improvement as the size increase, from 2.1% to 5%. The extra dark-bold-thick line with triangle mark in Figure 25 is the average improvement over 16K L1-D cache IPC in 15 benchmarks. The average results shows that, with larger cache size from 32K to 128K, the IPC improvement is from 0.35% to 0.85%. When L1-D cache swell to 1 MB with fixed three cycles hit latency, the improvement is merely 2.3%. This simulation result concludes that cache size is not always the panacea to gain performance.

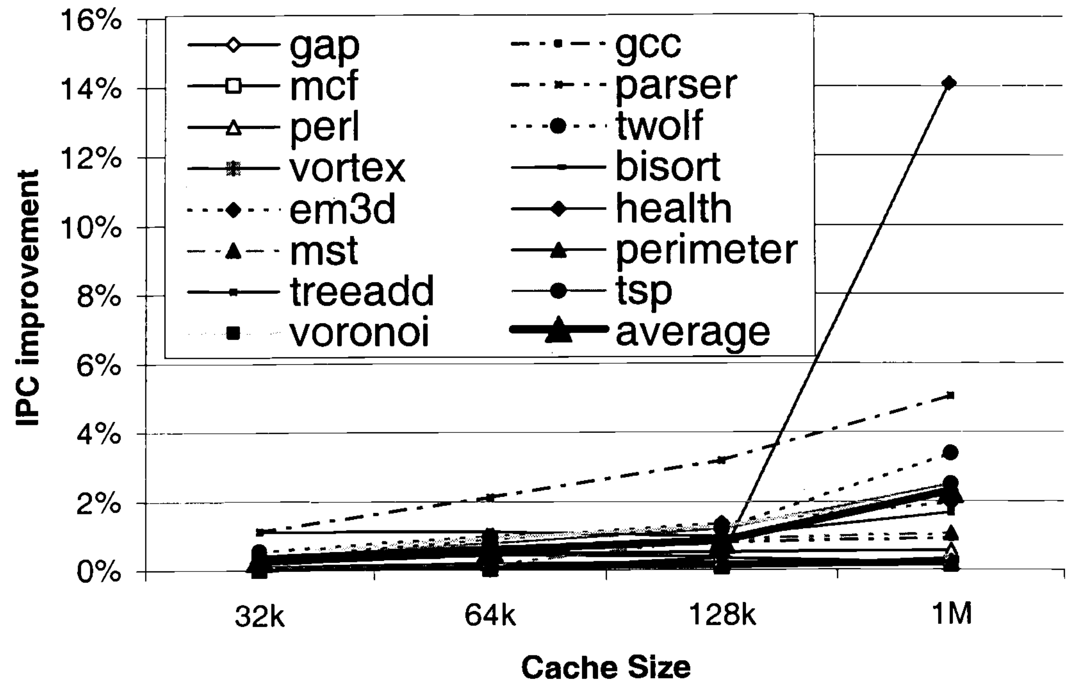


Figure 25 Different cache sizes IPC improvement over 16K L1-D cache size

From above simulation study, we proposed to choose the smaller L1-D cache size with added Prefetch Cache (PFC) and AC. For instance, if a base system has 16KB L1-D cache size with three cycles hit latency, our mechanism would have 8KB L1-D cache size with three cycles latency, 1KB~2KB PFC size and 64B~1KB AC size. Both PFC and AC has one cycles hit latency since they are much smaller than L1-D cache. Section 5 would present that even though our mechanism have about 10.5KB size (16KB L1-D cache, 1KB PFC, 1KB AC, 4Bytes TRB and 256Bytes BS), it performs better than 16KB L1-D cache system. Since section 3.3 details the implementation of AC, PFC design is given here. The PFC may either direct-map or fully-associative cache but the design configuration must be the one cycle hit latency to match the speed of processor core.

The PFC methodology is as following: whenever AC generate a predicted address or the base address is already in register after a load operation is decoded (detail in section 4.1), the data is prefetched from L1-D cache to PFC. If cache miss in L1-D cache, this prefetch request would go through L2 cache or main memory when memory port is idle. This request would not update the content of L1-D cache but fill the PFC when the data is return from lower-level memory hierarchy. After a load is decoded, it would fly though renamed, schedule, register read and address generation stage. If no memory dependency exist, the read request would send to cache memory and PFC at the same time. If PFC hit, the request to L1-D cache will be ignore/ cancelled. Since L1-D cache and PFC are separate physical memory structure, a store operation not only update L1-D cache but also, if this write request hit PFC, it would also update PFC's content as well to maintain the cache coherence in uni-processor system.

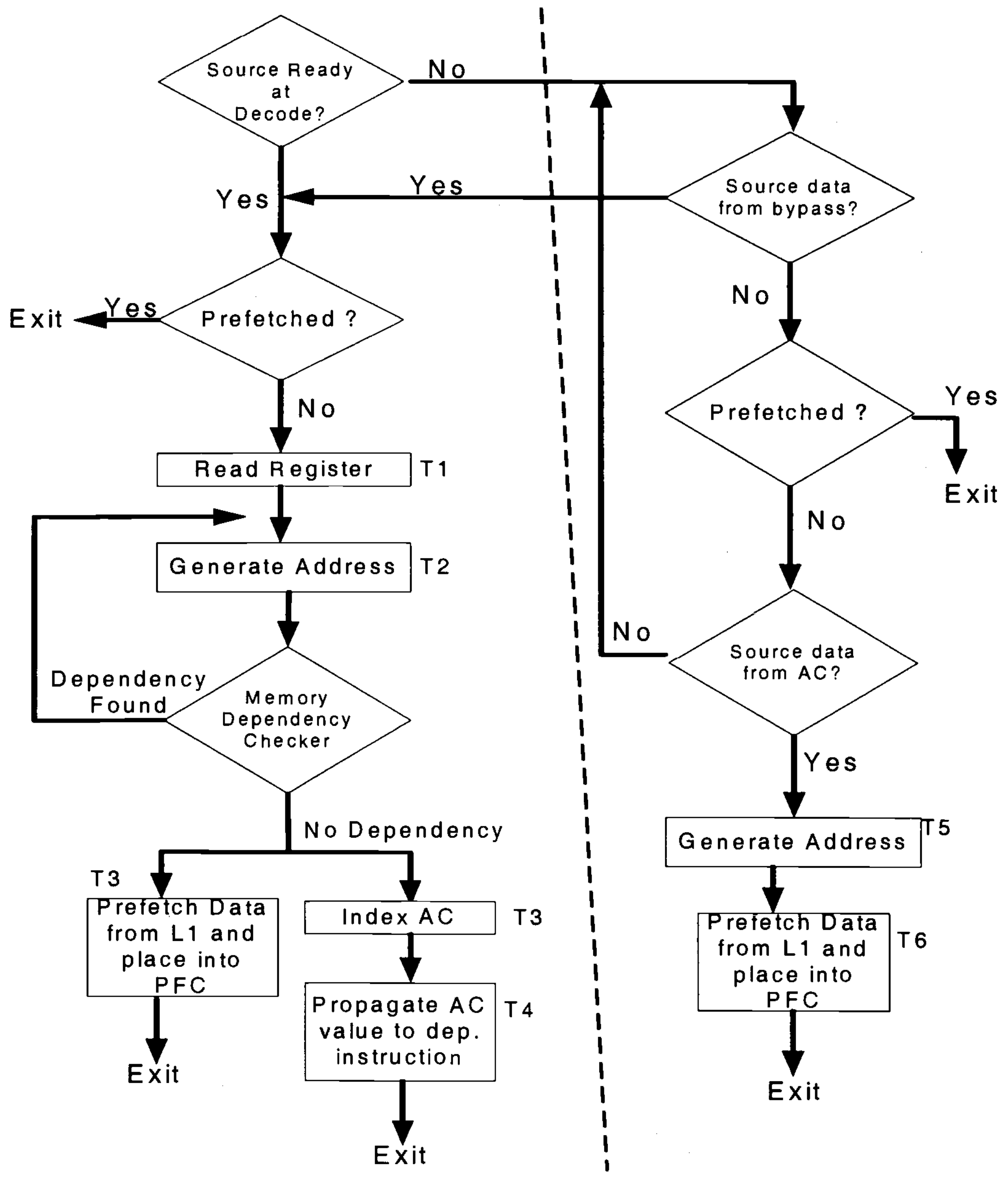


Figure 26 Flow Chart of Pointer Element Prefetch Unit

5.2 FLOW CHART OF POINTER ELEMENT PREFETCH UNIT (PEPU)

In this section, we present how PEPU manage to perform data prediction and prefetching. After a load instruction is decoded, if its source value is already in register, there is no need to predict its base address. In section 5.2.1, we describe while the base address of a load is ready, how PEPU handle address prediction and prefetching. In section 5.2.2, for those loads that its producer is still in transit state, we demonstrate how those loads use predicted data to perform data prefetching. Section 5.2.3 provides an example.

5.2.1When a load's base address is ready after decode stage

Our mechanism different from other proposed early prediction scheme [68][69] is that we does not use PC to index the prediction table at front-end pipeline and predict the load's base address before it was known. First of all, it would require extra hardware to pre-decode the instruction after it was fetched into Fetch Queue and pre-compute the address. Second, the prefetched data may early trigger load's dependent instructions. When the address of this data is miss-predicted, the hardware rollback mechanism need to enforce to rewind back to the point prior error occur. However, this error window may be variable cycles and make rollback scheme difficult to implement[80]. In order to reduce the system design complexity, we choose to perform our prediction between front-end and back-end pipeline stage.

In Figure 26, the left part of dash line is the source-ready load instruction flow chart. After a Pointer load is decoded, the renamer would map the programmer-visible register number to internal register number to eliminate write-after-write and write-after-read dependency. At the same time, since data is ready at register file, the load operation may read the register at T1 cycle and compute address at T2 cycles. As described in section 5.1.2, the Pointer load is identified after decode stage. At T3 cycles, if no memory dependency exists, the data may prefetch from L1-D cache into PFC. If it is Pointer load, it would index the AC at the same cycle. At T4 cycles, the output of AC would propagate to the load's dependent instructions, which are also load instructions. One thing need to address here is that, at T3 cycles, we call the data is "prefetched" because in the regular pipeline architecture, a load operations would fly through renamed, register read and then address computation. This left part of instruction flow may potentially reduces two cycles of load execution latency: first, the rename and register read action are overlap at the same cycle[4]; second, the schedule/issue stage is removed since an simple adder is dedicate to PEPU for generating address[5].

5.2.2When a load's base address is NOT ready after decode stage

Most of the time, after a load is decoded, its producer operation is still in transit. This is very common in LDS application since many iterations of loads may be fetched and decoded and waiting the destination value of producer load instruction return from cache memory. The right part of dash line in Figure 26 illustrates that if a pending load's source register value is forwarded from producer load, it would jump to the left part of the PEPU flow as described in section 5.2.1. If the pending load does not receive the forwarding data from producer load but from the output of AC, it would speculatively compute the virtual address at T5 and prefetch data form L1-D cache into PFC at T6 cycles.

[4] It assumes that no structure hazard in register file

[5] It assumes that only one source-ready load generate address at each cycle. However, there could be many source-ready loads ready to generate address in a cycle. The left part of flow may only save at most one cycles in such case.

5.2.3 A Linked-List Data Structure example

This example is from Figure 26(a). Here we assume this particular architecture is 4-wide issues superscalar processor with unlimited hardware resource and all the load operation would hit the L1-D cache with 3 cycles latency. Table 5 shows the normal pipeline execution. Without PEPU, even though the second iteration of instruction 7 is decoded at cycle 3, it will not begin execution until cycle 9. These two iterations of a loop take 27 cycles to complete.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1.Loop:lw r4,4(r4)	D	N	S	R	A	#1	#2	#3	W	C																	
2. be r4,r0, [Exit]	D	N							S	R	E	W	C														
3. lw r3, 8(r4)	D	N							S	R	A	#1	#2	#3	W	C											
4. addi r3, r3,4	D	N													S	R	E	W	C								
5. sw r3,8(r4)		D	N															S	R	E	W	C					
6. jmp Loop		D																				C					
7.Loop:lw r4,4(r4)			D	N					S	R	A	#1	#2	#3	W							C					
8. be r4,r0, [Exit]			D	N											S	R	E	W				C					
9. lw r3, 8(r4)			D	N											S	R	A	#1	#2	#3	W		C				
10. addi r3, r3,4			D	N																	S	R	E	W	C		
11. sw r3,8(r4)				D	N																			S	R	A	C
12. jmp Loop				D																							C

Table 5 Normal pipeline execution[6]

[6] D: decode stage, N: rename stage, S: schedule stage, R: register read stage, E: execute stage A: address generation stage, W: writeback stage, C: commit stage, #1-3: L1-D cache hit latency T1: read register in PEPU, T2: generate address in PEPU, T3: index AC/ prefetch in PEPU, T4: propagate AC value in PEPU, T5: generate address in PEPU, T6: prefetch data in PEPU

Assume that AC contain the valid address of these two iterations, Table 6 demonstrate the advantage of using PEPU. After instruction 1 is decoded, the load will go though the left part of PEPU flow shown in Figure 26. The instruction 3 may begin executing at cycle 7, which is 2 cycles less than Table 5. At cycles 5, the instruction 1 propagates the AC result to dependent instruction and instruction 3 and 7 may use this predicted address to prefetch data. Therefore, instruction 7 has the data ready at cycle 11, which is 4 cycles less than Table 5's result. Compared to Table 5 result, with PEPU, these two iterations take less cycle time to retire.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1.Loop:lw r4,4(r4)	D	N	T2	#1	#2	#3	W	C																			
		T1		T3	T4																						
2. be r4,r0, [Exit]	D	N					S	R	E	W	C																
3. lw r3, 8(r4)	D	N					T1	T2	T3	W	C																
						T5	T6	#2	#3																		
4. addi r3, r3,4	D	N								S	R	E	W	C													
5. sw r3,8(r4)		D	N										S	R	A	C											
6. jmp Loop		D														C											
7.Loop:lw r4,4(r4)				D	N		T1	T2	T3	T4	W					C											
						T5	T6	#2	#3																		
8. be r4,r0, [Exit]				D	N						S	R	E	W		C											
9. lw r3, 8(r4)				D	N						T1	T2	T3		W		C										
											T5	T6	#2	#3													
10. addi r3, r3,4				D	N										S	R	E	W	C								
11. sw r3,8(r4)					D	N												S	R	A	C						
12. jmp Loop					D																C						

Table 6 PEPU pipeline execution

5.3 SIMULATION RESULT

In this section, we will compare PEPU with some other proposed schemes and study the characteristic of our mechanism. Section 5.1 describes the base system configuration and our simulation environment. Section 5.2 compared the PEPU with different models. Section 5.3 shows the prediction accuracy of PEPU. Section 5.4 shows the result of varying the size of PFC. By increase the size of ROB in section 5.5, we study this effect on PEPU.

5.3.1 Simulation Environment

The architectural simulation used a modified version of the SimpleScalar toolset [10] to evaluate the performance impact on using PEPU. Seven integer SPEC 2000 benchmarks [52] and eight Olden benchmark [81] are used to evaluate the effect. These benchmarks are compiled with level-3 optimization. As shown in Figure 20, we choose these benchmarks because of higher Pointer load percentage, 18%~79%, than others. For SPEC 2000, after skipping the first 300 million instructions, statistics for 300 million committed instructions are collected. For Olden benchmarks, overall program execution's statistics data are collected.

Table 7 lists the base model configuration. It has 16KB L1-D cache and four integer pipelined ALU units. The PEPU model has 8KB L1-D cache, 1KB AC and 1KB PFC. These caches have its own two R/W port. This model use three integer pipelined ALU units and one simple adder for address computation. We also model the Dependency Based Prefetching scheme (DBP model) [65]and Direct Load scheme (DL model)[69].

DBP model has 256-entries of PPW and 256-entries of CT. It established the producer-consumer link at commit stage. The PFC is 1KB size. In this study, DL model has some different configures from [69]. First of all, sine we focus on the address prediction accuracy of LDS application, the stride predictor in [69] is not implemented in DL model. Second, since speculative scheduling dependent instruction is not the topic of this study, we choose not to include the rollback mechanism and all instructions are scheduled at data ready cycles. Therefore, not like [69], the prefetched data will place into PFC but not trigger the dependent instructions. The DL model has 256-entries of CRT and 256-entry of ULT/RUT. PEPU model, DBP model and DL model have the same 1KB of PFC and 8KB of L1-D cache.

Description	**Configuration**
Fetch, Decode, Schedule, Commit Width	8
Fetch/Decode/Rename/Schedule/Register/ **Execute/Writeback/Commit stage latency**	1/1/1/2/3/FU-latency/1/1
Architecture Register File IO port	8 reads/ 4 writes
Branch Predictor **Branch Prediction Table**	16-bits Gshare 8K-entry, 8 way
ROB/LSQ size	128/64
L1 I/D cache	16KB/16KB, 4-way, 32B line size
L1 I/D cache hit latency	1/3 cycles
L2 cache	4MB
L2 / memory latency	10/100 cycles
# of pipelined integer ALU/MULT/DIV	4/1/1
Integer ALU/MULT/DIV latency	2/6/40
Floating-point Adder/MULT/DIV latency	4/8/48
Cache Read/Write port	4

Table 7 Base Model configuration

5.3.2 Instruction Per Cycle (IPC) improvement

Figure 27 illustrates the result of three different models. All three models are performed better than base model with larger L1-D cache size. It means that all three models may improve the performance of LDS application. PEPU further outperform others model by about average 5%. On average, the DL model has less performance improvement since it required large table[7] to gain the 71% prediction accuracy. It is too costly in hardware implementation and, unlikely, able to access the table within one cycle latency. We believe the 256-entries of CRT and ULT are more suitable for this study's comparison. Using this size, DL model have merely 2.45% improvement on average. The DBP model has slightly better improvement, about 2.67% on average. Since DBP model updates and prefetch the data at commit stage, in many tight-loop codes, it was not able to return the data in time. While pipeline stages will be deeper and wider in future architecture, it is conceivable that the branch predictor will be smarter and lower the chance of polluting the prediction table. One should focus on designing the fast response prediction table.

[7] In the paper[69], it suggest 4K-entries (61KB) of CRT and 4K-entries (24KB) of ULT/RUT.

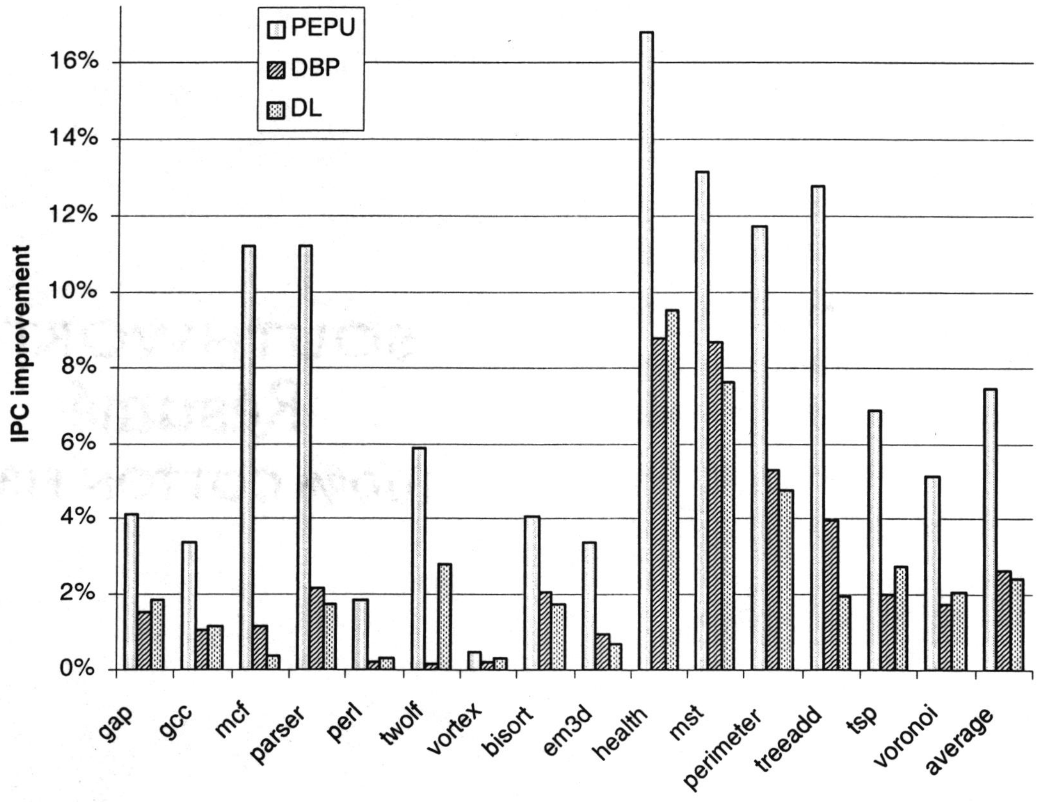


Figure 27 IPC improvement

PEPU model has better improvement than others, about 7.48%. "mcf" and "parser" have larger improvement , about 11%, among seven SPEC2000 benchmarks since these two benchmarks have larger percentage of Pointer loads as shown in Figure 20. "health", "mst" and "perimeter" results show that DBP model and DL model can handle LDS application better if such program have larger percentage of Data loads and Data-Address loads in Figure 20. If an application has more of these two types of Pointer load, more dependent instruction may benefit from data prefetching. Especially "health", since it has the largest percentage of Data-Address loads, it may reach 17% improvement. "vortex" has least performance improvement among three models since we observed it has highest instruction cache miss ratio, about 10%. Since the frond-end performance is ready hinder by sever instruction cache miss, little improvement may be seen at back-end stages.

There are two more reasons why PEPU outperform others model. First of all, in our study, we observed that store operations would update the behavior of Pointer loads. Figure 19(b) shows an exact example of this store-load memory dependency situation. DBP and DL model only use load operations information to keep tracking the possible future address. We waive this problem by allowing store operations update AC as well. Second, PEPU model also recognize the register-moving operations as a linker between child node and parent node as described in section 5.1.2. Therefore, we may identify more pointer loads in some applications and further improve the performance.

5.3.3 Prediction Accuracy

In this section, we present the address prediction accuracy of PEPU. Since when a Pointer load go through the left part of PEPU flow in Figure 26, the base address is known. There is no need to predict this load. Only those pending Pointer load at the right part of PEPU flow will have the predict address from the result of AC indexed by its producer. Except for "mst" and "tsp", most of the benchmarks show over 80% of prediction accuracy in Figure 28. We also compare the prediction accuracy of direct-map AC and 2-way associative AC. It shows similar result. The average prediction accuracy is from 83% to 87% as AC sizes vary from 128Bytes to 2Kbyes. The reasons why "mst" and "tsp" have worse prediction accuracy is caused by higher miss branch prediction ratio and pollute the AC. Therefore, "tsp" shows that for 128Byes AC the prediction accuracy is about 13% and 2Kbyes AC is about 34%. Figure 29 provides the prediction accuracy with perfect branch predictor. With perfect branch predictor, the address prediction accuracy of "mst" and "tsp" has impressive improvement, about 99%. This result also shows that the potential address prediction accuracy could reach 99% if more advanced branch predictor is used.

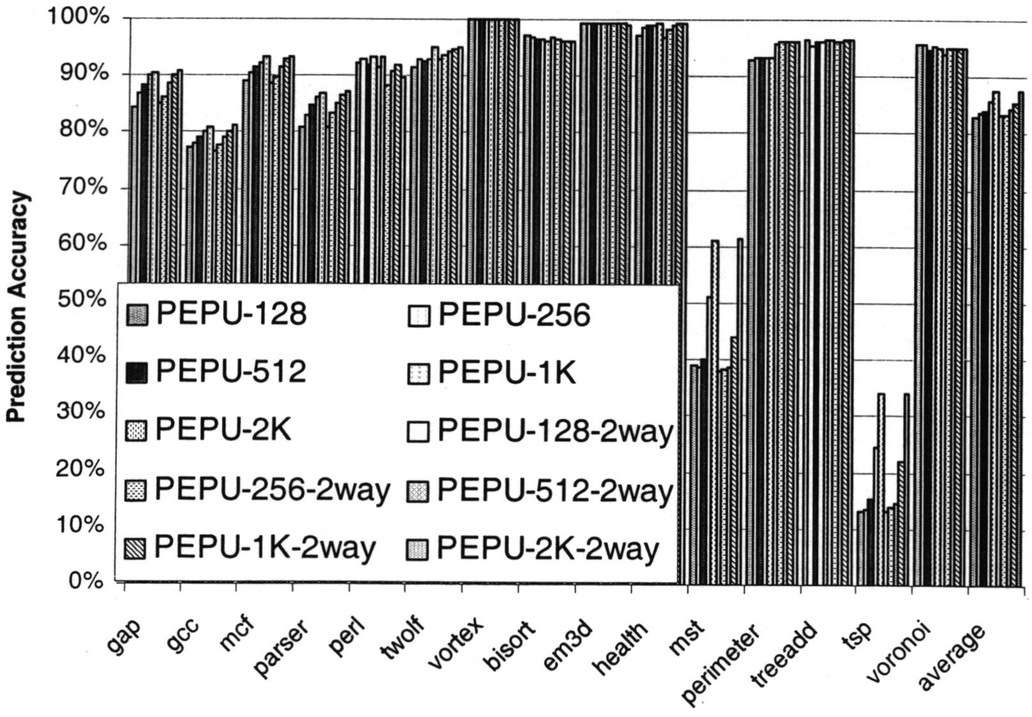


Figure 28 PEPU prediction accuracy with Gshare branch predictor

5.3.4 The Effect of Different size of Prefetch Cache

As the PFC varies from 128Byes to 4Kbyes, the performance improvement over base model is revealed at Figure 30. In this figure, we also compare the direct-map PFC and 2-way associative PFC. The average Olden benchmarks shows that direct-map PFC perform slightly better then 2-way associative PFC. It means that the address pattern in Olden benchmark may frequently hit to the same set and replace the line that will be needed soon. In other words, these benchmarks may frequently reference the nearby nodes. For SPEC2000 benchmarks, the result is opposite. Our study reveal that once a line is been replaced, it has less chance of been acquired again. Both direct-map and 2-way PFC illustrate that even with larger prefetch cache, the performance improvement is negligible.

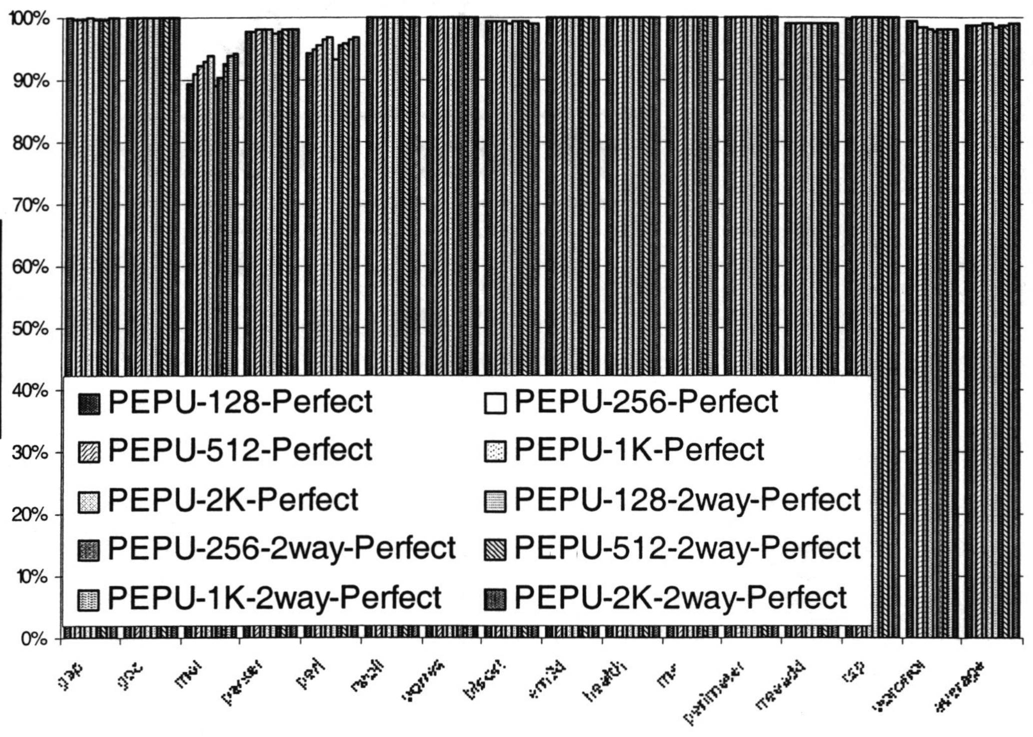


Figure 29 PEPU prediction accuracy with perfect branch predictor

In the study of Bloom filter scheme, data speculation allows instructions that are dependent on a load to be scheduled before the latency of the load is known. A simple approach is to speculate (predict) that the load will always hit the L1 cache and schedule its dependents accordingly. Unfortunately, whenever a prediction is wrong, the machine must recover all the mis-scheduled dependents, and performance suffers. In this paper we described how a Bloom Filter (BF) can be used to accurately predict cache misses. With a reasonably sized BF, we can correctly predict 99% of all misses. For the SPECint2000 benchmarks running on a modified SimpleScalar out-of-order model, the performance of a machine with a

BF improved by 19% over a machine that delayed the scheduling of the load's dependents until the load's hit/miss status was known, and by 6% over a machine that speculated loads always hit the cache. We have also shown that this performance improvement grows as the window size and branch prediction accuracy increase. We expect that our BF technique will have an even greater performance advantage as pipelines deeper and cache latencies increase.

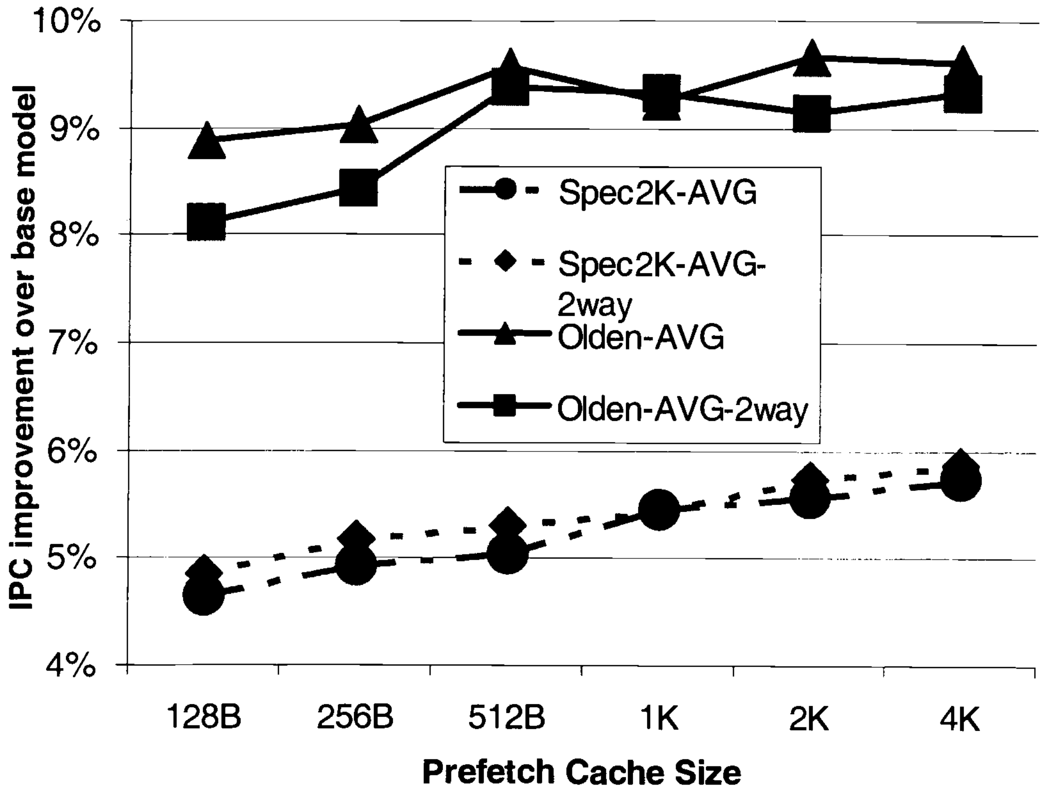


Figure 30 The effect of different size of prefetch cache size

CHAPTER 6 SUMMARIES

In the fault-tolerant microarchitecture we have presented the detail design of a fault tolerant superscalar processor called *Ditto Processor*. This processor re-fetches and re-decodes all instructions to protect all pipeline stages' logic from soft errors to assure high computation confidence. It requires little extra hardware on top of the baseline superscalar. We explain the additional microarchitecture resources needed and what units we can protect. We also explain how to handle the register renaming in *Ditto Processor*. We further identified that long latency operations have significant impact on time-redundant fault-tolerant superscalar processor. We studied the performance degradation of *Ditto Processor* in comparison with baseline superscalar and other published schemes. In general, *Ditto Processor* suffers only 1.8~13.3% of performance degradation for all benchmarks.

As *Ditto Processor* have only 1~6% more performance loss compared to O3RS scheme, our scheme have much better fault coverage. The degree of reduction varies with amount of contention on the resources brought about by duplication. We also observed that as memory latency increases, the performance degradation on *Ditto Processor* is reduced. While memory processor performance gap continues to grow with technology advancement, there will be more stalled cycles available for time redundancy. Our study reveals that different applications have different characteristics and have various requirements on hardware resources. Adopting the time-redundant fault-tolerant technique based on this knowledge would provide a balance designed fault-tolerant computing environment with less performance loss.

In the study of pointer-chasing problem, we present a novel hardware-only data prefetching mechanism to solve pointer-chasing problem. We first classify the pointer load into three types: Address load, Data-Address load and Data load. Most of the benchmarks are made of Data-Address load and data load. In order to identify the pointer load at run-time execution, we use a small bitmap structure to recognize the pointer load. As the instructions decoded, we further identify the types of pointer loads. We also implement a small address cache to profile the history of address pattern. We believe that once a data structure is established at program initial time, the address cache also recodes these addresses. After that initialize time, the program may expend or shrink the data structure. We use store operation to dynamically track these address changing. The prefetched data are also placed into a prefetch cache to not pollute the L1-D cache. In order to cooperate these units together, we define the flow chart of PEPU. The result shows that in an 8-way pipeline system, our scheme may outperform larger L1-D cache system by average 7% and 83% of address prediction accuracy. As more advance branch predictor is implemented, our predictor shows potential 100% address prediction accuracy.

BIBLIOGRAPHY

[1] J.L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach", San Mateo, CA 94403: Morgan Kaufmann Publishers, Inc., second edition, 1996

[2] D.A. Patterson, Thomas Anderson, Neal Cardwell, etc. "A case for Intelligent RAM: IRAM", IEEE micro, April 1997

[3] IBM Corp, "Understanding Static RAM Operation", Application note, March, 1997

[4] Jim Handy, "The cache memory book", ", San Mateo, CA 94403: Morgan Kaufmann Publishers, Inc., second edition, 1997

[5] P. Glaskowsky, "Pentium 4 (Partially) Previewed". Microprocessor report, Aug. 28, 2000.

[6] G. Hinton, et. al. "The Microarchitecture of the Pentium 4 Processor". Intel Technology Journal, 1st Quarter, 2000.

[7] R. Kessler, "The Alpha 21264 Microprocessor". IEEE Micro, Vol. 19(2), March/April 1999, pp.~24--36.

[8] A. Yoaz, M. Eerz, R. Ronen, S. Jourdan, "Speculation Techniques for Improving Load Related Instruction Scheduling". Proc. of 26th Int'l Symp. on Computer Architecture, Atlanta, Georgia, May 1999, pp. 42--53.

[9] B. Bloom, "Space/Time Tradeoffs in Hash Coding with Allowable Errors," Communications of The ACM, Vol 13(7), July 1970, pp.~422--426.

[10] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report #1342, CS Department, Univ. of Wisconsin-Madison, June 1997.

[11] B. Calder and D. Grunwald, "Next Cache Line and Set Prediction," Proc. of 22nd Int'l Symp. on Computer Architecture, S. Margherita Ligure, Italy, June 1995, pp. 287--296.

[12] K. Teager. "The MIPS R10000 Superscalar Microprocessor." IEEE Micro, 16(2):28-41,1996

[13] E. Morancho, et. al. "Recovery Mechanism for Latency Misprediction." Proc. of 10th Int'l Conf. on Parallel Architectures and Compilation Techniques, 2001.

[14] R. Kessler, R. Jooss, A. Lebeck, and M. Hill. "Inexpensive Implementations of Set-Associativity." Proc. of 16th Int'l Symp. on Computer Architecture, 1989.

[15] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. Jetty: Filtering Snoops for Reduced Energy Consumption in SMP Servers. Proc. of 7th Int'l Symp. on High Performance Computer Architecture, 2001.

[16] L. Liu. "Cache Design with Partial Address Matching." Proc. of 27th Int'l Symp. on Microarchitecture,1994.

[17] Compaq Computer Corporation. "Alpha 21264 Microprocessor Hardware Reference Manual", 1999.

[18] G. Chrysos and J. Emer. "Memory Dependence Prediction using Store Sets." Proc. of 25th Int'l Symp. on Computer Architecture, 1998.

[19] T. Juhnke and H. Klar, "Calculation of the Soft Error Rate of Submicron CMOS Logic Cicuits," IEEE JSSC, Vol. 30, No. 7, July 1995, pp. 830-834.

[20] J. Robertson, "Alpha Particles Worry IC Makers as Device Features Keep Shrinking," Semicond. Business News, October 21, 1998.

[21] N. Cohen et. al., "Soft Error Considerations for Deep-Submicron CMOS Circuit Applications," Proc. of IEDM, 1999, pp. 315-318.

[22] P. Hazucha and C. Svensson, "Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate," IEEE Trans. On Nuclear Science, Vol. 47, No. 6, Dec. 2000, pp. 2586-2594.

[23] T. Karnik et. al., "Scaling Trends of Cosmic Rays Induced Soft Errors in Static Latches Beyond 0.18um," Symposium on VLSI Circuits Design of Tech. Papers, 2001, pp. 61-62.

[24] U. Gunneflo, “Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation,” Digest of Papers in the 19th International Symposium on Fault-Tolerant Computing, 1989, pp. 340-347.

[25] G. Miremadi, and J. Torin, “Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection,” IEEE Transactions on Reliability, Volume: 44, Issue: 3 , Sept. 1995, pp. 441 –454.

[26] R. Horst et. al., “The Risk of Data Corruption in Microprocessor-based Systems,” Digest of Papers in the 23rd International Symposium on Fault-Tolerant Computing, Aug. 1993, pp. 576 –585.

[27] Barry W. Johnson, Design and Analysis of Fault Tolerant Digital Systems, Addison-Wesley, 1989.

[28] A. Avizienis, “Toward Systematic Design of Fault-Tolerant Systems,” IEEE Computer, April 1997, pp. 51-58.

[29] D. K. Pradhan , Fault-tolerant computer system design, Prentice-Hall , 1996.

[30] R.E. Blahut, Theory and Practice of Data Transmission Codes, Addison-Wesley, 1983.

[31] Parag K. Lala, Self-Checking and Fault-Tolerant Digital Design, Academic Press, 2001.

[32] G. Sohi, “A Study of Time-Redundant Fault Tolerance Techniques for High-performance Pipelined Computers,” Digest of Papers in the 19th International Symposium on Fault-Tolerant Computing, 1989

[33] J. H. Patel and L. Y. Fung, “Concurrent Error Detection in ALU’s by Recomputing with Shifted Operands,” IEEE Trans. On Computers, Vol. C-13, No. 7, July 1982, pp. 581-595.

[34] M. Nicolaidis, “Time redundancy based soft-error tolerance to rescue nanometer technologies,” Proceedings of 17th IEEE VLSI Test Symposium, 1999, pp. 86 –94.

[35] M. Franklin, “Incorporating Fault Tolerance in Superscalar Processors,” Proceedings of 3rd International Conference on High Performance Computing, 1996, pp. 301 –306.

[36] W. Torres-Pomales, “Software Tolerance: A Tutorial,” NASA Tech. Memorandum, TM-2000-210616, Langley Res. Center, Hampton Virginia, Oct. 2000.

[37] M. Franklin, “A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors,” Digest of Papers in the 25th International Symposium on Fault-Tolerant Computing, Aug. 1995, pp. 207–215.

[38] A. Mendelson and N. Suri, “Designing High-Performance and Reliable Superscalar Architectures The Out of Order Reliable Superscalar(O3RS) Approach,” Proc. Conference on Dependable Systems and Networks, 2000.

[39] Joydeep Ray, et. al., ”Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery”, In proceeding of 34th Microarchitecture, December,2001.

[40] T. Austin, “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design”, In proceeding of 32nd Microarchitecture, November, 1999.

[41] E. Rotenberg, “AR-SMT: a microarchitectural approach to fault tolerance in microprocessors,” Digest of Papers in 29th International Symposium on Fault-Tolerant Computing. 1999, pp. 84 –91.

[42] K. Sundaramoorthy et. Al. "Slipstream Processors: Improving both Performance and Fault Tolerance,” 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[43] F. Rashid et. al., “Fault Tolerance Through Re-execution in Multiscalar Architecture,” Proc. Conference on Dependable Systems and Networks, 2000, pp. 482 –491.

[44] Steven k. Reinhardt et. al. "Transient Fault Detection via Simultaneous Multithreading", In proceedings of the 27th International Symposium on Computer Architecture, June, 2000

[45] A. Avizienis and Y. He, Microprocessor entomology: a taxonomy of design faults in COTS microprocessors, Dependable Computing for Critical Applications 7, 1999, pp. 3 –23.

[46] A. Avizienis, "A Fault Tolerance Infrastructure for dependable Computing with High-Performance COTS Components", In proceeding of Dependable Systems and Networks, 2000.

[47] Y. He et .al ."Assessment of the applicability of COTS microprocessors in high-confidence computing systems: a case study," Proceedings International Conference on Dependable Systems and Networks, 2000

[48] Joel Nickel, "REESE: A Method of Soft Error Detection in Microprocessors," M. S. Thesis, Dept. of ECE, Iowa State University, Ames Iowa, 2000.

[49] T. Austin, "DIVA: A Dynamic Approach to Microprocessor Verification", The Journal of Instruction-Level Parallelism Volume 2, 2000.

[50] Saugata, et. al. "Effective Checker Processor Design", in proceeding of 33rd Microarchitecture, December 2000

[51] Doug burger, Todd M. Austin "Simplescalar Tool Set Version 2.0," Technical Report #1342, June 1997,University of Wisconsin-Madison Computer Science Department

[52] http://www.spec.org/osg/cpu2000/docs/readme1st.txt

[53] "Data prefetching with co-operative caching", C.H. Chi, S.L. Lau. 5th International Conference On High Performance Computing, 1998, page(s): 25 –32

[54] "Hardware versus hybrid data prefetching in multimedia processors: a case study", Pimentel, A.D., Hertzberger, L.O, et. al. International Conference on Performance, Computing, and Communications, 2000, page(s): 525 – 531

[55] “Hardware and software cache prefetching techniques for MPEG benchmarks”, Zucker, D.F.; Lee, R.B.; Flynn, M.J. IEEE Transactions on Circuits and Systems for Video Technology, Volume: 10 Issue: 5, Aug. 2000 page(s): 782 –796

[56] “Branch-directed and stride-based data cache prefetching”, Yue Liu; Kaeli, D.R., IEEE International Conference on VLSI in Computers and Processors, 1996, page(s): 225 –230

[57] “A comparison of data prefetching on an access decoupled and superscalar machine”, Jones, G.P.; Topham, N.P. IEEE/ACM International Symposium on Microarchitecture, 1997, page(s): 65 –70

[58] “Data prefetch mechanisms”, Vanderwiel, S.P., Lilja, D.j. ACM Computing Surveys, Vol. 32, No. 2, June 2000

[59] “Effective hardware-based data prefetching for high-performance processors”, Chen, T.F., Baer, J.L, IEEE Transactions on Computers, Volume: 44 Issue: 5 , May 1995, page(s): 609 –623

[60] “Tango: a hardware-based data prefetching technique for superscalar processors”, Pinter, S.S.; Yoaz, A. International Symposium on Microarchitecture, 1996, page(s): 214 –225

[61] “Dead-block prediction & dead-block correlating prefetchers”, A.C. Lai; Fide, C.; Falsafi, B.International Symposium on Computer Architecture, 2001,page(s): 144 –154

[62] “A dynamic data prefetching method of improving the memory latency”, J.F. Tu; Y.H. Wang; L.H. Wang, International Conference on High Performance Computing in the Asia-Pacific Region, 2000, page(s): 13 -18 vol.1

[63] “A new voting based hardware data prefetch scheme”, S. Manku, G.; Prasad, M.R.; Patterson, D.A. International Conference on High-Performance Computing, 1997, page(s): 100 –105

[64] "Speculative Execution via address prediction and data prefetching", J. Gonzalez, A. Gonzalez, International conference on supercomputing, July, 1997

[65] "Dependence based prefetching for linked data structures", A. Roth, A. Moshovos, G.S. Sohi, International conference on Architectural support for programming languages and operating systems October 1998

[66] "Memory-Side prefetching for linked data structures", C.J. Hugues, S. Adve, UIUC CS Technical Report UIUCDCS-R-2001-2221, May 2001

[67] "Modeling load address behavior through recurrences", Ramos, L.; Ibanez, P.; Vinals, V.; Llaberia, J.M., 2000. International Symposium on Performance Analysis of Systems and Software, 2000, page(s): 101 –108

[68] "Correlated load-address predictors", Bekerman, M.; Jourdan, S.; Ronen, R. et. al. International Symposium on Computer Architecture, 1999, page(s): 54 –63

[69] "Direct load: dependence-linked dataflow resolution of load address and cache coordinate", B. K. Chung; J. Zhang; J. K. Peir; S.C. Lai; International Symposium on Microarchitecture, 2001, page(s): 76 –87

[70] "Prefetching using Markov predictors", Joseph, D.; Grunwald, D. IEEE Transactions on Computers, Volume: 48 Issue: 2, Feb. 1999, page(s): 121 – 133

[71] "Predictor-Directed Stream Buffers", Sherwood, T.; Sair, S.; Calder, B. International Symposium on Microarchitecture, 2000, page(s): 42 –53

[72] " Compiler-based prefetching for recursive data structures", C.K. Luk, T.C. Mowry. International conference on Architectural support for programming languages and operating systems October 1996, Volume 30 Issue 5

[73] "The IA-64 architecture at work", Dulong, C. Computer , Volume: 31 Issue: 7 , July 1998, page(s): 24 –32

[74] "Data flow analysis for software prefetching linked data structures in Java", Cahoon, B.; McKinley, K.S. International Conference on Parallel Architectures and Compilation Techniques, 2001, page(s): 280 –291

[75] "Push vs. Pull: Data movement for linked data structures", C.L. Yang, A. R. Lebeck, International conference on Supercomputing May 2000

[76] "A prefetching technique for irregular accesses to linked data structures", Karlsson, M.; Dahlgren, F.; Stenstrom, P. International Symposium on High-Performance Computer Architecture, 1999, page(s): 206 –217

[77] "Effective jump-pointer prefetching for linked data structures", Roth, A.; Sohi, G.S., International Symposium on Computer Architecture, 1999, page(s): 111 –121

[78] "Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching", Z. Zhang; Torrellas, J. International Symposium on Computer Architecture, 1995, page(s): 188 – 199

[79] "Pointer-Cache Assisted Speculative Precomputation", J. Collins, S. Sair, B. Calder, D. Tullsen, International Symposium on Microarchitecture, 2002

[80] "Recovery mechanism for latency misprediction" Morancho, E.; Llaberia, J.M.; Olive, A. International Conference on Parallel Architectures and Compilation Techniques, 2001 Page(s): 118 –128

[81] Olden benchmark, http://www.cs.princeton.edu/~mcc/olden.html