STORE BUFFERS: IMPLEMENTING SINGLE CYCLE STORE INSTRUCTIONS IN WRITE-THROUGH, WRITE-BACK AND SET ASSOCIATIVE CACHES

by

Radhika Nagpal

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May, 1994

Signature of Author [illegible]

Department of Electrical Engineering and Computer Science, May 15, 1994

Certified by [illegible]

Prof. Anant Agarwal, Thesis Supervisor (MIT)

Certified by [illegible]

Rae McLellan, Company Supervisor (AT&T Bell Labs)

Accepted by [illegible]

F. R. Morgenthaler, Chair, Department Committee on Graduate Students

Eng.

Store Buffers: Implementing Single Cycle Store Instructions in Write-through, Write-back and Set Associative Caches

by

Radhika Nagpal

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of Bachelor of Science and Master of Science at the Massachusetts Institute of Technology.

Abstract

This thesis proposes a new mechanism, called *Store Buffers*, for implementing single cycle store instructions in a pipelined processor. Single cycle store instructions are difficult to implement because in most cases the tag check must be performed before the data can be written into the data cache. Store buffers allow a store instruction to read the cache tag as it passes through the pipe while keeping the store instruction data buffered in a backup register until the data cache is free. This strategy guarantees single cycle store execution without increasing the hit access time or degrading the performance of the data cache for simple direct-mapped caches, as well as for more complex set associative and write-back caches. As larger caches are incorporated on-chip, the speed of store instructions becomes an increasingly important part of the overall performance.

The first part of the thesis describes the design and implementation of store buffers in write-through, write-back, direct-mapped and set associative caches. The second part describes the implementation and simulation of store buffers in a 6-stage pipeline with a direct-mapped write-through pipelined cache. The performance of this method is compared to other cache write techniques. Preliminary results show that store buffers perform better than other store strategies under high IO latencies and cache thrashing. With as few as three buffers, they significantly reduce the number of cycles per instruction.

Thesis Supervisors: Rae McLellan (AT&T Bell Labs, DMTS)
Dr. Anant Agarwal
(MIT, Assoc Prof of Electrical Engineering and Computer Science)

Acknowledgements

The research for this thesis was performed at AT&T Bell labs. The design and implementation of the processor and 'store buffers' was done with my advisor, Rae McLellan, (DMTS, AT&T Bell Labs). Other support in terms of an assembler and compiler were provided by Phil Winterbottom and Sean Dorward. The thesis was written at MIT and supervised by my AT&T and MIT advisors, Rae McLellan and Anant Agarwal.

I would like to acknowledge the help of Daniel Coore, an MIT colleague, with whom I had several discussions while designing the processor, one of which directly lead to the idea of store buffers. I would also like to thank the following people for their help and support: Prof Thomborson (University of Minnesota), Alan Berenbaum (AT&T), Tom Syzmanski (AT&T). A special thanks to Sumeet Sandhu and Quinton Zondervan for their support and patience.

Finally, a thanks to my parents, Dr Vijay Nagpal and Mrs. Kamini Nagpal, and my brother, Gaurav, without whom I wouldn't have made it through these past 5 years at MIT. The due date for this thesis happens to be my parents' 25th wedding anniversary. Congratulations mom and dad!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In most programs compiled for RISC machines, the frequency of load instructions exceeds the frequency of store instructions. Loads also tend to be more of a performance bottleneck. They block the instruction stream since subsequent instructions that depend on the load cannot execute until the result of the load is available. For these reasons, more emphasis has been placed on optimizing the performance of loads in data caches. Not much literature is available on optimizing store instruction performance.

Store instructions in a conventional RISC pipeline are difficult to implement because in most cases, the cache tag must be checked for a hit before the data can be written into the data array. Sequential access to the tag and data arrays forces the store instruction to spend two cycles in the Data Cache Access (MEM) stage, thus stalling the pipe and wasting precious pipeline slots. Unlike load instructions, the result of the store is not required by subsequent instructions. Hence the execution is held up unnecessarily.

While there are ways to implement single cycle stores, most either require considerable silicon area or are limited to simple direct-mapped caches and exclude more complex cache organizations [11] [7] [4]. As processor speeds outstrip off-chip memory speeds, it is necessary to move from simple direct-mapped caches to more complex set associative and write-back cache organizations to reduce dependencies between pipeline execution and off-chip I/O speed. Hence for high performance designs, the speed of stores becomes an increasingly important part of the overall performance.

This thesis investigates the concept of **Store Buffers** as a technique for achieving single cycle store instructions and as a more general method of implementing delayed updating of the data cache.

1.1 Introduction to Store Buffers

Store buffers do not eliminate the problems associated with store instructions but simply take advantage of three characteristics. First, there are no inherent dependencies between the store instruction result and subsequent instructions. Store instructions affect execution solely as an artifact of the pipeline implementation. Second, the data cache stage unlike other stages is only used by load and store instructions during other instructions. Therefore it is not used by most types of instructions and is frequently free. Third, as long as data cache coherency is maintained, it is not necessary for the cache to be immediately updated. The result of the store instruction is not critical to the instruction stream flow.

The basic idea behind a store buffer is that the cache is queried as the store instruction passes through the MEM stage, but the actual updating of the cache is queued in a store buffer to be performed at a more convenient time without stalling the pipe.

Store buffers are functionally equivalent to write buffers. They can be implemented as a FIFO or a set of hardware buffers that are accessed in parallel with the data cache. A store instruction passing through the MEM stage of the pipe simultaneously reads the data cache tag and writes the store data into a store buffer. Later, during a cycle when the data cache is unused, either because there is an empty slot in the pipe or an ALU instruction, the data is written in the cache and the store buffer freed. While the store buffer is holding new data and waiting for a convenient time to update the data cache, load instructions can retrieve this data by performing an associative lookup on the store buffer. With multiple store buffers, several consecutive store instructions can proceed without stalling the pipe.

The advantage of the store buffer mechanism is its conceptual simplicity and straightforward implementation. It implements single cycle store instructions by making efficient use of pipeline slots when the data cache is unused. It can be implemented without increasing the critical timing path and increasing the hit access time of the data cache. Unlike other methods, e.g. blind-writing, it is not restricted to write-through direct-mapped caches but can also be used with write-back and set associative caches.

Store buffers can also support more complex caches. In write-through caches the write-buffers can be converted to store buffers with minimal additional state. For set associative caches, the set number is also stored with the data. For write-back caches the same applies, however it may also be necessary to provide multiple ports to the write buffers to allow both dirty data and store instruction data to be written.

1.2 Outline of Subsequent Chapters

Chapter 2 provides a survey of work related to the implementation of fast store instructions. Chapter 3 describes the implementation of store buffers in various cache configurations and how different hit/miss strategies can be supported. Chapter 4 describes the implementation of a RISC processor and its simulation environment. The design of the processor was the first phase of the thesis work. This processor was used as a testbed for experimenting with store buffers and comparing it to other store instruction strategies. Chapter 5 describes the implementation of store buffers in this processor. Chapter 6 compares store buffers quantitatively (through simulation) to other store strategies like 2-cycle stores and blind-writing.

1.3 Terminology

Before discussing store buffers, I define the terminology used in this thesis. More detailed explanations can be found in [7], [15] and [11].

Direct-Mapped Cache:

A direct-mapped cache is a cache in which the cache line is indexed by the low order bits of the address and the high order bits are stored as a tag associated with each line. A particular memory location may reside in only one entry of a direct-mapped cache.

Set Associative Cache:

A set associative cache is like a direct-mapped cache except that the low order address bits index a set of lines. Each line, or element, of a set has a tag associated with it. Hence a particular memory location may reside in any element of the set indexed by its low order bits. The higher order bits of an address match at most one tag in a set identifying the element in which the address is cached. An N-way set associative cache has N elements per set. A direct-mapped cache is a 1-way associative cache.

Thrashing:

In a direct-mapped cache, when two addresses have the same cache index but different tags they are said to 'collide' in the cache. If there are several memory operations to these locations, they will constantly miss in the cache because they overwrite each other. This phenomenon is called 'thrashing'.

Write-Through Cache:

Write-through is a write (store) policy in which any value being written to the cache must also be written out to the corresponding memory location. Memory data is always consistent with cache data.

Write-Back Cache:

Write-back is a write (store) policy in which values can be written to the cache without updating main memory. Such lines are marked dirty in the cache. The dirty data gets flushed to memory only when the cache line containing the data is being replaced.

Write-Allocate:

Write-allocate is a write miss policy in which a store miss results in the allocation of a line in the cache. If partially valid lines are not supported in the data cache, then this must be used with a fetch-on-miss policy.

No-Write-Allocate:

No-write-allocate is a write miss policy in which the address being written to by a store that misses is not allocated in the cache. Stores that hit still update the cache.

Fetch-On-Miss:

Fetch-on-miss is a cache miss policy in which the entire cache line is fetched for an address that misses in the cache.

Blind-Writing:

Blind-writing is a store instruction strategy for direct-mapped write-through caches in which data is written concurrently with the tag check. If the tag check indicates a hit, the store instruction proceeds down the pipeline. If it is a miss, then the store instruction stalls and either corrects the tag and valid bits or invalidates the cache line. This is also referred to as *write-before-hit.*

Write-Buffers:

A write-buffer is a set of buffers that queues memory write requests and decouples the completion of these requests from normal pipeline execution. Write-through caches use these buffers to write all store data to memory while write-back caches use them to flush dirty data to memory when the cache line is being replaced.

Load Reservation Stations:

A load reservation station is a set of buffers that queues memory read requests by loads that miss in the cache. It is used to implement non-blocking loads.

Single Cycle Store Instruction:

A single cycle store instruction is one that passes through the pipeline without spending more than one cycle in any stage.

In subsequent chapters, in describing store instructions that write data to cache and memory, cache writes are referred to as *cache updates* or *updates* and main memory writes are referred to as *memory writes* or *writes.*

Chapter 2

Survey of Related Work

Several papers and books discuss different cache configurations and compare their relative performance ([1], [8], [15]). However most of them emphasize load performance and focus on methods for dealing with reads and read misses. Jouppi[11] and Hennessey[7] are among the few that discuss the different policies for store hits (write-back and write-through) and store misses (write-allocate, fetch-on-write) in some depth and also give methods for performing fast stores. Jouppi provides a detailed and quantitative comparison of all the store hit/miss policies [11]. However neither discuss fast stores for set associative or write-back caches.

This is understandable because load instruction latency has a significant effect on performance. Loads can hold up the execution of the instruction stream because of data dependencies. However as caches become larger and clock speeds increase, the performance of store instructions begins to affect the overall performance as well. In a conventional RISC pipeline where the entire cache access is lumped into one stage – the MEM stage – a store instruction can take two cycles, one to check the tag and one to write the data. In contrast to other RISC instructions which execute in one cycle, this is can be a multi-cycle instruction. Although stores are about half as frequent as loads on average, if each store requires two cycles this will result in a 33% reduction in effective first-level cache bandwidth as compared to a machine that only requires one cycle per store [11]. Hence there is a motivation to eliminate this extra cycle penalty. The sections below describe some implementations of store instructions that attempt to achieve single cycle cache updates.

2.1 Blind-Writing Technique (or write-before-hit)

Jouppi and others suggest a method for performing fast writes ([7],[11]) where the data is blindly written into the data cache concurrently with the tag read, and hence before a *cache hit* has been determined. After the cache has been written, if the tag does not match, then the cache is incoherent and the pipe is stalled, either to fix the tag in caches with write-allocate or to invalidate the line in caches with no-write-allocate. Hence store instructions that hit in the cache execute in one cycle whereas store instructions that miss take two cycles. This shifts the extra cycle penalty from all stores to only those stores that miss in the cache. More aggressive techniques can be used to reduce this penalty further, such as stalling the pipe only if the next instruction after the store miss uses the data cache. If the subsequent instruction does not access the data cache, the invalidation/tag fix phase of the store that missed can be performed in the background. If the instruction immediately following the store instruction accesses the cache then it is necessary to stall since the cache is in an incoherent state. A no-write-allocate policy can be implemented efficiently by dual porting the valid bits so that in the case of a store miss the cache word can be invalidated in the background [11]. Write-allocate however requires writing the tag as well as valid bits and dual porting the tags can be slow and use extensive silicon area.

The disadvantage of blind-writing is that it has very limited application. It is restricted to only direct-mapped write-through caches. It can not be used with either set associative caches or write-back caches. In a set associative cache, the memory address being written by the store instruction may be cached in any of the elements of the corresponding set. It is not possible to determine a priori which element of the set to write the data in without first consulting the tags. In write-back caches, a line may contain unique data not yet written in memory. If store data is written to the cache blindly, it might overwrite this data. Therefore, it is necessary to determine the state of the tag and dirty bits before the update. In a write-through cache, both write-allocate and no-write-allocate policies can be implemented in conjunction with blind-writing. However, when a store instruction misses in a no-write-allocate cache, the old line gets invalidated even though the store instruction did not allocate that line.

2.2 IBM RS/6000

The IBM RS/6000 [6] implements a set associative cache by performing the tag access and comparison in series with the ALU. The tag lookup is done in the EXEC (ALU) stage and the data arrays are read in the MEM stage. This makes all store instructions single cycle but adds a severe delay to the ALU stage because the address calculation and tag lookup must be done sequentially. This increases the critical path through the pipeline. Another similar method is to move the data array access into the WB stage (stage after the MEM stage) after the hit or miss is known [11]. Then stores and loads would access the cache with the same timing and could be issued one per cycle. However, since load latency is of critical importance this is not really an attractive option. Both techniques achieve single cycle store performance by modifying the pipeline to remove the inherent problem of simultaneous tag and data array access. Both techniques can be used in direct-mapped caches as well as set associative and write-back caches with write-allocate or no-write-allocate miss policies.

2.3 VAX 8800

One method for fast writes mentioned by Hennessey [7] is implemented in the Digital Equipment Co. VAX 8800 [4]. This technique is very similar to the store buffer technique presented in this thesis, though less flexible. It implements single cycle stores without penalizing subsequent data cache accesses by using the data input latch of the cache as a backup register.

This technique takes advantage of the fact that the store instruction accesses the tag and data cache arrays in two separate phases. Therefore the tag and the data arrays are implemented with separate address selection and decode circuitry. A load instruction reads the tag and data simultaneously, hence it accesses both memory arrays. A store instruction *accesses the cache in two phases, one array per phase.* In the first phase, as the store is passing through the MEM stage, the corresponding tag is read (the data array is not accessed) and the store data is latched into the cache data input register at the end of the cycle. In phase 2, if there is a hit the data in the data-in register gets written to the data array. If there is a cache miss, the store instruction does not update the cache and the data in the data-in register is discarded.

None of the subsequent instructions need to be stalled. This can be verified as follows: If the subsequent instruction is not a load or store the data array is free therefore the data-in register can be written to the cache in the background. If the next instruction is a store, then the store will access only the tag array in phase 1 and the data array can be updated with the data of the previous store in the background. Hence consecutive stores can overlap. If the next instruction is a load there is a resource conflict since the load accesses both the tag and data arrays. Therefore the data in the data-in register is retained until a non-load instruction passes through the pipe, at which point it is written to the cache. A load must check the address of the data-in register to make sure it does not get the stale data cache value. The data-in register looks similar to a single entry write buffer.

Hence a store instruction always executes in a single cycle. This strategy is very simple to implement and does not aggravate any critical paths. It can be used with both direct-mapped caches as well as set associative caches.

The disadvantage of this strategy is that it requires a no-write-allocate miss policy. In the case of write-allocate, a store instruction that misses in the cache allocates a new line by writing both the tag and the data. This can no longer be performed in the background of another store instruction since the new 'phase 2' accesses the tag array too. Since there is only one backup register for the store instruction, it must be emptied before the new store instruction executes. Hence consecutive store instructions stall. If a line is not allocated on a store miss (no-write-allocate policy) this technique works perfectly. Although a no-write-allocate policy is easier to implement, it causes a significant increase in memory traffic since store data is often referenced soon after it is written [15] [8].

Store buffers use a concept similar to that of the backup register in the VAX 8800 to implement single cycle store instructions without constraining the cache configuration or penalizing the clock speed.

Chapter 3

Store Buffers

3.1 Concept

Store Buffer is a technique for achieving single cycle store instructions. More abstractly, it is a general method for implementing delayed operations on the data cache.

The concept of store buffers is similar to that of write buffers and load reservation stations. They both write to or read from main memory in the background without stalling the pipeline. Store buffers provide the same functionality for updating store instruction data in the data cache without holding up the pipe.

Store instructions are difficult to implement in a conventional RISC pipeline. Most RISC pipelines consist of five stages: Fetch, Decode, Execute, Memory(MEM) and Write-Back. The Execute stage is used for ALU operations and address generation, while the Write-Back stage writes the result back into the register file. The entire data cache access is lumped into the MEM stage. Cache access consists of accessing the data arrays and tag arrays in parallel and then checking cache validity by comparing the tag to the address.

This works well for load instructions. The data is read and its validity is checked simultaneously. If there is a miss the data is discarded and the word is requested from memory. In the case of a store instruction this arrangement does not work as well. Before the cache validity of the address is checked it is impossible to know if the data array should be written to or not.

For a load instruction the penalty of stalling the pipe on a load miss is justified since subsequent instructions may depend on its result. But a store instruction does not affect subsequent instructions, hence it should not affect the execution of those instructions by stalling the pipe.

The objective is therefore to have a *store instruction which passes through the pipeline without spending more than one cycle in any stage.* Store buffers accomplish single cycle stores by taking advantage of the following properties of store instructions to mask the penalty:

1. There are no *data dependencies* between a store instruction and subsequent instruc-

tions in the pipeline, therefore its result is not critical to the instruction flow.

2. Even though an extra cycle may be needed to write the data in the data array, the actual update need not be performed immediately. The *cache update may be delayed* if the information is preserved to prevent inconsistencies.

3. The MEM stage is used relatively infrequently compared to the DECODE or FETCH stages. This stage is used only by load/store instructions, and they are usually not a high percentage of the total instruction mix. Therefore the data cache is frequently available for reads or writes from outside the MEM stage.

In the store buffer model, as a store instruction passes through the MEM stage, it looks up the corresponding tag in the tag array and at the same time writes the data into a store buffer. The store instruction does not need to stall. When an instruction that does not access the data cache, such as an ALU operation, passes through the MEM stage, the buffered data is written into the cache and the store buffer is declared empty again. Hence the writing of the data occurs in the background without stalling the pipe whether the access hits or misses the cache.

With multiple store buffers, several consecutive stores can be tolerated without stalling the pipe. A store instruction would stall only if the store buffers are full. By increasing the number of store buffers, the probability of stalling a store instruction can be made arbitrarily small.

The store buffers look like a fully associative cache. The tag is the memory address and the data is the store value. For coherency, load instructions perform an associative lookup on the store buffer contents in parallel with the data cache read. If there is a store buffer hit, the load instruction takes the store buffer data, since it is the most recent write to that memory location and the cache has not been updated. If the data is also present in the cache, the store buffer data is given precedence since it is more recent than the cache entry.

Both write-through and write-back caches can use store buffers. The implementation is slightly different in each case.

3.2 In A Simple Write-Through Cache

ppIn a write-through cache the functionality of the store buffers closely resembles that of the write buffers. In fact they contain the same data since all store data is written to memory as well as the cache. Hence the write buffers can be readily converted to store buffers by enhancing them with an extra state bit to indicate a delayed cache update.

The purpose of the write buffer is to queue writes to memory. IO latency is high and the subsequent instructions are not concerned with the time it takes to write the data to memory. Therefore it does not make sense to hold up the entire pipeline until the write is completed. Write-through policy requires every store instruction to write its data to memory and therefore to the write buffer. There is a valid bit, called the *pending-write bit*), associated with each buffer that indicates that the data needs to be written to memory.

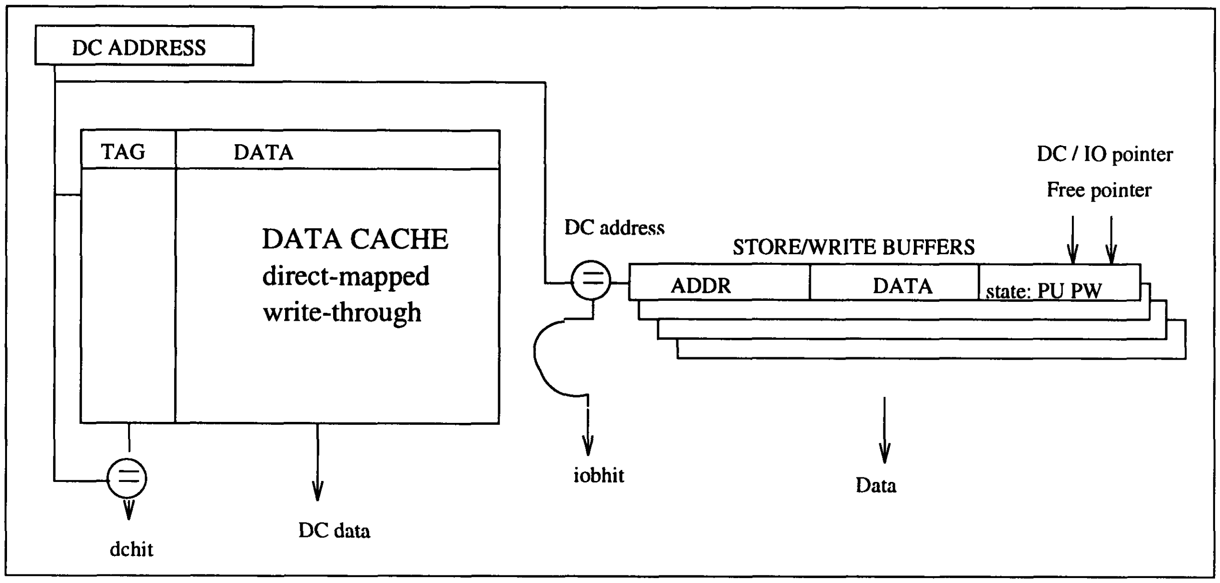

Figure 3.1: Store Buffers in a Write-through Cache

The functionality of store buffers can be achieved with a write buffer by adding some logic to the existing write buffer hardware. In addition to the original pending-write bit, there is an extra bit, called the *pending-update bit*), to indicate whether the data should be written to the data cache. Whenever a store instruction passes through the MEM stage, it is queued in the write buffer and both bits get set. Each valid buffer requests both the IO and the data cache. The data cache grants a a cache update if there is no instruction in the MEM stage accessing the cache. The cache is then updated from the write buffers. Once the data cache write is complete the pending-update bit is cleared. When the memory write is complete the pending-write bit is cleared. When both bits are off the buffer is considered free.

This makes the implementation inexpensive without adding delay through the MEM stage since the associative address lookup in the store buffers is done in parallel with the cache access (see figure 3.1). Notice that this lookup was performed in the write buffer implementation as well – a load instruction had to check the write buffer for values that had not yet been written in memory. Hence the store buffer lookup requires no additional address compare.

Since memory latency is in general greater than the availability of free cache pipe cycles, I expect the combined store/write buffers to fill up because of pending I/O, as opposed to pending data cache updates.

The following sections describe how store buffers can be implemented in direct-mapped caches with several words per line, and in set associative caches.

3.2.1 A Direct-Mapped Cache with Single-Word Lines

In a cache with single word lines, each word is associated with a tag and valid bit.

With a write-allocate policy, the new data is always written to the cache and the valid bit is always set to 1. Hence there is no need to check the tag at all. The store instruction updates both the tag and data during the MEM stage and always takes a single cycle. The store buffer becomes a simple write buffer.

For no-write-allocate, the pending-update bit is set only if the location was already in the cache as indicated by the tag comparison. Otherwise the write buffer gets the data with only the pending-write bit turned on. Hence, if the location exists in the cache, the store buffer marks that as a pending cache update, but if the location is not in the cache, space in the cache is not allocated and the data is only sent out to memory.

3.2.2 A Direct-Mapped Cache with Multi-Word Lines

In the case of multi-word lines, there is one tag associated with each line. The words in the same line are distinguished by the low order bits of the address. The valid bits can be per line or per word. We consider the case in which there is a valid bit per word to support partially valid lines. Multi-word lines with a single valid bit require the entire line to be resident in the cache. That makes store instructions particularly inefficient in the case of write-allocate because it forces a fetch-on-miss policy. To store a data word in the cache, an entire valid line must be allocated and fetched from memory. The IO overhead of a block fetch is incurred every time a store instruction misses. However most of the time this data is never read, and gets overwritten [11].

In a cache with a no-write-allocate policy, if the location is resident in the cache, the pending-update bit is set indicating a data and valid bit pending update. Otherwise the pending-update bit is cleared.

In the case of write-allocate with no fetch-on-miss, there are two types of cache misses:

Tag Hit, Valid Bit Miss: In this case a partially valid line already exists in the cache. Therefore the tag need not be written but the new word must be validated.

Tagmiss: In this case a whole new line needs to be allocated. Hence the tag needs to be updated, the valid bit for the new word needs to be set and the remaining words in the line needs to be invalidated.

Hence updating the cache requires three states: pending-update with no tag write, pending-update with a tag update and no pending-update. When the data cache is being updated from the store buffer, these state bits are used to generate the valid bits. These states can be implemented by two bits: the Pending-Update bit (PU) and the Tagmatch bit (TM).

PU	TM	STATE
1	1	Pending Update with No Tag Write
1	0	Pending Update with Tag Write
0	*	No Pending Update

Updates to the same location can be collapsed into the same store buffer as in coalescing write buffers. This has the advantage of reducing the traffic to memory as well as to the data cache. It also eliminates the problem of a load instruction finding multiple copies of the same location in the store buffer. This avoids the problem of detecting which was more recent.

In multi-word caches, there are also some non-obvious interactions of store buffers with subsequent load and store instructions. Interactions between pending-updates that modify the same cache line must be taken care of. There are two scenarios to be considered. For purposes of illustration, let a memory address be represented as $< Xyz >$ and a cache line as $< Xy >$, where X = tag, y = cache line index, z = word in line.

Scenario 1: Consider the case in which there are two store instructions S1, S2 to addresses $< Aa1 >$ and $< Aa2 >$. Let the cache initially contain the memory line $< Ba >$. When the first store, S1, queries the cache, it gets a tag miss, because A does not equal B, and is queued in the store buffer with state (PU,TM) = 10. Suppose when S2 queries the cache, S1 is still in the store buffer. Hence S2 will also get a tag miss and be queued with the state 10. If we let the data cache update proceed in a naive FIFO manner, S1 will update the tag in the cache line to A and invalidate all the words except itself. When S2 updates the cache it will not know that the tag has changed and it will re-write the tag and invalidate all the words except itself. It invalidates the data S1 wrote even though both had the same tag. Hence the line gets unnecessarily invalidated. The advantage of having multi-word lines to exploit spatial locality is lost. This is inefficient but not incorrect as it does not leave the cache in an incoherent state.

Scenario 2: Consider two store instructions S1, S2 to locations $< Aa1 >$, $< Ba2 >$ respectively. Let the cache contain line $< Ba >$. When S1 queries the cache it gets a tag miss and gets queued with (PU,TM) = 10. Suppose when S2 queries the cache, S1 is still in the store buffer. S2 gets a tag hit and its (PU,TM) bit = 11. When S1 updates the cache, it changes the tag from B to A and invalidates the remaining words. But since S2 has already passed the MEM stage and is queued in the store buffers, it does not see that the cache line's tag has changed. Therefore when it updates the data, it assumes the tag matched and is correct. It does not update the tag and validates itself. Hence cache location $< Aa2 >$ incorrectly contains the data of $< Ba2 >$. The cache is now incoherent.

This problem occurs because the state of a cache line may be changed between the time of query and the time the actual cache update is performed.

A simple way of keeping track of changes in the cache, is by placing the responsibility of correcting the store buffer state on the data cache update phase. When the store buffer updates the cache, it also performs a lookup in the store buffer and if a store buffer has the same cache index but a different tag, its TM bit is cleared since the cache no longer matches that store buffer address. If the store buffer has the same tag, its TM bit is set since the cache now matches the store buffer address. This corrects the problem since all interacting store buffers get corrected at the time the state of a cache line changes. Thus the state remains consistent. This works well for updates to contiguous memory, as in scenario 1, and each word in the same line remains in the cache. If the stores are conflicting, as in scenario 2, then the most recent cache update wins.

An alternate way of approaching this is to place the responsibility on the store instruction to cancel all updates to the same line is going to be invalidated. When a store instruction queries the cache, it does an associative lookup in the store buffers. If some buffer is scheduled to update the same cache line but has a different tag, turn off its PU bit and cancel the update since it would be overwritten anyways. This is a very efficient method since all spurious writes to the cache get canceled. However this seemingly simple method has a hidden problem. It is possible to leave incorrect data in the data cache and write the data to memory but not to the cache. The reason for this is that once the PU bit is canceled, there might be a stale copy of that location present in the data cache. Since the buffer has been freed there is no way of detecting the inconsistency. This can be corrected by not freeing the buffer until the cache line actually gets overwritten and there is no possibility of stale data being left in the cache. This method is more difficult to implement but is more efficient in reducing traffic to the data cache.

Just as it is necessary to take care of interactions between pending-updates to the same cache line, it is also necessary to pay attention to pending-updates and *blocking load instructions* that miss on the same cache line. The load miss modifies the cache line state and the store buffers need to be updated with this informations. During the load miss, the load address is compared to all the store buffers. The TM bit should be cleared for pending-updates with the same index but different tag and set for those with the same tag. This could be done during the time the load performs an associative lookup on the store buffers. But since this problem is only for loads that miss, the TM bits can be modified during the time the load is blocked in the MEM stage without penalizing normal load execution.

Non-blocking loads can be readily combined with store buffers. This is covered later in the section on IO buffers.

3.2.3 A Set Associative Cache

A set associative cache is similar to a direct-mapped cache but each address indexes a set of several lines, elements, instead of just one. An N-way cache has N elements per set. Each line has a distinct tag. The lines may be multi-word.

To lookup a word in the cache, the address indexes a set and checks the tag for each element of the set in parallel. The result of the tag comparisons is used to select the correct data array line.

The critical path through the cache tends to be the tag lookup and comparison selecting the data. The *access time = (tag array access) + (N tag comparisons checked in parallel) + (selecting multiplexor for data line).* For a direct-mapped cache the tag comparison is not needed to select the data therefore the access time is just *tag array access + tag lookup.* Hence a set associative cache access time is longer than that for a direct-mapped cache of the same size [9][10]. Set associative caches have several advantages over direct-mapped caches [15]. A set associative cache provides a higher hit rate than a direct-mapped cache of the same size. The problem of thrashing is greatly reduced. The tradeoff is between the performance gain from a higher hit rate and the performance decrease because of a potentially slower clock.

Implementing store operations in set associative caches is difficult because the element of the set in which the location resides is not known until after the tag comparison. Therefore methods like blind-writing cannot be used. Store buffers on the other hand can be readily modified to support set associative caches.

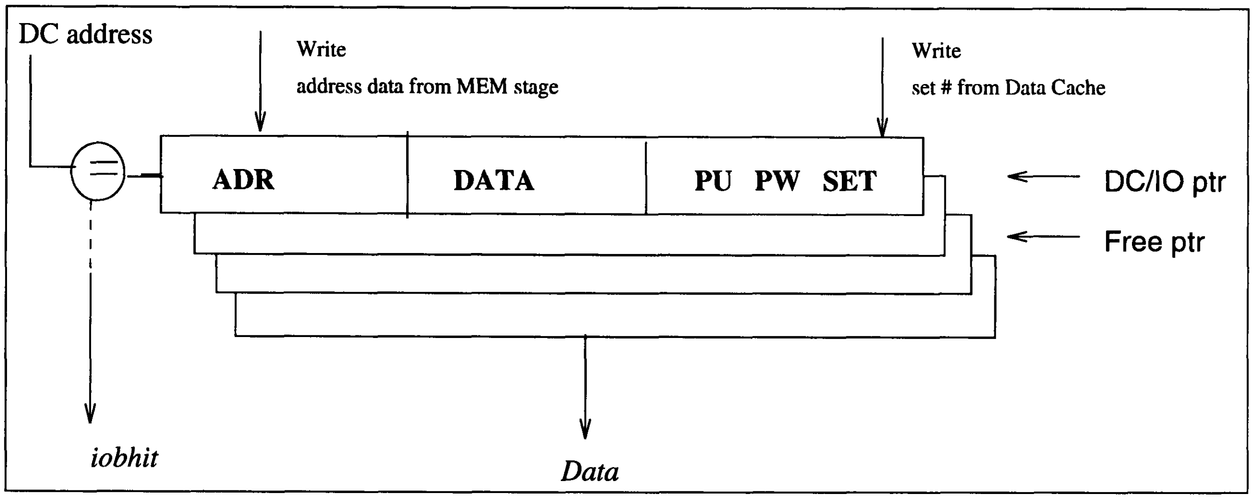

Figure 3.2: Store Buffers in a Set Associative Cache

The store/write buffer described in the previous section on direct- mapped caches can be extended to work for an N-way set associative cache simply by the addition of the element number in the store buffer state. The remaining state is the same as for direct-mapped caches with multi-word lines. The set number is then used to choose the set when the data is actually written to the cache. An important feature of this technique is that store buffers do not lie in the delay path through the set associative cache.

Store buffers make store instruction implementation in set associative caches very simple. The main disadvantage of set associative caches is that the access path for a hit is more than that for a direct-mapped cache of the same size. This problem can be alleviated by pipelining the cache over two stages, the MEM and Write-Back stages. The data array access is done in the MEM stage and the tag comparisons and line selection are done in the Write-Back stage. Hence, the access path through the cache in the MEM stage becomes the same as that of a direct-mapped cache. Store buffers can be easily extended to work in a pipelined cache. This is explained in Chapter 5.

3.3 In A Simple Write-Back Cache

In a write-back cache, a store instruction only writes data to the cache. This reduces memory traffic. The data already in the cache may be *dirty*, which means it is not yet updated in memory. If the line being overwritten by a store or a load miss is dirty, then it must first be written to memory. Most implementations incorporate a write buffer to queue the dirty cache line write to memory to allow the more critical load miss to be serviced first.

As a store instruction passes through the MEM stage, the tag and data cache arrays are read while the new data is written to the store buffer. If the tag misses and the line read is valid and dirty, it gets written into the write buffer. In this case the store buffer and write buffer do not contain the same data. A load instruction must perform a lookup in both buffers to maintain coherency.

There are at least two ways of implementing write-back caches with store buffers. The

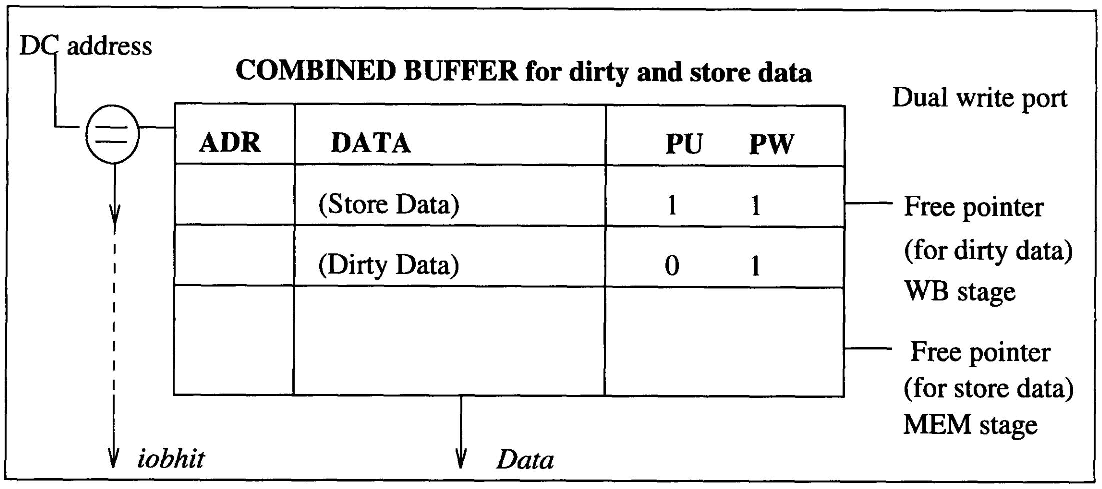

Figure 3.3: Store Buffers in a Write-back Cache

first implementation has separate write and store buffers. The second has a single set of buffers for both store and dirty data. Combining both the store buffers and write buffers is possible, since they have identical hardware and do associative lookups against the same address. However there is a disadvantage. Since a store instruction writes both the store data and potentially the dirty data to the buffers, the buffers must be dual-ported. The advantage of combining them is that the buffers get more efficiently partitioned amongst write buffers and store buffers.

3.4 Concept of IO Buffers

The purpose of delaying operations on the data cache is to avoid unnecessary pipe stalls as much as is possible. Only *real hazards*, caused by data dependencies with data not yet retrieved from memory, should stall the pipe. The same idea can be applied to load instructions that miss in the cache, leading to the idea of non-blocking loads. Store buffers can be modified to incorporate non-blocking loads.

3.4.1 Non-blocking Loads

Load instructions force the pipe to stall because of data dependencies. However blocking loads have the property that they cause *artificial stalls*, i.e. if the load misses in the data cache it holds up the pipe even if none of the instructions currently in the pipe require the result of the load. If the load instruction misses, it should be queued as a request to main memory. The pipe should continue to proceed until some instruction depends on the load data. This is the idea behind non-blocking loads. They allow compilers to hide load miss latency by moving the load instruction earlier than the use so that even if it misses the load latency can be hidden by the execution of other instructions.

Load reservation stations can be used to implement non-blocking load instructions [6]. Load instructions that miss are queued in a buffer and are removed from the pipe so that the

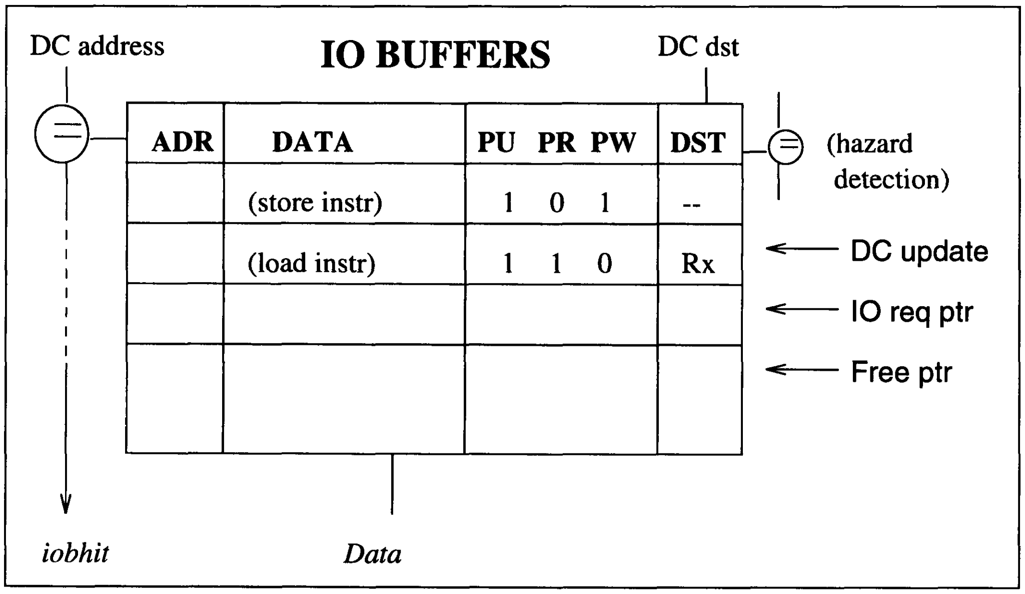

Figure 3.4: IO Buffers

instructions in the pipe can proceed. The buffer then requests memory for that address. Each buffer consists of an address and a register destination field. When the value arrives from memory it gets written to the register file as well as the cache. There needs to be hazard detection on these register destinations to detect a data dependency with subsequent instructions in the pipe. Also there must be detection for potential write-after-write (WAW) hazards on the register set caused by loads that were issued earlier completing out of order with subsequent register writes.

On the whole, the concept and the hardware logic is similar to that of store buffers and write buffers: IO requests are being queued in a set of buffers with a request bit set. Hence we can combine this set of buffers with our store/write buffers to create generic IO buffers.

3.4.2 IO Buffers

The advantage of having a combined set of buffers is:

1. Better allocation of hardware resources.
2. All external IO requests originate from a single source, the IO buffers.
3. All data cache writes, store or load miss, are performed from the IO buffers (hence the IO and Data Cache interfaces are simplified)
4. Easy to implement in hardware as a single set of buffers and control

What would the state of a combined IO buffer look like? Each buffer could contain a memory read request, a memory write request and/or a data cache update request. Hence each buffer has 3 state bits:

PU	PR	PW	State Description
1	1	0	Cacheable Load Request
0	1	0	Non-Cacheable Load Request
1	0	1	Cacheable Store Request
0	0	1	Non-Cacheable Store Request
1	0	0	Cache Update Request
0	0	0	Free Buffer
0	1	1	illegal
1	1	1	illegal

Table 3.1: IO Buffer State Bits

pending-read (load)
pending-write (store)
pending-update (data cache write)

A load that misses would get queued with state (PU, PR, PW) as 110 (or 010 if non-cacheable). A store would get queued as 101. The PR and PU bits request IO service and get cleared when completed. Only buffers with the state 101 or 100 can be updated in the cache. If the PR bit is set then the data has not yet been read from memory so the pending cache update is delayed. Table 3.1 enumerates the possible states.

Each buffer must contain, besides the state bits, an address that can be associatively compared to the address in the MEM stage. It must also contain a register to hold the store instruction data. This all exists in store buffers. There needs to be a register field per buffer for load instructions and hazard detection logic for register conflicts.

Load cache and register updates: Since the buffers already have space to hold store data, the data read from memory by a pending load can be written into the buffers. Rather than stalling the pipe to write the cache from IO, the cache can be updated from the store buffer when there is an available slot in the exact same way store requests update the cache. However the load data should be written to the register file as soon as possible as subsequent instructions depend on it. There are several methods for implementing the register update:

1. Mark buffers as *pending register writes* and then wait for a slot in the Write-Back stage when there is an instruction not writing to the register file or the Write-Back stage is empty. Neither situation occurs frequently in an efficient pipeline. Ultimately the DECODE stage will stall on the value of the load, forcing a register update. Therefore this is not a very efficient method.

2. Stall the pipe to write the data when it arrives from memory. This is still an improvement over blocking loads, since this causes a single stall cycle while blocking loads stall the pipe for the entire IO transaction.

3. Have a second write port to the register file so that when the data arrives it can be immediately written to the register file as well as the IO buffer for that load. This is more expensive but more efficient than the other methods.

Load Fetch strategy: One of the advantages of multi-word lines is that an entire line may be fetched when a load misses. This is because loads exhibit spatial locality and block transfers provide greater IO bandwidth than requesting each word separately. If the miss strategy is fetch-on-miss then the first load that misses in the cache can request the entire line and subsequent requests to the same line could be merged within the same buffer. That would require line size buffers. This makes it more expensive to combine load reservation stations with store buffers. There are different ways of implementing this (see Chapter 5). One way is to have a fixed number of line size IO buffers. For loads that miss on partially valid lines, or if the full line buffers are full, request only the word needed by the load instruction.

3.4.3 Scheduling Requests

The store buffers and write buffers request data cache update slots and IO service in a simple FIFO manner. There is a single pointer into the buffer that is incremented only when a buffer completes both the data cache update and the memory write. If load requests are included, it is still possible to schedule the requests in a FIFO order. But that is very inefficient since memory writes are not as critical as memory reads. Memory reads can produce real dependencies in the pipe. Similarly, store data can be written to the cache immediately from the IO buffers whereas load instructions must wait until the data arrives from memory to write it to the cache. Hence their cache update latencies are very different as well. Therefore it is advantageous to remove relative ordering between requests and simply request more important services first. This is called relaxed consistency [5].

Pending Memory Requests: Although a FIFO will keep the cache coherent, the relative order of loads and stores is not of much consequence as long as the order to the same address is maintained. Instead of having FIFO request order, the loads (pending-reads) are given preference. Pending-reads are requested in order to keep the pipeline from stalling while pending-writes are requested in order to free up IO buffers.

Coherency between operations to the same address can be maintained by requiring all loads requests to be completed before store requests in the store buffer. This reason why this maintains coherency is the following: Loads will see pending-updates in the store buffers during the associative lookup and thus will read the data values from the store buffer before the store request is completed. Therefore if there is a load and store to the same location present at the same time in the buffer, the store instruction must have been issued after the load instruction. If the store or memory write request is performed after the load request has completed, the load instruction will get the data prior to the store, which is correct.

Pending Data Cache Updates: Similarly, all cache updates can be done in a FIFO ordering which preserves coherency automatically. However pending-reads cannot be written to the data cache until the word arrives from memory, whereas pending-writes already have data and can be immediately updated in the cache. Updates may be unordered as long as the order to the same address is maintained. Using a FIFO ordering is unnecessarily inefficient. Only pending cache updates (PU = 1) to the same location need to be considered. The order to the same address is maintained by canceling the PU (pending-update) bit for the word in the buffer. If a store instruction passing through the MEM stage finds a pending-update to the same address in the IO buffers, it clears that pending-update bit.

The store then allocates a new buffer with the more recent data and sets the PU bit for this buffer. In this way only the most recent load or store request to a memory location is queued in the buffer as a pending cache update.

The scheduling of pending-updates and pending read and write requests is implemented by using three pointers: *IO request pointer, Data Cache Update pointer*, and *the Free pointer*. The IO request pointer checks to see if there are any pending-read requests, if not then it looks for a write request. After requesting it waits for an acknowledge (write) or data (read). The data cache update pointer looks for any buffer with state 101 or 100, i.e. pending-update but not pending-read. The Free pointer simply points to any buffer with state 000. If the Free pointer points to a buffer whose state is not 000, then the IO buffers are full. None of the pointers imply any ordering on the requests.

3.5 Qualitative Comparison with Other Techniques

Store buffers provide several cool advantages:

- All stores appear to be single cycle instructions – they do not stall unless the store buffers are full. There is no extra penalty for store instructions that miss in the data cache.
- A store instruction does not stall further instructions accessing the cache since, unlike the blind-writing method, it does not leave the cache in an incoherent state. Subsequent loads can proceed without getting incorrect or stale data. Subsequent stores do not have to stall since their data gets queued in the store buffer.
- This method is applicable to direct-mapped, set associative, write-back and write-through caches. It does not constrain the cache configuration and is compatible with different store hit and miss strategies (like write-around, write-allocate, etc).
- The store buffer hardware can be overlapped with that of the write buffers, since they provide similar functionality. It can also be efficiently combined with load reservation stations as IO buffers.
- The store buffer is accessed in parallel with the data cache and does not add delay to the hit access time.

For a write-through, direct-mapped, multi-word line cache, which is a common cache organization, the various penalties of different techniques are summarized in the table 3.2.

Store buffers have no penalty cycles as long as the number of buffers is sufficient to match the number of consecutive stores in the code. Even with fewer buffers, under stressed IO conditions the write buffer for any technique is bound to fill up and stall the pipe for any store strategy. In the case where the store misses are negligible, both store buffers and blind-writing provide the same performance. But store buffers allow the flexibility in cache configurations that blind-writing cannot support.

TECHNIQUE	HIT penalty	MISS penalty	Number of Buffers =	In other caches?
Store Buffers	none	none	max(pending-writes,pending-updates)	all
2-Cycle Store				
-with write-allocate	1 cyc	1 cyc	number of pending-writes	all
-with no-write-allocate	1 cyc	none	number of pending-writes	all
Blind-Writing	none	1 cyc	number of pending-writes	none

In comparison to the other methods, store buffers only have a penalty when the data cache stage is stressed and this penalty can be eliminated by increasing the number of buffers.

Table 3.2: Comparison of Store Strategies

Other functionality can also be easily incorporated into store buffers, for example victim caching. Values can be held in the store buffer until the buffer fills up beyond some high water mark, rather than updating the cache immediately.

Using a set of associative buffers to implement delayed operations on a cache is a powerful idea and can be used for several different purposes. One example is snooping. Invalidations or other cache coherency operations can be queued in the buffer rather than stalling the pipe to make the cache coherent. All subsequent load/store instructions will see these operations during the lookup phase and act accordingly.

This thesis focusses on implementing IO buffers and comparing the performance of store instructions using store buffers, and other techniques. Chapter 6 presents statistics gathered from simulations of the various store mechanisms under different cache and IO conditions. Preliminary results seem to confirm the better performance of store buffers. Future work involves further quantifying the advantage of using store buffers and looking at the implications of 'delayed operations' to the cache.

Chapter 4

Processor and Instruction Set Architecture

4.1 Introduction

The RISC microprocessor described in this chapter was designed by the author and Rae McLellan, DMTS AT&T Bell Labs. This architecture was designed as a testbed for investigating the various tradeoffs in cache and pipeline design, branch prediction techniques and instruction set enhancement for DSP applications in a simple, single-issue processor. It served as the environment for implementing, modeling and simulating Store Buffers and for comparison with other techniques. The first phase of the thesis work consisted of designing the instruction set and datapath for this architecture and creating the simulation environment.

A great deal of effort was dedicated to designing the caches and memory instructions. The main premise of the design is that as technology advances it is possible to incorporate larger on-chip caches that will have significantly better hit rates. However the critical path is determined by the cache access time. Therefore, it becomes necessary to move the cache out of the critical path. The design of the caches was aimed at alleviating this problem and investigates strategies such as splitting the instruction cache and data cache accesses over multiple stages, using non-blocking loads and using store buffers or blind-writing to prevent stores from stalling the pipeline.

The sections below describe the processor architecture and some of its novel features that are relevant to this thesis.

4.1.1 Processor Description

The processor has a typical single-issue RISC, load-store architecture with 32 bit fixed length instructions, separate Instruction and Data Caches, 32 bit wide datapaths and a 6-stage pipeline. It bears many similarities to the MIPS and Alpha architectures.

There are several reasons for choosing a RISC pipeline as the basis of this design. The performance of a task on a particular architecture is determined by the following equation.

Execution time = *(cycle/instr) * (cycle time) * (no. of instructions in task)*
= *(time to execute 1 instr) * (no. of instructions in task)*

RISC architectures strive to improve performance by optimizing the first two terms in the equation in hardware and minimizing the third through compiler optimizations and other techniques. By limiting the instruction set to the simplest and the most frequent occurring basic operations, the time to execute an instruction is minimized. This follows the adage of *optimizing the most common case* and is also how the acronym RISC (reduced instruction set computer) got coined [7]. RISC architectures aim at providing an execution rate of one cycle per instruction which reduces the first term to one. This goal is achieved by two techniques: pipelining and load-store architectures.

Pipelining breaks the execution of an instruction into sequential stages where the output of one stage becomes the input of another. A stage is determined by the longest indivisible operation. This also determines the clock speed at which the pipeline is run. Pipelining covers the latency of an instruction by allowing instructions that require different resources to execute in parallel in different stages thus obtaining a speedup over sequential non-pipelined execution. The number of stages in the pipeline and the latency determine the throughput.

Throughput = latency / number of stages

In a load/store architecture, all operations are performed on operands held in registers – only load and store instructions access memory. Since main memory tends to have a high latency, this technique has several advantages. Compilers can optimize register allocation and keep the operands in the registers, thus reducing the number of memory accesses. Limiting memory accesses to loads and stores simplifies the instruction set and pipeline design. In addition, compilers can schedule the memory accesses between register operations to try and hide the latency of memory access.

Processor Features:

- 32 bit operation: All instructions and addresses are 32 bits wide. Most datapaths are also 32 bits wide. The integer register file contains 32 32 bit registers.
- Pipelined RISC architecture: The CPU has a 6-stage fully interlocked pipeline. Most instructions are executed in one cycle. The only instructions that access memory are the load and store instructions. All other instructions operate only on registers which is characteristic of a load-store/RISC architecture.
- Caches: There are separate Instruction and Data Caches. In the initial implementation both are direct-mapped and have parameterizable cache sizes. Each caches is pipelined over two pipeline stages. The access to the cache has been split into the memory array access phase and the tag comparison phase.
- Register file: The register file contains 32 general purpose 32 bit registers. All registers are treated symmetrically except R0 and R31. R0 is hardwired to a zero value. It may be specified as a target for instructions whose result is to be discarded. R31 is the implicit link register for branch and link instructions.

This section describes the instruction set architecture and the various formats or types the instructions can be grouped into. A detailed listing of the instruction set can be found in Appendix A.

4.1.2 Instruction Set Overview

The instruction set of this machine is characteristic of a single issue, load-store RISC machine and is very similar to that of the MIPS and Alpha processors. All instructions are a fixed length, 32 bits, and either operate directly on register contents or load or store data from memory. The addressing modes and data types supported are listed below. Instructions are primarily operations on a 32 bit word, however there is support for multiple and subword memory load and stores. The pipeline is fully interlocked and there are no explicit delay slots.

Addressing modes:

immediate	constant
absolute	REGFILE[Rx]
register indirect	loadmem(REGFILE[Rx])

Data types:

byte	8 bits
short	16 bits
word	32 bits
single precision floating point	(32 bits, 24 bit mantissa)
double precision floating point	(64 bits)

The integer instruction set can be divided into the following groups:

Register-to-Register Instructions (referred to as ALU operations)

These are computational instructions that perform arithmetic, logical and shift operations on values in registers. The destination and one source are registers, the second source may be a register or an immediate.

OPCODE RX RY RZ	where RX OPCODE RY – > RX
or OPCODE RX IMM RY	where RX OPCODE IMM – > RY

The IMM is specified by 16 bits and is sign-extended to 32 bits for arithmetic operations and zero filled for logical operations. Examples are: ADD, SUB, OR, AND, ADDI

Special cases: In SLLI, SRLI, SRAI the IMM is specified by 5 bits. In LUI (load upper immediate) the format is OPCODE IMM RY where IMM is loaded into the upper 16 bits of the register RY.

Store Instructions

Store instructions move data from the general registers to memory. The memory address is computed by adding a 16-bit signed immediate offset to the base register.

OPCODE RX RY OFF   where RY – > OFF[RX]

Examples are: SW, SB and SH. SB and SH are store operations defined to write subword quantities to memory in which the low two bits of the address specify which byte/half-word to write and the data is considered to be in the low bytes of the source register.

Load Instructions

Load instructions move data from memory to registers. The memory address is computed by adding a 16-bit signed immediate offset to the base register.

OPCODE RX RY OFF   where OFF[RX] – > RY

Examples are: LW, LH and LB, where the two low order bits of the address [RX + IMM] correspond to the sub-word to be extracted, sign extended and loaded into the destination register. In the case of LHU and LBU the sub-word is not sign-extended. Unsigned subword data is supported with LBU and LHU.

Control instructions (referred to as jumps and branches)

Jump and Branch instruction modify the control flow of the program. Hence, they modify the PC as a side effect. Jump instructions unconditionally change the PC to a new computed target. Branches compute a condition which if true changes the PC to the new computed target else the PC continues on the sequential path. The format of the instructions and the corresponding method for computing the target are given below.

Branches:

```
OPCODES RX RY OFFSET   where new PC = ((RX OP RY) ?  (PC + (OFFSET*4))
                                                     : PC++)
                       and the OFFSET is a word offset
```

For conditional branches the possible OPCODES are: equal(EQ), not equal(NE), greater than(GT), greater than or equal to(GE), higher than, unsigned comparison(HI), higher or same, unsigned comparison(HS). By reversing the operands it is possible to get the complete set of comparisons.

There is also a static prediction bit called the *taken* bit. If the taken bit is true the branch is predicted to be taken otherwise it is assumed to follow the sequential path.

There are also link versions of conditional branches. The instructions store the next sequential PC in R31 which is the designated link register for conditional branches.

Jumps:

JMP OFFSET	where new PC = PC + (OFFSET*4) and OFFSET is a word offset,specified by 26 bits and sign-extended to 32 bits.
JAL RY OFFSET	where new PC = PC + (OFFSET*4), PC++ − > RY and OFFSET is a word offset, 16 bits and sign-extended to 32 bits and the next sequential PC is stored into RY (link register)
JALR RX RY	where new PC = PC + RX, PC++ − > RY and RX is a byte offset (lower 2 bits must be zero) and the next sequential PC is stored into RY (link register) The link can be discarded if RY is set to R0

Special Instructions:

KCALL where new PC = 0xVECTOR, PC++ − > EPC, PSW − >privleged

This instruction jumps to priveleged code allowing for system calls. The return PC is stored in the EPC (Exception Program Counter).

KRET where new PC = EPC, PSW − > non-priveleged

This instruction jumps back to the return PC stored in the EPC and sets the Processor Status Word (PSW) to non-priveleged.

Instruction Set Design Choices

- **Fixed length instructions**
  RISC architectures use fixed length instructions because they make fetching and decoding of the instruction extremely simple. During a fetch it is not necessary to decode the instruction to figure out the length, since each fetch is a complete instruction and the instructions are aligned on word bounderies. This simplifies the design of the instruction cache since the data unit can be made equal to the size of the instruction. The disadvantage is that some instructions need less space to be encoded while other more complex instructions get excluded because of lack of space. The limited space also constricts the choice of register file size as well as the size of constants. Both of these result in slightly larger code size. However the simplification in design and the increased speed of decoding offset these disadvantages.

- **No Delayed Branches**
  Branch and jump instructions modify the control flow of the program. However in a RISC pipeline, as the instruction is fetched it is not known to be a branch and it is only in the DECODE stage that the existence of a branch is detected. This results in a delay slot. There are two schools of thought regarding whether this slot should be exposed to the compiler, called delayed branches, or should be the responsibility of the hardware [7]. Although exposing it to the compiler simplifies the hardware, it severely constrains further implementations of the pipeline and prevents the hardware from taking advantage of multi-issue or branch prediction strategies like branch target buffers. Therefore we chose to have synchronous non-delayed branches instead of delayed branches. This is further explained in the next section on the pipeline architecture.

- **Branches on register-register comparisons (no compares to zero, or separate compare and branch instructions)**
  All the branches are comparisons between registers. Comparisons to zero can be easily generated by setting one of the registers to R0. Comparisons to zero are more frequent, but it is difficult to make them faster than a normal register to register compare. If the comparison to zero is done in same stage as the register file read, it must be done after the bypass paths which are usually a critical path. Hence it is preferred to use the ALU stage to do the compare to zero, in which case both operands could have been fetched from the register file.

  An alternative is to have condition codes, in which case a comparison to zero can be made faster. But condition codes tend to become a bottleneck and end up being yet another resource for the compiler to allocate. Therefore the branch uses the ALU output directly.

  Unlike the MIPS instruction set [13], there are no separate compares and branches. Compares without branches are extremely infrequent. Hence it does not warrant an extra instruction. Also we do not gain anything by separating the compare and branch. In this processor, the branch on a register being equal to zero takes the same amount of time as a branch incorporating the comparison.

  If the result of a compare is needed in a register then it can be done by the following code:

  ```
  BRNC R1 R2 true (comparison R1 C R2)
  MOV R0 R3
  ....

  true: MOV 1 R3
  ```

- **Branch and Link**
  All the branch instructions have a version that saves the PC in the link register. The register R31 is the implicit link register. It is implicit so that there are enough bits to specify the branch offset. Most instruction sets do not have branch and link. Branch and link instructions were incorporated to investigate potential compiler optimizations.

  Jump instructions can use any register as the link register, which avoids creating a resource bottleneck.

- **Support for Byte and Half-Word Loads/Stores**
  Several languages (like C) have data types that manipulate bytes and half words. Supporting these in a machine optimized for words is difficult since all the datapaths and the IO interface are word-width. It also complicates the design decisions such as whether or not the cache and write buffer should accept sub-word operations. The alternative is to only load and store full words and provide extra instructions to shift and mask the values once loaded in a register. This makes each sub-word manipulation expensive. Code like string manipulation, which moves bytes around, occurs frequently enough to warrant the extra complexity of implementing loads and stores for sub-word quantities.

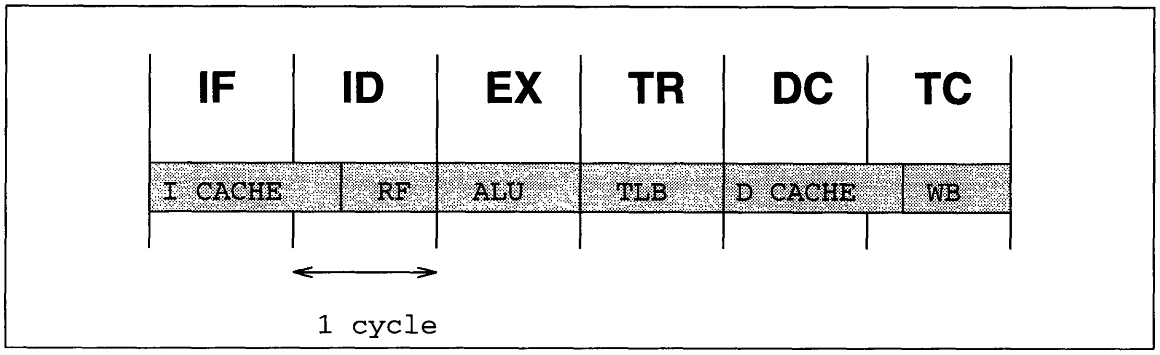

Figure 4.1: Pipeline Stages

- **No Delayed Loads**
  Load instructions are different from ALU instructions because they produce a result later in the pipe. Hence instructions immediately following a load instruction cannot access the value through a bypass path since the data is not yet available. Either we must insure that the instructions following the load do not use its destination register (compiler's responsibility), or that the pipe stalls any such instruction (hardware interlocks) [7]. There may be more than one delay slot depending on the number of stages between the operand read stage and the data cache access stage. Exposing the delay slot to the compiler removes the need for an interlocked pipe. But this severely constrains further implementations of the pipeline by fixing the number of pipeline stages and exposes the implementation to the compiler. Hence we chose to implement pipeline interlocks instead of load instructions with explicit delay.

4.2 Description of Pipeline

The microprocessor is a load-store architecture with a six stage pipeline. Appendix B has the detailed schematics of the various stages of the pipeline. The execution of an instruction can be divided into the following stages/steps.

IF = Instruction Fetch stage

Fetch the instruction at the address in the PC (program counter) from the instruction cache or memory.

ID = Instruction Decode stage

Read any required operands from the register file while decoding the instruction. Also perform the tag comparison for the Instruction Cache.

EX = Execution stage

Perform required operation on operands

TR = Translation stage

Translate virtual address to physical address for load/store instructions Access Translation Lookaside Buffer (TLB).

DC = Data Cache stage

Access Data Cache data and tag arrays.

TC = Tag Check stage

Perform Tag comparison for Data Cache and write-back result to register file.

IOB = IO Buffer stage

Loads that miss and store instructions get queued in the IO buffers and request IO. This stage which implements the store buffer protocol. It is a separate stage only for load and store instructions.

The pipeline achieves an execution rate approaching one instruction per cycle by overlapping the execution of 6 instructions. Each stage uses different resources (ALU/ Reg file/ IO, Data Cache) on a non-interfering basis. In other words most resources are accessed only during one stage in the pipeline, one exception being the register file.

4.2.1 Salient Features of the Pipeline

6 stage pipeline (TLB lookup stage)

The pipeline described above is different from the standard RISC pipeline in several ways. First, it is a longer pipeline. This is because of the addition of a separate stage for the TLB lookup. Having the TLB lookup before the cache access allows for the cache to be *physically indexed and tagged.* Otherwise if the TLB were in parallel, the cache would have to be *virtually indexed* with either *physical tags* in which case the size of the cache is restricted or *virtual tags* which results in many aliasing problems. This way there is no constraints on the cache size.

Of course there is a disadvantage to adding this extra stage. It adds a load delay slot. The data cache access is now one more stage further away from the register file read stage. This penalty is mitigated if the compiler is able to schedule loads to fill these delay slots. This stage has been added as an experiment to investigate the tradeoff between having a completely physical cache with the penalty of a longer pipeline, or having a shorter pipeline with a smaller cache. This investigation is not covered in this thesis. Since the compiler schedules the use of the load values as far away from the load as possible and this pipeline is constant over all the measurements taken, the addition of this stage does not affect the analysis.

Pipelined Caches

As technology advances it is possible to incorporate larger on-chip caches that will have significantly better hit rates. It is possible to gain a great deal of performance from increasing the size of the cache. However the clock speed of a pipeline is usually determined by the cache access time in the DC stage, which makes it imperative to move the cache out of the critical path.

One way of accomplishing that is to pipeline the caches. That implies that the cache access is split over more than one stage. Hence the cache access is removed from the critical path.

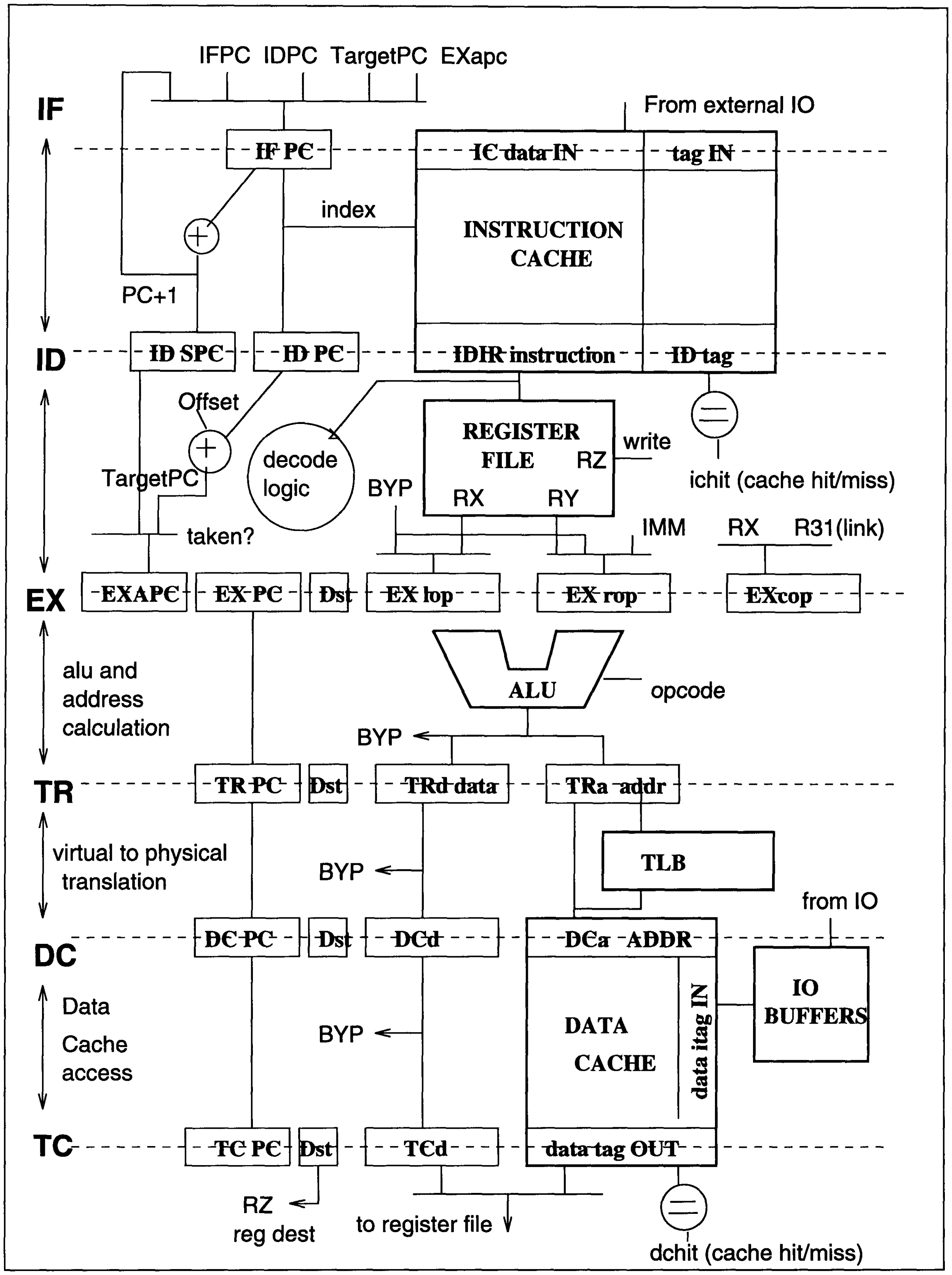

Figure 4.2: Detailed Pipeline Schematics

In this pipeline both the IC and DC are pipelined caches.

IOB stage

There is an extra stage for load and store instructions, called the IOB stage. Therefore for load/store instructions the pipe is a 7-stage pipeline while for others it is a 6-stage pipe. This stage contains the IO buffers.

The sections below describe the implementation of the various groups of instructions and how they interact with the various resources in the different pipeline stages. The last section describes the implementation of the caches which are spread over two stages. The IOB stage of the pipe is explained in the next chapter. This pipeline is constant across all the store methods implemented to insure that all conditions, other than those being measured, do not affect the statistics. The implementation of the entire pipeline is illustrated in figure 4.2.

4.2.2 ALU Operations

The ALU operations are register-to-register operations and therefore do not interact with the data cache or TLB. After an ALU operation is fetched in the IF stage from memory, it is decoded in the ID stage and the operands are read from the register file. The opcode chooses the function to be performed by the ALU in the EX stage and the result is latched in the TR data register *Trd*. The instruction then passes through the TR and DC stages without interacting with any of the resources. In the TC stage the *Tcd*, or result of ALU operation, is written back into the register file using the destination register specified by *Tcdst*.

Notice that the destination register does not get written to until the last stage in the pipe although the result of the operation was available since the EX stage. This is to avoid out-of-order completion problems and allow for exceptions and restartability. However, if some subsequent instruction in the ID stage requires the result of an ALU operation further down the pipe, the data gets routed to the ID stage via bypass paths. There are four bypass paths from the EX, TR, DC and TC stages to the left and right operands, *Exlop* and *Exrop*, in the ID stage. Therefore data dependencies between ALU operations do not stall the pipeline.

4.2.3 Branches and Jumps

Branch and jump instructions affect the control flow the pipeline by changing the IF stage PC register. The instruction is fetched in the IF stage, but it is decoded to be a branch or jump only in the ID stage. There is a separate adder to compute the target address. This could have been computed by the ALU in the EX stage, but if the address is available in the ID stage the IF stage PC can be updated immediately. When a jump is detected in the ID stage, the instruction fetched by the IF stage is incorrect and therefore discarded, which causes a single cycle penalty.

Branches on the other hand are conditional. They compare two registers using the ALU in the EX stage to determine whether or not to affect the PC. If the branch is taken in the EX stage, the instructions in the IF and ID stage must be discarded resulting in a two cycle

Instruction	**prediction** correct	**penalty** incorrect	**no prediction**
JMP	n/a		1 cycle
BRN taken	1	2	2
BRN not taken	0	2	0

Table 4.1: Branch Prediction Penalties

penalty.

One way to eliminate this penalty is to promote instructions before the branch into these delay slots. This technique is called 'delayed branches'. However not only is it a difficult task for a compiler to find such instructions, this technique forces the pipeline length for future implementations to remain the same. It also prevents the use of prediction techniques to mitigate the cost of taken branches.

Static Branch Prediction (taken) bit: In this processor there are no delayed branches and prediction is used to reduce the branch penalty. All the branches have a 'taken' bit which is set by the compiler causes them to be treated like jumps in the ID stage. and the IF PC is modified. In the EX stage this prediction is verified. If the prediction is incorrect the PC must be corrected. The penalties for branches with and without prediction is given in the table 4.1.

The success of this strategy depends heavily on the compiler's ability to predict branch outcome correctly. There are several heuristics that allow compilers to predict with a high rate of success. For example, branches in loops are predicted taken. Profiling can also be used to predict branches.

Branch Target Buffer: There is at least one cycle penalty with or without prediction for branch taken or JMP. This is because the instruction is decoded as a branch/jump only in the ID stage. Another branch prediction technique that actually eliminates this penalty as well by using a *Branch Target Buffer (BTB).* This buffer is indexed by the IF stage PC in the IF stage and contains the expected target associated with the PC if the instruction at that PC was a branch. Hence the branch/jump and the target address are predicted at the IF stage and if prediction is correct, there is no penalty.

This processor will be used as a testbed to compare the performance of both branch techniques. But for this thesis the branch prediction is done statically using the *taken* bit.

4.2.4 Load instructions

Load instructions retrieve data from memory and write it into the register file. See Figure 4.3. The memory address is computed by adding the R_{base} value to the offset specified by the instruction. The IF and ID stage fetch and decode the instruction and read the base register value, R_{base}, from the register file. In the EX stage the ALU is used to compute the virtual memory address, which is then translated by the TLB in the TR stage. In the DC stage this address is used to index the data cache and the data and tag are latched into

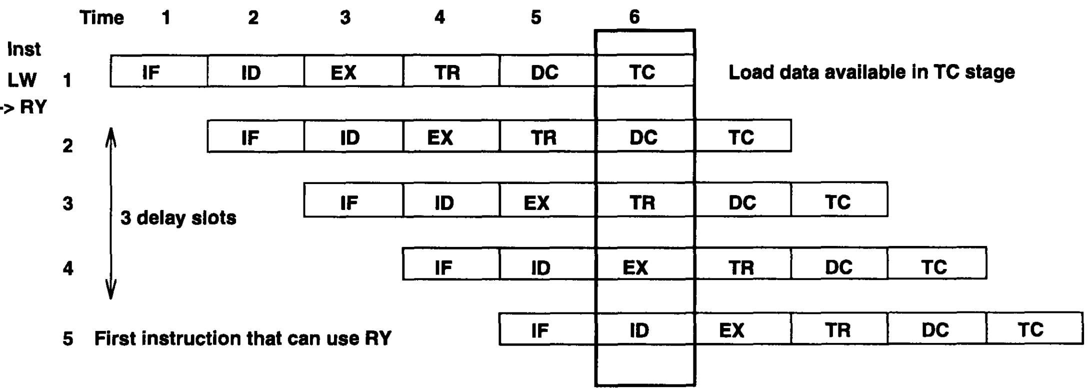
Time 1 2 3 4 5 6
Inst
LW 1
-> RY
2
3
4
5 First instruction that can use RY
3 delay slots
IF ID EX TR DC TC
Load data available in TC stage

Figure 4.3: Load Delay Slots

the Tcd and Tctag registers respectively. In the TC stage the tag is checked against the memory address *Tca* to determine hit or miss. The verified load data is available at the TC stage (see figure 4.3).

If any instruction in the ID stage requires the data from a load instruction in the EX, TR, or DC stage, it must wait until the load reaches the TC stage and the load data has been read and verified. Hence the bypassed value cannot be used. This is called a *bypass hazard.* Therefore there must be at least three intervening instructions between a load and its use to avoid stalling the pipe. These delay slots can be exposed to the compiler (delayed loads) but that fixes the length of the pipe for future implementations of the processor. Instead this processor has an interlocked pipe which will stall an instruction in the ID stage if it requires the data of a load in the EX,TR or DC stage.

The extra TR stage and pipelined cache access both affect the performance of the pipe by increasing the number of delay slots. For loads that miss, this penalty is insignificant compared to the IO latency. For loads that hit, there is a penalty if the instruction depending on the load data cannot be moved at least 3 instructions away. Scheduling load instructions earlier than the use of the destination register is often difficult since loads cannot be promoted easily past block boundaries. The actual penalty paid as a result of these extra stages in the pipe, depends on the compiler's success at pipelining loads and dependent ALU operations.

Loads that miss can be dealt with in several ways. The most common way is to block the execution of the pipe until the load address is read from memory even if the instructions in the pipe are not dependent on the load value. In this processor we are implement non-blocking loads: if a load misses in the data cache, the pipeline is not stalled until some instruction in the ID stage cannot proceed without the load data. When the load misses in the data cache, a memory read request is queued up in the IO buffers and is processed in the background of the pipeline execution. Hence the load miss is serviced out of order with the subsequent instructions. Once the value arrives from memory it gets written to the register file directly and is queued to be written into the data cache. This is explained in further detail in Chapter 5.

4.2.5 Store Instructions

Unlike load instructions, store instructions do not have a register result. Hence subsequent instructions do not have data dependencies with store instructions. The IF and ID stage fetch and decode the instruction and the register file supplies the store value and base for the memory address. The ALU is used to compute the memory address and it is latched in the *Tra* register and the data is latched in the *Trd* register. The TR stage translates the virtual address to a physical one.

During the DC stage a store instruction needs to verify the tag and then write the data. As explained in Chapter 3, this poses a problem since both tags and data are accessed concurrently. To access the tags and data arrays serially, a store instruction needs to spend two cycles in the DC stage. There are several techniques for avoiding this second cycle penalty, one of which, Store Buffers, is being proposed in this thesis. Chapter 5 describes the implementation of store buffers. The basic idea is that a store instruction checks the tag on first pass through the data cache and then using this information a data cache update request is queued in the IO buffer, to be performed when the DC stage is unused.

4.3 Memory Hierarchy

The memory hierarchy can be divided into three parts:

1. Instruction and Data caches
2. External Memory Interface
3. IO buffers (interface between the Memory and Data Cache)

The sections below describe the implementation of first two in detail. IO buffers are covered in Chapter 5.

4.3.1 Instruction and Data cache

The processor has two separate caches for instructions and data (harvard architecture). Separate caches allows each cache to be optimized for either instructions data or program data, both of which have different characteristics. It also simplifies the pipeline design by allowing simultaneous access to cached instructions and data.

The access for both caches is pipelined. The premise behind the design of the caches is the following: Advanced VLSI techniques make it possible to incorporate larger caches on-chip. At the same time on-chip logic speed is outstriping IO speed making off-chip communication a severe bottleneck. Larger caches significantly increase performance by decoupling the pipeline speed from the IO latency. However larger caches adversely affect the pipeline speed. Currently the critical path is almost always determined by the cache access time. This negates the performance gain of higher hit rates by decreasing the throughput of the pipeline. Therefore to take advantage of larger caches, it becomes necessary to move the cache out of the critical path.

One way of accomplishing that is to pipeline the cache. In this processor the cache access is split over two stages so that the memory array lookup for both tags and data is done in the first stage and the tag comparison is performed in the second stage.

In a single stage cache the access time is *(tag/data array access time + tag comparison).*

In a pipelined cache, the longest path through the cache is *(max (memory access time, tag comparison)).*

Hence the cache tag and data array can be made larger without affecting the pipeline speed. There is a performance gain as a result of the higher hit rate of the cache. Of course this performance gain too has a limit after which the cache size becomes the critical path again.

The disadvantage of this strategy is that the detection of a miss is delayed by a stage. In both caches it increases the penalty of a miss by one cycle, which is not significant compared to the penalty incurred because of IO latency. The lower miss rate also offsets this penalty. The data cache however, has the additional penalty of increasing the number of load delay slots. As a result, the instruction using the load data needs to be moved a cycle earlier to prevent stalls because of data dependencies. The effectiveness of the pipelining depends on the degree to which the increased cache size and hence reduced miss rate, offsets the higher miss penalty. In the data cache it also depends on the compiler's ability to schedule loads early enough with respect to their use to avoid pipeline stalls.

Pipelining the cache over more than two stages is difficult since it is physically difficult to split the memory array access into smaller sections. Also the miss penalty quickly overcomes the performance gain of increased hit rates, as the cache is split over more stages.

Instruction cache

The simulator implements an instruction cache as a direct-mapped cache with a variable number of words per line and is specifiable at run time. There are valid bits per word in the line to support partially valid lines. The access is split over two stages. During the IF stage the PC indexes the data and tag arrays. The output is latched into the *IDir* and *IDtag* registers respectively. During the ID stage the tag is compared to the high order bits of the translated PC, if they match the instruction read is correct. This comparison is done in parallel with the rest of the decode logic. See figure 4.4.

```
/*------------------------------------------------------------*/
/* CACHE INITIAL VALUES */

#define INIT_ICSIZE    256       /* 256 lines -> 1K words       */
internal ICSIZE;                 /* Instruction cache size      */
#define INIT_ICBLK     4         /* 4K words (16K bytes)    */
internal ICBLK;                  /* Instruction cache size      */
internal LOGICBLK;               /* Instruction cache size      */

/* CACHE DATA STRUCTURE */
struct regIC {  unsigned int *inst;
                unsigned int tag;
                unsigned int v;
              };
```

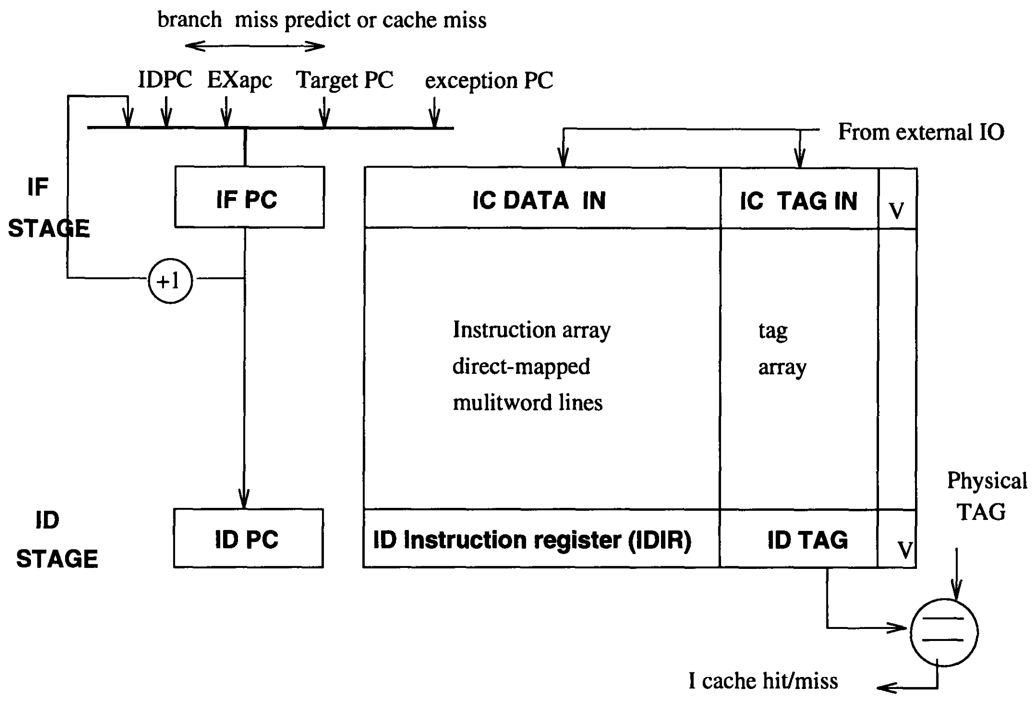

Figure 4.4: Instruction Cache

/*--*/

If the tag comparison is a hit the instruction flow is uninterrupted and hence the hit performance is the same as that of a single stage instruction cache. If however there is a miss in the ID stage, the instruction fetched in the IF stage must be discarded. The IF stage PC is modified to the ID stage PC that missed and the IF stage is stalled until the word arrives from memory. Hence the miss penalty is 1 cycle + IO latency. Since the miss rate for instruction caches is very low and IO latency tends to be anywhere from 6-20 cycles, the miss penalty for the pipelined cache is the same as that of the single stage cache. Hence the pipelined instruction cache wins by allowing larger caches at the same clock speed with same hit and miss penalties. Notice however that pipelining the instruction cache over more than two stages introduces additional penalties, for example the number of jump and branch delay slots increase and can severely degrade performance.

The cache miss strategy is fetch-on-miss. Therefore if a PC misses in the cache, it requests the cache line starting at the word that missed in the cache.

This takes advantage of the spatial locality exhibited by instruction streams. As soon as the first word is received from memory, the IF stage is allowed to proceed, it does not wait for the entire line to be written into the cache. If the instruction stream proceeds without any control flow changes, then the miss penalty on subsequent instructions in the same cache line that missed is reduced since they are requested from memory before the miss occurs. The miss latency for these words is also reduced because of the higher bandwidth of block transfers. The performance tradeoff is when a line is being fetched but the PC is modified to some new value by a jump or branch. The IO is tied up by a useless block fetch and it

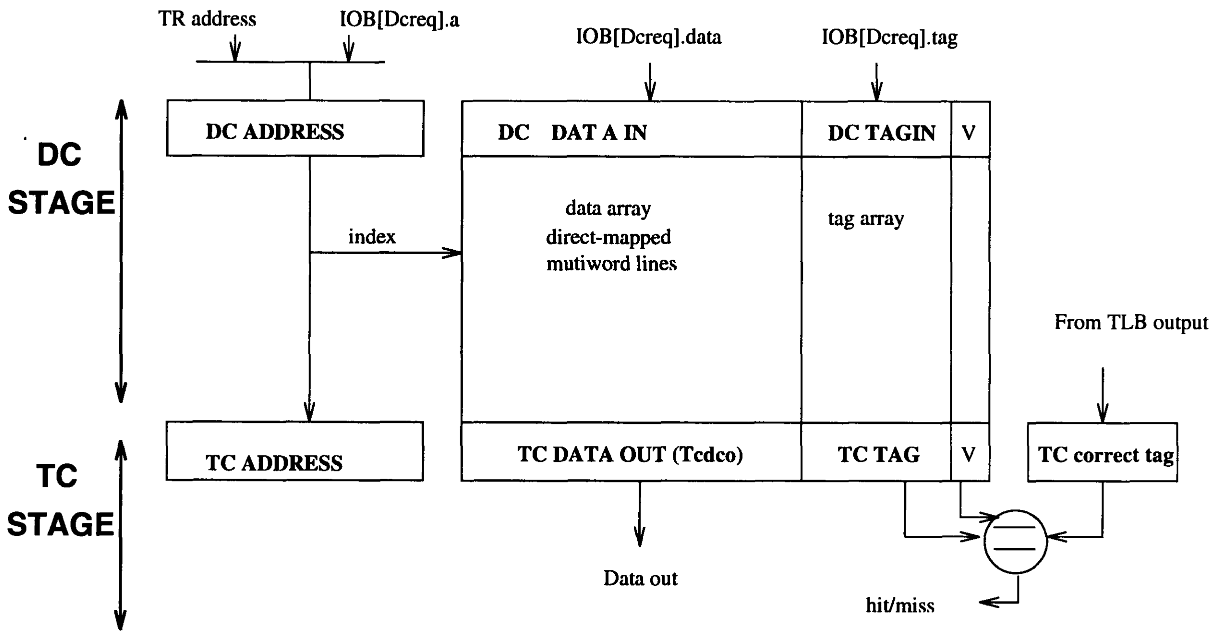

Figure 4.5: Data Cache

is too difficult to cancel the off-chip IO operation already in progress. However this occurs infrequently and in general the fetch-on-miss policy proves to be very efficient.

In the layout the IF stage PC, *IDir* and *IDtag* registers can be placed right against the memory fabric. The critical path through the IF stage is then just the data array access time. The critical path through the ID stage is determined by the max(register file lookup, bypass path settling, target address computation, tag comparison).

Data Cache

The simulator implements the data cache as a direct-mapped write-through cache with a variable number of words per line specifiable at run time. Like the instruction cache, it has valid bits per word in the line to support partially valid lines. Since it is write-through the data in the cache is also present in memory hence there is no unique data in the data cache that cannot be replaced or discarded. This also means that all store instructions must write their value to memory. See figure 4.5.

```
/*-------------------------------------------------------------------*/
/* DCACHE INITIAL VALUES*/

#define INIT_DCSIZE 256                 /* 256 lines-> 1K words        */
internal DCSIZE;                        /* data cache size             */
#define INIT_DCBLK 4                    /* 4K words (16K bytes)        */
internal DCBLK;                         /* data cache block size       */
internal LOGDCBLK;                      /* log2(data cache block size) */

/* DCACHE DATA STRUCTURE */
struct regDC {  unsigned int *data;
                unsigned int tag;
                unsigned int v;
```

```
            };
/*-----------------------------------------------------------------*/
```

The cache access is split over two stages. During the DC stage the DC address register *Dca* indexes the data and tag arrays. The output is latched into the *Tcd* and *Tctag* registers respectively. In the TC stage the tag is checked against high order bits of the physical address. If the tags match the data read is valid. If the tags do not match there is a miss.

Unlike the instruction cache pipelining the data cache penalizes both the hit and miss performances and also constrains store instruction implementation. The tradeoff then is to use compiler techniques to overcome the penalties and increase the cache size enough to still obtain a performance gain.

Hit penalty: Verified data is not available until the TC stage, instead of the DC stage as would be the case for a single stage data cache. This increases the number of delay slots for a load instruction. If the compiler is able to find instructions to fill these slots then this penalty is not seen. It is also possible to bypass the data from the DC stage, but then there needs to be a way to back out if the data turns out to be incorrect. This eliminates the hit penalty but is complicated to implement. Instead this processor only forwards data after its validity has been confirmed and relies on compiler techniques to eliminate the hit penalty.

Miss penalty: The data miss is detected in the TC stage, hence the IO request for a read is made a cycle later than for a single stage data cache. This does not significantly affect the performance because IO latency is usually 6-20 cycles and therefore miss penalty is already high. The miss rate is more important in the equation and that is decreased by the increased cache size.

Another disadvantage of pipelining the cache is that it makes store implementation more complicated. Since hit or miss is not determined until the TC stage, a store instruction can only write data with surety after the TC stage. In the meantime there may already be a store instruction in the DC stage accessing the cache arrays. Since the cache is incoherent and may have stale data or an incorrect tag, the instruction in the DC stage needs to be stalled until the data write is completed. This can increase the penalty to two cycles. Implementing blind-writing also encounters similar problems. Only in the TC stage is a miss detected in which case the instruction in the DC stage is accessing an incoherent cache. Therefore the instruction must be stalled until the tag fix is completed. This is explained in more detail in Chapter 5 where these techniques are implemented in the simulator for comparison with store buffers.

Store buffers, on the other hand, are readily implemented with a 2-stage cache access since the cache update is delayed already and is present in the IO buffer for subsequent instructions to observe. It does not leave the data cache in an incoherent state at any time.

Store instructions queue their data into the IO buffers to be written into the cache when the DC stage is free. Load instructions that miss queue the read request in the IO buffers without stalling the pipe. The data from IO is written to the IO buffer and queued as a cache update just like store instruction data. Hence all cache updates are done from the IO buffers. This simplifies the interface to the data cache since all updates are performed by the store buffers.

4.3.2 External Memory Interface (arbiter)

Requests for IO service come from the Instruction cache and the IO buffers. The Instruction cache requests a memory read of a block of data corresponding to the remaining line in the cache. The IO buffers can submit two different types of requests, read requests for one word or an entire cache line, or write requests. Subword requests are not supported in the memory interface. The arbiter chooses the next request to be granted and gives precedence to the instruction cache read request. The IO buffer requesting pointer gives precedence to read requests. Hence the relative priority of grants is: Instruction fetch, Data load, Data store.

The arbiter chooses the request and sends back the appropriate grant signal. The address is latched into the address register *ioaddr* and the data into the data register *iodata* . For block reads, the block number is latched in *iocount*. The off-chip memory services the request word at a time since the IO interface is 32 bits wide. For each word the *dtack* signal is sent to the on-chip arbiter that sends the corresponding done signal. The data in the case of a read is latched into the data in *dtin* register.

The IO interface is actually a bit more complicated. There is a separate address on the pins and a separate address on the master of ioaddr. This way a second request is latched while the first is on the pins.

4.3.3 IO Buffers (interface between the cache and memory)

The IO buffers act as combined write buffers, store buffers and load reservation stations. The IO buffers act as an interface between the cache and the IO frame. The cache queues memory read requests and memory write requests in the IO buffers. All cache updates are performed from the IO buffers when the data cache is unused. The IO buffers submits read/write requests to the arbiter. For read requests it receives the data in the IO buffer and for write requests it provides the data from the corresponding buffer. Hence the data cache and IO are interfaced through the IO buffer which is responsible for maintaining coherency on either side and efficiently scheduling all update, read and write requests. IO buffers are explained in detail in chapters 3 and 5.

Chapter 5

Store Buffers: Implementation

5.1 Implementation

The purpose of delaying store operations on the data cache is to avoid unnecessary stalls in the pipe as much as possible. Only unavoidable hazards caused by data dependencies with data not yet retrieved from memory, should cause the pipe to stall. Store instructions writing to memory or to the data cache, as well as blocking loads are artificial dependencies and unnecessarily stall the pipe.

In this processor, store buffers are implemented as a part of more general IO buffers. IO buffers act as combined write buffers, store buffers and load reservation stations and prevent unnecessary interruption of pipeline execution. As mentioned earlier, the advantage of having a combined set of buffers is that it allows better allocation of hardware resources and simplifies the interface between memory and the data cache. It also schedules off-chip memory reads and writes more efficiently by unordering them and is easier to implement than several sets of buffers and control to the same effect. On the whole IO buffers prove to be very effective in preventing pipeline stalls by delaying cache update and memory write operations.

5.1.1 Implementation of IOB stage

After the TC stage there is an extra stage for load/store instructions, called the IOB stage. This stage contains the IO buffers. The IO buffers are checked for consistency in the DC stage in parallel with the data cache. New operations are entered in the IO buffer at the end of the TC stage. This is because the tag check is performed in the TC stage and it is required for setting the tagmatch bit and for determining if a load needs to request data from memory. This spreads the logic for the associative lookup and queuing over two stages and reduces the critical timing paths. However this does introduce a delay between the time of lookup and the time of IO buffer queuing where data dependencies can arise.

The state of an IO buffer consists of five bits, the three mentioned before, PU, PR, PW, and two additional bits, TM (tagmatch) and FS (fetch strategy). The table 5.1 explains the possible states and the auxiliary bits.

PU	PR	PW	State Description
1	1	0	Cacheable Load Request
0	1	0	Non-Cacheable Load Request
1	0	1	Cacheable Store Request
0	0	1	Non-Cacheable Store Request
1	0	0	Cache Update Request
0	0	0	Free IO buffer
			All other states are illegal

TM	Tagmatch Bit. Only useful if PU bit is set
	SET if cache tag comparison returns true
	CLEAR if tag match returns false
FS	Fetch Strategy Bit. Only useful if PR bit is set
	SET if requesting an entire line from memory
	CLEAR if requesting a single word

Table 5.1: Expanded IO Buffer State Bits

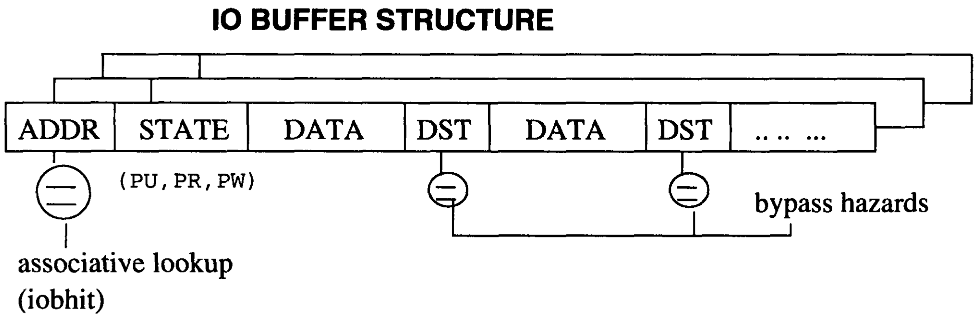

Figure 5.1: IO Buffer Implementation

The IO buffer has cache line size buffers for requesting reads of entire cache lines. Hence the IO buffers are multi-word. There is a PU bit per word in an IO buffer. This is more efficient for implementing a fetch-on-miss strategy for load misses. If there is a subsequent store to a word requested by a load, the load's update request is cleared by the store to preserve the coherency of cache updates to the same address. If an entire line is being requested, a single PU bit can be cleared without canceling the update for the remaining words in that line.

Besides the state bits, each IO buffer contains an address that is associatively compared to the address in the DC stage. It also contains a register to hold the store instruction data, as well as load data arriving from memory. For each data register there is a destination register field for loads and a 5-bit comparator for hazard detection.

Instead of having cache line size buffers for each IO buffer entry, a pool of cache line size buffers can be associated with any IO buffer requesting a full line. In the simulator, to simplify code and ease design, all IO buffers are simulated as full line size buffers with a destination register field per word.

To facilitate the testing of different configurations, the IO buffers are implemented in the simulator as a parameterizable entity, just like the caches. This allows the number of buffers to be specifiable at run time and the size of the buffers can track the number of words per line in the data cache.

```
/*----------------------------*/
/* IO BUFFERS             */

internal IOBSIZE;                         /* Number of load reservation stations */
struct Lbuf {
    unsigned int pw;                      /* pending write to IO, 1 per byte     */
    unsigned int pr;                      /* pending read      to IO, 1 bit      */
    unsigned int pu;                      /* pending DC update, 1 per word       */
    unsigned int fs;                      /* fetch strategy, 1=line 0=word       */
    unsigned int tm;                      /* tagmatch (invalidate other words?)  */

    int a;                                /* word address to be written/read     */
    int *data;                            /* data to be written/received         */
                                          /* fs=1 => cacheline size, else 1 word */
    int *dst;                             /* LD register dest, 1 dst/word        */
    int *regw;                            /* register write enable / dst field   */
};
/*----------------------------*/
```

This completes the description of the IO buffer model. The following sections describe the flow of load and store instructions through the IO buffer and the interaction of the IO buffer with the data Cache and off-chip memory. A detailed diagram of the pipeline is given in appendix B. Simulator code for selected signals is also included in this section to illustrate the implementation. The simulator code describes the register level logic design and corresponds directly with the pipeline diagram.

5.1.2 Implementation of Non-Blocking Load Instructions

The simulator implements non-blocking load instructions. If a load instruction that misses in the cache does not stall the pipeline but instead gets queued as pending read request, it is called a *non-blocking load.* The flow of instructions continues past the load instruction even though the load has not yet completed. Only a data dependency with this load value stalls the pipe.

As a load instruction passes through the DC stage, an associative lookup against the *Dca* address is performed on the IO buffers. If the load address matches an IO buffer address (*iobhit*), then the corresponding IO buffer data is latched into the *Tcwbd* register. In the TC stage the data cache tag comparison takes place and data cache hit, *dchit*, is determined. If there is an *iobhit*, the IO buffer data is given precedence since it is the most recent update to that memory location, else if there is a dchit the data cache output, *Tcdco*, is taken. If neither *dchit* nor *iobhit* occur, then the load is considered to have missed in the data cache and a memory request is queued in the IO buffers.

A load miss is queued into the IO buffer with state bits (PU PR PW) set to 110. A fetch-on-miss policy is used to take advantage of the higher bandwidth of block transfers and the spatial locality exhibited by load instructions.

For block transfers the data arrives in word size chunks, with the initial word taking the longest latency and subsequent words arriving at a quicker rate. A load that misses allocates a line in the cache. The cache update by the load is treated as another pending-update in the store buffers. Hence the entire data line is gathered in the buffer before updating the data cache. The advantage of this strategy is that the number of data cache updates is reduced because the entire line is updated at once. The disadvantage of requesting entire lines is that the size of the buffer to hold data needs to be of cache line size and a register destination is required per word. Hence the size of each IO buffer increases. This cost can be mitigated by having a pool of full line buffers as mentioned before in section 5.1.1.

As the load instruction passes through the DC stage, it also detects any existing IO buffers requesting the same cache line. If the load misses and the cache line is already requested in the IO buffers (*iobmatch*), the load simply adds its register destination to that IO buffer. If no IO buffer is allocated to this line but the line is partially valid in the data cache or there is a pending-update to a word in the same cache line, the load requests only a single word (FS bit = 0). If the line does not exist in the data cache or IO buffers, the load gets queued as a full line request with the register destination set to *Tcdst*.

The data cache update for a load instruction is not time critical. However the register file needs to be updated as soon as possible to prevent subsequent instructions from stalling the pipe if they depend on the load data. A second write port to the register file is dedicated for updating from the IO buffers. When the data arrives from main memory, it can be immediately written to the register file without stalling the pipe.

After the load data is received from main memory and written to the data cache, the IO buffer can be freed.

5.1.3 Implementing Store Instructions

Store instructions are implemented using the Store Buffer technique proposed by this thesis. This strategy uses a set of buffers to delay the cache update phase of store instructions. The pending update is performed when the DC stage is empty or the data cache is unused to avoid stalling the pipe.

During the DC stage, the store instruction also performs an associative lookup in the IO buffer to detect any pending-writes to the same location. If there is such a buffer, the new store data is simply appended to this buffer. This provides the functionality of a coalescing write/store buffer. During the associative lookup it also cancels the pending cache update, or PU, bit for any other queued request to the same word address. This is necessary to preserve an ordering amongst cache updates within the IO buffers. The new store instruction data is the most recent update to that location so only its PU bit is set and all other updates to that location are canceled.

A store instruction is queued into the buffer during the TC stage with the state (PU PR PW) 101 and the TM state bit remembers the result of the data cache tag comparison.

5.1.4 Background Data Cache Update

Updates to the data cache are performed in the background of the pipeline operation. When the DC stage is empty or the data cache is unused, an update from the IO buffer can be performed. The data cache is only used when there is a load/store instruction in the DC stage. During an ALU operation the data cache address register *Dca* is free to be used by the IO buffer for a cache update. After the update is performed the IO buffer is freed. Updates can be performed one per cycle.

Scheduling: Within the IO buffers, pending-reads cannot be written to the data cache until the word arrives from memory, whereas pending-writes already have data and can be immediately flushed to the cache. Since there need not be an ordering on the updates a long as the order to the same address is maintained (relaxed consistency [5]), using FIFO ordering is over constrained. With relaxed consistency the ordering amongst pending-updates is eliminated and coherency is maintained by not allowing more than one pending-update to a single memory address in the IO buffers. The order of operations to the same address is maintained by canceling the PU bit for that word in the buffer. Only the most recent request is queued in the buffer as a pending cache update.

The IO buffer scheduled for an update is selected by the *DCreq* pointer. Any buffer with the PU bit set and PR bit clear, i.e. pending-update not waiting for the data to arrive from memory, is a valid choice for the new *DCreq* position. When the update is completed the buffer's PU bit is cleared and the *DCreq* pointer advances to the next register.

```
/* Bdcreq: if bg_dc_write                                          */
/* OR if current buffer no longer valid pending cache update       */
/* bg_dc_write = DC stage empty                                    */
/*               OR next instruction is not a load/store           */

bdcreq = (bg_dc_write || !Iobuf[Bdcreq].pu) ? next_dcreq
                                            : Bdcreq;
```

Several IO buffer entries with pending-updates may interact with the same cache line. The TM bit for an IO buffer is set if the address of the IO buffer matches the current tag in the data cache. When the data cache tag is modified by a background cache update, the TM bits for these buffers must reflect this change. Therefore the background cache update performs an associative lookup to identify buffers requesting the same cache line. If an IO buffer has the same cache index as the background cache update but a different tag, the TM bit is cleared. This indicates that the IO buffer address no longer matches the location resident in the data cache. Similarly, if an IO buffer has the same cache index and the same tag, the TM bit is set. Hence the new TM bits reflect the updated state of the cache line. This preserves the coherency of the cache.

5.1.5 IO Request State Machine

The IO buffers submit read and write requests to off-chip memory. The type of request is determined by the PR and PW bits: if the PR bit is set it is a read request, else if the PW bit is set it is a write request. Once the request is complete the corresponding bit is turned off. In the case of a pending-read the request is complete once the data has been latched into the IO buffer, i.e. when the last IO done signal is received. For a pending-write the request is complete as soon as the data is latched into the outgoing data register, i.e. when the grant signal is received from the arbiter.

The requests can be generated in FIFO order, thus preserving the ordering between the load and store operations. But that is very inefficient since memory writes are not as critical as memory reads. Memory reads can produce real data dependencies in the pipe. Although FIFO will keep the cache coherent, the relative order of loads and stores is not of much consequence as long as the order to the same address is maintained. Instead of having FIFO requesting, the loads (pending-reads) are given preference. Pending reads are requested in order to keep the pipeline from stalling while pending-writes are requested in order to free up IO buffers.

Coherency between operations to the same address can be maintained by requiring all loads requests to be completed before store requests in the store buffer. This works because of the following reason: If the data is a pending-update in the store buffer then a load will see that information during the associative lookup and read the value from the IO buffer. Therefore if there is a load and store to the same location present at the same time in the buffer, the store instruction must have been issued after the load instruction . Therefore if the store or memory write request is performed after the load request has completed, the load instruction will get the data prior to the store, which is correct.

```
/* Bioreq: if valid io request AND grant received    */
/* OR granted arrived in past OR not valid anymore   */
/* => choose new location to request                 */
bioreq = (iobgrant || (Bioreq == Bwait)
        || !(Iobuf[Bioreq].pw || Iobuf[Bioreq].pr)) ? next_ioreq
                                                     : Bioreq;
```

The *IOreq* pointer selects the next memory request. When the memory operation for the current buffer indexed by *IOreq* is completed, the new *IOreq* points to a pending-read if one exists, else it points to a pending-write.

5.2 Implementation of Alternative Store Strategies

In order to compare the store buffer implementation to other store implementation techniques, several alternative methods were also implemented in the simulator. The implementation of load instructions remains the same across all strategies. The two competing techniques investigated were 2-cycle store instructions and blind-writing. For both strategies, implementations in pipelined and non-pipelined caches were considered and are described below.

The tables 5.2 through 5.5 represent the flow of instructions for a particular store strategy. An instruction is represented by a letter which indicates the instruction type, for instance 'S' indicates a store instruction. Each column represents the state of the TR, DC and TC stages of the pipeline at a particular clock cycle. Subsequent columns represent the progression of the instructions in the pipe as the clock advances.

SYMBOL	
X	= any instruction
S	= store instruction
-	= empty slot
X*	= stalled instruction

2-CYCLE STORE

(a) single stage data cache

CYC	1	2	3
TR	X1	X1*	X2
DC	S	S*	X1
TC	X0	-	S
	(hit/miss)		

penalty cycle = cycle 2

(b) in pipelined cache

CYC	1	2	0	3
TR	X1	X2	data	X2*
DC	S	X1	tag	X1*
TC	X0	S	write	-
		(hit/miss)		

penalty cycle = cycle 3

Table 5.2: 2-Cycle Store Instructions

3-CYCLE STORE
in a pipelined cache

CYC	1	2	3	4
TR	X1	X2	X2*	X2*
DC	S	X1	X1*	X1*
TC	X0	S	-	-
		(hit miss)	(data write)	(redo)

penalty cycle = cycles 3 and 4

Table 5.3: 3-Cycle Store Instructions

2-Cycle Store Instructions

As explained in chapters 2 and 3, a store instruction implemented in a generic five stage RISC pipeline, can take two cycles to complete. In the first cycle the tag is checked and in the second cycle the data is written. The data cache miss is determined during the DC stage and in the next cycle the store instruction stalls while the cache is written. Thus there is a one cycle penalty for the store instruction. This is illustrated in table 5.2(a).

However, in the pipelined data cache in our simulator, this is difficult to implement. The data cache hit or miss is not determined until the store instruction reaches the TC stage. The next cycle (clock 3) could be used to write the data, but an intervening instruction, X1 may access the incoherent cache and receive stale data. Instead, the behavior of 2-cycle stores in a non-pipelined cache is modeled by introducing a clock period, *clock 0*, between clocks 2 and 3 where the data is updated. This is illustrated in table 5.2(b). The instruction X1 is stalled in clock 3 and there is a single cycle penalty. The final state of the pipeline is the same as that for the 2-cycle store in a non-pipelined cache. This was done in order to make a comparison of store buffers with 2-cycle stores in a non-pipelined cache,

3-Cycle Store Instructions

If store instructions are implemented in a pipelined cache, there will be two penalty cycles. The first penalty cycle occurs after the data cache hit/miss is determined in the TC stage. This is illustrated in table 5.3. During the first penalty cycle, clock 3, the data gets written. The second penalty cycle occurs because of the intervening instruction X1, which may have accessed stale data in the cache. It is necessary to re-execute this intervening instruction to make sure that it sees the updated data cache.

Of course there are optimizations that can be performed, for example only if X1 is a load/store instruction does the pipe need to be stalled. In that case when X1 is an ALU operation, there is only one cycle penalty. However the performance of the optimized version is limited by the performance of the 2-cycle store instruction.

2-Cycle Blind Writing and 3-Cycle Blind-Writing

Blind-writing is a store instruction strategy for direct-mapped write-through caches in which the data is written concurrently with the tag check. If the tag check is a hit then the store

2-cycle BLIND WRITING
(a) single stage data cache

CYC	1	2	3
TR	X1	X1*	X2
DC	S	tagfix	X1
TC	X0	S	-
	(miss)		

penalty cycle = cycle 2

(b) in pipelined cache

CYC	1	2	0	3
TR	X1	X2		X2*
DC	S	X1	tagfix	X1*
TC	X0	S		-
		(miss)		

penalty cycle = cycle 3

Table 5.4: Blind-Writing Store Instructions

instruction can proceed. However if the tag check is a miss, the data cache has been incorrectly updated and the store instruction stalls to either correct the tag or invalidate the word. Chapter 2 explains blind-writing in more detail.

Figure 5.4(a) illustrates how blind writing works in single stage data cache. A store instruction passing through the data cache stage writes the data and checks the tag concurrently in clock 1. If there is a hit, the store instruction just proceeds without stalling. However if there is a miss, the cache is incoherent. A penalty cycle, clock 3, is used to fix the tag and valid bits for the store instruction. Hence there is a one cycle penalty only on store instructions that miss in the cache.

As in the 2-cycle store instruction case, this is difficult to extend to pipelined caches. The tag hit or miss is determined in the TC stage at which point there is already an intervening instruction in the DC stage. The tag fix must consume an extra cycle and the intervening instruction X1 must be run a second time. Thus there will be two penalty cycles, which is called 3-cycle blind writing technique in the rest of the chapter. The implementation of 3-cycle store instructions is illustrated in figure 5.5.

Figure 5.4(b) shows how 2-cycle blind writing is simulated in a pipelined cache. A tag fix cycle, *clock 0*, is introduced between cycles 2 and 3 so that when X1 is re-executed in cycle 4, the cache is coherent.

Implementation In Simulator

The above store methods were implemented in the simulator as run-time options. Both the 3-cycle store instruction and the 3-cycle blind-writing options were implemented at register-level while the 2-cycle store instruction and 2-cycle blind-writing options were simulated by

3-Cycle BLIND-WRITING
in pipelined cache

CYC	1	2	3	4
TR	X1	X2	X2*	X2*
DC	S	X1	X1*	X1*
	blind-wr		tagfix	
TC	X0	S	-	-
		hit/miss		redo

penalty cycle = cycles 3 and 4

Table 5.5: 3-Cycle Blind-Writing Store Instructions

artificially inserting an extra cycle.

In order to model these store strategies the simulator was modified in mainly three areas: the Pending-Update bit of the IO buffers, the Data Cache logic and the DC stage control logic. Firstly, the store buffer option sets the pending-update bit in the IO buffers when a store instruction is queued while the other strategies update the cache directly. Secondly, all strategies except store buffers, write into the cache from the TC stage as well as the IO buffers. Finally, the pipeline control logic for stalling the DC stage is customized for each strategy. The cache is still write-through and load instructions are still implemented as non-blocking therefore the remaining IO buffer logic is unchanged.

Chapter 6

Store Buffers: Comparative Analysis

6.1 Simulation Environment

The simulator for the processor was written in C. It has a register-level description of the pipeline and all the store strategies. The simulator is highly configurable with runtime options for chosing a store mechanism, varying the IO latency of both the initial word and subsequent words of a block transfer, and varying both the number of lines and words per line of the data cache.

Non-blocking loads were used across all store implementations so that the isolated performance of the store strategy could be measured. Although non-blocking loads are only implemented in a few of today's microprocessors, they will become more prevalent and necessary to decouple the IO latency from the pipeline execution to achieve higher performance. Non-blocking load instructions should not have an effect on the performance of any of the store strategies.

Statistics were gathered for each of the store strategies on clocks per instruction (CPI), pipeline stalls, number of store penalty cycles and load performance under different cache and IO parameters. Additional statistics were gathered for store buffers on the utilization of the IO buffers and the various optimizations.

6.2 Performance Measurements

In the next section preliminary results comparing the various store strategies are presented. These results were generated by a random code generator and do not reflect actual compiler output in terms of instruction mix. Instead they represent extreme conditions in terms of IO load/store behavior and reflect worst and best case behaviors. As a result, even though exact performance gain numbers are not obtained, they give an idea of the relative performance of store buffers compared to other store implementations with different IO behavior, data cache sizes and cache thrashing.

Random code generator:

The code generator produces a random sequence of instructions with a mix of 1/3 ADD instructions (representative of ALU operations) 1/3 LOADS and 1/3 STORE instructions. Registers for the add and load instructions were chosen randomly from R1 to R5 to generate sufficient data dependencies between ALU operations and data loads. Load and store memory locations were chosen to be (nrand(2)*1000 + (nrand(3)<<2)) to produce 32-bit addresses of the form 0x0000[2/3]00[0/4/8/c]. Hence all memory operations operate on basically two data sets from 0x3000 to 0x300c and 0x2000 to 0x200c. By varying the cache size these two data sets can be made to index two different cache lines or collide in the same cache line.

Notice that there is an extremely high percentage of load/store instructions operating on a relatively small data set – about 2/3rds of the code sequence accesses the data cache. This has several effects.

- The IO is stressed by the large number of memory read and write requests.
- If the cache size is such that the two memory address sets represent two lines that conflict in the cache (i.e. same cache index but different tags) then the miss rate becomes very high. This is because nearly all the load and store instructions thrash each other.
- On the other hand, if the cache size is large enough, the two line addresses index different lines and never conflict. Then after the initial load of the data cache working set, none of the loads or stores ever miss in the data cache. These two situations represent the extreme cases of having a thrashing cache and having no cache misses at all, thereby representing both the worst case and best case scenarios.
- Since there is such a high percentage of instructions that access the cache, the DC stage is highly utilized. There are many sequential memory instructions so the data cache has very few empty slots to allow for background updates.
- Both load and store instructions access the same small memory region. This exacerbates the interaction between store instructions and load instructions queued in the IO buffer.

Thus the random code compiler generates a mix of instructions that, together with the variable cache and IO parameters, represents most worst case scenarios for memory operations.

The code generator produces a sequence of a 1000 instructions and a jump instruction to the beginning to form a loop. There are no other control flow instructions in the code. The Instruction cache size is chosen to be 256 lines, 4 words per line, which is large enough to hold the entire code sequence.

Therefore after the first loop through the code the instruction cache never misses. The instruction cache no longer contends for IO and the IO behavior directly affects the load/store performance. This does not reflect the behavior of most application code. Instead it represents the instruction cache as an ideal cache and thus isolates the behavior of the store method being analyzed.

Another point to note is that since there are no IC misses and mostly sequential code, the ID stage never generates an empty slot after the first iteration and the issue rate becomes exactly one instruction per cycle. This further stresses the store buffer performance by representing the extreme case in which the data cache is highly utilized and there are very few slots for background data cache updates.

Performance measurement in terms of CPI

The measurements taken by the simulator consist of running the 1000 instruction long code sequence on the machine and computing the CPI from the number of cycles taken to complete 10^6 instructions. The CPI is used as the main comparison of the performance for the different store mechanisms. The CPI is used because in the case of a direct-mapped write-though cache the critical path times for the various store methods do not differ significantly and more importantly, the cycle time for the store buffer implementation is no more than that for any of the other strategies.

This can be seen from the register level implementation of store buffers described in previous chapters. Since the data cache is usually the critical path in a pipeline it determines the clock speed of the pipeline. Store buffers increase the delay path through the write buffers by adding additional state. However no logic is added to the data cache array access or tag comparison. Hence the critical path through the cache is not increased.

Blind-writing however uses the output of the tag check comparator to determine whether the store instruction missed and whether the tag needs to be fixed the next cycle. Hence the data cache hit signal is used to determine the cache tag array write enables and the control logic for stalling the DC stage. This increases the delay path through the DC stage and may decrease the clock speed. Either way the cycle time for store buffers is no worse than that for any of the other techniques being measured.

2-cycle and 3-cycle implementations (non-pipelined vs pipelined cache)

In the comparisons below, store buffers are compared to 2-cycle and 3-cycle stores, and 2 and 3-cycle blind-writing. The store buffer strategy, however, is only implemented in a pipelined cache. Its performance in a single stage pipe would be similar. Since the cache update phase is not dependent on the tag check timing and is delayed, moving the tag comparison to the TC stage does not affect the performance adversely. The only difference between the two is the introduction of a gap between the store buffer lookup in the DC stage and the buffer allocation in the TC stage when data dependencies need to be checked. This is handled by a comparator between the TCa and DCa address registers.

However, for the other store options there is significant performance decrease from a single stage cache access to a pipelined cache. This is because the alternative methods depend on the second phase of the store (whether data write or tag fix) occurring immediately after the data cache is read and before any other instruction accesses the data cache. Moving the tag check to a later stage introduces an extra penalty cycle in both blind-writing and 2-cycle stores. Therefore for those methods both pipelined and non-pipelined caches are considered and measured separately.

6.3 Quantitative Analysis

All the following plots were generated from data obtained by executing a thousand instruction loop sequence until the completion of a million instructions. Store buffers have no penalty for cache hits or misses, but can stall the pipe if the IO buffers fill up. Out of the alternative store techniques proposed, 2-cycle blind-writing performs the best since it stalls the pipe only for store misses. It does not depend on the number of IO buffers. Therefore it is not obvious which of the two strategies, store buffers or blind-writing, will perform better under different circumstances. The performance of store buffers and 2-cycle blind-writing was compared in more detail to determine the performance tradeoffs more accurately.

A. Number of IO Buffers

Figure 6.1 shows the performance of the various store strategies with different number of IO buffers. The IO latency is (4 ,2) which implies four cycles for the first word and two cycles for subsequent words in a block transfer. The cache size is 256x4 (256 lines, 4 words/line). Therefore the addresses 0x2000 and 0x3000 index the same cache line and the cache exhibits thrashing.

In this test the cache has a very high miss rate because of thrashing and the number of memory read requests is very high. Increasing the number of IO buffers increases the performance across all strategies by allowing more pending read requests. When the number of IO buffers exceeds four the performance gain levels off because four IO buffers were sufficient to hold all outstanding pending-reads.

Store buffers perform consistently better than any of the other techniques measured. Even with as few as three buffers, there is a significant performance gain. This is because the store miss rate is very high. All store methods except store buffers pay a high penalty for store misses. Only the store buffer strategy is able to eliminate store penalties by taking advantage of the existing IO buffers and hence provides significant performance increase.

From figure 6.1 we see that, as the number of IO buffers increase, the performance also increases but with diminishing returns. The initial increase in performance is because of the availability of buffers for load requests. The performance of store buffers continues to increase beyond four buffers because of the availability of more IO buffers for store cache update requests. The performance becomes steady when all cache update requests are being efficiently serviced.

The number of IO buffers required by the store buffers technique to completely eliminate pipe stalls by stores is equal to the greater of the number of pending-write requests or the number of pending cache updates. The data shows that even with a high percentage of store/load instructions, as few as four IO buffers gives a significant performance advantage over other techniques. This gives us an idea on the number of buffers that will be required to implement store buffers.

B. IO Latency

Figure 6.2 shows the results of varying the IO latency while keeping all the other parameters constant. Data points were taken by varying both the initial latency and the block transfer latency, but only the former is plotted. Varying the block size did not generate significant

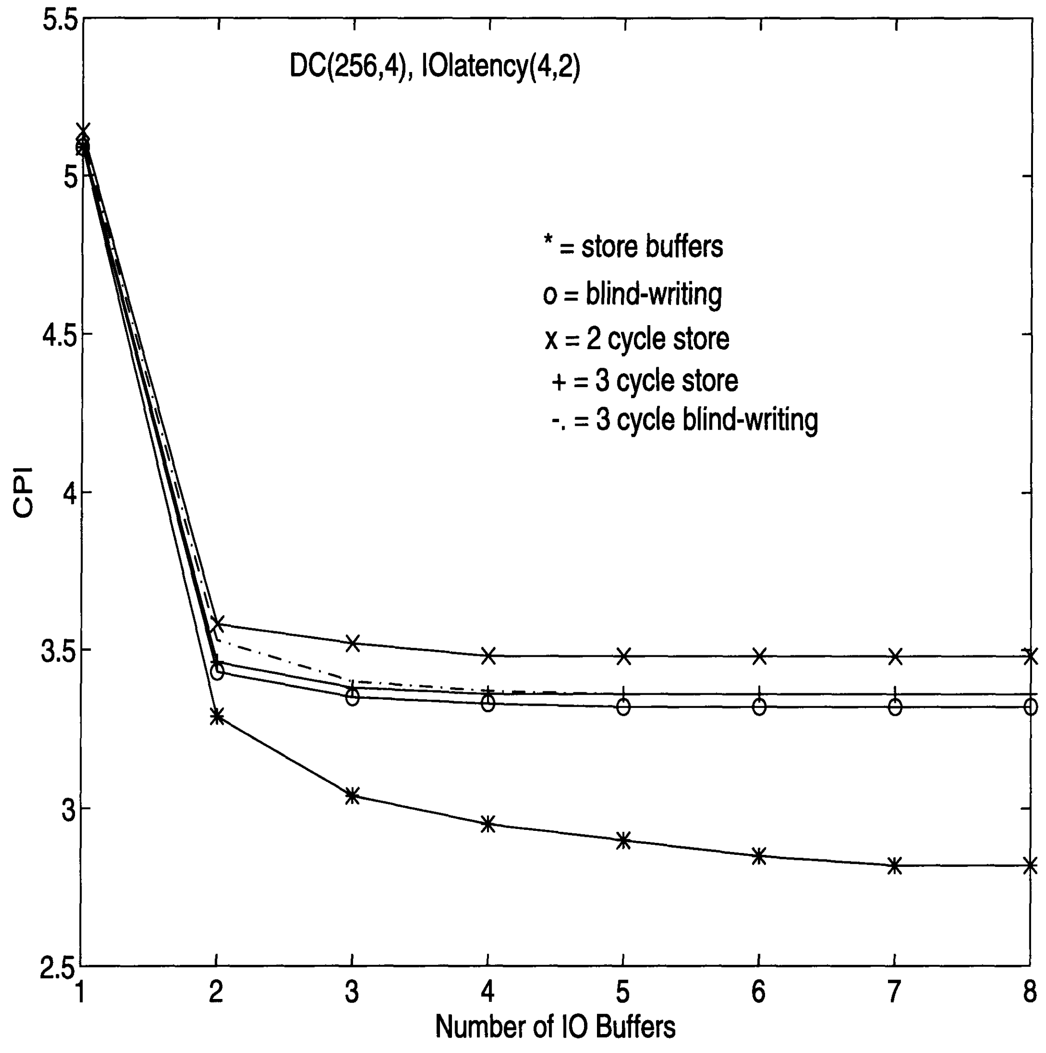

Figure 6.1: Plot: CPI vs Number of IO Buffers

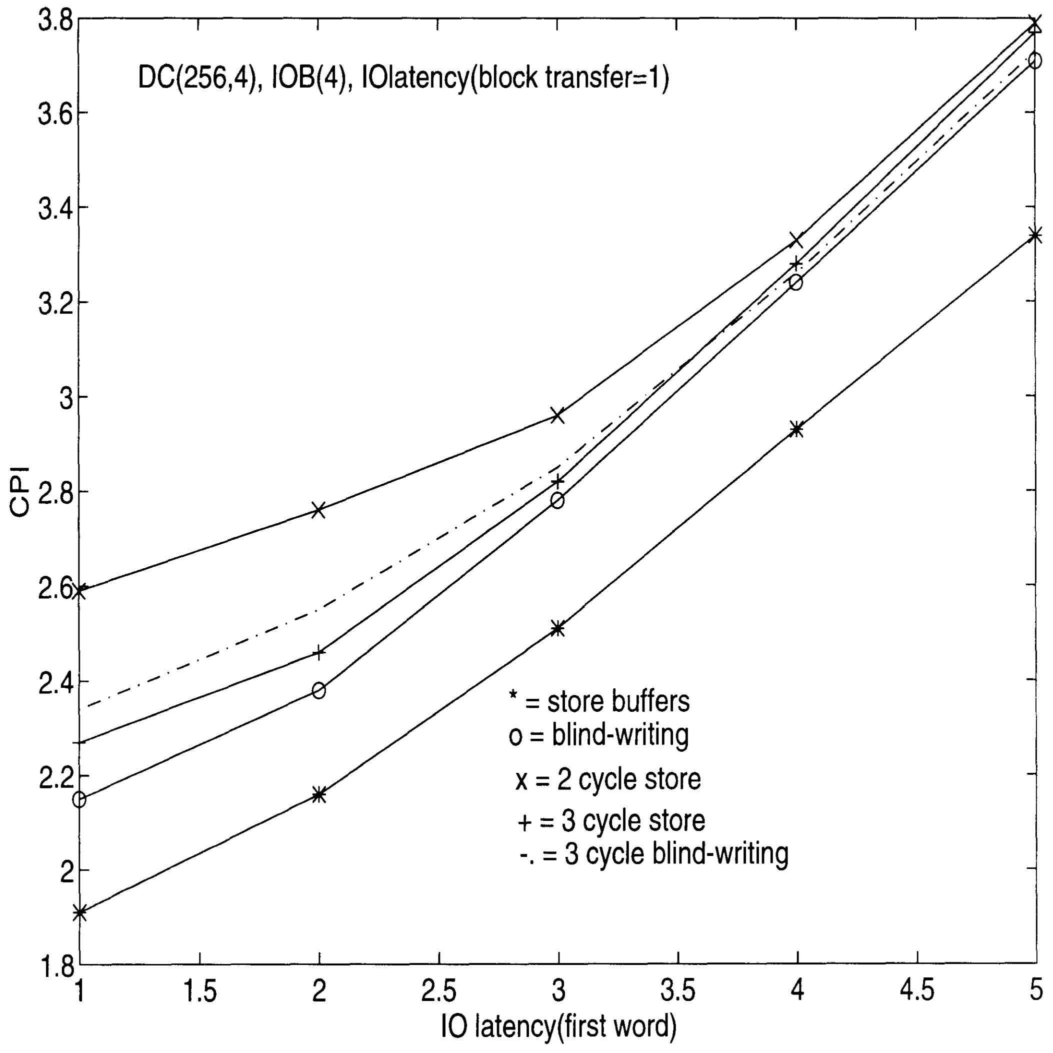

Figure 6.2: Plot: CPI vs IO Latency for Initial Word

Store Method	DC lines	Words 1	per 2	line 4	8
store buffers		2.42	2.70	1.45	1.45
2 cycle store		2.74	3.04	1.68	1.68
3 cycle store	512	2.85	3.14	2.01	2.01
blind-writing		2.69	2.99	1.45	1.45
3 cycle blind-writing		2.75	3.04	1.46	1.46
store buffers		2.42	2.70	2.95	1.46
2 cycle store		2.74	3.04	3.36	1.68
3 cycle store	256	2.85	3.14	3.48	2.01
blind-writing		2.69	2.99	3.33	1.45
3 cycle blind-writing		2.75	3.04	1.37	1.46
store buffers		2.42	2.70	2.95	3.02
2 cycle store		2.74	3.04	3.36	3.53
3 cycle store	128	2.85	3.14	3.48	3.73
blind-writing		2.69	2.99	3.33	3.50
3 cycle blind-writing		2.75	3.04	1.37	3.56

Table 6.1: Performance with Different Data Cache Sizes

differences in performance, the initial latency was the main factor affecting the CPI. This is because the block transfer mode was not utilized – store instructions cannot take advantage of block transfer rates, the instruction cache never misses and there were not many load requests for entire lines. The data cache is 256x4 and there are four IO buffers. Four IO buffers was the least number of buffers in plot 6.1 that provided a significant performance increase for all strategies.

The performance for all strategies severely deteriorates as the initial IO latency is increased. This is because the cache has a very high miss rate and this increases the miss penalty for both memory read and write requests. With this test code, the pipe stalls frequently because of data dependencies from pending-reads and because of the IO/write buffers filling up with memory requests.

In general we can see that store buffers perform consistently better than all other techniques. However, the relative performance gain from using store buffers is somewhat constant. This is because the performance gain is obtained from the number of store cache updates that are serviced in the background and is not directly affected by IO latency. Cache updates are easily serviced in the background since the pipe is often stalled by a full IO buffer or an ID stage data hazard. Hence high off-chip IO latencies generate sufficient DC stage slots to retire pending updates from the store buffers which makes store buffers a more efficient strategy under these circumstances.

Figure 6.3 shows the performance of store buffers with varying initial IO latency and block transfer latency. The performance for a given initial IO latency stays almost constant even when the block transfer rate is varied. This is because the number of memory write requests and single word read requests outnumber the full line fetches. Hence the block transfer rate does not appear to affect the performance of store buffers in this test code.

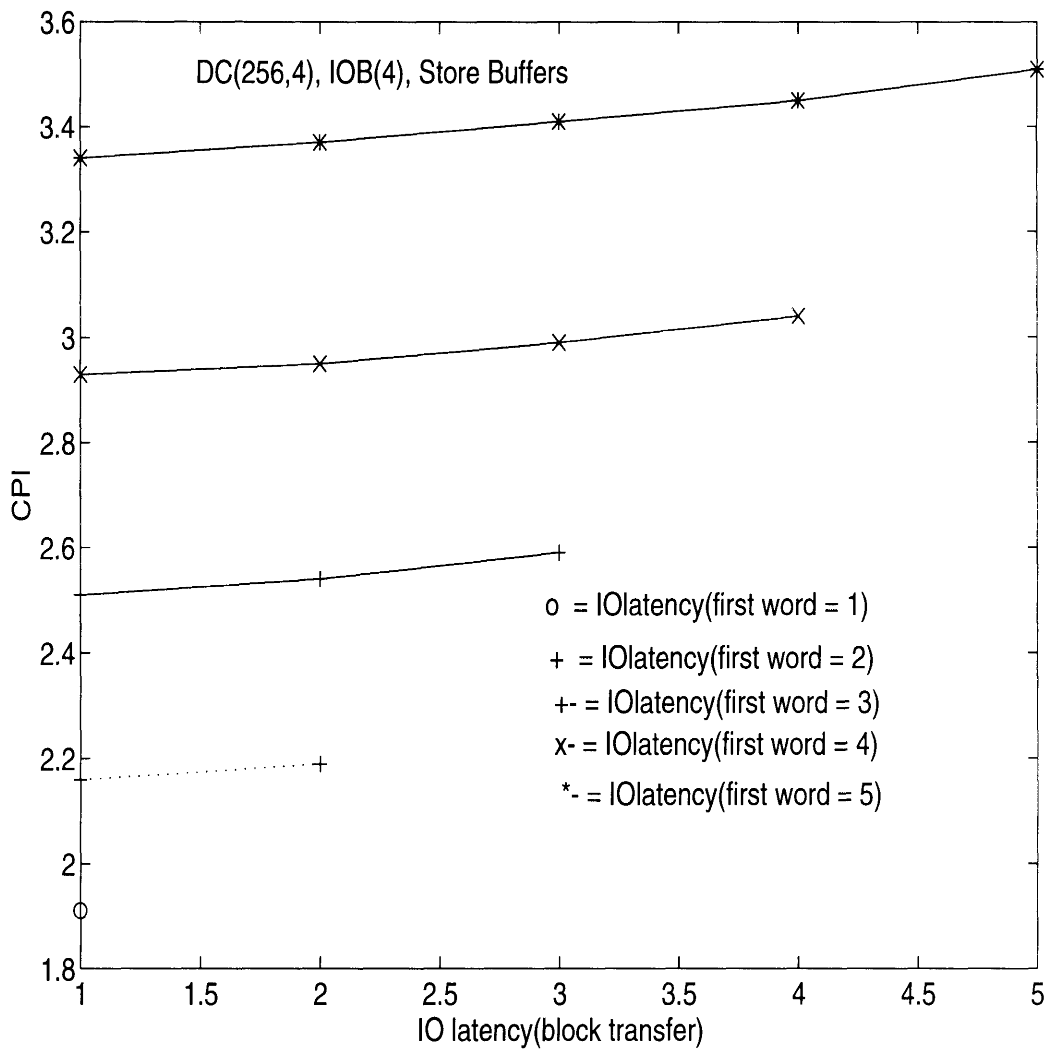

Figure 6.3: Plot: CPI vs IO Latency for Store Buffers

C. Data Cache Size

Table 6.1 shows the performance of all the strategies with varying data cache sizes. Both the number of lines and words per line are varied. The IO latency is kept constant at (4,2) and there are four IO buffers. The cache size varies from 128 to 1024 lines and 1 to 8 words per line.

All previous experiments were performed with a cache size small enough for all store and load addresses to collide. Different cache sizes have different hit/miss properties. The data for each of the strategies is presented in a table and a graph is plotted for store buffers and blind-writing.

The code for this experiment is characterized by a very high percentage of loads and stores to the same memory region. An ideal cache is one that can hold the entire working set. The hit rate for such a cache is 100%. A worst case cache is one where all the loads and stores collide. This implies that all data cache accesses miss in the cache, request IO, update the corresponding cache line and then get overwritten by subsequent cache accesses. The miss rate for such a cache is very high.

Hence this code represents two extreme cases, an ideal cache and a thrashing cache, depending on the overall cache size rather than the division into lines and blocks.

This can be observed from the table. For any strategy, all caches of size 4K words or larger have no cache misses. Therefore the performance over all such caches is constant irrespective of the number of words per line. For caches smaller than 4K words the miss rate is very high as a result of cache thrashing. Increasing the block size decreases performance because of the fetch-on-miss policy for load misses. Pending reads tie up the IO fetching entire cache lines that later get invalidated by subsequent load misses. Hence all loads unnecessarily incur the penalty of a block transfer. As the block size increases this penalty increases.

The difference in cache size can affect performance significantly, more than any other factor measured in these experiments. Therefore trading space, miss penalty and load delay slots for a larger cache at the same clock speed is a very beneficial tradeoff. This result shows that the advantages of pipelining a cache will outweigh other factors. With an optimizing compiler the performance of a pipelined cache can be further increased.

Figure 6.4 plots the data points from the table for store buffers and blind-writing. For caches larger than 4K words, blind-writing represents the ideal case in which all store instructions execute in a single cycle. This is because the cache is large enough to hold the entire working data set. Therefore there are no store misses and no penalty cycles. Store buffers perform as well as blind-writing which implies that it is able to effectively perform all cache updates in the background with as few as four buffers.

In smaller caches, blind-writing pays a high penalty because of the high store miss rate whereas the store buffer strategy is able to perform all store cache updates in the background without stalling the pipe. Store buffers perform better than all other techniques for small caches.

D. IO Buffers and IO Latency

Figure 6.5 investigates the effects of varying the number of IO buffers and off-chip IO

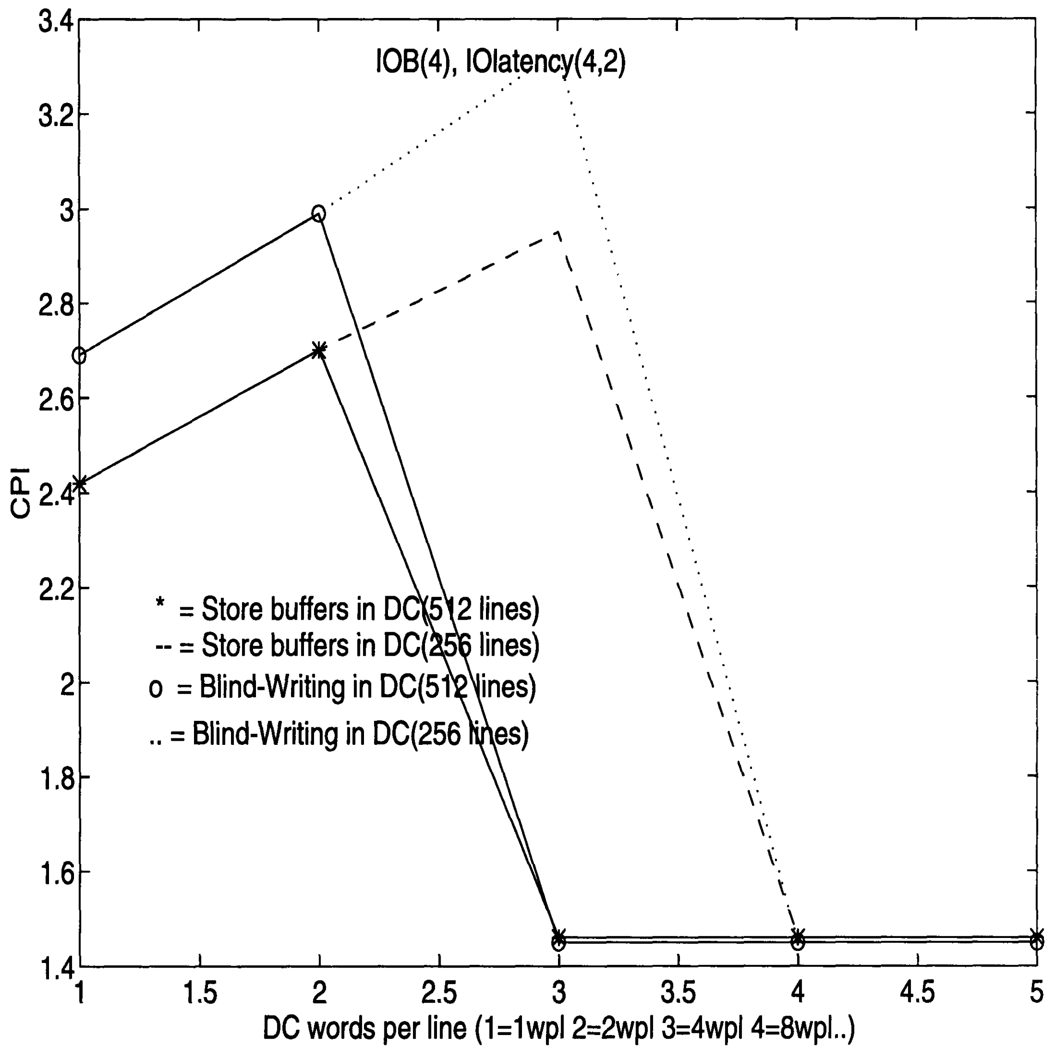

Figure 6.4: Plot: CPI vs Data Cache Size

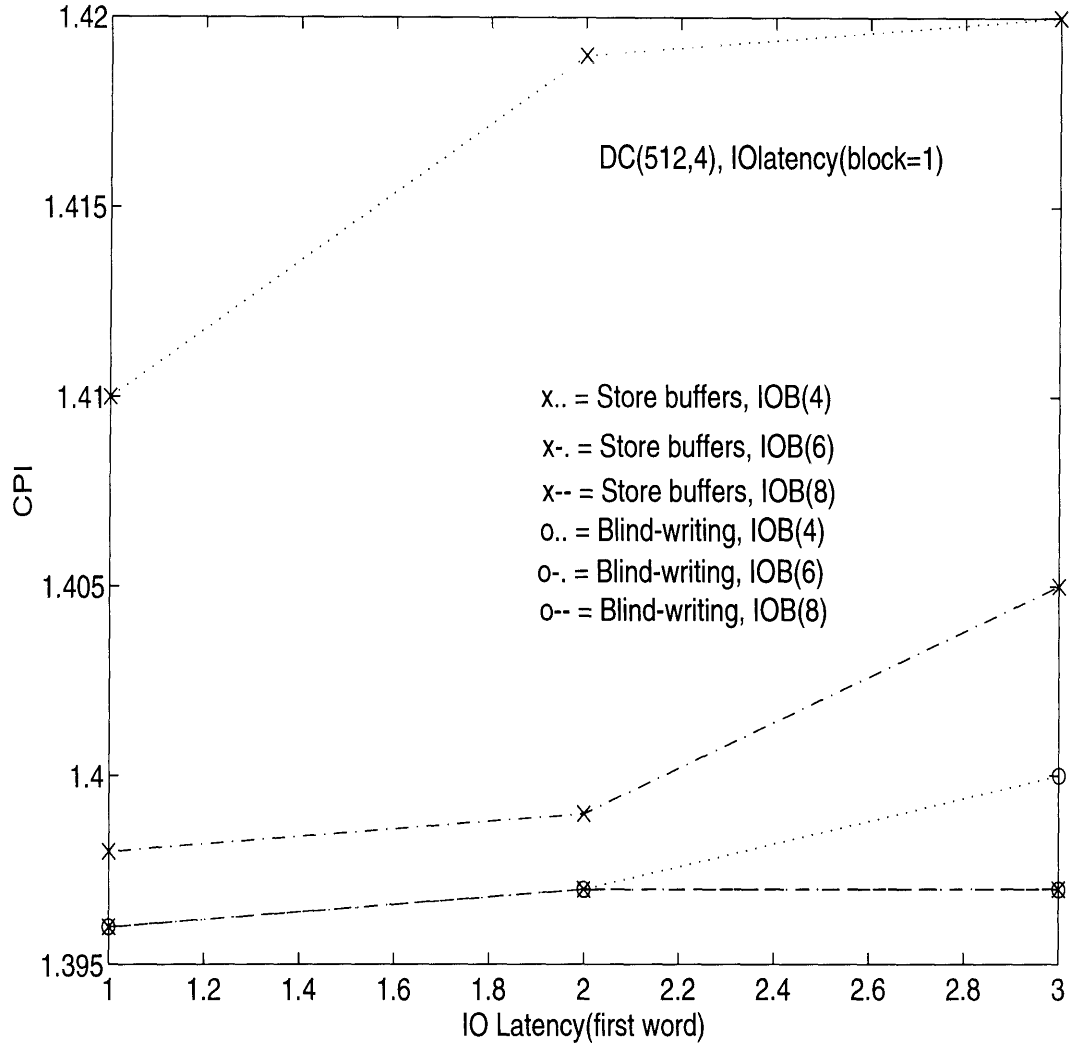

Figure 6.5: Plot: CPI vs IOB Size and IO Latency

BLIND-WRITING

IOB	stat	IO(1,1)	IO(2,1)	IO(3,1)
4	CPI	1.396	1.397	1.40
	IOB full	0	0	1
	max PWs	3	3	4
6	CPI	1.396	1.397	1.397
	IOB full	0	0	0
	max PWs	3	3	3
8	CPI	1.396	1.397	1.397
	IOB full	0	0	0
	max PWs	3	3	3

STORE BUFFERS

IOB	stat	IO(1,1)	IO(2,1)	IO(3,1)
4	CPI	1.41	1.419	1.42
	IOB full	12974	14970	14971
	max PUs	4	4	4
6	CPI	1.398	1.399	1.40
	IOB full	2994	4990	4990
	max PUs	6	6	6
8	CPI	1.396	1.397	1.397
	IOB full	0	0	0
	max PUs	6	6	6

Table 6.2: Performance with Different IO Latencies and IO Buffer Sizes

latency in an ideal cache. The blind-writing technique performs ideally, with all store and load instructions being single cycle. Store buffers however show slightly worse performance for higher IO latencies and fewer buffers.

Additional statistics were gathered and are shown in table 6.2. As can be seen, the maximum number of pending memory-writes is less than the minimum number of buffers provided. As a result the blind-writing technique never stalls on a full IO buffer. However the maximum number of outstanding cache updates is six for store buffers so, for less than six IO buffers, the IO buffers fill up and cause the pipe to stall. Hence the number of cache updates exceeds the pending-writes. This causes store buffers to perform slightly worse than ideal. It is expected that IO latency would be the more constraining factor and that the maximum number of outstanding memory writes would exceed the maximum number of pending cache updates. This is not true in this case because of the nature of the test code used in this experiment. Only 1/3rd of the instructions are ALU operations which allow background cache updates. Also, the ideal IC and DC generate no empty slots in the pipe. Hence there are very few DC stage slots for cache updates. In compiler generated code, more slots for cache updates would be available and the pending-updates would be retired sooner. This experiment represents the extreme case in which the pipeline is highly utilized. It is important to note that store buffers perform well even under these circumstances.

6.4 Conclusions

From the preliminary data, store buffers seem to perform significantly better than other store techniques in most cases.

- In caches that have a high miss rate, store buffers perform significantly better even with as few as three IO buffers. This is because all other techniques pay a penalty for store misses. Even with a very high percentage of store instructions and load misses, as few as four to six buffers suffice to virtually eliminate stalls caused by stores.

- The comparison to blind-writing in a thrashing cache is very important. It implies that even if blind-writing is optimized to eliminate the penalty for some store misses, for instance only stall for tag fix if subsequent instruction is a load/store to same cache line, it cannot eliminate the penalty. The case in which a tag fix must stall the pipe, i.e. when there are subsequent load/store to same cache line, is a common occurrence in compiled code. Back to back store operations to consecutive data locations are exhibited by many applications. Store buffers on the other hand can virtually eliminate all stalls by having sufficient buffers.

- In a cache that never misses, which represents an ideal data cache, store buffers perform as well as blind-writing and better than all other techniques, given enough buffers. Blind-writing in this case is the same as single cycle store instructions, since there are never any store misses and therefore no penalty cycles. Store buffers achieve the same performance, even with such a high percentage of data cache accesses, with as few as six buffers.

- The store buffer mechanism does however seem to require more buffers than a normal write buffer would need. This is because there are very few DC stage slots to retire pending cache updates. Compiled code does not show such a high percentage of load/store instructions and the data cache is unused often enough to allow pending-updates to retire sooner than pending-writes. As a result store buffers would probably provide the same performance in an ideal cache with just enough buffers to service outstanding pending-writes.

- Under long IO latencies, the same performance measures for ideal and thrashing caches apply. This is because IO latency does not affect store buffer performance. It causes the performance under all methods to degrade. The relative performance of store buffers compared to other strategies remains the same.

- Pipelined caches allow for larger caches without decreasing the clock speed and provide a significant performance increase through improved hit rates. The performance of blind-writing in a pipelined cache shows significant degradation. The store miss penalty is higher and adversely affects the performance. Hence store buffers is a better strategy when the cache access is split across more than one stage.

These results have been very encouraging. However simulations run on compiled code will be needed to make more accurate performance estimations of real-life performance.

6.5 Future Work

Future work will consist of porting a compiler and studying software techniques to optimize code for IO buffers. Different styles of application code will be run to get exact performance measurements and to quantify the performance gain of using store buffers. The simulator could be changed to also simulate set associative caches. Once that is complete, the simulator could also be extended to study other implications of store buffers, such as using the store buffers as a victim cache or as a delayed cache for snooping operations.

Appendix A

Instruction Set Listing

Arithmetic Operations	
ADD	add
ADDI	add immediate
SUB	subtract
LUI	load upper immediate
MUL	multiply
DIV	divide

Logical Operations	
AND	AND
OR	OR
XOR	exclusive OR
ANDI	AND immediate
ORI	OR immediate
XORI	exclusive OR immediate

Shifts	
SLL	shift left logical
SRL	shift right logical
SRA	shift right arithmetic
SLLI	shift left logical by immediate
SRLI	shift right logical by immediate
SRAI	shift right arithmetic by immediate

Jumps	
JAL	jump and link
JMP	jump immediate
JMPR	jump to register and link

Branches	
BEQ	branch on equal
BNE	branch on not equal
BGT	branch on greater than
BGE	branch on greater than or equal
BHI	branch on higher
BHS	branch on higher or same
BEQL	(link version)
BNEL	
BGTL	
BGEL	
BHIL	
BHSL	
BEQT	(predict taken)
BNET	
BGTT	
BGET	
BHIT	
BHST	
BEQLT	(predict taken and link)
BNELT	
BGTLT	
BGELT	
BHILT	
BHSLT	

Load/Store Operations	
LW	load word
LH	load half
LHU	load unsigned half
LB	load byte
LBU	load unsigned byte
SW	store word
SH	store half
SB	store byte

Special instructions	
KCALL	kernel call
KRET	kernel return

Floating point instructions	
LS	load single precision number
LD	load double precision number
SS	store single
SD	store double
FADDS	add single
FSUBS	subtract single
FMULS	multiply single
FDIVS	divide single
FADDD	add double
FSUBD	subtract double
FMULD	multiply double
FDIVD	divide double
FMAC	multiply and accumulate
FCVTDS	convert double to single
FCVTDW	convert double to word
FCVTSD	convert single to double
FCVTSW	convert single to word
FCVTWD	convert word to double
FCVTWS	convert word to single
FEQ	compare equal
FGT	compare greater than
FGE	compare greater than or equal
FUN	compare not a number

Appendix B

Processor Block Diagrams

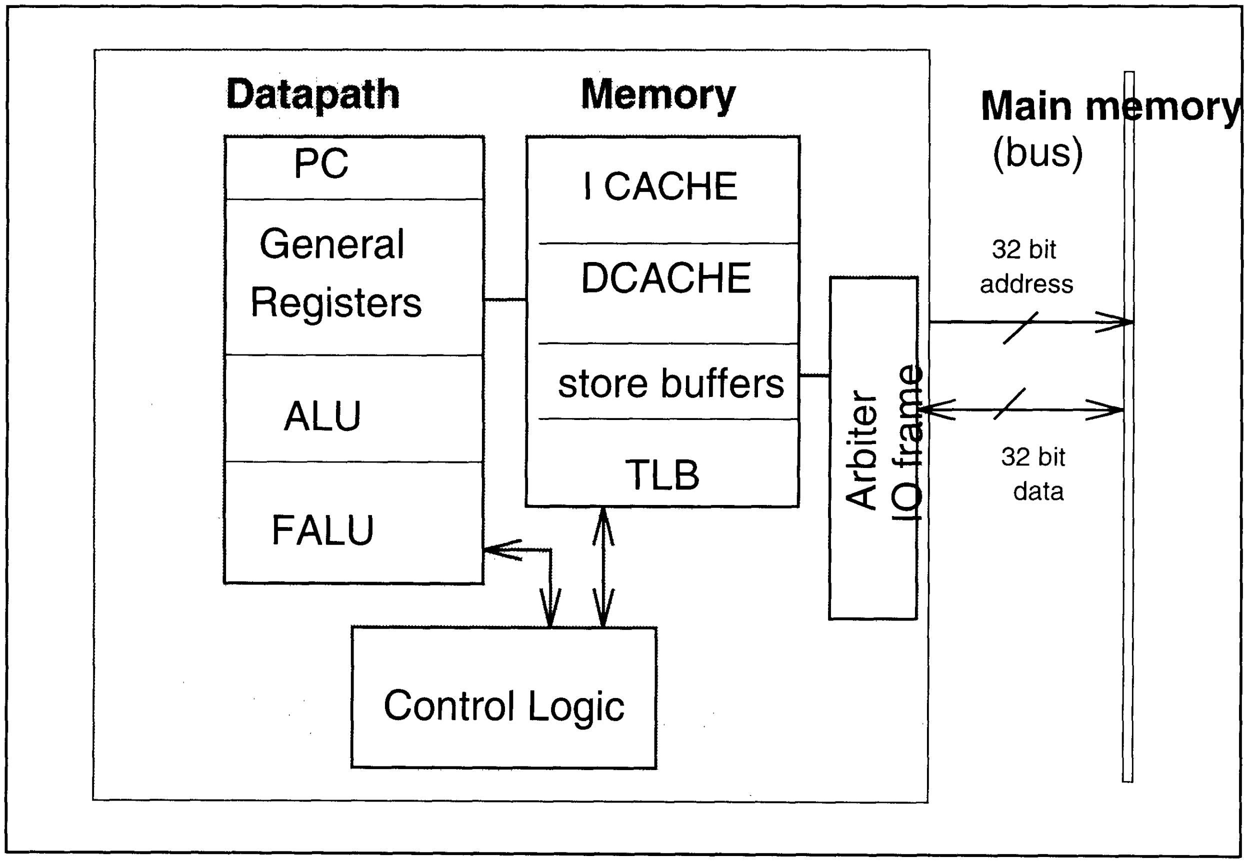

Figure B.1: Block Diagram of Processor

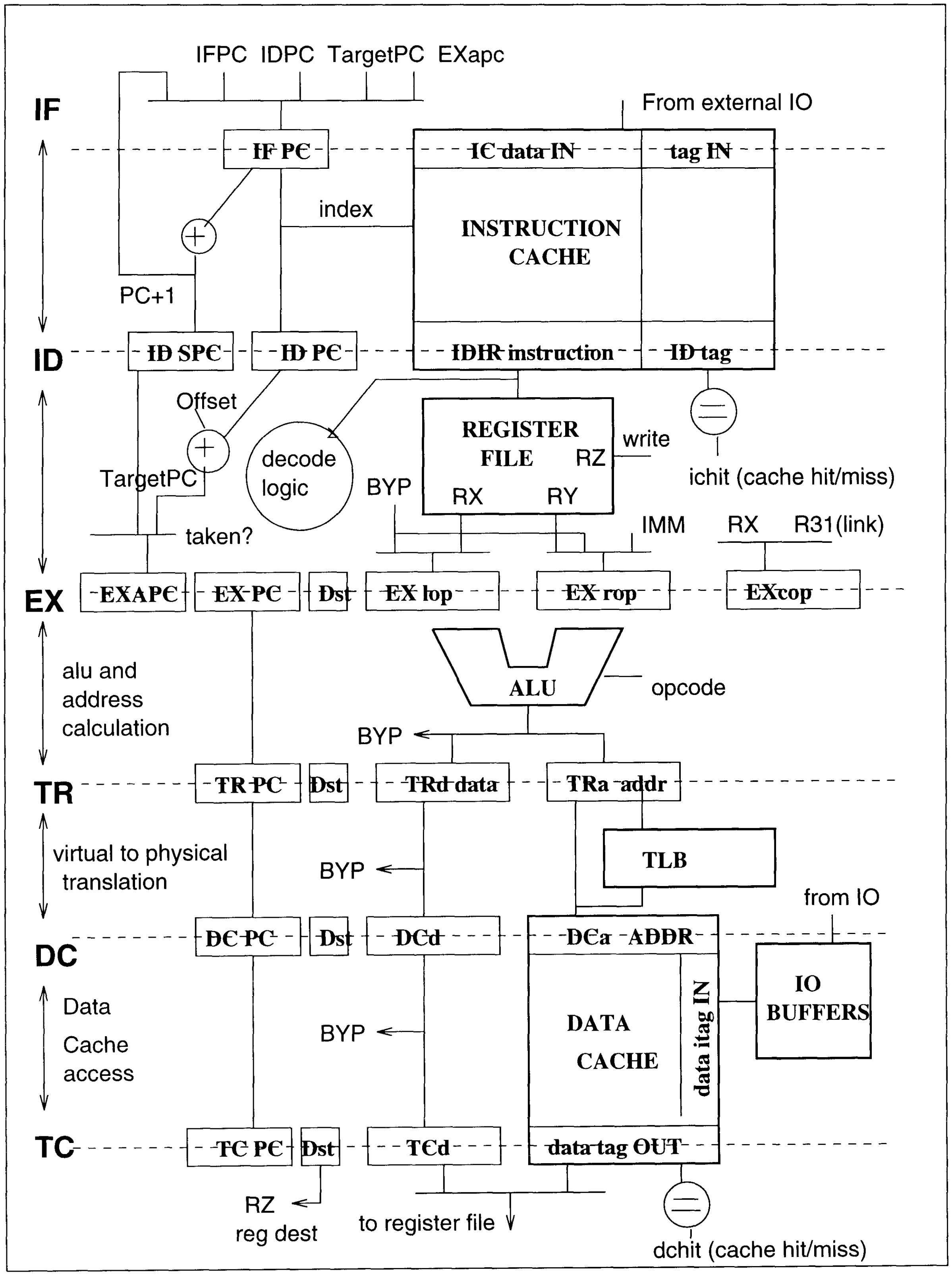

Figure B.2: Pipeline Architecture

Appendix C

Simulator Code Excerpts

The simulator is a register level implementation of the processor described in Chapter 4. The code for this simulator was written in C and run on an SGI challenger at AT&T Bell Labs.

Separate code modules were written for the control and data paths. Physical modules like caches and register files were implemented as separate C procedures. The data structures chosen to represent the module reflect the actual implementation at a register level. Special C conventions were used to describe actual circuit elements like multiplexors, latches, AND OR gates, etc. The code almost directly corresponds to a register level schematic except it allows for parameterizing certain structures. In addition to the register level simulator there is an interpreter that verifies the output of the simulated pipeline and the processor memory state at every cycle to make sure the simulator is performing correctly.

Below are some excerpts of the code for the data structures and the IO buffer implementation.

```
/***************************************************************************/
/*---------fsim.h (excerpts): Data structures-------------------------*/
/***************************************************************************/

/* REGISTER FILE*/
#define REGSIZE         32
#define R31             (REGSIZE-1)

/*----------------------*/
/* INSTRUCTION ENCODING      */
#define IMM(x)        (((x) & 0xFFFF) | (((x) & 0x8000) * 0x1FFFE))
#define OFF26(x)      ((((x) & 0x03FFFFFF) | (0 - ((x) & 0x02000000))) << 2)
#define OFF16(x)      ((((x) & 0x0000FFFF) | (0 - ((x) & 0x00008000))) << 2)

/*----------------------*/
/* STORE METHOD      OPTION      */
internal          ST_OPT;
#define TCYC      1
#define THCYC     2
#define BLND      3
```

```
#define THBLND    4
#define SBUF      5

/*----------------------*/
/* IO INITIAL VALUES      */
#define INIT_IOCYCLES    4
internal IOCYCLES;
#define INIT_BLKCYCLES   2
internal BLKCYCLES;

/*----------------------*/
/* CACHE INITIAL VALUES */
#define INIT_ICSIZE     256        /* 256 lines -> 1K words       */
internal ICSIZE;                   /* Instruction cache size      */
#define INIT_ICBLK      4          /* 4K words (16K bytes)        */
internal ICBLK;                    /* Instruction cache size      */
internal LOGICBLK;                 /* Instruction cache size      */

struct regIC {      unsigned int      *inst;
                    unsigned int      tag;
                    unsigned int      v;
              };

/*----------------------*/
/* DCACHE INITIAL VALUES*/
#define INIT_DCSIZE     256         /* 256 lines-> 1K words       */
internal     DCSIZE;                /* data cache size            */
#define INIT_DCBLK      4           /* 4K words      (16K bytes)  */
internal     DCBLK;                 /* data cache block size      */
internal     LOGDCBLK;              /* log2(data cache block size)*/

struct regDC {      unsigned int      *data;
                    unsigned int      tag;
                    unsigned int      v;
             };

/*----------------------*/
/* IO BUFFERS            */
#define INIT_IOBSIZE     4
internal IOBSIZE;                  /* Number of load reservation stations */
struct Lbuf {
            unsigned int pw;       /* pending write to IO, 1 per byte */
            unsigned int pr;       /* pending read      to IO, 1 bit  */
            unsigned int pu;       /* pending DC update, 1 per word   */
            unsigned int fs;       /* fetch strategy, 1=line 0=word   */
            unsigned int tm;       /* tagmatch (whether to invalidate other words) */

            int a;                 /* word address to be written/read */
            int *data;             /* data to be written/received     */
                                   /* (if fs=1 then cacheline size, else 1 word)   */
            int *dst;              /* LD register dest (1 dst/word)   */
            int *regw;             /* register write enable/dst       */

            unsigned int sz;       /* LD data size                  */
            unsigned int sgn;     /* LD signed/unsigned            */
           };

/*-----------------------------*/
/* machine internal registers     */
intern_s      regIC      *icache;          /* pointer to instruction cache array  */
intern_s      regDC      *dcache;          /* pointer to data cache array         */
intern_s      Lbuf       *iobuf;           /* pointer to load buffers masters     */
intern_s      Lbuf       *Iobuf;           /* pointer to load buffers slaves      */
internal      regf[REGSIZE];               /* pointer to register file            */
intern double      freg[REGSIZE];


/**********************************************************************/
/*------ IOB.C (excerpts): IO buffer implementation--------------------*/
/**********************************************************************/
/*                  IO BUFFERS                                        */
/**********************************************************************/
/* IOB = PW:4 PR:1 PS:wpl TM:1 FS:1 ADR:1 DATA:wpl DST:wpl regw:wpl    */
/*--------------------------------------------------------------------*/
/* PW = 0xf => write word to memory                                   */
/* PR && PW !=0 => request word to merge subword, then write to memory */
/* PR && PW = 0 => if no conflict FS =1 (requesting cache line)       */
/*                  else FS = 0 requesting single word                */
/* PS && !PR => write word to data cache (if TM don't invalidate rest  */
/*            of word in line, else consider new cacheline            */
/*--------------------------------------------------------------------*/

iob(){

int i,j,tmp;
int eq_tag, eq_index, eq_word;
int update_word, word_arrived, in_arrival_window;
if (CLK1 && CKearly) {      /* early only for simulation calling order      */
      Bfree = bfree;        /* POINTERS: next free location                 */
      Bioreq = bioreq;      /* location to request io (waiting for grant)   */
      Bwait = bwait;        /* location waiting for iodone                  */
      Bdcreq = bdcreq;      /* location requesting data cache slot          */
      Blag = blag;          /* location in IO master (granted)              */

      for(i=0; i < IOBSIZE; i++) {          /* copy masters to slaves       */
            Iobuf[i].a = iobuf[i].a;
            Iobuf[i].pw = iobuf[i].pw;
            Iobuf[i].pr = iobuf[i].pr;
            Iobuf[i].pu = iobuf[i].pu;
            Iobuf[i].tm = iobuf[i].tm;
            Iobuf[i].fs = iobuf[i].fs;
            Iobuf[i].sz = iobuf[i].sz;
            Iobuf[i].sgn = iobuf[i].sgn;
            for(j=0; j < DCBLK; j++) {
                  Iobuf[i].data[j] = iobuf[i].data[j];
                  Iobuf[i].regw[j] = iobuf[i].regw[j];
                  Iobuf[i].dst[j] = iobuf[i].dst[j];
            }
      }
}

if (CLK1 && !CKearly) {
      if(tcclken) {
            Iobhit = iobhit;   /* LD: data available        */
            Tcwbd = tcwbd;     /* LD: output of write buffer */
            Tcfsx = tcfs;      /* LD: fetch strategy        */
            Tcu = tcu;         /* LD: cacheable word        */
            Iobmatch = iobmatch; /* LD: already IOB requesting   */
                              /* ST: location in IOB to merge to */
            Tcloc = tcloc;    /* ST: location to merge with      */
            Lbfree = lbfree;  /* last bfree location (b2bld)     */
       }
      /* TCA DCA comparators      */
      tc_dc_sameindex = (DCL_INDEX(Dca) == DCL_INDEX(Tca));
      /* Tca and Dca index same cache line          */
      tc_dc_sametag = (DCLINE(Dca) == DCLINE(Tca));
      /* Tca and Dca have same cache tag and index    */
      tc_dc_sameword = (DCW_INDEX(Dca) == DCW_INDEX(Tca));
      /* Tca and Dca have same word in line          */

      /* If the line already exists in the DC do not request the entire line      */
      /* since result of the tagmatch is not known until TC stage, must do here    */
      Tcfs = Tcfsx && !dctagmatch && !((Dcld || Dcst) && !tc_dc_sametag
&& tc_dc_sameindex);
      /* Dcld and Tcld to same cacheline last cycle  */
      same_iob = B2bld || Iobmatch;
      /* Tcld is collapsing into an existing request */
      /* IOB is available if free pointer is not pointing to used location */
      /* TC stage must not advance unless Free pointer has been updated    */
      iobfull = (Iobuf[Bfree].pw || Iobuf[Bfree].pr || Iobuf[Bfree].pu);
}

/*----------------------------------------------------------------*/
/* Operations performed on IOBs by LOAD/STORE/DC_UPDATES during DC stg  */
/*----------------------------------------------------------------*/
/* STORE => lookup for a location to collapse to,not waiting (IOBMATCH) */
/*          turn off dc for any loads (IOB or TC) to same word          */
/*----------------------------------------------------------------*/
/* LOAD => lookup for a matching store/pending-update to get data from  */
/*            or a pending read with this word present (IOBHIT)         */
/*         lookup for a pending read for same cache line and data not   */
/*            arrived or in window (add your register dst (IOBMATCH))   */
/*         if Tcld requests same cacheline, merge (B2BLD)               */
/*         if load to same cacheline & cannot merge, request only       */
/*            single word (TCFS)                                        */
/*         if store to same cacheline, request only single word (TCFS)  */
/*          hzrd if match a sub-word write (LDHZRD_FLAG)                */
/*----------------------------------------------------------------*/
/* DC_UPDATE => lookup for any pending update lines to the same cache   */
/*            line and turn on/off .tm bit (tag matches now)            */
/*----------------------------------------------------------------*/
/*----------------------------------------------------------------*/
```

```
/* Operations performed on IOBs by LOAD/STORE/DC_UPDATES during TC stg  */
/*---------------------------------------------------------------------*/
/* STORE => If Iobmatch and location marked by Tcloc is not already     */
/*            waiting for IO, or granted this cycle, collapse store     */
/*          else allocate new buffer (Bfree)                            */
/*---------------------------------------------------------------------*/
/* LOAD => If Iobhit or dchit, then load does not get queued into IOB   */
/*         else if Iobmatch and data not just arriving on IO bus, add   */
/*            register dest to requested line                           */
/*         else if B2bld, collapse into the buffer marked by Lbfree     */
/*          else allocate new buffer and request entire line if Tcfs==1 */
/*---------------------------------------------------------------------*/

if (CLK2) {
  /* Store: if the state of the Tcloc buffer has not changed (granted)      */
  /*      then do not allocate new buffer (collapse), use Tcloc buffer      */
  collapse = Tcst && Iobmatch && !((Tcloc==Bioreq)
&& iobgrant && !Iobuf[Bioreq].pr);
  loc = (Iobmatch && !(Tcst && (Tcloc==Bioreq) && iobgrant
                                     && !Iobuf[Bioreq].pr)) ? Tcloc :
                                                      B2bld ? Lbfree
                                                            : Bfree;
/*-----------------------------------------------------*/
/* LOGIC for each STATE BIT in each buffer in IOB      */
/*-----------------------------------------------------*/
/* contains all DC stage, TC stage, IO stage changes   */
/* except the hzrd flags which must be in phase 1      */
/*-----------------------------------------------------*/
store = -1;
tcfs = 1;

for(i=0; i < IOBSIZE; i++) {
/* DC STAGE ASSOCIATIVE LOOKUP (Iobhit, Iobmatch, Tcfs, Tcu)      */

  if(Dcst && eq_tag && eq_word && Iobuf[i].pw &&
    !( !Iobuf[i].pr && ((i==Bioreq) && iobgrant)) )  {
     /* ST: stores only COLLAPSE with stores      */
        if (tcstline == -1)          /* (Iobmatch, Tcloc)                */
            tcstline = i;           /* to preserve sanity in fetch-line world */
        else error('w', "write buffer (st) multiple match\n");
  }
  if ((Dcld || Dcst) && eq_tag) {     /* LD/SB/SH: IOB data hit (iobhit)      */
                                      /* ST subword uses data for merging     */
     if ((eq_word && Iobuf[i].pw) ||           /* LD: matched a store word to loc  */
       (update_word && word_arrived)) {        /* LD: matched request to that word */
             if (store == -1) store = i;       /*     that has already arrived     */
             else error('w', "write buffer (ld) multiple match\n");
     }
  }
  tcfs &= !(eq_index && (Iobuf[i].pw || Iobuf[i].pu || Iobuf[i].pr));

/* LD: cannot fetch line if .PW to some other word in         */
/* same cacheline OR same cacheline already requested         */
/* OR data cache update scheduled from some word in same line */
/* PENDING WRITE BITS (4, 1 per byte) */
/* If store word OR store sub-word that hits DC word OR full IOB word, set .pw = 0xf */
/* else if store sub-word collapsing into IOB, set appropriate bits and or in old .pw*/
/* else if store sub-word is allocating new buffer, set appropriate bits           */
/* NOTE: in case of conflict between Tc ST collapsing and IO read returning, Tcst    */
/*      STALLS thereby preventing conflict                                           */

  iobuf[i].pw = (reset1 ||
        ((i == Bioreq) && iobgrant && !Iobuf[i].pr && (Iobuf[i].pw == 0xf))) ? INVALID :
        ((i == loc) && new_store && (Tcsz != WORD) &&
        (collapse || (!dchit && !Iobhit)))     ? (vbytes(Tcsz, Tca) | Iobuf[i].pw) :
                (((i == Bwait) && Biodone &&  Iobuf[i].pr && Iobuf[i].pw) ||
                                                /* merge incoming word   */
                 ((i == loc) && new_store && ((Tcsz == WORD) || dchit
                                               /* full word/dchit       */
                                  || (Iobhit && !collapse)))) ? 0xf
                                                               : Iobuf[i].pw;
/* PENDING READ BIT (1 per iob)      */
  iobuf[i].pr = !reset1 && (Iobuf[i].pr
              ? !((i == Bwait) && Iobuf[i].pr && Biodone && Lastio)
              /* IO: iob's last dtack      */
              : ( ((i == loc) && new_store && (Tcsz != WORD) && !dchit && !Iobhit) ||
              /* ST: new partial st      */
              ((i == loc) && new_load && !same_iob) ));
              /* LD: alloc iob       */

/* PENDING DC_UPDATE BITS (1 per word per cache line)      */
  new_loadwords = ( ((1 << DCBLK) - 1)
                  & ~((Dcst && tc_dc_sametag) ? (1 << DCW_INDEX(Dca))
                                              : 0) );
  /* turn off.pu bit if Dc store to same word*/
  new_word = (!(Dcst && ((Tca & -4) == (Dca & -4)))) << (DCW_INDEX(Tca));
  /* don't turn on .pu bit if DCst to same word as TCa      */

  if (ST_OPT == SBUF) {
  iobuf[i].pu = (reset1 || ((i == Bdcreq) && bg_dc_write &&   /* DC_U: Free iob          */
                                                              /* DC update next cycle    */
                   !((i==loc) && new_store))) ? INVALID : /* unless a store collapse*/
              (Dcst && eq_tag && Iobuf[i].pu)   ? (Iobuf[i].pu & ~(1 << DCW_INDEX(Dca)))
                                                              /* clear stale data pu      */
              ((i == loc) && ((new_load && !same_iob && !Tcfs && Tcu) ||
                          new_store))         ? new_word :
              ((i == loc) && new_load && !same_iob && Tcfs) ? new_loadwords
                                                : Iobuf[i].pu;

/* For OTHER STORE METHODS, do not set .pu bit on store instructions     */
/* also use the old method of turning off .pu bits so that tagfix and    */
/* other store operations maintain cache tag  coherency, otherwise it    */
/* will be necessary to update .tm bits during a tagfix/store_write phase*/
   } else {
     iobuf[i].pu = (reset1 ||
            ((i == Bdcreq) && bg_dc_write) ||
                            /* DC_U: Free iob, DC update next cycle        */
            ((Dcst || Dcld) && !eq_tag && eq_index && Iobuf[i].pu)) ? INVALID :
                            /* LD/ST: follow most recent update            */
            (Dcst && eq_tag && Iobuf[i].pu)? (Iobuf[i].pu & ~(1 << DCW_INDEX(Dca))) :
                            /* clear stale pu      */
            ((i == loc) && (new_load && !same_iob && !Tcfs && Tcu) &&
                            /* LD: single word      */
            !((Dcst || Dcld) && !tc_dc_sametag && tc_dc_sameindex)) ? new_word :
                            /* invalidate if indices match but not tags      */
            ((i == loc) && new_load && !same_iob && Tcfs &&
             !((Dcst || Dcld) && !tc_dc_sametag && tc_dc_sameindex)) ? new_loadwords
                            /* LD: schedule write to DC for line      */
                            : Iobuf[i].pu;
  }

/* PENDING TAGMATCH BIT (1 per iob)                  */
/* tm bit signifies if tag needs to be written and if*/
/*   other words in line need to be invalidated      */

  if (ST_OPT == SBUF) {
     iobuf[i].tm = !reset1 && (((i == loc) &&
            new_store || (new_load && !same_iob))) ? ((Bg_dc_write && tc_dc_sameindex)
                                                   ? tc_dc_sametag
                                                   : dctagmatch)      :
          (Bg_dc_write && Iobuf[i].pu && eq_index) ? eq_tag
           /* DC_U: update the state of conflicting lines */
                                        : Iobuf[i].tm );
   } else {
     iobuf[i].tm = !reset1 && ( ((i == loc) && new_load && !same_iob)  ?
                          ((Bg_dc_write && tc_dc_sameindex) ? tc_dc_sametag
                                                   : dctagmatch)      :
                  (Bg_dc_write && Iobuf[i].pu && eq_index) ? eq_tag
                                                   : Iobuf[i].tm );
   }

/* PENDING DC_UPDATE BITS (1 per word per cache line)           */
/* assume fetch strategy is only checked with a pending read    */
/* therefore, set or clear only when iob is allocated           */
  iobuf[i].fs = ((i == loc) && new_store)            ? 0 :
                /* ST: store partial word         */
               ((i == loc) && new_load && !same_iob)     ? Tcfs
                /* LD: request full line depending on Tcfs      */
                                                     : Iobuf[i].fs;
/* ADDRESS (32 bits)                    */
/* address always comes from Tca        */
   iobuf[i].a = ((i == loc) &&
     (new_store || (new_load && !same_iob))) ? ((Tcsz == WORD) ? (Tca & -4)
                                                        : Tca)
                                         : Iobuf[i].a;
/* DATA, DST AND REGW ARRAYS (1 per word in cacheline)  */
   for(j=0; j < DCBLK; j++) {
          iobuf[i].data[j] = Iobuf[i].data[j];
          iobuf[i].regw[j] = Iobuf[i].regw[j];
   }
   k = DCW_INDEX(Dioaddr);
   if ((i == Bwait) && Biodone && Iobuf[i].pr)
        iobuf[i].data[k] = mergeb(Dtin,Iobuf[i].data[k],Iobuf[i].pw);
                                                          /* LD, partial ST */
   j = DCW_INDEX(Tca);                                   /* writing from TC stage */
   if( (i == loc) && new_store )
       iobuf[i].data[j] = iobdin;                         /* ST, take data from TC stage */
       iobuf[i].dst[j] = ((i == loc) && new_load &&
                          (!same_iob || !data_arriving)) ? Tcdst
                                                          /* LD: add dst if hit           */
                                                          : Iobuf[i].dst[j];

   }
   if( (i == loc) && new_load && (!same_iob || !data_arriving) ) {
            iobuf[i].regw[j] = 1;        /* set regw if new valid load in TC stage  */
   }
   if( (i == Bwait) && Biodone && Iobuf[i].pr )
            iobuf[i].regw[k] = 0;        /* clear regw if IO completes for this IOB */
   }


/*----------------------------------------------*/
/* results of ASSOCIATIVE LOOKUP in DC stage    */
/*----------------------------------------------*/
iobhit = ((Dcld || Dcst) && (store != -1));
      /* Iobhit => LD found data location       */
      /* SB/SH => data from IOB for merging     */

tcfs &= !(tc_dc_sametag && (Tcst || Tcld));     */
/*----------------------------------------------*/

/* INCREMENT POINTERS      */
/*----------------------*/
/* Bioreq: if valid io request AND grant received */
/* OR granted arrived in past OR not valid anymore*/
/* => choose new location to request             */

bioreq = (iobgrant || (Bioreq == Bwait) || (Bioreq == Blag)
      || !(Iobuf[Bioreq].pw || Iobuf[Bioreq].pr)) ? next_ioreq : Bioreq

/* Bdcreq: if bg_dc_write OR not valid anymore    */

bdcreq = (bg_dc_write || !Iobuf[Bdcreq].pu) ? next_dcreq
                                            : Bdcreq;
/* Bfree: if allocated a new buffer to TC LD/ST request, or if NOT empty */
bfree = ((new_load && !Iobmatch) || (new_store && !collapse)
      || Iobuf[Bfree].pu || Iobuf[Bfree].pw || Iobuf[Bfree].pr) ? next_free
                                                                : Bfree;

}
```

Bibliography

[1] Agarwal, Ph.D thesis, 1987, Stanford: Analysis of Cache Performance for Operating systems and Multiprogramming.

[2] Aho, Sethi, Ullman, Compilers: Principles, Techniques and Tools, 1988.

[3] D. W. Clark, ACM Transactions on Computer Systems 1, 1983: Cache Performance in the VAX 11/780.

[4] Fu, Keller, Haduch, Digital Technical Journal 1,6, 1987: Aspects of the VAX 8800 C Box Design.

[5] K. Gharachorloo, A. Gupta, J. Hennessey, , ASPLOS 4, April 8-11, 1991: Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors.

[6] Hardell et al, IBM RS/6000 Technology manual, 1990: Data Cache and Storage Control Units.

[7] Hennessey and Patterson, Computer Architecture: A Quantitative Approach, 1990.

[8] M. D. Hill, Ph.D thesis, 1987, Berkeley: Aspects of Cache Memory and Instruction Buffer Performance.

[9] M.D Hill, A.J Smith, IEEE Transactions on Computers, vol 38, no 12, Dec 1989: Evaluting Associativity in CPU caches.

[10] M.D.Hill,IEEE Computer, Dec 1988: A Case for Direct-Mapped Caches.

[11] N.P.Jouppi, ISCA 20, 1993: Cache Write Policies and Performance.

[12] N.P.Jouppi, ISCA 17, 1990: Improving Direct-Mapped Caches by the Addition of a Small Fully Associative Cache and Prefetch Buffers.

[13] MIPS R4000 Instructions Manual, 1992.

[14] K. Olukotun, T. Mudge, R. Brown, ISCA 19 proceedings, May 1992: Performance Optimizations of Pipelined Primary Caches.

[15] S.A. Przybylski, Cache Design: A Performance-Directed Approach, 1990.

[16] A. J. Smith, Journal of ACM 26, 1979: Characterizing the Storage Process and its Effect on the Update of Main Memory by Write-Through.