Micropipelined Cache Design Strategies for an Asynchronous Microprocessor

A thesis submitted to the University of

Manchester for the degree of Master of Science

in the Faculty of Science.

by

Rahul Mehra

Department of Computer Science

October 1992

Contents

List of Figures

List of Tables

Abstract

The design of modern, pipelined, VLSI microprocessors is based almost entirely within a synchronous framework. Clock distribution in such circuits suffers from many problems. A large portion of the design effort must be devoted to overcoming these problems when fabricating synchronous designs. The throughput of synchronous pipelines is determined by the time taken for the slowest operation in any stage since this determines the maximum global clock frequency.

An asynchronous methodology may provide solutions to these short-comings by not requiring the distribution of a global clock signal. Instead circuits are constructed from small, *self-timed* sub-circuits within which temporal dependencies are maintained. Larger, pipelined circuits are constructed by joining these sub-components together using a simple communication protocol. This arrangement allows asynchronous pipelines to have a flexible depth and a data dependent throughput.

A block level simulator has been written that models microprocessor caches developed using a particular asynchronous methodology − *micropipelines*. This thesis describes the development of such caches. It notes the effects of varying various cache parameters and using different strategies whilst attempting to optimise performance for a specific architecture. The target architecture is a micropipelined version of the ARM microprocessor which is currently being developed within the AMULET research group.

DECLARATION

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification at this or any other university or other institute of learning.

Acknowledgements

At this time I would like to express my sincere thanks to; Jim Garside who endeavoured to keep me on the tortuous path that is known as research, to Steve Furber for the opportunity to be involved in such ground breaking work, to Nigel and Paul for putting me in my place, to Paul Garnett who turned out to be a veritable treasure chest of useful information and everybody else in the AMULET group.

About the Author

Rahul Mehra graduated from the University of Manchester in 1991 with an honours degree in Computing and Information Systems having originally been undertaking a joint honours degree in Mathematics and Computer Science and then making the transition in order to gain more experience of computer architecture issues. His final year project involved writing a program to investigate the effectiveness of neural networks at recognising hand printed characters.

To mum and dad. Cheers dad − for everything.

Chapter 1

Introduction

The increasing penetration of microprocessor controlled devices into consumer markets has resulted in demands for devices that exhibit many properties. If products are to meet consumer expectations they must have complex and fast user interfaces which frequently need high performance processors to implement them. The consumer also demands that products be low cost, small and convenient. This should be reflected in the price and size of a product's sub-components. Small lead times are essential if manufacturers are to be able to bring products to markets in a minimum amount of time. Small size and portable convenience also imply that the processors at the heart of these consumer devices should have low power consumption, thus being able to be battery powered. When directed at a single component, many of these requirements are conflicting in nature, for example high performance at low cost is often contradictory. This has resulted in new market openings for devices that maximise these tradeoffs.

Traditional microprocessors are synchronous in nature. They rely on the presence of a global clock to coordinate their internal and external operations. To improve performance higher clock speeds are employed and also architectural features such as pipelining. Pipelining exploits *parallelism* by having many operations executing concurrently thus increasing data throughput. However increasing the clock rate introduces problems such as clock skew, increased electromagnetic radiation, greater demands on the power supply and so forth.

As chip fabrication techniques have improved there has been a tendency to incorporate a greater number of architectural components onto a single silicon die. This has resulted in increasing the difficulties involved in distributing a single clock signal correctly on such a chip. The stage has now been reached where a significant percentage of the design effort has to be spent on designing the clock distribution scheme to overcome problems such as clock skew and ground bounce. Existing synchronous processors cannot be scaled to gain benefits from improved fabrication techniques because clock distribution is not scalable and must be carefully redesigned each time.

Increases in the clock rate and higher degrees of integration in a synchronous design also carry the penalty of greater power consumption since clock distribution is responsible for large amounts power usage within a design. On every clock cycle a significant (and ever increasing) number of transistors concerned with clocking change state and as they do so power is dissipated. This occurs even if a individual synchronous sub-component need not do any processing in this cycle.

This has led to a recent resurgence in interest in asynchronous circuit design.

Circuits designed in this fashion do not require global clocking; instead they rely on internally generated timing and use a communication protocol in order to exchange data with other stages and are often − and perhaps more correctly − referred to as *self-timed circuits*. By avoiding the use of global clocks problems such as clock skew and ground bounce are avoided and silicon area maybe saved since the signal no longer has to be distributed over a large area. Removal of circuitry concerned with clock driven control also saves power in asynchronous sub-circuit elements whenever they are not required to perform a computation since operation is now only dependent on input data and not regular clock events.

Design within an asynchronous framework can be more modular than in a (purely) synchronous scheme. Asynchronous sub-circuits can be combined to provide the same architectural features as found in synchronous processors. Most notably asynchronous pipelines can be constructed in which the throughput is determined by the data flowing through the pipeline; where simple processes are quick whilst complex ones take longer. Such pipelines generally exhibit an average case performance whereas synchronous pipelines must allow for the worst case and therefore have their global clock adjusted accordingly.

Although individual sub-circuits can be harder to design than their synchronous counterparts, the use of a simple communication protocol allows a more extensible design method to be used by actively encouraging simple piece-wise development of circuits.

The AMULET group at the University of Manchester is undertaking to investigate the applicability of Sutherland's asynchronous design methodology. At the 1989 Turing Award Lecture Ivan Sutherland presented an asynchronous design technique known as *micropipelines* [SUTHERLAND89]. Micropipelines employ an event based bundled request/acknowledge protocol for communication between self-timed sub-components. One of the ESPRIT funded projects OMI/MAP (Open Microprocessor Initiative/Microprocessor Architecture Project) is aiming to implement an existing synchronous RISC design using micropipelines. As it transforms a successful synchronous processor it aims to address commercial considerations such as the applicability of CAD tools, design time, ease of design, performance of finished product etc. This can best be done by comparing two compatible versions of the same chip; one synchronous and the other asynchronous.

A large degree of the flexibility of an asynchronous micropipelined processor would be lost if it were to use a standard synchronous memory interface. This coupled with the discrepancy between the access speeds of large, low cost, high density, main (dynamic) memory and the speed at which the processor could operate would nullify many of its inherent advantages. Enhanced performance can be achieved if the processor were to employ an asynchronous memory interface.

The asynchronous ARM processor that is being developed, like other RISCs, requires the memory sub-system to have a high bandwidth and low access latency.

Modern DRAM does not meet these requirements. One possible solution used to alleviate this problem is to employ a *cache memory*. Caches are small, high speed memories that are located between main memory and the CPU in order to lessen the effect of slow memory access(es) on processor throughput. This is achieved by holding a copy of the most recently used sub-set of main memory in a small fast store. Caches are an established technology and have become a common architectural feature. The cache design for a particular system is very much a engineering task with the simulation of various designs and evaluation of their relative performances. Synchronous caches are well understood and evaluation and comparison techniques to aid their development are well known.

The synchronous ARM3 variant of the ARM processor has an on-chip cache in order to improve its performance. As a continuation project from OMI/MAP, OMI/DE (Deeply Embedded) seeks to effectively extend the micropipeline technique into the memory sub-system by the inclusion of an on-chip asynchronous cache. Use of classic synchronous cache design techniques is not well suited to the development of such a cache. This thesis describes the development of asynchronous cache designs and explores methods by which different designs can be evaluated and their relative performance compared. It describes how an event-based simulation may be used to model the behaviour of a micropipelined cache, and how simulation of a cache system can aid system designers in the development of new architectures by allowing them to make or test predictions about system behaviour and performance. The simulator provides a cheap and easy means for evaluating effects of different strategies and allows structures to be optimised for efficiency.

Chapter two gives background information on pipelines, caches and the ARM processor. It describes the operation of traditional synchronous pipelines followed by an introduction to the different asynchronous pipelining techniques in particular micropipelines. The chapter also describes the nomenclature, structure and operation of synchronous caches. The section dealing with the ARM processor describes salient features of both the synchronous and asynchronous versions and how they affect possible caches designs. It also describes the operation and reasoning behind the structuring of the synchronous ARM3 cache.

Chapter three starts with an overview of how the performance of a synchronous cache can be evaluated, explaining why the methods employed need not be applicable to micropipelined implementations. It then discusses how performance may be evaluated in the asynchronous framework. The second section of the chapter explains the design of the event based cache simulator and how the memory reference traces that it uses were collected.

In chapter four specific issues in cache design are discussed in more depth. It introduces the characteristics of ARM code programs and discusses the tradeoffs that are normally considered when designing caches. Chapter five presents prototype micropipelined cache designs explaining the evolutionary steps involved. Using a particular design the usefulness of various strategies is

investigated. This chapter also addresses possible changes to cache structure that may be beneficial. Finally chapter six provides conclusions about the simulation and evaluation methods. It also outlines the recommended cache structures based on the results obtained in chapter five and describes future work that can be done.

Chapter 2

Background

2.1 Pipeline Design

In a quest of higher performance in computer systems many techniques of exploiting parallelism have been exploited. One such technique is *pipelining*.

2.1.1 Synchronous Design

Modern general purpose (micro)processors have typically been synchronous in design. Much time and effort has been devoted in exploiting the features of this methodology in order to increase their performance. One commonly used technique used is a pipeline.

The operation of a pipeline is analogous to a production line. It aims to increase the rate at which results appear (the *throughput*), but it does not decrease the time taken for an individual result to emerge (the *latency*). Indeed the latency may be higher since there is now more control associated with the flow of data through the pipeline.

In a synchronous processor the system is partitioned into separate functional units each of which performs a different task; thus an operation is performed as data moves through each stage and is manipulated/acted upon. At any one time different stages will be operating concurrently on different data as it passes through the processor. Figure 2.1 shows a three stage pipeline whose operation is governed by a global control circuit and clock signals. Concurrent operation is illustrated by the letters "**A**", "**B**" and "**C**". During the first cycle stage one operates on **A** which gets passed on at the start of the second cycle to stage two. This leaves the stage one free to start processing **B**. By the third cycle all three stages are operating concurrently on different data.

In a synchronous design the control of the units and the movement of data between them occurs in 'lockstep' and is governed by the globally distributed clock. A stage receives its data and associated control signals from the preceding stage at the start of a clock period and must process it before the next clock, at which point it passes the data onto the next stage. The maximum clock frequency is determined by the time taken for the slowest pipeline stage to complete processing and this imposes an upper limit on the throughput of the pipeline. The latency of the pipeline is given by the product of the clock frequency and the number of stages. The throughput is equivalent to the clock frequency.

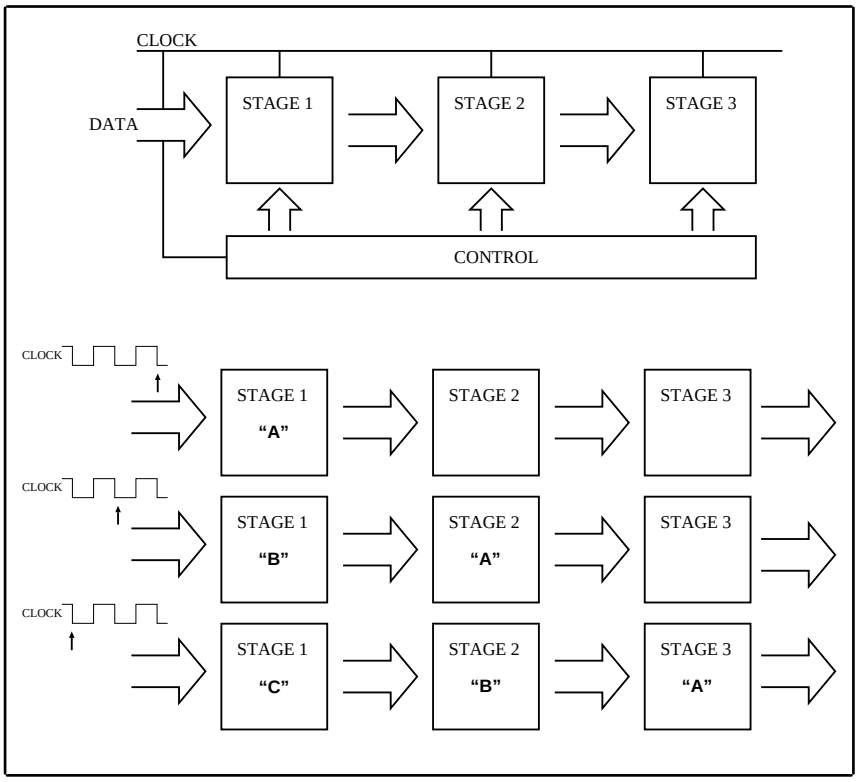

Figure 2.1 Operation in a synchronous pipeline.

To maximise system performance the system should be partitioned into units which take an equal amount of time for processing. To (potentially) further improve the throughput of a processor the number of pipeline stages can be increased by further subdividing the existing stages, although doing so can pose a problem as it may significantly increase the amount and the complexity of control required for certain functions. Another solution is to devote extra logic (and normally therefore more silicon space) to improving the worst case performance of an individual pipeline stage. Unfortunately both these solutions increase the size of a chip, dissipate more power and exacerbate a fundamental problem in synchronous design, that of clock distribution.

As both the chip area and the clock rate increase, the problem of how to distribute a single, consistent clock over the entire chip becomes significant [SEITZ80]. A clock signal goes to all parts of a chip and controls multiple circuits, as the clock cycle times get faster and the chip gets bigger the distance of a particular stage from the clock source becomes more significant. This occurs because the inherent resistivity and high capacitive loading on clock lines limit the speed at which voltage changes are propagated through the length of the wire. This causes the rising and falling edges of a clock pulse to arrive a different times at different locations on the chip − this is known as *clock skew*.

Clock skew can cause stages to become out of step because communication between stages relies on a globally accurate point of reference − the clock edge − at which to synchronise and exchange data. If this reference point differs across

the circuit it is possible for one stage to pass on data before the next stage is ready to receive it because it has yet to 'see' its clock edge. This can cause data to be lost which must not be allowed to occur. Hence significant effort is devoted in ensuring that clock signals are consistent in a synchronous design.

These problems with clocking can be controlled by using a *clock tree*. Initially the clock signal is taken from the point of origin, split up and distributed to clock driver trees by lines of close to equal length. Clock trees are made of transistors with similar charcteristics and are of the equal depth. The type of transistors, the depth of the trees and the length of the clock lines must all be carefully crafted for each new design and can take a large about of design effort. For example, there are continuous developments in process technologies which allow feature sizes to be shrunk. Older chips could then be scaled onto these newer technologies in order to exploit the advanatages that they offer. Unfortunately the problems of clock distribution are not scalable (outside certain limits) so extra effort is required.

A particular example of the clock distribution problem is the DEC Alpha processor [DEC92a] and [DEC92b], which employs a very large primary clock driving transistor. In total, 30% of the chip area and one entire metalisation layer (shared with power distribution) is used to distribute the clock signal. The chip also exhibits a average case power consumption of 30Watts, 17Watts of which is accounted for by the clocking circuit.

As illustrated clock distribution is also responsible for a significant amount of the power consumption within a processor both directly and indirectly: the clock lines are large capacitive loads which require large transistors to drive them. Power consumption occurs indirectly due to circuits controlled by the clock, switching on every clock edge. CMOS transistors dissipate significant power only whilst switching. The repeated high frequency switching inherent in any clocking scheme thus dissipates a significant amount of power. Even during an inactive clock period a synchronously pipelined stage dissipates power as the transistors in its control circuitry undergo clock driven transitions.

Clocked circuits are also prone to problems of *current spiking* and *ground bounce*. These occur as a clock edge causes large numbers of transistors to switch at the same time. As they switch some transistors discharge current into ground lines and some draw current from supply lines. Ground and supply lines provide points of voltage reference for the transistors on the chip. Ground bounce occurs as switching transistors dump current into the ground rail and temporarily raises its absolute level. Similarly current spikes occur when large amounts of current are drawn from the supply rail which is unable to meet the instantaneous high demand and so drops its voltage. These problems have to be addressed at the transistor level, ensuring that not too much current is drawn/dumped in too short a period of time whilst still allowing time for circuits to settle before the end of the clock period.

2.1.2 Asynchronous Design and Micropipelines

Asynchronous design methods attempt to avoid the problems associated with synchronous circuits by not using global clocks to control and synchronise the exchange of data. Instead they rely on pipeline stages operating independently and autonomously using a communication protocol in order to temporarily synchronise for the exchange of data. In this framework circuits can be constructed from smaller circuits within which temporal dependencies are maintained internally. At the interface of the individual units a protocol that provides temporary synchronisation for data exchange must be implemented and observed. This makes the design process more modular since individual stages are independent of each other (other than in the control signals that they exchange) and can be designed and tested in isolation.

Asynchronous pipelines exhibit characteristics not seen in synchronous systems. For example component stages can be organised into pipelines that exhibit a data dependent rate of processing; complicated operations may take a long time to pass from input to output whilst simple ones pass through more quickly. This may be done using a fixed pipeline with individual stages allowed to process in variable amounts of time since they are no longer obliged to produce a valid result within a given time span. Pipelines may also have forks and joins that allow the data to be directed so that it only passes through those operations that are required.

Asynchronous pipelines exhibit an *elasticity* of depth that is not apparent in synchronous ones; there is no requirement to enforce a strict one-in one-out correspondence between input and output. In an asynchronous pipeline any stage is free to generate and *inject* an extra data value into the stream. Similarly any stage can remove a data item from the stream by accepting it but not producing a corresponding item on its output. These may be useful for dividing complex tasks into simpler steps and then recombining them into a single result, or if − for example − a conditional operation is not to be executed it may be eliminated from within the processing pipeline.

The development effort in an asynchronous framework is directed at the design of individual stages and a simple, low overhead communication protocol to facilitate data exchange. In contrast with synchronous pipelines (which use the supplied global clock for all timing), asynchronous pipeline elements requiring timing information for processing must generate it internally. The design of such elements can be classified into three categories [GOPAL90];

- **Delay Insensitive.** No assumptions are made about either gate or interconnection delays.
- **Speed Independent.** Gates may have arbitrary propagation delays but interconnections are instantaneous.
- **Bounded Delay.** Both gates and interconnections have finite delays.

Once pipeline elements adhering to one of these models have been generated they can then be combined to create asynchronous pipelines. Synchronisation between these these elements (required to pass data from one to the next) requires the use of *handshake signals*. Figures 2.2 and 2.3 illustrate the two alternative types of signalling conventions, known as *four-phase* and *two-phase*;

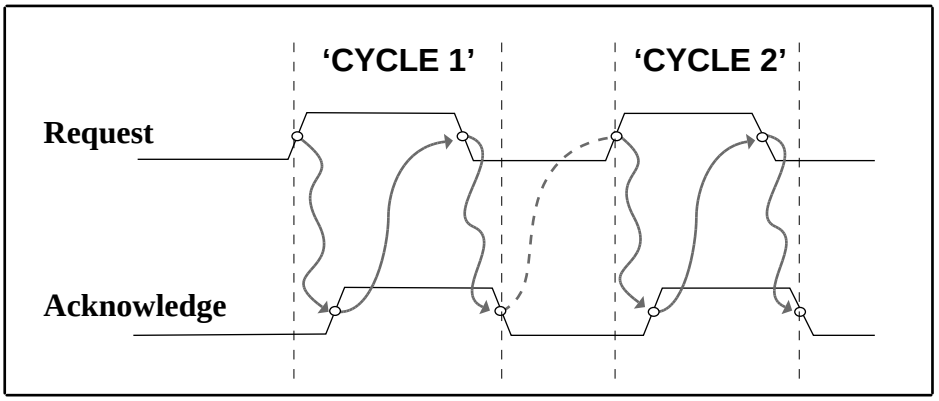

Figure 2.2 Four−Phase Signalling.

Four-phase signalling is made-up of four distinct stages; first the sender takes the request line active indicating that it wishes to transfer data. Then the receiver takes its **acknowledge** line active thus synchronising and indicating that it has received the data. At this stage the sender is free to let the **request** line become inactive after which the receiver is also free to let the **acknowledge** return to its in active state. Data transfer occurs only during the first two phases the latter two are used just to allow the signal lines to return to their original logic level.

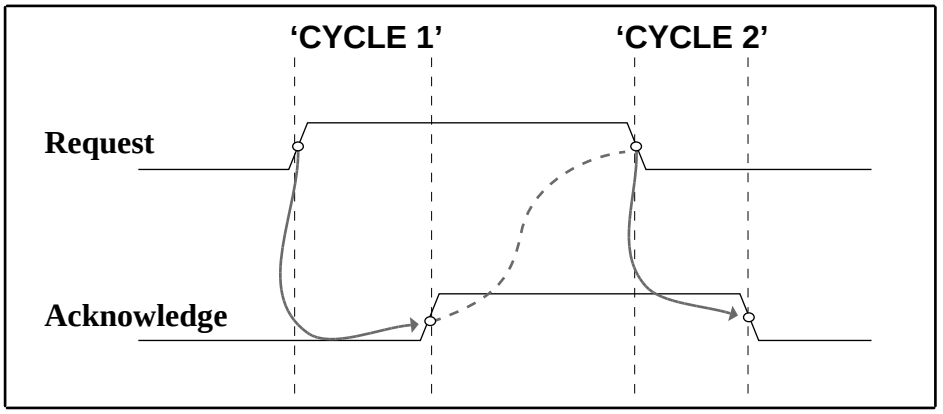

Figure 2.3 Two−Phase Signalling.

The two-phase signalling scheme employs only two phases of operation; *request* and *acknowledge*. It does not have the overhead of extra phases to return the signals to their original levels. Instead transitions, either rising or falling edges on a signal line, are used to indicate events. This gives alternative names for two-phase signalling of *transition* or *event* based signalling. When the sender has data ready to send, it causes a transition on its **request** line. Once the receiver has received the data it acknowledges this by causing an event on the **acknowledge** line. The sender is now free to initiate another data transfer. It is important to note

that in this protocol there is no logical difference between a rising and falling transition.

When designing any asynchronous circuit in CMOS technology, transition signalling offers some advantages over four-phase signalling; there is potential for a higher speed of operation and lower power consumption by avoiding the needless transitions as control signals return to their original states. Care must be exercised however, since circuits that use events to control processing are harder to design than ones in which a particular logic level triggers processing.

These signalling protocols can either be used in a *bundled* or *unbundled* fashion. A bundled communication protocol packages any data and control signals together with two extra request and acknowledge lines. The transmission of data can then be governed by normal use of either two or four phase signalling on the control lines. The bundled data convention requires certain ‘set-up’ and ‘hold’ timing constraints to be met by the data bundle and is thus not suitable for delay-insensitive circuits.

To transmit asynchronous data in an unbundled fashion *dual-rail encoding* is usually used. In this scheme each data bit is represented by two wires; a *0 line* and a *1 line*. The value of a given bit during a particular transfer is then decided by a single ‘event’ on one of the two lines. In this case an ‘event’ may refer to either a two or four phase transition scheme. Delay insensitive circuits may be constructed in this fashion; the connection between one stage an the next being facilitated by using dual-rail encoding for the data/control bits and a single additional signal wire for the returned acknowledge. Thus during a transfer the receiver waits until its sees an event on each of the data wires. Then it generates an event on its acknowledge wire. At this point the communication has completed in a two-phase scheme whereas in the four-phase approach the logic levels for each of the bits must return to its orginal state at which point so does the acknowledge.

Using combinations of such techniques different forms of asynchronous circuits have been constructed. Martin et al [MARTIN85–89] have used four-phase, dual-rail encoding and synthesis techniques to design and implement an speed independent asynchronous microprocessor. van Berkel et al [BERKEL88] also use formal techniques to construct four-phase, dual-rail, delay-insensitive circuits. Brunvand [BRUNVAND89] takes a more pragmatic approach and uses an event based bundled communication protocol when implementing pseudo-delay-insensitivity. Most of this work has taken place in the CMOS design arena although Brunvand has also investigated the applicability of asynchronous techniques in other frameworks [BRUNVAND92].

An event based protocol was used by Sutherland when describing asynchronous *micropipelines* [SUTHERLAND89]. Summarised in figure 2.4, micropipelines are bounded-delay circuits in which synchronisation between stages is facilitated by a bundled data, event based, protocol. When the transmitter has stable information ready to send it generates an event on the request line. On seeing the **request** the

receiver is free to latch the data. Once latched an event on the **acknowledge** line indicates that the transmitter is free to remove the data value from its output. This scheme is bounded-delay since care must be taken that the request never over-takes its associated data since this may allow the receiver to latch incorrect data.

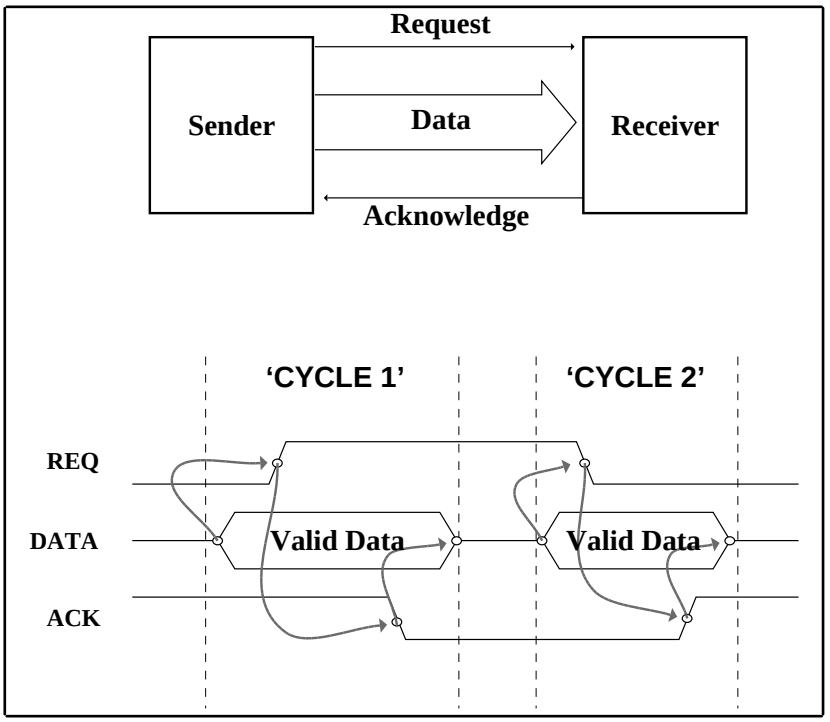

Figure 2.4 Micropipelines: Bundled Event Based Communication Protocol.

It is this is basic strategy that has been employed within AMULET group for the development of an asynchronous microprocessor. Although the logic elements required to implement and use such a transition based protocol are not as simple as level based ones, this method does provide some advantages; there is less redundancy than in four phase protocol, so it allows for higher speed and since CMOS circuits dissipate power whilst switching, the fewer transitions there are then the less power is consumed.

A micropipeline is an example of a particularly simple and elastic asynchronous pipeline. The operating frequency of an asynchronous pipeline is not strictly determined by the speed of the slowest stage (although in sustained operation it does adopt this throughput rate), but is based on 'average case' performance. These circuits themselves can then be combined to give a more hierarchical design methodology. As techniques mature and designs develop, just the micropipeline stages that are involved needed to be redesigned.

2.2 Caches

This project has been concerned with micropipelining cache designs. It is therefore appropriate to introduce structures of typical synchronous caches along

with their associated terminology. The following sections introduce the concepts employed when cacheing main memory. Following this the structures that are used and their nomenclature along with illustrating examples are discussed. Finally the issue of power consumption within caches is briefly touched upon.

2.2.1 Overview

In a cached computer, memory is organised into a hierarchy with the CPU at the upper-most level, main memory at the lower-most and one or more caches inbetween (see figure 2.5). The CPU performs two basic types of memory operation; reads and writes (reads can further be split into data reads and instruction fetches) which are ultimately satisfied by data held in main memory. Caches between the CPU and memory attempt to mitigate the effects of slow access times and long cycles times exhibited by the cache/memory immediately below them. They do this by holding copies of small areas of data from the layer below. When a request arrives at a particular level the cache attempts to satisfy it itself; if it cannot, it passes the request onto the next level and (optionally) takes some other action (e.g. alters its contents so that a similar request in the future can be satisfied). This thesis is primarily concerned with the design of a two-level memory hierarchy with just one level of cacheing so multi-level hierarchies will not be discussed further.

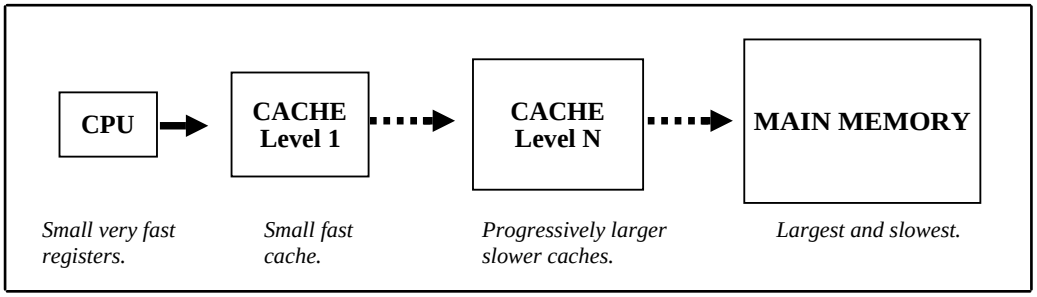

Figure 2.5 Memory Hierarchy.

Since cache memories are orders of magnitude smaller than main memory, an implied mapping must exist. A cache dynamically maintains a *working-set* of the memory locations that are currently in use. This allows it to satisfy the majority of memory requests quickly fom the cache, only occasionally referring to main memory for data. This is done by exploiting characteristics of programs such as *Spatial* and *Temporal* locality.

Spatial locality is exhibited when, for example sequential code is executed. If an instruction at a particular point has just been executed, then unless it caused change of program flow (a branch), then the next instruction will be fetched from the sequentially next location. Even if a branch does occur, it is probable that it will be to somewhere “fairly close” (e.g. small tight loops used to implement traversals of data structures). Spatial locality can also be noted during data references since data that is likely to be used at one time is normally stored together.

Temporal locality occurs when, over a given time, there exists a set of locations that are accessed repeatedly. For a given algorithm, there probably exists a subset of variables that are used repeatedly, whilst others are only accessed occasionally.

2.2.2 Terminology

Various structures have been developed that exploit both spatial and temporal localities of reference. The nomenclature of these structures has arisen at different times by many independent groups and therefore some inconsistency is apparent. One of the the first survey papers to collate and address this issue was [SMITH82]. As caches have developed and become more widespread, terms have come into common usage that are inconsistent or even contradictory. The nomenclature presented below is based on [PRYBYLSKI90] but incorporates terms that are found in common usage along with their older naming.

A *cache-line* or *block* is a area of cache. This has a *tag* associated with it indicating which portion of main memory is mapped into that line at the moment. A *set* is a collection of cache lines whose tags are checked in parallel. A contiguous group of bits from the address is usually used to determine in which set a block of main memory may lie.

The *degree of associativity* of a cache is determined by the number of cache-lines in a single set. Thus if there is only one set then all the address tags have to searched in parallel and the cache is said to be *fully associative*. If however there is only one cache line per set then the cache is said to be *direct mapped*. In a direct mapped cache only one tag that has to compared with an address of a memory request in order to determine whether the request can be satisfied from within the cache or whether it has to be passed on to main memory. Normally a real cache lies some where between these two extremes.

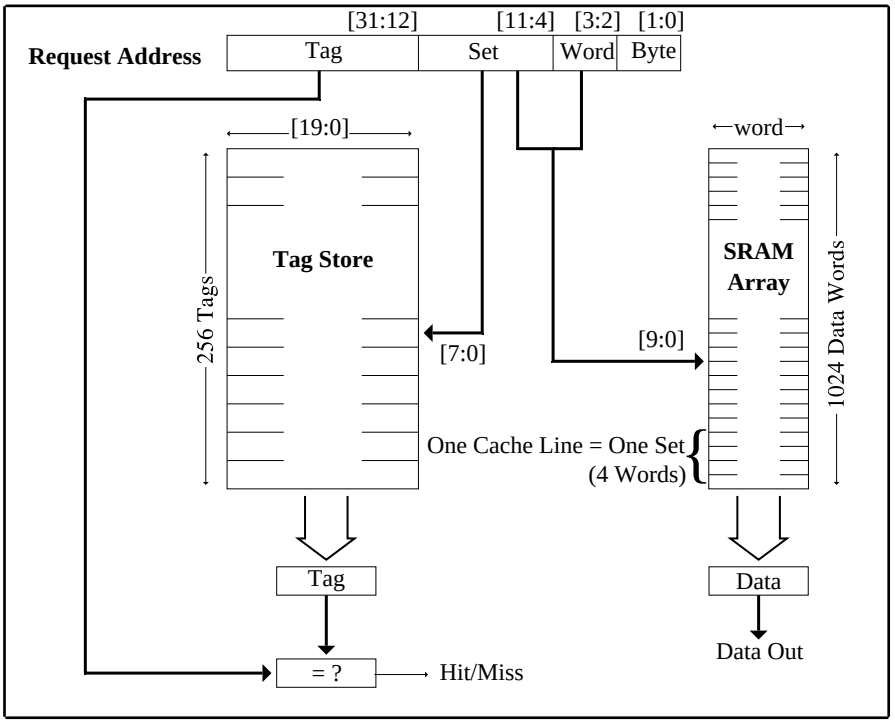

Figure 2.6 A 4kbyte Direct Mapped Cache.

As an example, figure 2.6 show a 4Kbyte direct mapped cache. It has four 32-bit words per cache line and this illustrated in the diagram by the four lines that are marked as being in the same set (direct mapped = one line per set). Address tags associated with each cache line are stored in the tag store − one for each of the 256 cache-lines.

Caches are also named by nature of the addresses the operate on. The memory addresses used for comparison can be either *real* or *virtual*, depending on the position of the memory management unit (MMU), which leads to the terms *real cache* and *virtual cache* (see figure 2.7). For further discussion see section 4.2.3.

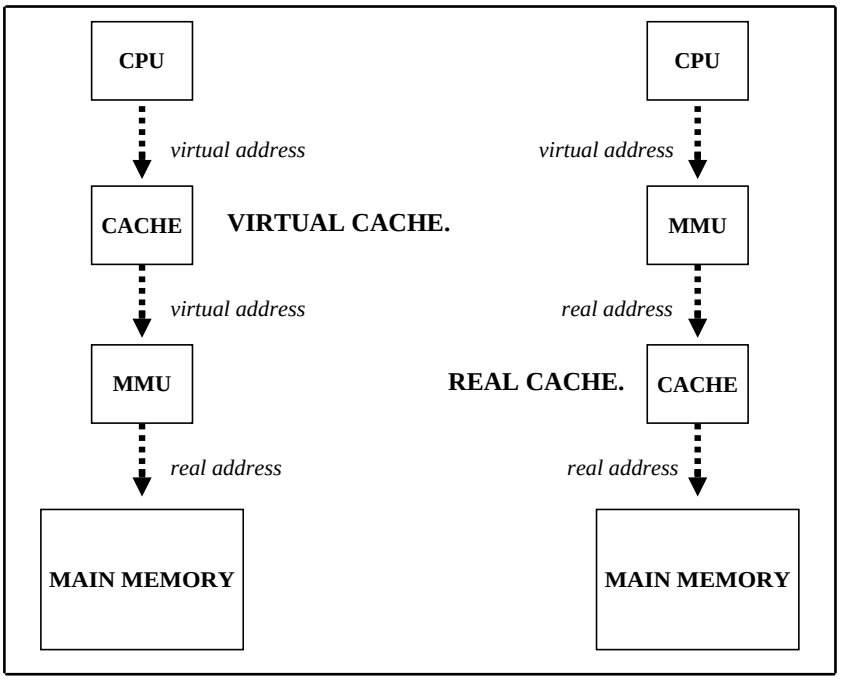

Figure 2.7 Virtual and Real Caches.

Fetch strategy is the algorithm deciding when to fetch data and what data to fetch from the next level in the memory hierarchy; it also determines which order operations are performed in within a level (e.g. start by getting the requested word or fetch all the new data first etc...).

The *fetch size* is the size of the unit of interaction with main memory and determines how many discrete operations it takes to fill a cache-line. If the fetch size is equal to the size of the cache-line then a whole cache-line either contains valid data or not. In certain cacheing schemes the fetch size is smaller than the cache-line and so the cache-line is further divided into *sub-blocks*. This requires additional information to be stored such as a *valid bit* (V) indicating if this sub-block has yet to be fetched into the cache. In addition a *dirty bit* (D) can be placed here to implement a write-back cache (see below) by indicating if the data in this sub-block has been altered.

Write strategy determines details of how writes are handled by a cache. There are two main variants: *write through* and *write back*. A write through cache passes all write requests onto the next level whilst also maintaining any copies of the data already held. A write back strategy initially alters only the cached values, copying the altered data back to main memory only when the line is about to be replaced.

Both write through and write back techniques can be augmented by the use of a *write buffer*. This buffers write operations before they are sent out to main memory and means that the cache/CPU does not have to wait for the whole operation to complete before continuing. Write buffers have different *depths* and *widths*; the depth is number of outstanding writes that can be buffered and the width is number of words written for each write operation (ie. the number of data

words associated with each address).

Another way that caches are classified is by the policies that they employ for cache line allocation. The *replacement strategy* governs which cache location is used to store a new block of memory once all available ones have been used. *Cyclic replacement* simply replaces the next line in sequence when a line fetch occurs, so if the last line to be loaded was *n* the next will be *(n+1)*. Upon reaching the last line in a set the cycle restarts with *n=0*. This is the simplest policy to implement (uses a simple counter) but it is not very effective since it fails to exploit any form of temporal locality. The line least recently allocated is ejected despite the fact that this may not necessarily be the 'best' one to eject. Some frequently used data may be resident in the cache for long periods of time, cyclic replacement would cause it to be repeatedly and regularly ejected needlessly from − and then reloaded back into − the cache.

The *least recently used* (LRU) replacement algorithm is closest to the concept of temporal locality. It replaces the cache entry that has been accessed least recently. This assumes that data accessed recently is likely to needed in the future and so should kept in the cache. Unfortunately it is hard to implement this algorithm efficiently in hardware for all but the simplest cache structures (low associativity). Therefore in cases when LRU cannot be implemented efficiently (e.g. highly associative caches), *random replacement* is often used. The line to be replaced is selected on a pseudo random basis. Random replacement is similar in performance to an LRU policy but offers a *graceful degradation* in performance which LRU does not. Consider the pathological case of a loop with access patterns which just over fills a cache employing LRU replacement. In these circumstances the LRU degenerates to a cyclic replacement. For loops smaller than the critical size the cache performs as expected with most reads being satisfied from within the cache. As the loop gets bigger then LRU 'breaks' suddenly as the line that is just about to be used gets replaced just before use, thus giving a sudden decrease in performance. This sudden decrease in performance is not as apparent in a cache with random replacement. Pathological cases are harder to construct since the replacement behaviour is not as deterministic − the performance degrades more slowly and consistently.

2.2.3 Generic Cache Structures

How a cache is implemented is very dependent on its structure especially its degree of associativity. Direct mapped caches and caches with low associativity tend to be used in situations where a small cache cycle time is required. This is because their implementation allows for concurrent accesses for both the tags and the cached data. In highly associative caches the tags are stored in a content addressable memory (CAM) array. This is because CAM allows comparison of a requested addresses tag against the many cached line tags in parallel. Unfortunately this means that access procedure is divided into two stages − tag comparison and data read − thus making the access time in any highly associative cache longer than in a direct mapped equivalent. In this section the structure and

access procedures employed in both high and low associativity caches is discussed.

Consider the 4-Kbyte 2-way set associative cache in figure 2.8. It is very similar in structure to the direct mapped equivalent shown in figure 2.6. Both have 32-bit words organised into 256, four word cache-lines. The direct mapped cache is made of two banks of SRAM; one for address tags and one for cached data values. In the set associative cache however cached tags are stored in two separate banks of SRAM. This done to allow for the two-way associativity. It should be noted that the 1024 cached data words could still be held in a single bank of SRAM (as in the direct mapped one), but if parallel tag and data reads operations are to be supported then a two array structure as in figure 2.8 must be used.

In the 2-way set associative cache when an address request arrives the seven set bits are used to select the two of the 256 cache lines on which cached data can reside. The tags for these lines are obtained by using the set bits to index into the two tag store arrays and reading out the tags stored there. Simultaneously, the two candidate words are read out of the SRAM array. This is done by combining the seven set bits and two word bits to index into the two data store SRAM arrays and reading the words into buffers.

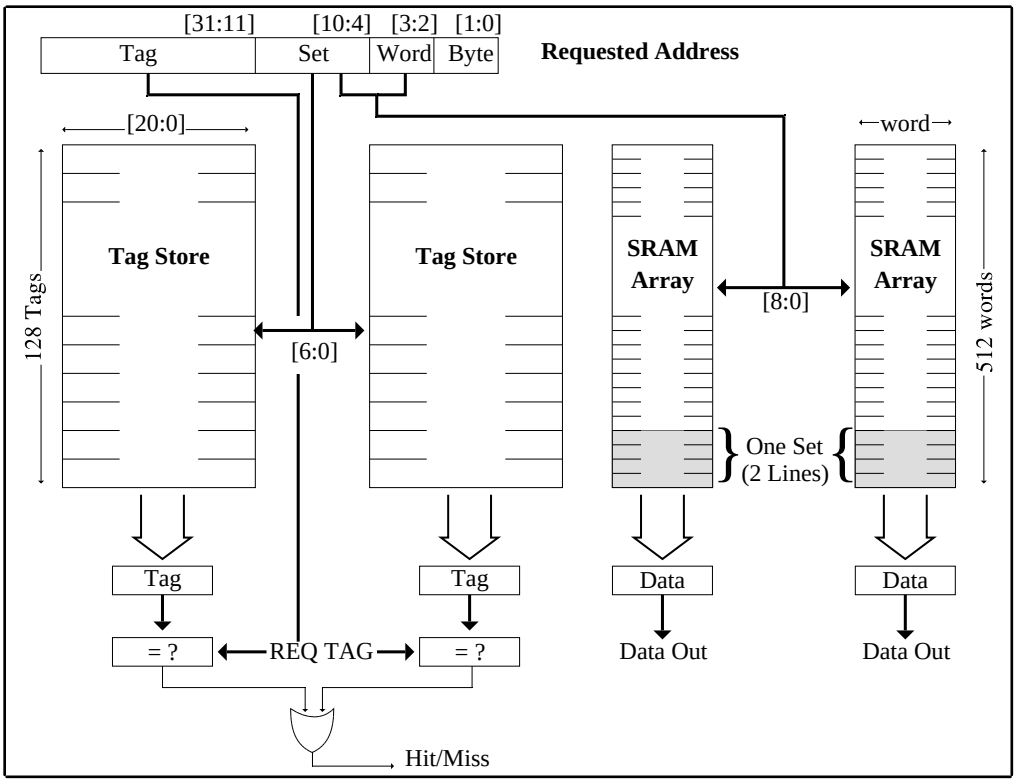

Figure 2.8 Structure of a 2-Way Set Associative Cache

Once the tags have been read out of the tag store arrays each tag is compared against the tag bits obtained from the request − this is the 2-way associativity. If either one matches then a *cache-hit* occurs − the requested data words are in the cache − and the data from the associated data buffer can be read out. If neither of

the tags match then a *cache-miss* is said to have occurred i.e. the required data words are not in the cache. If a replacement occurs then the request's tag value must be written into one of the tag store arrays and the line data must be fetched from main memory and written into corresponding SRAM data array.

If the fetch size differed from the size of the cache line − meaning that entire cache line may not contain valid data − then a valid bit would have to be used. In a set associative caches like figure 2.8 these would best be stored alongside the tags. Then as a tag is read from the array all its associated valid bits would also be read into buffers. After determining that a tag matched, some high order bits out of the word field in the requested address would be used select one of the valid bits in the buffer. This is done to check that the sub-block in which the requested word resides has indeed been fetched. If not a fetch for sub-block must occur.

It can be seen that by using structures similar to that shown in figure 2.8 the degree of associativity in a cache can be improved. There is a practical limit however, to how far it may be increased. As associativity increases, so does the number of parallel SRAM arrays, comparators and associated circuitry used when accessing the tags. Above about sixteen-way (usually no more than eight) it becomes impractical to use this method to further increase the degree association. At this point different structures must be employed.

With reference to figure 2.9, a typical cache with a high degree of associativity also has two main structures; a tag-store made of Content Addressable Memory (CAM) which contains address tags and an area of fast SRAM to hold the cached data and any 'dirty' or 'valid' bits. The diagram show schematically how each cache-line is associated with a tag address containing high-order address bits indicating the contiguous region of main memory from where it originated. Normally the data words are kept in a single contiguous block of one-word wide SRAM as in the direct mapped cache (figure 2.6). Any valid bits may also be kept in this array structure or in a similar parallel one.

Thus to decide whether an associative cache holds a particular datum the following procedure is followed: a memory request (real or virtual) arrives. The high-order bits are presented to the CAM. If the tag field of the address matches any one of the tags store in the CAM then a cache hit has occurred. This is indicated by the hit/miss signal out of the CAM array. The CAM array also provides the position number (in the array) of any matching line.

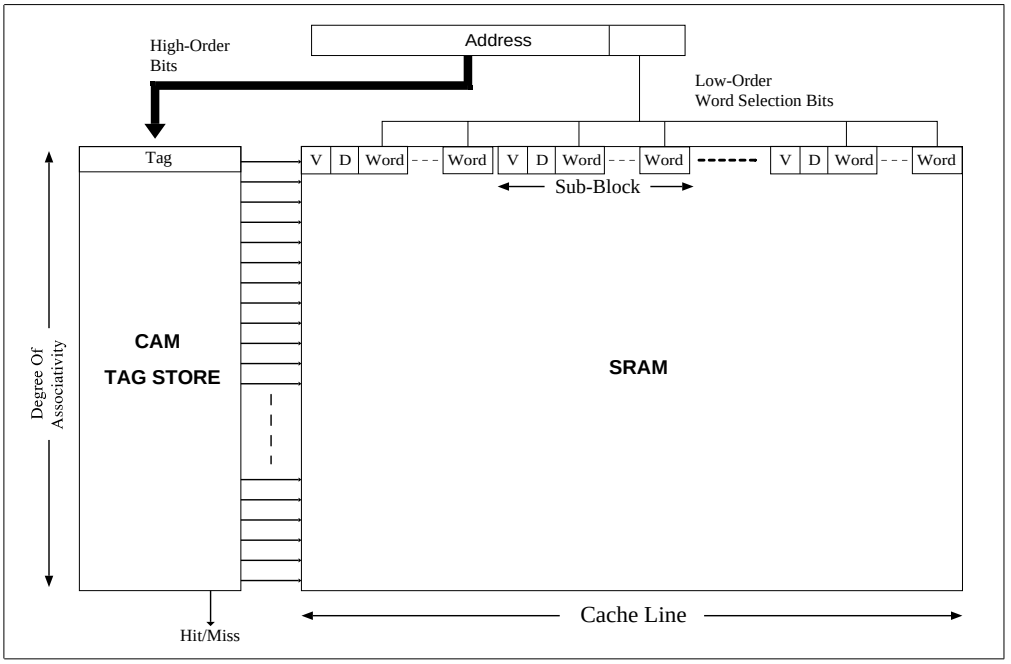

Figure 2.9 Generic Associative Cache Structure

Some low-order bits of the requested address are combined with the address of the matching line in the CAM in order to read the valid bit for the sub-block in which the requested word lies. If the sub-block is present then the address of the matching CAM line is combined with the remaining low-order bits of the address to select the requested word. If sub-block is not present then it must be fetched and written into the SRAM array before the operation can continue. If however, no matching tag in the CAM was found (cache miss) then a line replacement may occur. For an associative cache any tag entry in the CAM array may be replaced. If the line to replaced has dirty bits set (in a write-back cache) then the line data must written out to memory. Then the new request's tag is written into the CAM array and the cache-line − or part thereof − loaded into the SRAM data array.

2.2.4 Power Usage in Caches

Since one of the possible benefits of asynchronous design is a reduction in power consumption it is worth considering the major causes of power usage within caches. Most of the power is dissipated when accessing the CAM and SRAM arrays. Both these structures are constructed from basic storage cells. These have a pair of wires (**sense** lines) running vertically down through each cell that are used to read and write the actual data values. **Write** and **select** lines run horizontally through the SRAM array; similarly **match** and **write** lines run horizontally through the CAM. All of theses lines are physically long and present a high capacitive load. In order to minimise the loading and reduce the transistor count these blocks tend to have dynamic operation. i.e. they use a system of pre-charging the logic (and all outputs) into either a high or low state. On the 'evaluate' phase of operation the outputs are pulled only up or only down as required. This lowers the transistor count for a particular circuit thus reducing the

capacitive loading and therefore the time of operation.

The **sense** lines carry differential signals during read operations. These are required because during a SRAM read operation; first all the bit and not-bit lines are precharged high and then the selected cache-line is activated by driving its select line high. This causes a pull-down transistor in each cell of the selected line to either discharge the bit or not-bit line depending on its internally stored value. It would take a relatively long period of time to discharge the **sense** line to a logical low level due to the large amount of stored charge on the **sense** line. Thus small changes in the relative levels of the two sense lines are measured using a sense amplifier at the bottom of each column in order to determine the state of each bit more quickly.

Unfortunately sense amplifiers are analogue components and they dissipate power continuously whilst they are in operation. The amount of time that they are on is therefore critical and needless use should be minimised. This can be done by using ‘feed-back’ circuits that switch off each sense amplifier as soon as the state of its bit has been determined. However it may be better if techniques that minimise the use of sense amplifiers could be found.

2.3 The ARM Processor

2.3.1 Existing ARM Processors

Initially developed between 1983 and 1985 by Acorn Computers Ltd, ARM processors are general purpose 32-bit RISC microprocessors. Currently the development of the ARM series is being undertaken by Advanced Risc Machines Ltd. with the processors being fabricated by VLSI Inc. and GEC Plessey Ltd. The current ARM variants are listed in table 2.1.

Model	Description
ARM1	Non commercial, internal development version
ARM2	First commercial version.
ARM3	Processor core same as ARM2 but incorporates 4Kbyte combined instruction/data cache on chip.
ARM6	Macro cell. Extended address space to 32 bits and provided new processor modes.
ARM60	A Packaged Macro cell with JTAG boundary scan.
ARM61	Macro cell with JTAG boundary scan packaged and pin compatible with ARM2.
ARM600	Macro cell, 4Kbyte combined instruction/data cache, write buffer, MMU, co-processor and JTAG boundary scan on single packaged silicon die.
ARM610	Same as ARM600 except without co-processor interface.

Table 2.1 ARM Processors.

All ARM processors feature a small die size (under 30,000 transistor for the 7mm^2 processor core) and there are support chips (memory, video and input/ output controllers) available which are optimised for use in cheap DRAM based workstations. The small die size and the use of standard CMOS fabrication techniques means that the processors have low power consumption per unit cost and deliver a reasonable level of performance due to the pipelined architecture. This makes them suitable for embedded applications, hardware control (low interrupt latency) and as the CPU in low cost workstations.

The ARM6 is unusual in that it is not a self contained processor, but is offered as a macro cell (i.e a large VLSI layout component). In this case the macro cell is a complete 32-bit microprocessor, allowing system designers the flexibility of software based development augmented by specifically designed hardware modules. Using this approach the ARM6 is offered in various standard configurations (eg. with/without cache − ARM60/ARM600, configured 26/32-bit program counters versions − ARM60/ARM61 etc.) and other companies have tailored the design to meet specific needs (ARM610).

Internally the ARM employs a three stage pipeline; instruction fetch, decode and execute units. There are sixteen 32-bit registers available at any one time with one of them (R15) containing the program counter. Extra registers are overlayed over the existing ones when different processor modes are selected. A feature to note is that the processor mode is available to the memory subsystem and used to provide different memory maps for different modes.

An interesting feature of the ARM this generation of a *sequential* signal. This

signal originates from within the processor core and appears with each memory access indicating that the current memory address is sequential with the last one. Its existence is primarily due to the requirement for the chip set to work efficiently with cheap paged mode DRAMs. These devices have access modes in which memory locations within the DRAM row can be accessed without the need for a new row address to presented each time. This significantly reduces the access time by removing need for row address multiplexing and nRAS hold times.

The sequential pin is utilised by the external memory management unit (MMU) to optimise memory accesses; if the sequential pin is set and still accessing the same DRAM row in the same fashion (read versus write) then a faster access mode can be used. Typically this optimisation manifests itself mainly during instruction fetching and the loading and storing of multiple registers. Both these operations utilise the address incrementer within the ARM.

2.3.2 Existing Synchronous ARM Caches

A synchronous cache has been developed for the synchronous processor described in section 2.3.1. It is a 4kbyte, 4-set, 64-way associative virtual cache with demand fetch, random replacement and write through policies, its structure is shown in figure 2.10. It is currently used in the the ARM3, ARM600 and ARM610 although in the 600 series the cache is not necessarily virtually addressed since the MMU is also on chip.

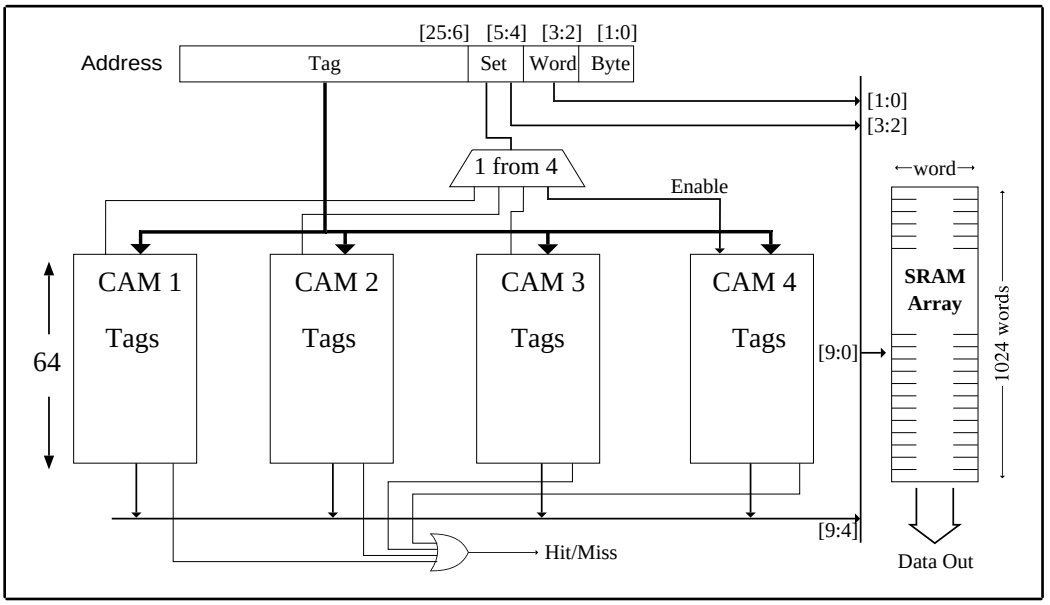

Figure 2.10 **ARM3 Cache**

A virtual memory address arrives and the bottom two bits are ignored (only used during byte writes). The next two bits choose the word in the quad-word line. Bits 4 and 5 select which CAM array is active this cycle and the rest of the address is presented to the active CAM. Three extra bits are also factored into the tag address. One valid bit per line which can all be cleared in parallel on cache flush

operations and two access control bits. Use of different memory translation tables for user and supervisor code can be handled using these two extra access bits and control registers.

The tag comparison stage supplies a hit/miss signal and a six bit address of the matching line if a hit does occur. These are combined with the set selection and word number bits to give a ten bit SRAM address. The SRAM is organised as a 1K by 32-bit word long array and the ten bit data address locates a word within this array. Once selected the data is read out via sense amplifiers or new data is driven into the array on write cycles.

This cache structure was used in the ARM3 and later in the 600 and 610 variants. The design was primarily motivated because typical ARM samples were capable of being cycled at twice their regular clock rate but were limited to 10MHz because of slow DRAM parts that made-up main memory. With the possibility of shrinking the design rules for new batches of the processor to at least 1.5 micron and thus obtaining an even faster cycle times. Regular main memory is unable to cycle at these sorts of speeds. Thus instead of just producing smaller processors it was decided to develop a code-compatible, cached version of the ARM, the ARM3, for use in applications that demand more processing power [FURBER89].

By utilising the reduction in area used for implementing processor core, the remainder can be used for an on-chip cache memory, giving a single chip solution. Another goal was to maintain the memory interface allowing the new chips to be substituted in systems via the use of small daughter boards thus keeping system development costs to minimum. Since address translation is done off chip by the MEMC, only virtual addresses are available to the cache. The cache is therefore virtually addressed.

The ARM is strongly von Neumann in nature, transferring only one instruction or data word every cycle. This − coupled with the fact that space for the cache on the silicon die was limited − made the use of a mixed instruction and data cache desirable. A mixed cache is better at adapting to and balancing the amount of data and instructions stored in the cache for the currently executing workload, whereas fixed partitioning can be wasteful of resources for the small cache sizes considered.

Organisation of the cache was based on results of simulation of different structures. Real-time memory reference traces were collected from full ARM based system by using a hardware add-on. This allowed the execution of a typical work load (complete with user interaction) whilst recording all reference to memory. Such traces are more realistic than ones obtained by architecture simulation since simulated traces do not tend to model user or operating system activity. These traces were then used as input for simulations of different cache structures. It was found that a caches with a high degree of associativity gave better performance than direct mapped alternatives that would fit in the limited chip area.

A completely associative cache requiring the overhead of one 24-bit tag entry per cached data word was very inefficient and so a four-word cache line was employed (one 22-bit tag per 128 bits of data). This resulted in a 256-way associative structure but this was found dissipate too much power when accessing the CAM array. Experiments were done and it was found that splitting the CAM array into 4 sets of 64 entries had little effect on the performance but did lower the power consumption by requiring only one bank of CAM be active for each request.

A write-through strategy was used to avoid the need for a complex control circuit for cache flush or line copy back operations. The organisation of the cache as described allows the hit/miss decision to be made early since it is based entirely on the contents of the tag store with a single valid bit. A write back cache would possibly use a ‘dirty’ bit per word − indicating data to be written out. Such bits are more logically stored with the actual data in the SRAM thus making any hit/miss decision time slower. Given a write-through cache, a no-allocate on write policy was used since allocating on writes was found to be ineffectual.

Demand fetch and random replacement strategies were chosen for reasons of ease of implementation. Demand fetch is the most straight-forward of fetch policies. Although not quite as good as the least recently used algorithm, random replacement is much easier to implement in hardware. It also offers more graceful degradation in performance when pathological loops just break the LRU algorithm. Write buffers to avoid stalling the processor during write through operations were not added because this would exclude the use of existing translation exception mechanisms implemented by the MEMC.

The additional benefit of the cached processor over its uncached predecessor was that it utilises much less main memory bandwidth (10% of the original amount). This makes it less prone to performance degradation in situations where memory bandwidth is limited, for example in a system where the CPU and video subsystem are in contention for the same memory bus. Low bandwidth requirements also make the cached chip an attractive multiprocessor.

The development of the ARM600 saw the introduction of a write buffer in addition to the on-chip cache and MMU. The MMU has unusual features to aid in implementing an object orientated environment and will be not be described further (see [ARM600]). The presence of the MMU on chip allows address translation to be done up-stream of the cache and write buffer and so preserves the ability to have exact memory aborts.

The write buffer takes the form of eight data slots and two address slots. Up to eight writes occupy the data slots and the two address words control where in memory the data is sequentially written. This means at worst case only two independent write operations can be buffered but the buffer has been engineered in recognition of a feature of the ARM whereby multiple data values can stored (and loaded) in a single instruction. During a “store-multiple”, writes occur to sequential memory locations, in these situations the buffer need only hold the

address of the first location written to. When a sequential write arrives, it is associated as being part of the current group of sequential writes by having an extra bit in the data slot set to indicate which address tag to use. When an non-sequential (unconnected) write arrives then the other address tag must be used (if free). If no address and/or empty data slot are available the processor is stalled until space becomes available.

The buffer maintains control of the port to main memory and data is written out on a continuous basis and all writes pass through it whilst reads are being satisfied from within the cache. Thus when the need for a read from main memory arises (eg. line fetch, uncacheable read etc), it has to wait while the write buffer empties. This done to preserve strict ordering of reads and writes. The use of this write buffer produces up to 10% improvement over a cached system without buffer for certain classes of program.

2.3.3 The Asynchronous ARM

As part of the ESPRIT funded OMI/MAP initiative the AMULET research group are currently in the process of developing an micropipelined version of the 32-bit ARM6 architecture. The project aims to investigate the applicability of the micropipelined design and to demonstrate the low power consumption characteristics of the methodology when implementing a processor which can be used outside of an academic arena. The ARM is a good candidate for implementation since it has a low transistor count, is a relatively simple architecture and, as a commercially successful processor, is often used in low power situations.

As a direct effect of implementing the ARM in an asynchronous framework, the datapath was micropipelined. This means that the execution phase which used to be a single synchronous stage is now internally micropipelined. This allows data loads to be issued and normal CPU operation to continue unhindered until the data requested by an earlier load operation is actually required. The control and management of this in asynchronous ARM is facilitated by a novel, patented structure called the *lock FIFO* (see [PAVER92a] and [PAVER92b]).

Decoupling of data read operations from normal CPU operations by the addition of the lock FIFO has implications for the cache design. When a data load operation is issued, the destination register for the loaded value is locked. The CPU is now free to carry on executing other instructions; this includes generating requests for more data. Operations within the CPU cease, either when the lock fifo becomes full (currently four data words), or some of the requested data that has yet to arrive, is actually required for a computation. Contrast this behaviour with that found in the synchronous version where the CPU stalls on every data fetch until satisfied.

In effect, the presence of the lock fifo allows the execution pipeline to extend out, into the memory subsystem before coming back into the processor. One major implication of this is that all read and write requests to the memory/cache sub-

system must maintain a strict ordering − any requested data must emerge from the output of the memory pipeline in the same order as the requests entered. To maximise the benefits from this mode of operation, code should be designed so as to maximise the time between data being loaded into a register and the time at which it is actually used. This would have the greatest possible chance of allowing processor operation to continue whilst the data load(s) were being satisfied by external memory. This needs changes to programs and/or compiler since this feature did not exist in previous implementations.

In other respects the asynchronous ARM differs little, in terms of the programmer's model, from the architecture defined by the ARM6 (see section 2.3.1). Within a processor there are sixteen 32-bit registers available at any given time, one of which is the PC. It has numerous processor modes (some for backward compatibility with 26-bit processor). It possesses 32-bit address and data buses and is capable of generating exact data aborts.

Chapter 3

Experimental Method

3.1 Evaluating Cache Performance

3.1.1 Overview/In Synchronous Systems

When designing a cache the aim is to try to maximise the **system** level performance. This includes the CPU, cache, memory and any other sub-system that influences the system. The choice of parameters used for measuring the performance of a system is subjective; it could be measured as the number of jobs that are completed within a given time (the *throughput*) or the time taken for a single job to finish (the *response*/*execution* time or *latency*). Decreasing the response time generally increases the throughput although this is not necessarily true. For example, if multiple computers are used for a number of separate tasks (e.g. a distributed booking system) then the throughput can be increased but response time is the same, since no single job completes more quickly.

When discussing the performance of a cache system, measurement is normally done with reference to the time taken for a *job* to complete. The selection of *benchmark* programs that constitute a 'typical' job has to be done carefully. Short of actually using a wide variety of real programs which perform real tasks but which would be too cumbersome to use, a benchmark should reflect the general make-up of typical programs embodying all significant features found in real code without too much superfluous detail. It should be noted that most 'real' tasks involve user interaction which is hard to simulate accurately.

Real systems often employ multitasking operating systems and perform a significant amount of processing in addition to running simple user jobs. The operating system provides functions that allow many programs to 'run' at the same time by scheduling programs and managing resources to make the best use of the system for a given selection of jobs and devices. Thus if a program is waiting for a disc to become available, if it has been executing for a comparatively long time or if any operating system code has to be run, then a *context switch* may occur. This involves saving a program's state, restoring the state of a previously executing one and maintaining any associated data. This overhead can significantly alter the both the general make-up of instruction set usage and the temporal localities exhibited by programs when run in isolation. This feature of modern operating systems must be borne in mind when choosing benchmark programs.

Given that benchmark programs are chosen such that they exhibit *typical* behaviour, the total time taken for a task to complete is a good indication of the

performance of a system. In a synchronous system the total execution time $t_{execute}$ is given by;

$$t_{execute} = N_{job} \times t_{cycle} \qquad \text{EQ 3.1}$$

In this simplistic equation N_{job} is the number of CPU cycles executed when running a job and t_{cycle} is the cycle time of the CPU. The total number of cycles required for a job can be estimated from the product of the instruction count and the average number of cycles per instruction.

On modern RISC processors data processing instructions execute in a fixed number of cycles (typically one) since they are usually register−register operations and are therefore fast as they take place entirely on-chip. The loading and storing of registers and fetching of instructions involves access to off-chip memory which is significantly slower.

Memory accesses can be broken down into three types; instruction fetches, data reads and data writes. In a synchronously cached system these can either be satisfied by the cache or main memory. Regular advances in processing technology and developments of CPU design techniques (e.g. RISC) has meant that the speed of on-chip processing within the CPU has steadily increased. Now-a-days the speed of CPU operation regularly outstrips the speed of external memory. This means that a modern CPU tends to be limited by the speed at which it can get data and instructions. This has resulted in the cycle times of synchronous CPUs with an on-chip caches being determined more by the speed at which an effective cache strategy (high hit rate) cycles, than the speed of the slowest CPU operation. For this reason modern cache designs operate on the principle that the minimum cycle time of the cache determines the cycle time of the processor and therefore the fundamental ‘time unit’ for the system.

Consider a synchronous system featuring a combined instruction/data cache with a write through policy. Rather than calculating the performance directly in terms of the time taken to complete a job, the calculation can be done in done in terms of the number of cycles. Thus the total number of cycles spent executing a job can be factored into parts; the number of cycles spent executing code on-chip − not accessing memory, cycles spent accessing the cache for instructions and data, plus the number of extra cycles spent accessing main memory for instruction and data on cache misses, and finally the number of cycles devoted to writing data out to main memory.

Assuming that all read operations take a single cycle for hits − and misses can be modelled by a fraction of the number instruction fetches/data reads that miss, multiplied by the average number of cycles satisfying them − then a mathematical equation can be constructed:

$$N_{total} = N_{execute} + N_{Ifetch} + N_{Ifetch}.m.n_{MMread} + N_{load} + N_{load}.m.n_{MMread} + N_{store}.n_{write}$$

EQ 3.2

Where

$N_{execute}$	Number of cycles spent not performing a memory reference.
N_{Ifetch}	Number of instruction fetches.
N_{load}	Number of loads.
N_{store}	Number of stores.
n_{MMread}	Average number of cycles for a read miss.
n_{write}	Average number of cycles for a write.
m	The miss ratio for the cache.

This equation can be further simplified. In synchronous RISC machines the time spent executing completely internally whilst not performing any memory reference, $N_{execute}$, is zero − a pipelined processor will, at least, have to fetch another instruction to fill space left by the currently executing one. It may also need to access the memory for operands which, in a system employing a mixed cache, means that all reads (instruction and data) are multiplexed onto the same port and can be combined. If one also assumes that a read hit takes a single cycle to satisfy, then by using a miss ratio which is a weighted average of the cache's instruction and read miss ratios, equation 3.2 can be simplified thus;

$$N_{total} = N_{read}\ (1 + m_{read} \times n_{MMread}\ ) + N_{store} \times n_{write}$$

EQ 3.3

Using equation 3.3 various synchronous cache architectures can be designed and evaluated (minimise N_{total}). A given benchmark will have a fixed number of cycles devoted to reading, N_{read} , and writing, N_{store}. If the write policy is then fixed (n_{write} is constant − though it may be reduced by employing a write buffer), the only free variables left are the miss rate, m_{read}, and miss penalty, n_{MMread}. These are more important anyway because the ratio of reads to writes is approximately 9:1.

The primary organisational characteristics of a cache, **C**, determine its miss ratio, m(**C**). The fetch and write policies also affect the miss ratio but their involvement is not straight forward and so will be ignored in the following discussions. A cache's major characteristics are; its size, C, its associativity, A, its number of sets, S and its block size, B; the miss ratio can be written as a function of these thus:

$$m(\mathbf{C}) = f(S,C,A,B) \qquad \text{EQ 3.4}$$

As noted previously a relationship holds between these parameters − the total size $C = A \times S \times B$ − so when doing experiments only two of these are independent variables. It is generally accepted that as these parameters increase the miss ratio decreases asymptotically and, after a certain point, further changes provide little benefit (see figure 3.1).

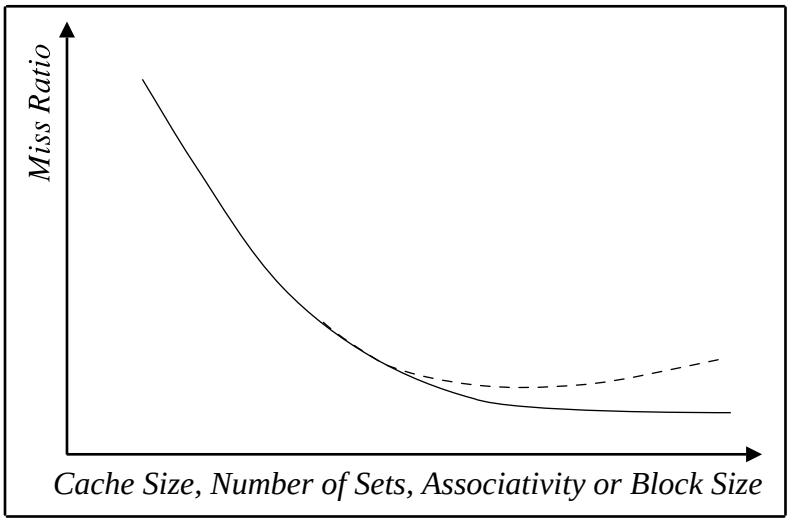

Figure 3.1 Miss Ratio versus Cache Parameters.

The block size is unique; there exists a block size that minimises the miss rate. Increasing the block and hence the fetch size decreases the miss ratio due to the spatial locality of programs − the fetch for a missed word is likely to bring words that will soon be required. Due to other constraints imposed on the cache parameters, as the block size increases the degree of associativity decreases; this means that there can be contention for the remaining tags, reducing the cache's ability to hold widely distant data. In a cache that fetches data concurrently with CPU operation further increases could mean the fetch unit is still in use when another miss occurs thus requiring the new request to have to wait before being satisfied. In a cache that fetches an entire block before continuing this effect gets transferred to the miss penalty.

With reference to equation 3.3, the miss penalty, n_{MMread}, is the overhead incurred due to main memory when a miss occurs (in average extra cycles per read miss). It is a function of the latency, *LA*, and transfer rate, *TR* (expressed in terms of number of CPU cycles), of main memory and also the block size, *B*, used;

$$n_{MMread} = j(B,LA,TR) \qquad \text{EQ 3.5}$$

When a miss occurs a block must be fetched. Initially a delay of *LA* cycles is incurred before the first word arrives. After this the rate at which the remaining words arrive is dictated by the rate of transfer, *TR* (in cycles per word), and the number of words transferred, *B*. When designing for a specific system the main

memory characteristics are fixed, effectively leaving the the miss penalty entirely dependent on the block size. A cache can be designed and simulated in order to obtain figures for the miss ratio and miss penalty. Using these in equations 3.3, 3.4 and 3.5 the relative performance of different designs can be compared.

A significant relationship between the cache parameters and minimum cycle time of the cache has thus far been overlooked. This fixed cycle time is the basis of the evaluation procedure presented so far but this relationship is very difficult to quantify. This is primarily because it is extremely dependent on the silicon level implementation of the structure and control logic used to implement the fetch, write and replacement policies. It is difficult to determine optimal cache configurations without designing these units in detail. At a higher level the strongest statement that can be made about the relationship between cache parameters and the cycle time is that it is monotonic, though not strictly so (as is illustrated in figure 3.2).

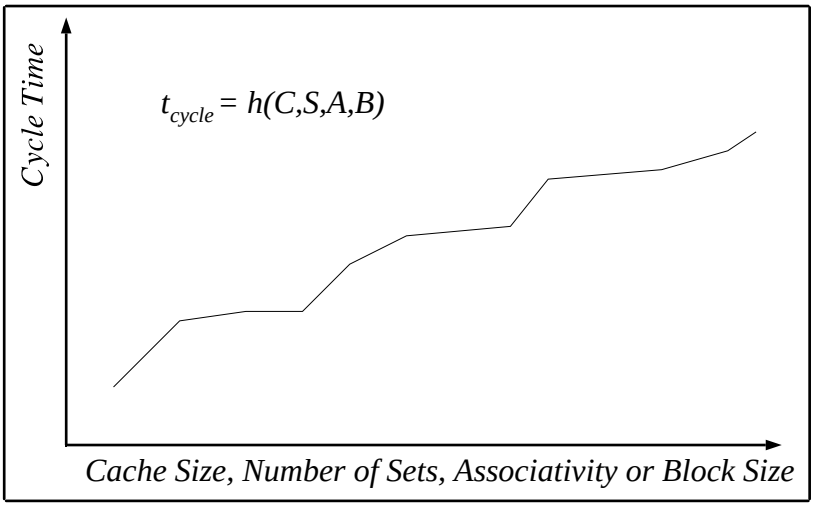

Figure 3.2 Cycle Time versus Cache Parameters

3.1.2 Evaluation in Asynchronous Framework

From the previous section it can be seen that the evaluation methods used in synchronous systems rely on the presence of a global measure of time in the form of a CPU clock cycle. An asynchronous pipeline does not possess such a measure; there is not a fixed time in which the cache will satisfy a request nor does it take a fixed time to perform a miss or write through. Instead a request propagates along the pipeline at a rate determined by what preceded it and what sort of action it requires. Since each stage may take a data dependent time − and there may more than a few stages − the number of possible paths and times soon multiply.

The delays through the pipeline will be bounded. When a request resulting in a cache hit is applied to an empty pipeline, it will invariably take less time to process than one which is delayed behind a cache miss. The time taken for a cache hit to pass through an empty pipeline puts a lower limit on the read latency. Similarly there also exists a ‘pathological’ case in which, say, a write is occurring

from a write buffer, which is delaying a cache miss operation whilst a new request just received is also about to be a cache miss. The new request will thus have to wait for both the write through and the following line fetch to complete before it is satisfied. Such a scenario illustrates an upper limit to a notional cycle time for an asynchronous cache. This general behaviour can be illustrated graphically as shown in figure 3.3.

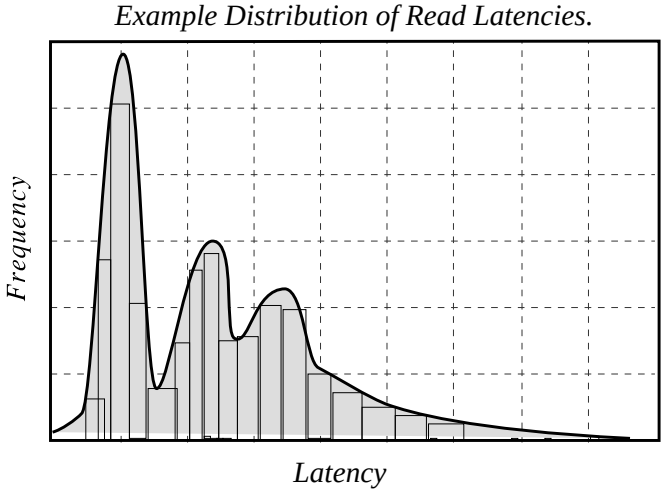

Figure 3.3 Example of Latency Distribution.

If each read request is timed from the point at which it enters the pipeline to the point at which it emerges, then the total time taken is its *read latency*. If a histograms of latencies is drawn (the latency vs. the frequency of that latency), it is expected that a graph similar to that in figure 3.3 will emerge which shows the distribution of read latencies that can be expected from a micropipelined cache. The primary features that are expected are illustrated; a tall sharp initial peak, several lesser peaks and a slow tail-off. These peaks correspond to different paths through the cache and the relevant frequency at which read requests take them.

The first tall peak corresponds to cache-hits − these are memory read operations from within the cache and so can be satisfied quickly. A degree of *smear* (smaller bars surrounding the larger ones) is likely to occur reflecting the fact that not all cache-hits take exactly the same amount of time to satisfy. Some are a little quicker because the micropipeline was empty ahead of them and some may take a little longer if the pipeline ahead is busy. The area contained under this peak reflects the amount of time that requests were satisfied from within the cache. For an effective cache organisation the area contained underneath should be large with the peak starting earlier rather than later and it should not have a very wide base.

Temporarily ignoring write operation since they are ‘fire and forget’ and as soon as they are dispatched CPU operation can continue, this total time also indicates how long it has taken to execute a program (EQ 3.6);

$$t_{read} = \int_0^{\infty} g(x)\ dx$$

$$\approx \sum_{i=0}^{n} x_i \cdot f(x_i)$$

EQ 3.6

Where

t_{read}	Time taken reading instructions and operands.
$g(x)$	Continuous distribution function of read latencies.
x_i	Latency of the i^{th} interval.
$f(x_i)$	Frequency of the i^{th} interval.

If, in equation EQ 3.6, the frequency of occurrence of a particular latency, $f(x_i)$, is replaced by the probability of a read taking that length of time to satisfy, then the equation provides the average read latency for a particular cache organisation whilst executing a particular program. Thus when evaluating cache organisations the effects on the latencies of altering structure or strategies can be analysed using the distribution and comparisons can be made using the average latencies.

Values and distributions for the rate of throughput can also be obtained in a similar fashion. The throughput of the pipeline can be measured by noting the time between the new requests being either consumed or satisfied. This can be done in two locations; at the start or end of the pipeline. If measured at the end just the rate at which requested data values emerge would be obtained. Measuring at the input of the cache pipeline allows better metrics for the peak throughput to be gathered since at this point the effects of performing writes is also considered.

As a result of the elasticity inherent in a micropipelined cache there is a slight reduction in the need to emphasise consistently fast responses to memory requests. This is a result of the autonomous nature of micropipeline stages where many operations can occur concurrently. A stage will only impose a delay on another stage if it needs to. For example, in normal operation the cache satisfies the majority of read requests from within the cache (i.e. hits). This leaves the stage that is responsible for generating line fetches free for significant periods of time. If desired, the idle time can be utilised to *prefetch* other data in an attempt to reduce the miss rate. This can be done by assessing when the cache is unlikely to perform a line fetch in the near future (e.g. as data is being read sequentially from a existing line) and instigating a prefetch for the sequentially next line. The circuitry for checking the conditions can be made self-timed − completing quickly if no prefetch is likely to occur, and only occasionally will it take longer − possibly adversely effecting the latency of following requests.

Such a strategy can go awry if a request that requires the use of the fetch unit occurs before the prefetch has completed. This means that the new request must

wait longer and also that the prefetched line is less likely to be required immediately, or at all. Thus techniques such as prefetching can increase the average latency for reads. In synchronous terms this is analogous to increasing the number cycles taken to satisfy a read hit, N_{load} (in equation 3.2), when such a read hit occurs. Similarly, for read misses the analogy is an increase in the miss penalty, n_{MMread}. This need not adversely affect an asynchronous cache in the same way as in a synchronous one.

More generally, since the design space is small, simulation of the different architectures can be performed and histograms of their performance generated. From these histograms the effects of different strategies can be seen and used to adapt designs in order to reduce the time taken for different operations. For example choosing a prefetch strategy which affects normal operation least − i.e. the height of the histogram for such an operation is small. In synchronous terms this has the effect of reducing the penalty for such cases and therefore reduces the total execution time for a job.

When comparing different architectures at the highest level, a single metric is required − the total execution time for a particular benchmark program. This can be obtained by summing the product of the latency (in seconds) and frequency of occurrence over the entire latency histogram. If the probability of occurrence is used in preference to the frequency, a normalised measure of performance is obtained.

3.2 Simulation

3.2.1 Simulator Design

The primary issues in the design of the cache simulator are modularity, flexibility and efficiency. Modularity is important since the cache is internally pipelined and attempts were made to determine the optimal partitioning of the system. Thus a modular approach allows the piece-wise construction and refinement of different strategies. Flexibility refers to the ability of a single simulator to model a wide variety of target systems so that within a particular partitioning scheme it is possible to alter various cache structures size, associativity, block size etc. The efficiency is important because like most simulations, cache simulation is compute intensive. To obtain realistic results large programs must be run and a large number of metrics kept to determine the effect of different cache scenarios. Thus a complex simulator that does a lot of processing per memory reference may not be as good as a simpler, more efficient one. To allow simulations to run in a reasonable amount of time a complex one must use shorter, less realistic program traces than a smaller faster simulator.

The nature of the micropipelined cache means that there are many stages operating concurrently, capable of exchanging data at any time and in an arbitrary order; thus the simulator takes the form of a event driven scheduler and client modules. The scheduler and modules are written in C with the scheduler

providing functions to control and interface the modules written specifically to model each part of a simulated cache. Individual modules implement specific cache structures (one or more micropipelined stages) which are connected to each other via events with associated data. The nature of the interconnections between modules (the events) is registered with the scheduler at the start.

The code associated with an event is invoked when a corresponding event occurs. Each module is responsible for taking its input data, manipulating it and producing other events at some later time. Internally a module acts upon the data it receives and maintains any internal state that it needs − specifically the state of all input/output and any pertinent statistics for that module. Such an organisation exhibits the same characteristics found in real micropipelines. Once a stage is working in isolation it can be transplanted into other situations as long as the interfaces between it and the other stages remain the same.

In all the simulated models there exist three additional modules; a producer, a consumer and memory. The memory module simulates the action of dynamic random access memory (DRAM). Since this remains constant over the entire design and evaluation phase, it was standardised and written early on. It presents an idealised and simplified view of main memory; based on a memory made from 512bits-per-row paged-mode DRAM parts, operation is optimised for sequential accesses on the same DRAM row (see figure 3.4).

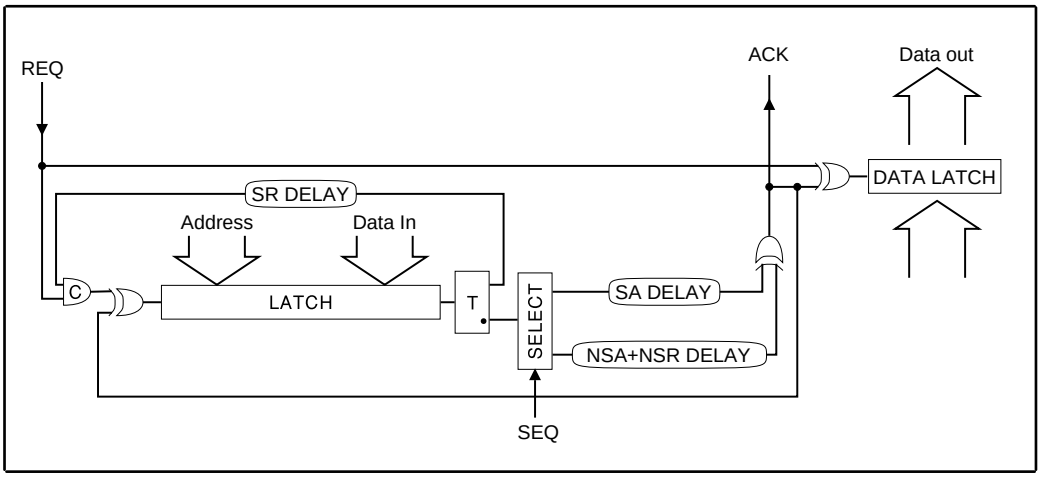

Figure 3.4 Memory Module.

In the memory module a request arrives and if it is of the same type (read/write), on the same DRAM row and sequential with the previous request then a fast sequential access is performed and the data is available/written a short *SA* (sequential access) time later. Otherwise a longer *NSA+NSR* (non-sequential access and non-sequential recovery, respectively) delay is incurred as memory is setup for a full access. This corresponds well with actual memory operation. Modern DRAM parts have such access modes which are inherently fast and do not require external multiplexing of both new row and column addresses on every access. The requirement for the sequentiality of access is a little stricter than necessary but can be accounted for by the need determine the type of access

quickly and simply. Since the sequential signal is generated directly from within the processor core it is always available and it is simpler to employ this rather than to add more complex logic to implement the test. Once the transfer has completed, *SR* (sequential recovery) time is set aside in which memory can recover and the logic for the next access (assumed sequential) is setup. No other memory access is allowed during this time.

Conceptually the producer and consumer are parts of the same module which in a complete system would be the CPU. Since it was decided to keep the simulator trace-based to aid modularity the functionality of the CPU was split into two phases, generating and accepting requests respectively. The producer simply reads information about a memory request from a file, 'bundles' it and passes it onto the first stage of the cache model. The consumer is responsible for accepting the data from the final stage of the model and gathering statistics. As each request flows through the cache one of the things it has associated with it is timing information. The scheduler is responsible for updating this information. The data associated with each request is time stamped when it is passed onto the next stage and when it is accepted there. The consumer is responsible for keeping a histogram of the read latencies through individual stages and the entire cache. Only reads can be dealt this way because writes go to memory and terminate there, whereas the data resulting from a read is passed back to the processor.

The decision to split the CPU operation and have a trace based simulation overlooks some important features. The most important of these is that there is a **data dependency** between the producer and consumer. The cache is internally pipelined and is attached to a processor capable of decoupling data reads from data usage. Thus the CPU is able to carry on generating requests until it finally has to stop, waiting for data requested earlier. The only way for the cache to stall such a processor, is to delay acknowledging a request. Even so this delay will still have to propagate back within the processor, stalling each unit in turn before the whole system comes to a halt. When the request is finally acknowledged, a quick succession of requests will appear on the output since many were partially processed and backed-up behind the stalled one.

Holding the time between memory accesses constant would not however, illustrate all the possible implications of overlapped operations within a cache. Thus it was decided that in lieu of an accurate asynchronous architecture simulator, the problem of relative timing between operations would be deferred. In the interim an upper limit on the maximum sustainable throughput rate for the cache could be obtained by supplying new memory requests to the cache pipeline as soon as space becomes available. The simulator design is sufficiently flexible to allow integration of a processor model, but the timescale of this project did not permit this.

A feature of trace based simulations that is commonly overlooked is the number cache misses during a *cold start*. These occur at the start of a simulation run when the cache is empty. For example if a cache initially contains one thousand empty

cache lines then an address trace of one million references will yield results in which up to one thousand misses are due to the cold start of the cache. One solution to this is to make the address trace long enough in comparison to the number of cache entries that the ratio of cold start misses to total misses is small. A more elegant solution [PRYBYLSKI89], [SHORT88], [WANG89], is to divide the address trace into two parts, in which the first part 'primes' the cache with the current working set. After this has been done gross level statistics such as miss rate can then be gathered to illustrate normal working behaviour. This solution was implemented and tried during these experiments, however the combination of small cache size and large multiprocess address traces meant that there was no significant effect on the hit/miss rates. The multiprocess nature of the trace meant that the cache was periodically flushed normally so it was decided to remove cold start handling from the simulator for simplicity.

The basic structure of the simulation environment consisted of: a scheduler to order and control execution of modules; the three pseudo modules implementing memory and the CPU, some utility code that maintained a generalised form of tag store and organised cache line structure and some code that created and maintained histograms of results. In this skeleton form the simulator comprised of about two thousand lines of source code into which different cache models then had to be incorporated. The source code for different models could be quite long and depended on the complexity of the model and the statistic maintained within it.

Due to the design and to improve efficiency of the simulator the penalty for flexibility was mainly concentrated in the compilation phase. Once a specific cache model had been written and incorporated into the simulator, various values pertaining to the structure were defined via C header files and the complete simulator compiled. Execution of this code using a trace as input resulted in the production of files containing statistical information. These were then post-processed and supplied to graph drawing programs in order to produce the illustrations in this thesis.

3.2.2 Address Trace Collection

A combined cache and CPU system is a single large micropipelined structure. The ideal way in which to evaluate the effect of architectural decisions within such a system is to construct it and take measurements. This is usually impractical and a reasonably accurate software model is used instead. At the time of the investigation there did not exist a suitable simulation of asynchronous ARM architecture so the execution of benchmark programs was kept separate from the actual process of cache simulation.

The choice of programs from which address traces are gathered is important when benchmarking. They should exhibit a normal mix of instructions and correspond to a typical workload. The ideal method of obtaining such a trace is to use a hardware device that monitors memory accesses on an actual system during

normal use. Traces gathered in this fashion provided a much better mix of instructions usage and loading since they can incorporate the effects of user interaction. Since the asynchronous ARM has yet to be fabricated and the required hardware monitors were not available, a software based trace gathering scheme had to be employed.

A collection of benchmark programs was gathered (see table 3.1). They consisted of two *real-world* application programs (the cache and instruction set simulators), a few other programs and some specific programs to model user interface behaviour. *Windows* was a code segment that mimicked the code found in user interfaces − block moves of memory.

Program Name	Total References	Instruction Fetches	Data Reads	Data Writes
cache_sim	2,001,527	1,438,296	278,695	284,536
ARMem	1,875,981	1,336,706	281,397	257,878
bytedemo	635,680	451,536	90,246	93,898
simple	89,260	61,323	12,072	15,865
windows	112,534	49,621	32,529	30,384

Table 3.1 Benchmark Programs.

Programs used for benchmarking can exhibit temporal effects; when started they first execute initialisation code that can be very different from code run during normal operation. This is known as the *warm-start boundary* and is especially true of complex, time consuming programs that possess a large setup time but an even greater main body execution time. Since none of the programs used for this benchmarking exhibit such behaviour (both the cache and architecture simulators used for benchmarking have relatively quick start up phases), the effect of this characteristic was not developed any further.

A trace of all memory references generated by a program was obtained by running the program on a synchronous ARM simulator. The simulator provided values for all the pins on the ARM: address, data, processor mode and control pins. The control pins indicate in which direction the memory transfer occurs (read/write), whether the current address is sequential from the previous one (SEQ), if it is a instruction or data fetch (OPC) and if the transferred value is a byte or word long (BYTE). In addition to these there is also a pin used to allow the implementation of semaphores, SWP; This indicates the occurrence of a read-modify-write cycle that comprises of a data read immediately followed by a data write which should remain atomic and uninterruptable.

For simulation purposes there is no need to know about the actual data being transferred and so the data pins are ignored along with the BYTE and SWP pins. Timing information about the relative time between memory requests could be kept but this would be significantly different in an asynchronous chip. For

example in the synchronous version instructions are prefetched every **cycle** up to a maximum depth of three (one executing, one being decoded and one being fetched). In the asynchronous version the maximum depth is five but the times between them are flexible. In normal operation the CPU fetches an instruction each time the CPU removes one from the head of the prefetch buffer for execution (i.e. each time a instruction is executed). After a branch the CPU loads the address incrementer with a new address and discards incorrectly prefetched instructions. Thus after branches, memory accesses occur in quick succession whereas normally they occur at rate dependent on the instructions preceding it. Such operations are compounded further by the interaction of decoupled data reads which merge and therefore interfere with the prefetch stream.

Once traces for all the benchmark programs had been obtained they were ‘stitched’ together into one large trace to simulate a multiprocess work load. The smaller programs were kept whole since they complete within one time slice (nominal 0.5 seconds). The larger simulations were broken down into two parts and all traces were then interleaved to result in a single address trace nearly five million requests long. An address trace of this magnitude was deemed adequate since the caches being designed were not large − a total combined size not exceeding 16Kbytes.

Chapter 4

Cache Design

4.1 Profile of Memory Traffic

When designing a cache for a specific architecture the nature of the memory traffic can suggest the types of spatial and temporal localities that are exhibited. The gross level statistics presented in this section as to the sequentiality and distribution of memory references were obtained by examining the the memory traces listed in table 3.1. The figures are mainly concerned with a property specific to the ARM of a sequential pin which, when set, indicates that a reference to the sequentially next memory location is occurring. Statistics were recorded as to lengths of sequential access and the distance between them, i.e. how many references occurred before sequentiality was lost and the distance between last reference in a block and the first reference in the next one.

The proportions of reads and writes can be of primary importance when considering the type of cache employed − a high percentage of write operations suggest that a write-through cache would be overwhelmed by a large amount of write traffic and that perhaps a write-back cache may be better suited. Examination of the benchmark traces indicates that, for those programs used, writes constituted approximately 14% of memory operations. Analysis of the distribution of writes demonstrates that they exhibit a high degree of spatial locality with about 80% occurring within 32 bytes of the previous block of sequential writes.

The statistics also showed that about 30% of reads occurred in blocks of between four and twelve sequential operations but distance between blocks of reads was much more distributed than that for writes. Even so about 20% of non-sequential reads were within 48 bytes of the end of the previous block. These figures suggests a high degree of locality of reference for writes − most occur as parts of small blocks of sequential writes separated by short distances. There is less locality for reads with about 70% of reads being parts of blocks less than four sequential accesses in length. The read locality that is present manifests itself as fewer, larger sequential blocks separated by slightly larger distances.

These figures suggest that writes do not demonstrate the same randomness in distribution as seen for read operations. The blocking of reads also suggest that smaller fetch sizes may be better suited for code produced on this architecture.

4.2 Major Decisions

4.2.1 Overview

When designing caches there is a very diverse problem space where, unfortunately, the decisions that one takes are not all orthogonal. There is a degree of interrelationship between most of the design parameters for a cache. Thus when examining prospective cacheing strategies it is best, at least initially, to fix some of the variables and concentrate on specific design tradeoffs.

Some of the most fundamental decisions that have to be made concern the basic nature of the cache; for example unified versus split cache, write thorough organisation versus write-back etc. This section discusses these major decisions, identifying what parameters were fixed and why. Attention is drawn to those parameters that were investigated and why details concerning their implementation were thought to affect the performance of a micropipelined cache most. The degree of interdependence between the value of some parameters is also highlighted explaining why and how these interactions arise.

4.2.2 Unified vs. Split

One strategy is to split the cache design in to two parts; one for data and the other for instructions. This has the advantage of increasing the potential bandwidth by allowing data and instructions to be fetched simultaneously and also enables more selective use of different structures and strategies within each cache. Instructions tend to be clustered spatially over short segments of sequential code followed by branches to code that is often nearby. In such circumstances a long line length or prefetch strategy might be useful since it is likely to bring in code that will soon be needed. Data on the other hand exhibits a greater degree of temporal locality which requires a higher degree of associativity in the cache if many spatially distant data items are to be held. Each type of cache can be optimised arbitrarily for access latency. In an instruction cache simplified control logic and reduced associativity (quicker hit/miss decision time) lowers cache overheads. However the need for two sets of control logic can increase the amount of die area taken by control, reducing the amount available for cache memory.

A major problem with split caches is coherence. There are two possible areas where copies of the main memory can be held. Consider an example of self-modifying code; initially an instruction is read into the instruction cache and executed. Later the same instruction is read except this time as data via the data cache. It is then processed and it written back out to the data cache and − in a write-through scheme − to the store. This written back data is in the form of a new processor instruction. A problem arises when this instruction is due to be executed − the opcode exists in two places; in the data cache and the instruction cache. If no communication between the two caches takes place then differing contents for the same location exists in two places simultaneously.

There are solutions to this problem. For example all data writes can be checked against the contents of the instruction cache. If present the line in the instruction

cache can either be updated or marked as invalid thus resulting in a fetch from memory to get the correct value. Alternatively, for modern processors, software that exhibits such properties is outlawed or is handled as a special case (either mapping such data to uncacheable areas of memory or using instructions that are not cached). However such operations are not necessarily rare and can occur − for example − when program code is loaded.

Another problem with split caches is that they *statically* partition resources between the two caches. Therefore to maximise performance the relative cache sizes should be based on the relative frequency of instruction and data references (adjusted to minimise miss rates). Unfortunately programs differ greatly in their mix of instruction and data usage; in small caches the combined miss rate for the split cache is higher than the corresponding unified one when a program varies significantly from the normal mix of instructions and data. This occurs because the programs utilise one cache in preference to the other and thus the net effect is to have a smaller effective cache hence increasing the miss rate.

For the version of the asynchronous ARM being considered there is also the problem of applicability to be considered. Split caches rely on two separate paths to memory, one each for data and instructions. In the ARM, instructions are always requested via the address incrementer and data requests from the data path are multiplexed into this stream. These could be separated although the address incrementer is also used, for convenience, to generate addresses during load/store multiple instructions.

The actual act of multiplexing the two asynchronous streams requires the use of a potentially time consuming functional unit called an *arbiter*. The arbiter decides which of two asynchronous events to let through to a shared resource at any given time. Splitting the cache just moves the arbiter into the cache logic to the point where the caches communicate to try to maintain coherence. At this point the data cache tries to check that a write does not affect the contents of the instruction cache. Since the instruction cache’s operation is autonomous to that of the data cache an arbiter is again required. Admittedly, possible arbitration only on data writes would be more efficient than arbitration which occurs each time the instruction and data streams are multiplexed, however since a split cache policy would require a major change to the processor architecture and the merits of such a cache are dubious for small cache sizes, the use of split caches is not investigated further in this thesis.

4.2.3 Real vs. Virtual Cache

The existing synchronous cache described in section 2.3.2 uses *virtual* addresses. This organisation was motivated primarily by the need to keep the cached version of the ARM compatible with existing systems in which memory translation performed by the memory management unit (MMU) was done by external circuitry. Since the form of the external MMU was known, specific support was provided for it within the cache. This meant that the problems normally associated

with virtual caches were largely unfelt.

A virtual cache must be flushed on every context switch because the virtual addresses now refer to a new part of real memory so the data actually in the cache is no longer valid. Such global purging can be avoided if the width of the tags in the cache are extended to include information as to which process owns this virtual address tag (known as the *process identifier tag*). The number of processes supported in such a scheme is subject to hardware imposed limits.

Another problem arises with the use of a virtual cache known as *aliasing*. This occurs when different programs use different virtual addresses for the same real address (e.g. for shared data). The two virtual addresses are said to be *synonyms* for each other. This means that it is possible to have two separate entries in the cache for the same location. The operating system and hardware used separately or together can detect and avoid or disallow such synonyms from occurring. For example a software solution may be to force shared data to occupy an area of uncacheable memory thus requiring all read and writes to go though the external MMU.

In isolation, virtual caches can be beneficial as they reduce the hit time by not requiring address translation before a cache access. However, pipelining the translation phase is relatively easy and can be done quite quickly. This results in a small increase in latency but potentially improves throughput by allowing the use of a simpler, more efficient real cache (no process ID field or synonym control overhead). Ultimately it is hoped to incorporate a micropipelined MMU on chip to give a complete, integrated solution. It was thus decided to assume that the cache was down-stream of any memory translation and thus a *real* cache. It should be noted that the addition of a very crude memory translation unit that mimicked the operation of the downstream MMU would allow a real cache to be used as virtual one.

4.2.4 Sequential Accesses

The presence of the *sequential* signal may be used for two purposes within the cache; speeding up cache accesses and reducing cache power consumption. Sequential accesses can be used to reduce the latency of cache read operations by removing the need for a full tag comparison when previous access was a hit. This can be done because it is known that if the sequential pin is asserted, the next access will also be a hit to the next word on the current cache line although extra logic is required to confirm that the next sequential access has not ‘fallen off’ the end of the line.

This extra logic would consist of a counter that was set up on each new cache line hit with number of the word that is currently required. On each consecutive sequential read access the count is incremented. If the count exceeds the length of the cache line then a full tag comparison operation must be forced. Alternatively if a new non-sequential request arrives then a full tag comparison must also take

place. In both cases the counter is again set up for this new non-sequential access on the assumption that the following requests will be sequential.

The ability to bypass the need for a full tag comparison also results in reduction in power consumption. This reduction in power consumption can be extended into the main SRAM array in the cache. This requires a potential reorganisation of the SRAM data store. Cached data values are normally stored in a SRAM column that is one word wide and data is accessed a word at a time. The SRAM can be reorganised so that a single line contains all the data words for a single cache-line. This allows the data for an entire cache-line to be read/written in one operation, whereas it previously it needed one operation per word accessed.

Reading from the reorganised SRAM can now be divided into two stages by the use of a SRAM line latch. Once selected, an entire cache-line is read into the line latch at the bottom of the SRAM array. The required word can then be read out of the latch using just its word offset. For sequential operations the line-latch need not be reloaded each time. Once loaded, individual words can be read more quickly, using less power, directly from the line latch rather than directly from the SRAM array. Power and size trade-offs need to be studied, however since the reorganisation of the SRAM has increased the power used in reading a SRAM line by increasing the number of sense amplifiers required. For sequential accesses, any increase in power consumption is offset by a reduction in use of CAM and SRAM array access circuits.

A possible extension to this system is to relax the need for sequentiality and leave only the requirement that consecutive operations be on the same cache-line. This would reduce the power consumption when executing code that has many short backward (or even forward) branches to locations that lie within the same cache line. Implementation of such a scheme would not require a counter but a special tag for the last line accessed that could be compared with more quickly and using less power than a full tag store comparison. New requests would first be compared against the special tag and if they matched the access is performed from the line latch otherwise full tag store/SRAM accesses must be performed.

Using techniques such as these effectively frees parts of the circuit from continuous use. There is potential to trade the power reduction gained in this way for improved performance by utilising these units during unused phases. For example in the cache optimised for sequential accesses, a little extra logic could trigger a cache *prefetch* operation by noting when the sequential accesses were approaching the end of a line (see section 4.2.6).

4.2.5 Degree of Associativity

A reasonable tradeoff between the degree of associativity and the effectiveness of the cache is one of the primary goals of small cache design. The associativity determines the number of disjointed threads a cache can hold. A large number of small cache lines allows many different areas of memory to be held simultane-

ously. It should be noted that increasing the associativity has implications for the tag store. Tags are required to hold the address of the stored location and if there is a high degree of associativity, then a greater number of stored tags need to compared against the requested memory location to determine a hit or miss condition.

When there is a high degree of associativity blocks of content addressable memory (CAM) are used for the comparison function. CAM stores the tags and allows direct parallel comparison of all stored values against another single value. It then produces a hit/miss signal indicating which, if any, entry matched. Unfortunately a CAM array comparison takes a *worst case* time which is determined by the width of a tag and the size of the array (i.e. its degree of associativity). The greater the associativity the longer the hit/miss decision time.

An alternative way for comparing an address against the tags uses discrete comparators. Tags are stored in SRAM, the ones to be checked against are read out and compared with the address using discrete comparators. This is faster than CAM and uses less space − SRAM is more space efficient than CAM because it avoids the overheads of comparison logic for each cell (see section 2.2.3). However it is only effective for caches with low associativity since it is impractical to read and compare more than about four tags at time in this way. Rather than increasing size of cache lines to account for a reduction in the associativity, normal practice is to maintain the line size and increase the number of sets. Thus more of an address' low order bits are used to locate the group of tags for comparison.

Caches in synchronous systems tend to have low associativity (using SRAM tag store) or be direct mapped due to the need to keep the processor cycle time down. Low associativity reduces their ability to hold many threads. Conflict misses arise due to needed data being ejected from the cache by new data that resides in the same set. The only solution for this is to increase the number of sets by increasing the overall size of the cache but increasing the number of sets to compensate incurs a greater likelihood of conflict misses.

In the small caches being considered for this project, large, low associativity caches might result in a higher miss rate and reduced effectiveness − this was investigated. Having long cache lines would make the same-line-access power reduction scheme (see section 4.2.4) more effective as long as the effectiveness of the cache was not compromised severely. Finding a good tradeoff between the associativity, the line size and number of sets is of high importance.

4.2.6 Fetching Strategy

The fetch strategy decides when and in what order to bring data from main memory. *Demand fetching* is normally used − a read miss occurs for a particular word on a cache line and only then is the whole line/sub-block fetched into the cache. The sequence of fetching can be done in two ways, either the whole block

is fetched in sequentially starting with the first word on the line and forwarding the requested word as it arrives, or the requested word can be fetched first followed by the 'tail' of the line, finally wrapping around to fetch words at the beginning. Fetching the requested word first has the benefit of reducing the latency for the initial data request but the following requests may be delayed because of the loss of sequentiality of access at wrap around. This effect was investigated by simulation and it found that for small line lengths the throughput decreased since sequentiality of access was lost at wrap around.

In addition to demand fetching, *prefetching* can be used. These algorithms attempt to predict what information will soon be needed and obtain it in advance. In order to keep the decision to prefetch as fast and simple as possible only sequential lines/sub-blocks are considered. A prefetch acts like a traditional cache read: first the address of the next line must be generated, then a check should made to see if it is in the cache. If it is not then the line to be ejected must be chosen and a request to fetch the required data from main memory issued. It can be seen that prefetching can cause problems; if the prefetch causes ejection of a line that is more useful than the prefetched one then the effectiveness of the cache deteriorates. In the worst case this translates into a completely unused line being prefetched and ejecting one that is required immediately. The fact that the cache is now committed to a prefetch can also increase the latency of following operations. In addition to this the increase in memory bandwidth due increased fetching activity can be detrimental to the system as a whole as by putting a greater strain on main memory.

There are different types of prefetching algorithms; *Always prefetch* is when every access to a line automatically results in a prefetch of the next. In *prefetch on miss* a prefetch for the next line only occurs when a regular cache access is a miss. There is also technique called *tagged prefetch* in which lines have an extra bit flag which is initially (and on every prefetch) marked as zero indicating it has not been accessed. Every time it is accessed the flag is set to one and as it changes from zero to one a prefetch of the next line is initiated. In synchronous systems "always prefetch" can be effective but greatly increases the memory bandwidth consumed by the cache (leaving less for peripherals) whereas "prefetch on miss" uses less memory bandwidth but is less effective [SMITH78]. In [SMITH82] it was noted that tagged prefetching was almost as effective as "always prefetch" but generated less extra memory traffic.

In the asynchronous cache being investigated it is also possible to initiate a prefetch only in certain circumstances. For example initiating a simple prefetching on an instruction fetch may not be necessarily be useful. It is likely that the instruction was generated by the address incrementer in the CPU and is prefetched itself. Thus having additional prefetch is not necessary when an instruction address originated from the address incrementor (easy to identify OPC=1 and SEQ=1). The remaining situation occurs when instruction fetches immediately follow a change of program flow. In this case the address is generated by the data path (or as a direct loading of a value into the PC) and is non-sequential.

Following this the incrementer is reloaded and proceeds to prefetch instruction sequentially from the new point of execution. A similar action takes place when multiple data transfers are multiplexed onto the memory port. In summary there is little point in performing prefetch on instruction fetches.

Data fetches however provide more opportunity for prefetching. A simple policy might be capable of fetching spatially close data such as a high-level language data structure. Programs compiled from high level languages exhibit a large number of array accesses which would benefit from a prefetch policy that was triggered during sequential line accesses. This could be facilitated using a counter as described in section 4.2.4.

Extra control logic which notes where in the cache line the first (non-sequential) access took place in conjunction with the counter (used to reduce power consumption) is all that is required to initiate a prefetch operation. Consider a situation were a sequential access has started near the beginning of a cache line. Starting a prefetch as the sequential access nears the end of the line may be more efficient than an always prefetch policy. The applicability of this sort of prefetch policy was investigated, first by noting the sequentiality of data accesses within traced programs and then by modelling simple systems.

4.2.7 Write Strategy

In synchronous cache designs one of the major decisions is how write operations are dealt with. Write through caches always send write operations to memory. Write back caches only alter the cached copy and write the altered data back to main memory when the cache-line is replaced.

Write back caches tend to deliver better performance by allowing localities of writes to be exploited. Allocating a line (or writing just to the cache) on a write miss means that new data areas can be held in the cache reducing write latencies and demand on memory bandwidth. Implementation of this policy requires the use of extra data to maintain the state of whether a line is altered or not. This “dirty” bit must be consulted before a line is replaced in the cache. If it is set then a write back of the altered data is required which imposes added design complexity and an extra burden on the memory bus during a line replacement. Another problem is that coherence between the cache and memory is lost. Other devices can, incorrectly, read the old contents of memory because the new value is still only in the cache. Control for a write back cache can thus be difficult to implement.

Write-through caches are easier to implement. They maintain coherence since all writes are sent to memory as well as the cache (where applicable). However this also causes an overall increase in the memory bandwidth requirement because all writes (which typically account for 10% of all memory references) appear on the memory bus. Contrast this with a write-back cache where write traffic is reduced since data is only altered internally − with writes normally only being sent to main memory when a dirty line is replaced.

In order to allow the write to complete and the memory to recover, write operations utilise extra cycles in synchronous systems. Within an asynchronous system that is not required, write operations become ‘fire and forget’ operations, i.e. the request for a write is generated and as soon as the acknowledge returns it can be deemed to have finished as far as the originating stage is concerned. This occurs at each stage down the pipeline until the interface to memory performs the write and does not acknowledge until completed. Similarly the memory recovery time is hidden in that no more operations will be accepted until memory has recovered. In this scheme store coherence is maintained by ensuring the strict ordering of read and write operations.

To decouple the effects of writes from other cache operations a simple addition of an empty pipeline stage will suffice. This is equivalent in function to a *write buffer* as used in synchronous systems. A write buffer simply buffers write operations allowing the rest of the cache to operate whilst the buffer completes writing to memory. The amount of buffering provided is dependent on the depth and the width of the buffer. Each write operation consists of an address and data tuple. The depth of the write buffer is the number of tuples the buffer can hold, the width refers to the number of data values associated with each address. Normally there is only an single address-data value pair, but this need not be so. The ARM’s write buffer can support up to eight data words to one address (see section 2.3.2). The method used in the ARM which coalesces adjacent writes can be generalised to a fully coalescing buffer which compares new writes with currently waiting ones, allocating a data slot when a match occurs. This is similar, though not as wide in scope, to the *write cache* strategy discussed in [BRAY91].

Caches containing write buffers have to deal with other cache operations that require access to main memory. Consider the example of a read miss causing a line fetch in a system employing a write buffer. In such a situation contention for the memory path can exist; write operations waiting in the write buffer to be written out and the pending line fetch. The simplest method of handling this is to force the read to wait until all the writes have completed. This maintains strict ordering of accesses and so guarantees cache consistency but can impose large latencies as the write buffer empties. A possible solution is to allow the reads to proceed ahead of waiting writes so reducing the miss latency. However coherence problems can arise if the location that needs to be read is also waiting in the write buffer. Bypassing the write buffer means the read will read the wrong value from main memory. Any bypassing techniques must therefore take measures to maintain consistency. This can be done by comparing the addresses in the write buffer against the read request. If there is a match the cache can either wait till the buffer empties or forward the data to the CPU directly from the contents of the write buffer.

Sophisticated techniques require complex control and so were not investigated however the effectiveness of simple write buffers was addressed. The results from these and other areas mentioned are examined in in the following chapter.

Chapter 5

Experiments and Results

5.1 Initial Experiments

5.1.1 Prototype Partitioning

In order to test various different cacheing strategies and designs, a model of a micropipelined cache was produced. It was decided to use a structure similar to that employed in the ARM3 cache as this provides a basis for comparison. Any model constructed on this basis would tend to highlight areas of difficulty either in terms of implementability or performance degradation.

The non-pipelined synchronous write through cache was partitioned into two stages (see figure 5.1); the tag comparison with hit/miss determination and the SRAM access phase. This corresponds with the actual structure of the synchronous cache; tag comparison is done during the first phase of the clock and, if a hit occurs, the access for the data value from the SRAM array takes place during the second phase. If, however, any main memory activity has to occur the second phase gets *stretched* to synchronise with the slower external access.

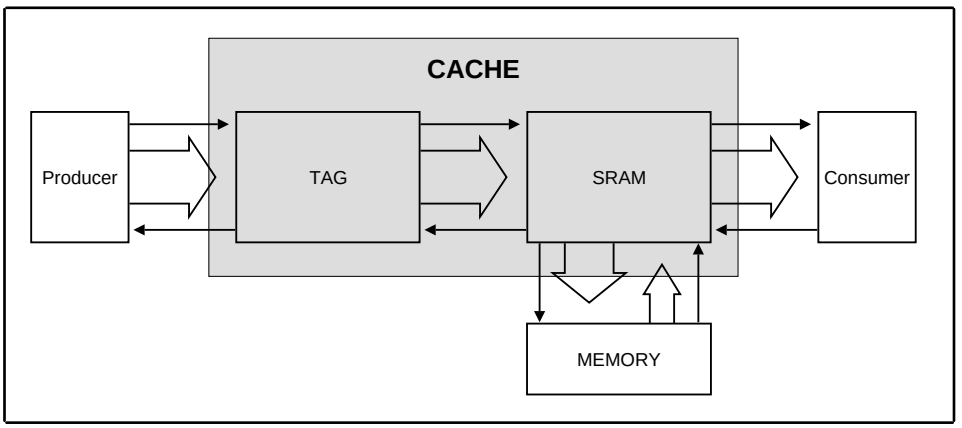

Figure 5.1 Prototype Cache Partitioning.

In the asynchronous version the tag comparison occurs in the *TAG* module. This also determines the form of activity for this request in the *SRAM* module. For cacheable read operations that are hits in the tag module, a simple lookup is performed in the SRAM array. In the model the time taken for these operations was based on the actual cycle time of the equivalent structures within the fabricated ARM caches plus an arbitrary amount for overheads (about 20ns for both tag lookup and SRAM access). This conservative estimate is justified since the structure of the cache is similar to that of the synchronous design.

The basic cache operation has two stages of pipelining as follows: a memory

request is generated within the CPU and presented at the first stage of the cache pipeline − the tag store. Here the request's tag is compared against those in the CAM. The nature of subsequent operations is determined by whether the comparison was a hit or miss and whether the operation was a read or write. For both hit and miss write operations the request is passed onto the SRAM stage leaving the tag store free to accept new requests. In the SRAM stage (see figure 5.2) writes are held in a latch until the write to main memory has completed. In addition during a write hit operation the data also has to be written to a line in the cache's SRAM array the address of which was passed on by the tag store.

For read hits the tag store passes the address of the matching line to the SRAM stage (select lines into SRAM array), whereupon the line is selected and the required word read out and passed to the CPU. When a read miss occurs and line replacement must be initiated, an entry in the tag store is chosen and the address of the corresponding cache line is passed to the SRAM stage. Here the SRAM stage is responsible for performing a line fetch; it does this by combining the request's tag bits with the output of a counter. This results in the fetching of the required line starting with word zero in the line and increasing. Assuming the slightly altered cache SRAM structure shown in the rough schematic in figure 5.2, it can be seen how each word is loaded into its relevant place in the line latch. Either a discrete comparator, extra signal out of the counter or something similar can be used to cause a request to the CPU to be generated as soon as the required data word arrives thus allowing CPU operation to proceed. The SRAM stage however, is still committed to completing the entire line fetching and writing it into the correct SRAM cache line before it will accept a new request.

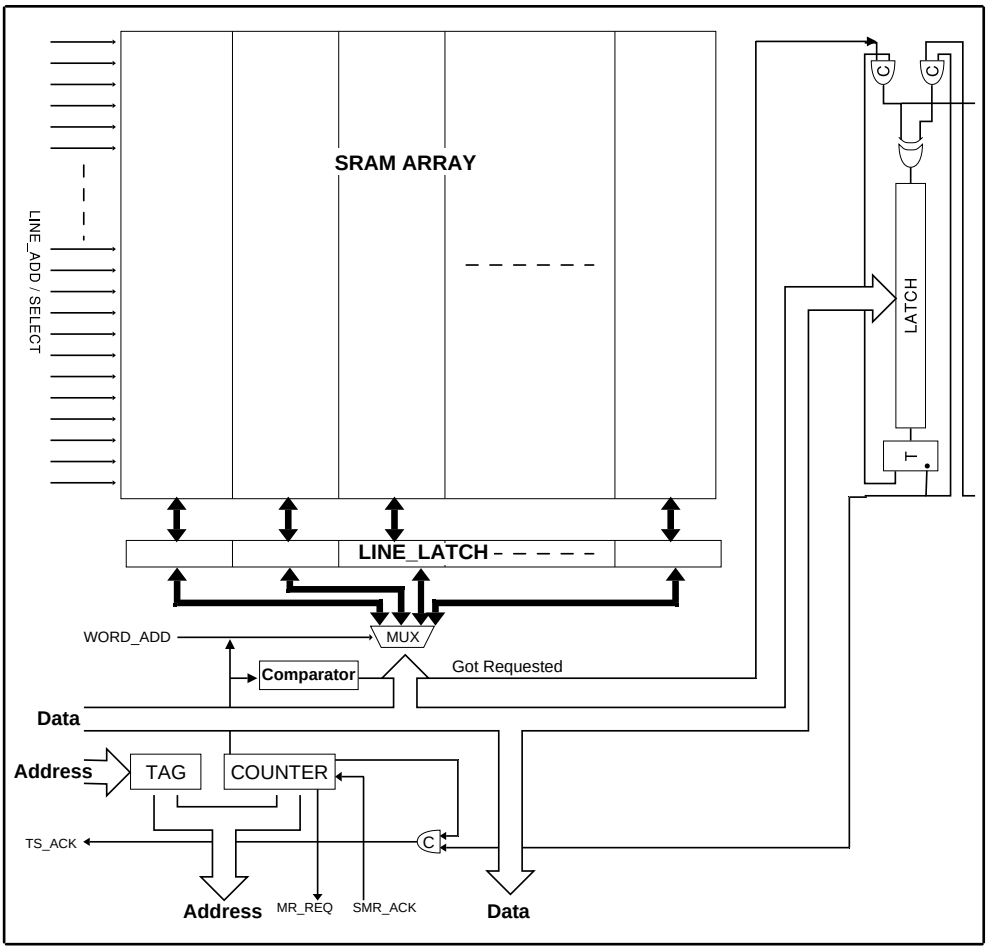

Figure 5.2 Cache Prototype: SRAM Module.

5.1.2 Results

A cache pipeline was partitioned as described in section 5.1.1. The combined, 4Kbyte cache was organised into 4 sets of 64-way associative four-word cache lines − the configuration used for the cache in the ARM3 and ARM600 series. Using random replacement, write through, demand fetch and no-allocate-on-write policies a model was constructed which was similar in structure to that used in synchronous ARM caches. A control simulation was run on this model to establish that it performed as expected − giving similar hit/miss ratios to that documented for the ARM3 cache in [FURBER89].

Figure 5.3 shows a histogram of combined instruction and data read latencies for this model. Latency is timed from the point at which the read request is presented to the first stage till the time it emerges at the output. It can be seen that there is a wide distribution in latencies which is dominated by a large peak and several lesser ones, much as predicted in section 3.1.2.

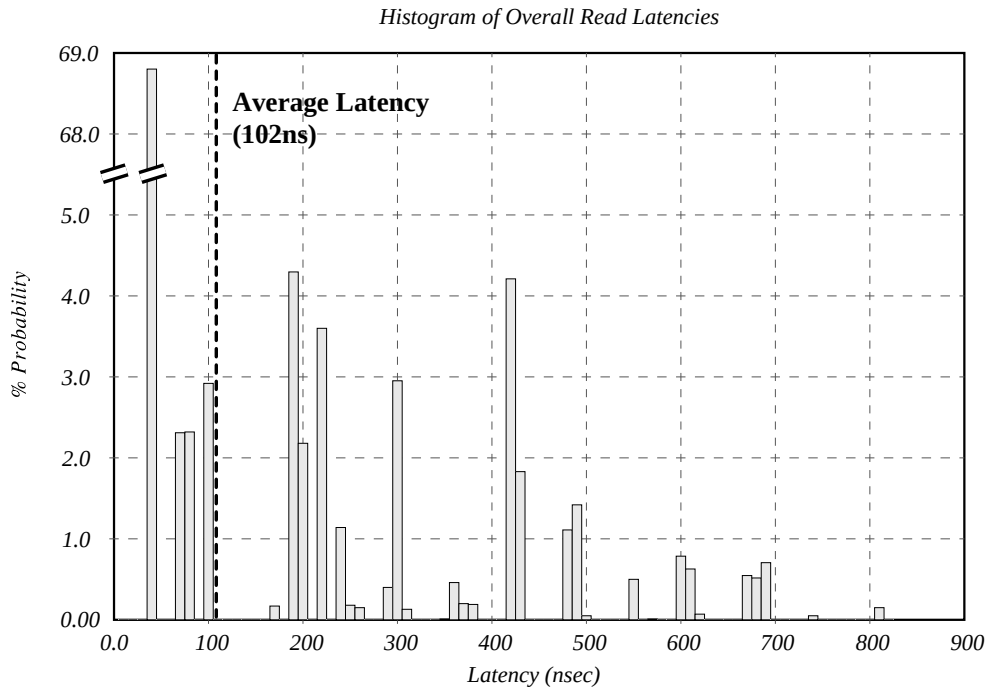

Figure 5.3 Cache Prototype : Overall Read Latencies.

The largest peak (at 40ns) coincides with majority of memory requests − read hits which are serviced quickly. The next three peaks (between 70−100ns) are read hits that immediately follow ‘writes’. Between 160−400ns there are approximately three groups of four peaks which correspond to read misses. The individual peaks in each group of four match with the word that caused the miss. There are four words in a cache line and they are fetched in strict sequential order starting with word zero. As soon as the word that caused the line fetch arrives, it is forwarded onto the next stage, thus giving four possible fetch latencies. The distance between these groups corresponds to the ‘page mode’ cycle time of the DRAM model used; set up for a sequential accesses, forced to perform a non-sequential access or recovering from a write. Also at the start of the range the distinction between and within the sets is complicated by the presence of writes in the preceding operation leaving memory in different states.

The large peak at 420ns occurs when a read hit is queued behind a read miss. In this cache organisation the hit cannot be serviced until the line fetch caused by the preceding miss has completed. Similarly the groups that tail off after this point are hits and misses identical to the ones described in the previous paragraph except that in this case they a held up by a preceding cache miss and/or write through. The latencies for such requests can quite long − causing the CPU to stall for up to a micro second as entire cache-lines are filled. Of the 4% of hit accesses that take 420ns most are likely to be on the same line as has just been fetched since majority of cache reads are for instructions and there is high degree of locality in instruction fetches. In fact most of these will be to the next sequential word which arrived a some time previously during the preceding line fetch. As this situation is quite frequent and causes a long delays to occur between sequential read

operations, there is a desire to reduce this shortcoming.

In the synchronous ARM3 cache the need to wait for the whole line fetch to complete before handling the next request is avoided. At the start of a cycle during a line fetch operation, the address of a newly requested word is compared with the last location obtained from memory for the line fetch. If they are the same then the fetched word can quickly be forwarded to the CPU. Otherwise the CPU must wait until the requested word arrives (as part of the current line fetch) or until the current line fetch finishes. Such a strategy can be employed because of the synchronous nature of operation − the rising clock edge − provides a fixed point of reference which triggers both the comparison of new request with line fetch address and the memory operation.

A similar scheme cannot be used directly in a micropipelined system because a new request and the ongoing memory operation occur asynchronously with respect to each other. To decide which of these may have exclusive access to the shared resource of the comparator requires the use of an *arbiter*. An arbiter placed between the source of new requests and acknowledges from memory arbitrates between the two asynchronous events. It only allows one request through at a time to trigger the comparator. Without an arbiter either request could trigger the comparison function and there is risk that the both events could occur in a very short period of time and hence − in a micropipelined system − *cancel each other out*.

Arbitration usually takes a small amount of time to happen when there is no contention for the resource. Unfortunately when contention arises arbitration can take an indeterminate (and potentially infinite) length of time. In a critical path such as this, contention or near contention is likely to occur. e.g. for sequential accesses for instructions from a newly fetched line. Any delays incurred due to this would disadvantageous. Thus an alternative cache organisation, not requiring arbitration, which forwards line fetch words would be preferred.

5.2 Refined Cache Structure

5.2.1 Structure

To overcome the need for arbitration an alternative design was devised. The rough structure of this is shown in figure 5.4. The cache is still partitioned into two parts; the tag store and SRAM. The majority of the cache structure and operation is the same as that described in section 5.1. However some additional structures have been added and some reorganisations undertaken; a special *fetch tag* has been added in the tag store and the SRAM stage has been sub-divided into three functional units; the main SRAM array − where most of the cached data is held, a fetch engine which performs line fetches, storing them in its line-latch and a synchronising circuit that allows requests for fetched words to be matched with the word as it arrives from memory.

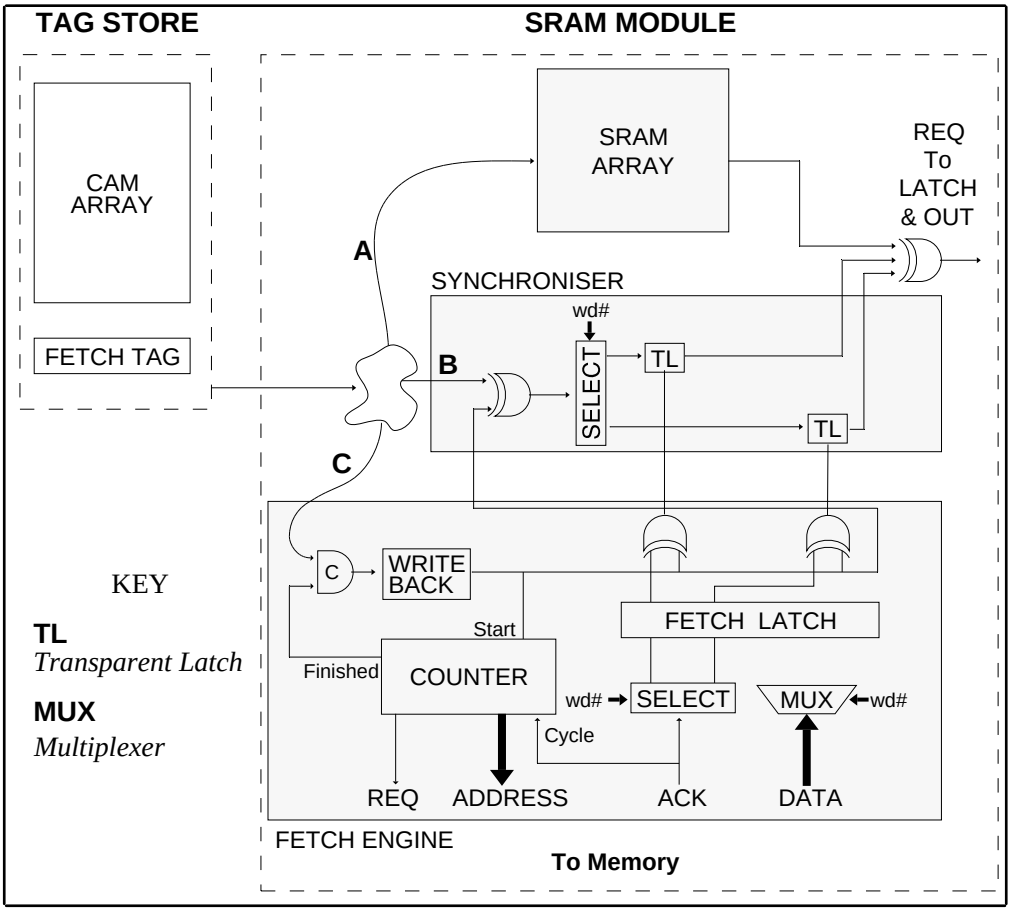

Figure 5.4 Pipelined Cache with Autonomous Fetch Engine.

The reorganisation and partitioning of the SRAM stage has meant that this system is capable forwarding other required word from the line currently being fetched without the use of arbitration. Of the three functional units in the SRAM module, only one of them can handle an active request at any given time. However the fetch unit can, once started, work concurrently with the either of the other two.

Control of which unit(s) is/are active for any given request is split between the tag store and the SRAM. A special tag location is needed in the tag store to store the address of the last line fetched. The tag comparison operation can then split into two separate or concurrent phases; a check to see if the line is in the main SRAM array − a full CAM comparison. Or whether it is the same line as the last fetch − compare against the special tag. Thus the mode of operation for each request is decided in the tag store.

If a cache read request matches with a tag in the main CAM array in the tag store then the request is directed to the array part of the SRAM stage via path **A** (by some random logic − represented by irregular shape in figure 5.4) where a normal cache line access is performed for the required word. When a read matches with the special tag, indicating that a word is in the last line fetched, then the request is sent to the synchroniser via path **B**.

The most significant activity occurs when a line-fetch is initiated. The tag store loads the special tag line with the initiating request's tag value but first it must replace a cache line by writing the old special tag value into to a main CAM array entry. This is required because the fetch latch in the SRAM stage only holds data for the line currently being fetched and if the previously fetched line is to be retained then it must be written into the main SRAM data array.

The line fetch request is directed to the fetch engine in the SRAM stage via path **C**. Here it must wait until the previous fetch has finished (the Muller-C element ensures this). Once the the previous fetch is finished the first thing that the new line fetch does is to cause the writing of the previously fetched data into the main SRAM array. It then starts the counter requesting data to fill the fetch latch and the request continues along clearing the control for the word synchroniser (resetting the state of the exclusive-OR gates). Resetting these control signals indicates that none of the line fetch words have yet arrived. The request then continues on and up into the synchroniser in order to wait for the word it requires to arrive.

The synchroniser is responsible for matching up requests with their data. It does this by using transparent latches (TL) to block the progress of a control signal until the required word has arrived. Initially, immediately after a new line fetch is started, all the transparent latches are disabled. So as a request arrives at the synchroniser it is directed the transparent latch corresponding to the word requested by the select block. Here it waits for the latch to be enabled. Meanwhile the fetch engine operation causes data to be fetched and placed into the the correct slot in the fetch latch. As a word arrives, it is directed to the correct location using the multiplexer (MUX). Word arrival also causes the state of the corresponding exclusive-OR gate to change. This in turn enables the associated transparent latch thus allowing any waiting event through to trigger the forwarding of the word to the CPU.

It can be seen that using this system a new line fetch operation starts the fetch engine and then waits for its required word to arrive. Until this point no further requests will be accepted into the SRAM stage. Upon the data's arrival it is forwarded to the CPU and the received acknowledge is steered back to the tag store stage thus allowing further requests to to be accepted into the SRAM stage. Further reads from same line now use the synchroniser and normal cache hits use the SRAM array. Any write throughs or further line fetches however, must wait till the previous one finishes.

All writes are checked against the main CAM to see if the copy in the cache needs to be altered but their requests also go via the fetch unit (path **C**) because it has the only port to main memory to accommodate these write-through operations. This means that the fetch unit is responsible for writes to the SRAM array using the same circuitry used during normal fetched line write back operations.

A point to note about the schematic in figure 5.4 is that for reasons of clarity only

two transparent latches and two exclusive-OR gates have been shown indicating a two word line. This can be extended any to line length by increasing the number of the TLs and XOR gates and by increasing the size of the line latch and degree of the select blocks.

The possibility of two asynchronous events occurring nearly simultaneously still arises in this design but in a some what altered form. It occurs when a transparent latch is enabled at almost the same instant as a new request event reaches its input. Since the transparent latch is controlled by a logic level that will not change, the event will eventually propagate through to the output after a certain setup time. The relative times of when the enable and event arrive may produce spurious levels on the output but this can be controlled by engineering the latch circuits.

5.2.2 Initial Results

A pipelined cache model partitioned as described in section 5.2.1 was constructed. Figure 5.5 shows the distribution of read latencies for cache of the same structure and employing the same policies as the synchronous ARM cache.

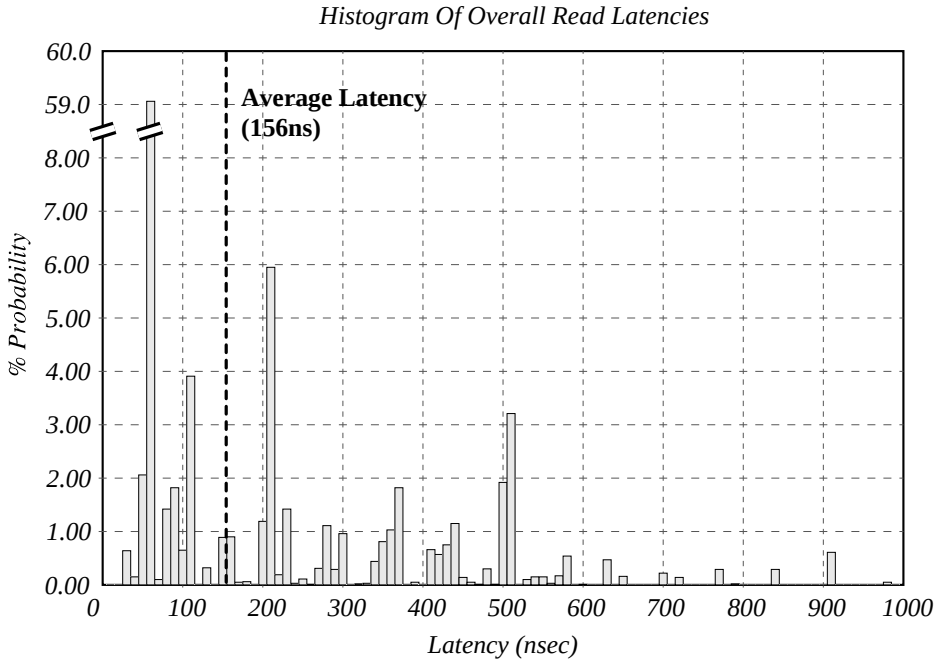

Figure 5.5 Distribution of Read Latencies
in Cache with Autonomous Fetch Engine.

There are two stages of pipelining present in this arrangement, one request in the tag store and one in the SRAM module. However it can be seen by comparing figures 5.3 and 5.5 that the average latency is in fact greater for the improved model. A better indication of the performance can be made by examining the throughput of the caches (a long waiting time indicates possible CPU stalls). Table 5.1 lists the average time spent waiting at the start of the pipeline for each cache structure; with and without the autonomous fetch engine. These figures

indicate that the throughput has in fact increased since less time is spent waiting for space in the cache pipeline.

	Initial Design	Refined Design
Mean Waiting Time (ns)	118	92

Table 5.1 Cache Designs: Indication of Throughput

The increase in latencies can be accounted for by two features; the method of latency evaluation and the increased complexity of the line fetch system. Timings for the synchronising process had to be added and this increases the latency of any request that passes through. It can also be noted by comparing figures 5.3 and 5.5 that the two read latency distributions are significantly different. Latencies in the initial design occur in a small range of discrete values; this indicates that there is a small number of discrete paths and hence different latencies that requests can have as they flow through the cache. In the refined model the interaction of line fetch operations with other requests affects the latency of many operations. The effect of this is to 'diffuse' and spread the latencies for different operations out over a wider area. For example the latency for cache hits that lie in the previously fetched line are dependent on when the line fetch started, how soon the request for the required word followed and the nature and number of any intervening requests.

It can be argued that the synchronising behaviour of this refined design is desirable in a cache. In a complete system it allows CPU operation to continue producing new requests without waiting the line fetch operation the finish. For this reason evaluation of different strategies covered in this chapter were based on additions to this design.

5.2.3 Sequential Access Support

There is considerable scope for specifically supporting sequential access modes within the refined cache design. As mentioned in section 2.3.1 the ARM asserts its *sequential pin* when an access occurs to the sequentially next memory location. It was suggested in section 4.2.4 that the state of this pin could be used to by-pass full CAM comparisons by noting that the access is sequential from the previous cache hit and that it has not 'fallen off' the end of the line and so must still be in the cache. This procedure can still be used.

Experiments were performed to judge how effective this strategy would be. Statistics were kept as to what percentage of hits occurred to same cache line as the previous access and how many had the sequential pin asserted. This should not be confused with operation of the line fetch engine which notes whether an access to the previous *line-fetched* line occurs. Preliminary results indicate that for the ARM3 organisation (four-word lines), approximately 50% of cache hits (reads or writes) occur on the same cache-line as the previous cache access and, of these, 46% were flagged as being sequential. These figures suggests that up to 40% of CAM comparison operations can be avoided thus saving significant amounts of power.

Such a scheme can be added to the refined cache structure by placing it up-stream of the CAM and (possibly) the special tag comparisons. Thus if the previous cache operation resulted in a read hit or line-fetch, and the current access has its sequential pin asserted, then (allowing for finite line-lengths) either the CAM or special tag comparisons can be avoided. If the previous operation was a line fetch, the request is directed to the synchroniser and if the previous operation was a regular cache hit then the request is directed towards the SRAM data array. If the previous access is not a cache-hit and/or the current access is not sequential then the request must be compared to the special tag and then the CAM array to determine the course of action. The full effectiveness of this strategy has yet to be determined. A more detailed study of the VLSI structures involved in implementing it would have to be addressed in order to discover if there is any resultant degradation in system throughput.

5.3 Adding a Write Buffer

5.3.1 Idealised Buffer

An experiment was performed to judge the possible effects of a write buffer on cache latency and throughput. All writes to main memory were deemed to take zero time thereby modelling an idealised write buffer of infinite depth. Figure 5.6 shows the read latency distribution for this arrangement once again using the same structure and strategies employed synchronous ARM cache.

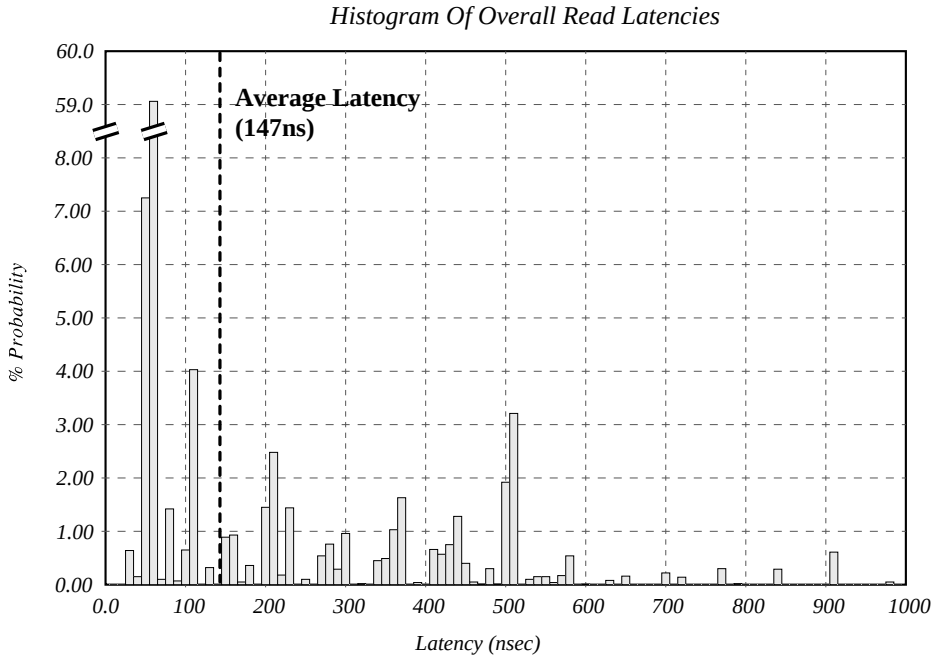

Figure 5.6 Cache with Idealised Write Buffer.

As can be seen the distribution is almost identical to that in figure 5.5. The major difference being that the large peak (and portions of surrounding ones) at 210ns (in figure 5.5) have moved to 50ns (in figure 5.6) as writes stop affecting the

response time for read operations. Consequently the average read latency drops from 156ns to 147ns − a 5.77% improvement. Examining the improvement in throughput reveals that the average waiting time for memory requests goes from 92ns in the unbuffered cache down to 87ns in the buffered − a 5.43% improvement. It should be noted that this is the improvement is for an idealised write buffer in a situation where the cache is (unrealistically) in constant use. This write buffer does not impose penalties when it has to be flushed to memory before a line fetch operation can take place. Neither is a time penalty associated with writing to the write buffer. The model could be improved to incorporate these features but its effectiveness is still largely dependant on internal CPU activity.

The reason that the addition of a write buffer seems to make little difference to the average latency is due to the way performance is measured. Consider the two stage cache pipeline with a write occupying the SRAM stage. The write is held in the fetch unit and is being driven out to memory. In a cache without a write buffer it will remain occupying the SRAM stage until the write completes. Any read request waiting in the tag store stage behind will thus also have to wait. If the following request is a read then waiting for the write to complete will cause its read latency to increase but this is the only request whose latency is affected. Any further requests are not allowed into the start of the cache pipeline and are therefore not time stamped until the write has completed and relinquished a slot within the pipeline. Adding a write buffer to this simulation therefore only affects the latency of any read requests immediately following a write since this the only request type that is speeded up by a write operation completing early.

In a real system the latencies of many following requests could be affected. In situations where a small groups of writes are interleaved with a large number of cache reads, the write buffer could improve the performance. All the writes could be held in the buffer and then be written out whilst the reads were satisfied from within the cache. The degree of improvement may not be as great as that obtained when adding a buffer to a synchronous cache. This is because a degree of buffering already exists in the micropipelined implementation as the processor core treats writes as ‘fire and forget’ operations. However the 6% improvement over an unbuffered cache does indicate the improvement that can be potentially be gained by adding a write buffer. Any performance gain would be limited by a combination of the depth of the write buffer, the depth of the pipeline preceding the buffer and the CPU’s ability to decouple reads from usage.

5.4 Prefetching

The need to design a cache which was capable of forwarding words as they arrive during line fetch operations resulted in the design described in section 5.2.1. This design contains an autonomous fetch engine which, when combined with elasticity inherent in micropipelines makes implementation of prefetch strategies relatively simple. A simple always prefetch can be implemented by altering the tag store stage and small changes to the SRAM stage.

In the tag store, when a new request enters and is identified as a candidate for initiating a prefetch, first the original request can be sent to the SRAM stage. Then, once acknowledged, rather than accepting a new request, a check can be made to see if the line to be prefetched is already in the cache. This is accomplished by an internal CAM look up operation against the new line address. If it is currently held then a nothing further need be done. Otherwise a request to the fetch unit within the SRAM stage can be generated. Additional information has to be sent at this stage to indicate that this is in fact a prefetch and the there is no need to forward a particular word to the CPU. Once the prefetch operation has started and the data from the previous fetch has been written back, the request that started the prefetch can be short circuited and fed back to the tag store instead of being steered to the synchroniser. Thus prefetch is effectively accomplished by injecting an extra request at the tag store stage and removing it again at the SRAM stage.

When considering the effectiveness of a prefetch policy, certain statistics prove to be useful; the *miss rate* which indicates if the effectiveness of the cache has been increased. The *prefetch test rate*, a figure that indicates what percentage of cache read operations resulted in an attempted prefetch. The *prefetch rate*, illustrating what percentage of the cache read operations actually resulted in a prefetch and the *prefetch ratio* which shows what percentage of prefetch tests resulted in prefetches. These prefetch metrics were used to compare several prefetch policies that were implemented by employing the autonomous fetch engine. Table 3.1 summaries the results for the basic strategies evaluated. This section describes the performance of each policy in turn and the rational that motivated it.

	Miss Rate %	Prefetch Test Rate %	Prefetch Rate %	Prefetch Ratio %
No Prefetch	5.52	—	—	—
Any Data Read	5.19	16.23	1.37	8.44
SRAM Array Hit	5.22	15.14	1.33	8.78
Seq. SRAM Array Hit	5.23	6.24	0.64	10.26

Table 5.2 Summary of Statistics for Prefetch Policies.

The model employed to compare various prefetch strategies uses the same standardised cache structure used in the synchronous ARM caches. Prefetching is only considered for data reads. This was done because, though instructions tend to reflect a higher degree of sequential locality then data, there is already some prefetch activity for instructions. This is done by the prefetcher within the processor, which fetches instructions sequentially until its prefetch buffer is full (currently three entries). On every change of program flow the prefetch buffer is emptied and the fetching begins at the new target address. This results in closer correspondence between instruction prefetch activity and actual instruction usage than could be implemented externally by the cache.

The first strategy considered attempts a prefetch for the next cache line on every data read operation. This is results in an excessive amount of prefetch activity with a prefetch test performed for 16% of all read operations (i.e. for this benchmark 16% of reads are for data and 84% are instruction fetches). The prefetch activity represents almost one in six extra internal full tag store CAM comparisons in order to determine if a prefetch is required or the next line already resides in the cache. This comparison overhead is clearly too high although the miss rate decreases (as compared to no-prefetch) indicating that prefetching is successful, but there is still ground for some improvement.

Reducing the number of prefetch tests whilst keeping the number of actual prefetch operation relatively high would increase the efficiency of the policy. In the above strategy a prefetch is attempted on every data read even if the read itself is a miss. Consider the situation when a request for a word in the bypass unit arrives in the SRAM stage. This can either be as a request for another word in the previously fetched line or as a new miss that requires the use of the line fetch engine. Normal operation continues when the line fetch caused by the original request has completed. Any following request that needs to read a word from the same line will have to wait whilst the originally requested line is fetched in full, written back to the SRAM array plus any time spent starting the prefetch. The problem is worse if a following request also requires a line fetch − it will, in addition, also have to wait for the prefetch to complete.

To reduce the possibilities of interfering with normal operation, prefetches should be avoided if it is known that the fetch engine is already in use. This is the approach that was taken in the second prefetch policy − ‘Prefetch on SRAM Hit’. A prefetch is only attempted when the current data read occurs from the main SRAM array. This causes the miss rate to increase marginally as compared to prefetching on any data read but it is still better than the no prefetch case. The rate of prefetch tests is still relatively high and the proportion of these tests that actually result in a prefetch is low.

A greater improvement in the latencies may be obtained if the ratio of prefetch tests to actual prefetches is increased. This would mean that less time would be spent performing needless internal CAM comparisons that do not lead to the prefetch taking place. Attempting a prefetch only when accessing specific words within the SRAM line might facilitate this. Also assuming that prefetched data will be used in the near future, it is interesting to see how the prefetch rates vary with the word fetched (see table 5.3).

	Miss Rate %	Prefetch Test Rate %	Prefetch Rate %	Prefetch Ratio %
Word 0	5.28	4.43	0.62	14.00
Word 1	5.21	3.47	0.47	13.54
Word 2	5.24	3.78	0.44	11.64
Word 3	5.23	3.49	0.36	10.31

Table 5.3 How Statistics Vary with Word Accessed for 'Prefetch on SRAM Hit' Policy.

The miss rates increase a little but the number of prefetch tests are greatly reduced and the ratio of test to actual fetches is increased. The variance in miss rate with the word accessed is non-uniform, with prefetches triggered by word one accesses resulting in a minimum miss rate. Spatial locality would suggest that words towards the end of the line would progressively give better improvements in miss rate. Since this does not occur a better method for reducing the overhead of prefetch tests must be found.

If prefetched data is not requested soon (if at all) then there is a chance that the line will be replaced and even if it is present then there is also an increased likelihood that the prefetch caused the ejection of another line that was required. A method to maximise the probability that the prefetched data will be used immediately or soon is to monitor the previous data read activity and identify when incremental reads of memory are occurring. This is cumbersome and inefficient to do on an on going basis. Fortunately the sequential pin can used to monitor approximate pattern of memory accesses. It is asserted when the current access is to a location sequentially following the preceding location thus, assuming the pattern continues, there is a increased likelihood that the next few locations will also be required.

The final prefetch policy uses the state of sequential pin to decide if a prefetch should occur. The last line in table 5.2 shows the miss, prefetch test and actual prefetch rates for attempts triggered by sequential data reads from any word within the SRAM array. This results in a slightly higher miss rate but the overhead of extra CAM comparisons and number of actual fetches has been halved in comparison to any previous policy. Table 5.4 breaks down these figures on a word by word basis to discover if the word position governs the likelihood of a fetch being useful.

	Miss Rate %	Prefetch Test Rate %	Prefetch Rate %	Prefetch Ratio %
Word 0	5.28	1.42	0.25	17.79
Word 1	5.27	1.47	0.27	18.48
Word 2	5.28	1.48	0.28	19.25
Word 3	5.27	1.90	0.24	12.87

Table 5.4 How Statistics Vary with Word Accessed for 'Prefetch on Sequential SRAM Hit' Policy.

It can be seen that miss rates for individual words is uniformly higher when compared to the break-down in table 5.3. Thus there is little to gained from instigating prefetches only on tail-end word accesses. Further, with reference to tables 5.2 and 5.5 it can be seen that only half the number of prefetches were triggered in the policy employing the sequential pin and this did not have as severe an effect on the miss rates as could be expected. This can be explained by the nature of the synchronous memory reference trace. Sequential data fetches only occur during load multiple instructions since normally data references are interleaved with instruction fetches. This is because the ratio of memory references per instruction is less than one for all but load store multiples. Even in the asynchronous version of the CPU, both **sequential** and not **opcode** can only be asserted simultaneously during load multiples since **sequential** is only generated when the address originates from the address incrementer.

5.5 Cache Structure

In this section general trends in read latency and miss rates are observed for differing cache sizes and structures. The graphs presented are based on multiple cache simulation runs. Since these can be quite time consuming to run, smaller than usual memory reference traces were used (approximately one million references long).

5.5.1 Associative Caches

Previously discussed evaluation of the performance of different strategies has been based on the standard ARM3 cache structure. This structure is highly associative in nature and so it was decided to investigate the performance of generic associative caches. First a specific feature of the ARM3 cache was examined − the relationship between cache line length and the degree of associativity within a set. The ARM3 cache has split the associativity into four sets of 64 entries each but was originally based on 256 associative entries in one set.

Investigations were performed into the consequences of varying some of the cache parameters. Figure 5.7 shows the effect on miss rate of altering the cache line length. This was done at the cost of varying the number of sets so that the total size of the cache remained constant. For example the first point plotted is for

a 4Kbyte cache divided into 16 sets of 64 one-word entries. The next has 8 sets of 64 two-word entries and so forth until the last one which has one set of 64 entries which are 16 words in length.

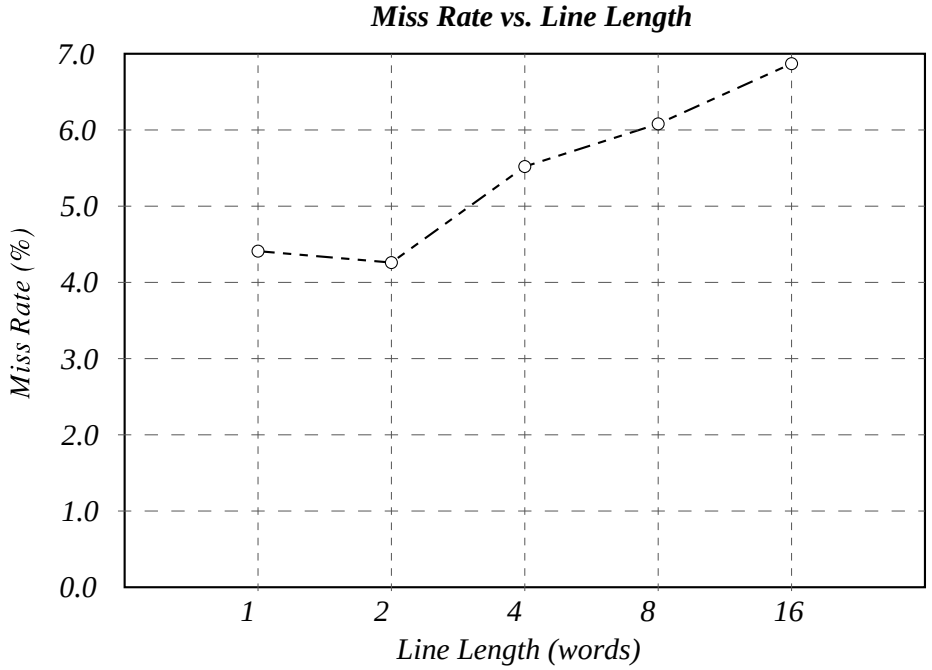

Figure 5.7 Effects on Miss Rate when Varying Line Length

It can be seen from the graph that a minimum miss rate is observed when the line is two words, either side of this size the miss rate increases. The small performance degradation for one word lines can be ascribed to the fact that each line fetch brings in no extraneous data whereas with a two-word line fetches an extra item thus two-word lines are able to exploit the spacial locality of reads. The increase in miss rate for lines longer than two-words is justifiable because as the line size is increased then the total associativity (number sets * set size) decreases since the number of sets decrease. This lowers the ability of the cache to hold large amounts of spatially distant data. With a small cache the occurrence of conflict misses increases as newly fetched data causes other, needed lines to be ejected. Figure 5.8 (read latencies for same cache structures as in figure 5.7) also suggests that data is too widely separated to benefit from being fetched in a single cache line.

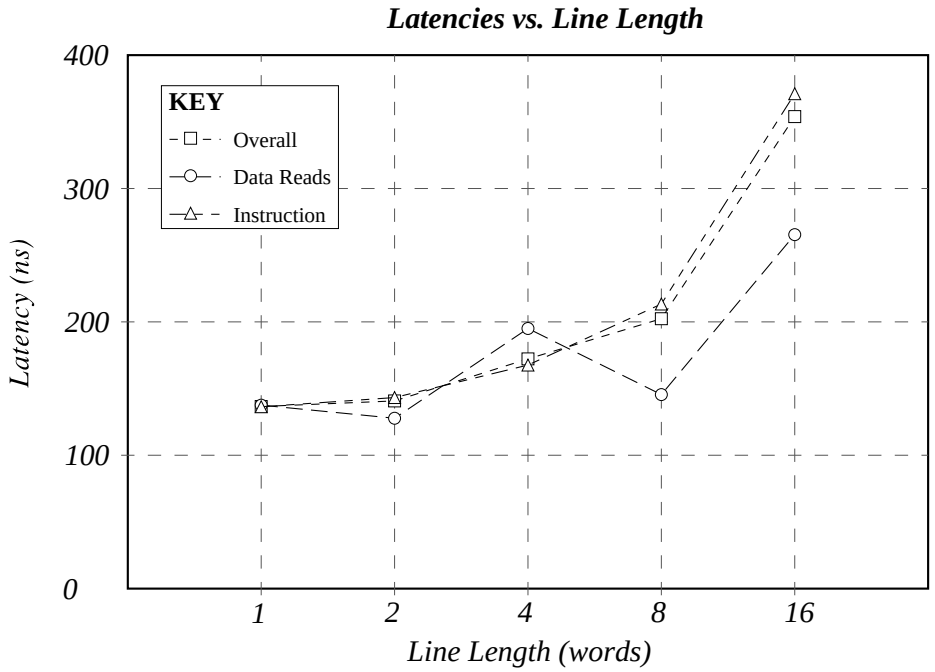

Figure 5.8 Effects on Read Latencies when Varying Line Length

The anomaly that occurs in figure 5.8 for data read latencies for a four-word cache line may account for the sudden increase in miss rate obsereved in figure 5.7 but this may be an artifact of the benchmark used. The fact that the latency returns to its original trend at eight-word lines may indicate that the sudden increase was just due to a pathological case. This situation should be unlikely to occur when random replacement is used as in this example cache. On investigation it was found that the data latency did increase in an anomalous manner for the particular benchmark used. Simulation using other memory traces resulted in gradual increase in read latency and miss rate. It should noted that with the alternative trace the minimum miss was still obtained when a two-word cache-line was employed.

General performance trends for associative caches were also investigated. Figure 5.9 maps overall read latencies for fully associative caches of varying sizes. For this figure the entire associativity was kept in a single set − reduction in the CAM size being facilitated by an increase in line lengths.

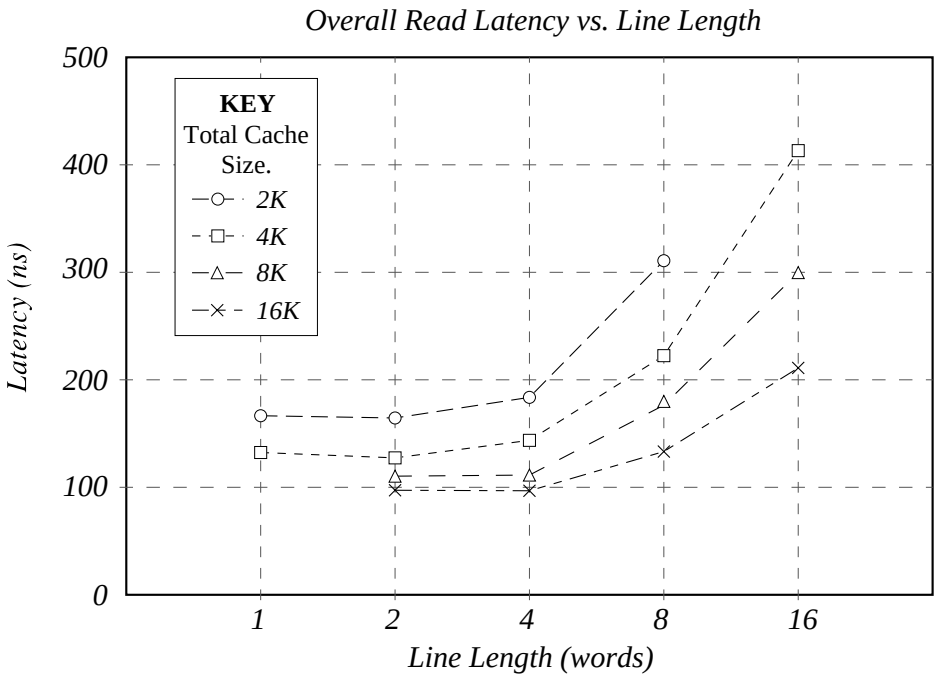

Figure 5.9 Fully Associative: Read Latencies vs. Line Lengths

Even using the technique of increased line length, the number of CAM entries that are required for the large caches is still excessive. For example the 8Kbyte cache with four-word lines requires 512 CAM entries and 16K, two-word one needs 2048. This may well be prohibitively expensive in silicon area − CAM cells are larger than SRAM cells and would use a significant area that could be better used as SRAM data store. Thus it was decided to examine more space efficient direct mapped alternatives.

5.5.2 Direct Mapped Caches

Direct mapped caches almost always have a worse hit ratio than similar sized associative counterparts, but as can be seen from figure 5.10 as size increases their miss ratio decreases. Note that increasing the line length also decreases the miss ratio, but the miss ratio alone can be a little misleading. The cache simulated in this example uses the autonomous fetch engine and any hit/miss decision is taken in the tag store stage. Thus a request that requires a word from the line currently being fetched is classed as a hit because storage for it has been allocated even though the word may not have been fetched as yet. This can result in an artificially low miss rate but with a correspondingly longer average read latency.

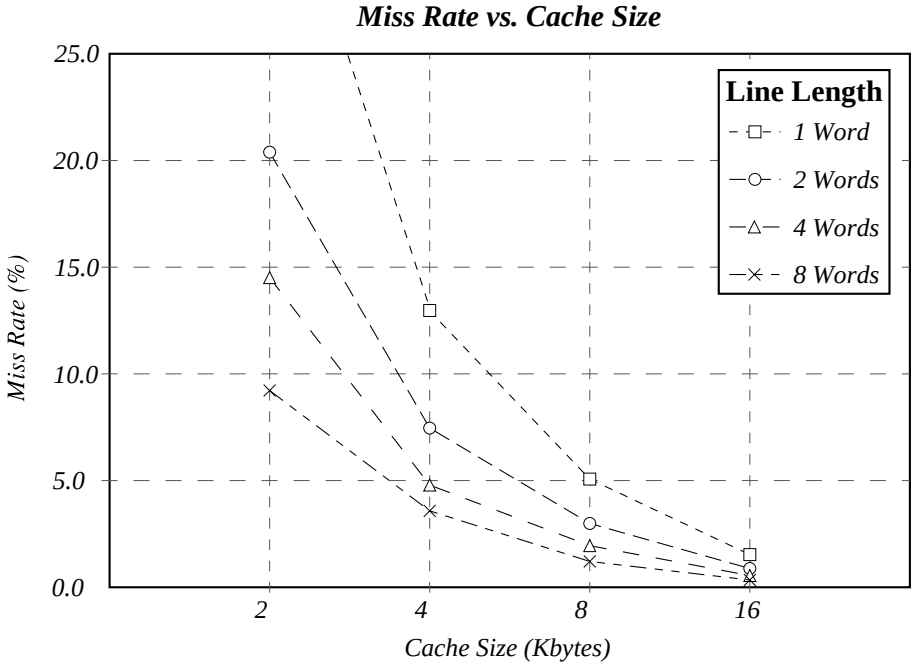

Figure 5.10 Direct Mapped: Miss Rate vs. Cache Size

Figure 5.11 provided a clearer picture of the effect of increasing line length for a direct mapped cache − it shows how the average read latency varies with line length. The line for the 2Kbyte cache shows the same characteristic kinks as the miss ratio one for the 4Kbyte associative one indicating the same pathological behaviour. Some interaction between a program's pattern of memory accesses and the structure of small caches can be expected but this should disappear as the cache size increases. Using alternative memory reference traces the read latency for the 2Kbyte cache with four-word line length was seen to decrease resulting in a more distinct 'cup shape'. The minimum point for the average latency still occurred for two-word lines. The lines for the larger cache sizes also exhibit a shallow cup shape with longer latency for both small and large line lengths.

For the caches larger than 2Kbytes the latency does not seem to increase excessively as the line lengths increase. This is probably due primarily to the fact that larger caches have a lower miss rate and so most of time is spent satisfying requests from within the cache, thus the mean time between cache misses increases. This allows the cache to satisfy concurrently requests from the main SRAM array whilst it finishes fetching a cache line, effectively hiding most of the time for long line fetches.

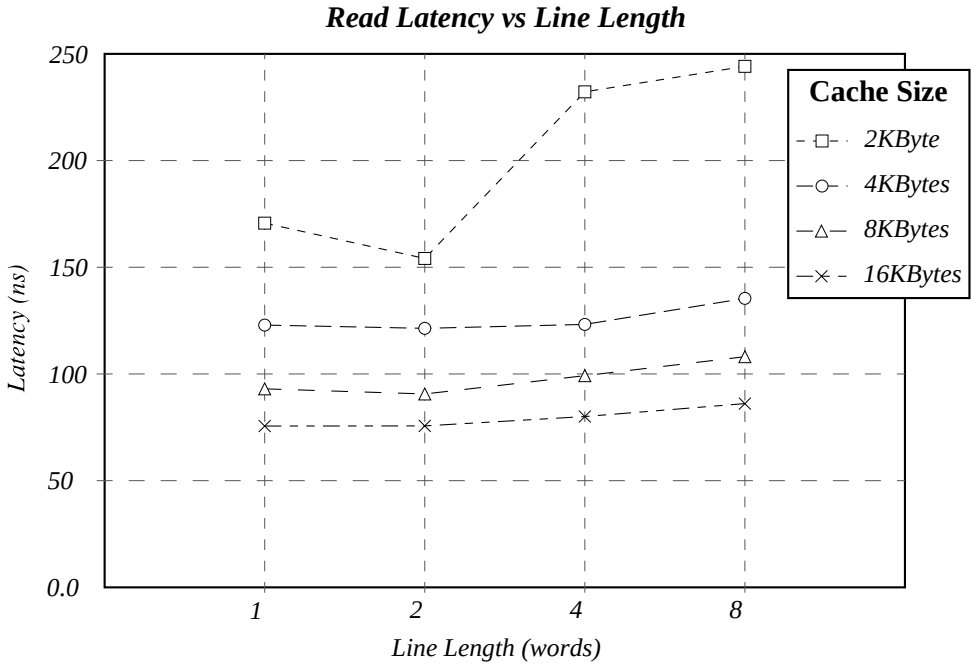

Figure 5.11 Direct Mapped: Read Latency vs Line Length

5.5.3 Two-Way Set Associative

As mentioned, generally, associative caches out perform their direct mapped counter parts but it has been found that even a small degree of associativity improves the performance of a direct mapped cache. Thus it was decided to investigate the effect of introducing two-way associativity to the caches discussed section 5.5.2.

The cache modelled has a structure as depicted in figure 2.8, the address tags are grouped into two columns of SRAM. Low order set bits select two entries either of which can hold the required location. When the need to replace an entry arises, the one selected is based on the one that was least recently used (LRU). This can be implemented by the simple addition of a single extra bit (LRU bit) associated with each tag. Each time data is read, the matching tags LRU bit is set and the other is reset. Thus on a replacement the least recently used one is the entry with a unset LRU bit.

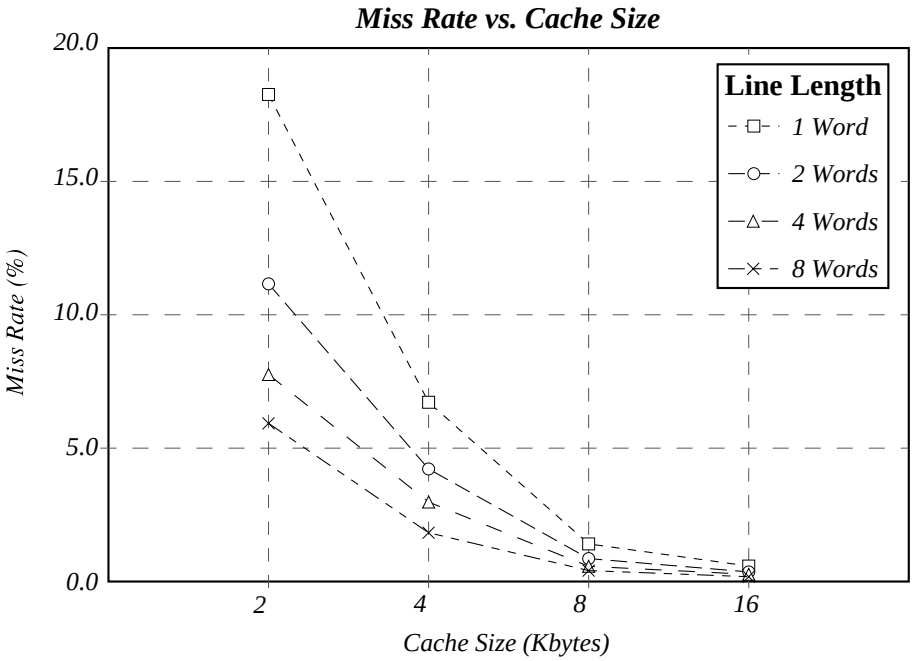

Figure 5.12 Two-Way Set Associative: Miss Rate vs. Cache Size

Figure 5.12 show how the miss rate varies for different sizes of two-way set associative caches with LRU replacement. Note that on this occasion miss rates for small cache sizes start lower than the direct mapped equivalent but then proceeded to improve at the same rate. As mentioned in section 5.5.2, miss rates can not be relied on in this case and so average read latencies must be used.

Figure 5.13 show how the average read latency varies with differing line length in a two-way set associative cache. All cache sizes show the cup shaped latency curve. The one for the 2Kbyte cache is distinct and deep whereas the one for larger caches are shallower. This behaviour is consistent with that observed for direct mapped caches but successive improvements in latency reduce more quickly as the total cache size increases.

The distinctive ‘cup-shape’ is characteristic when altering the block size (see section 3.1.1). For short cache lines/block sizes the miss rate starts high. As the line length is increased, the miss rate drops since each miss now brings in more spatially close data which also likely to be required. As the line length is increased further however, the caches ability to hold many disconnected areas of memory is reduced as the number of cache lines has been traded for longer lines. Also the cache is now committed to spending greater periods of time fetching these longer lines. This means that if a subsequent memory request results in a cache miss and requires a line fetch, then it must wait till the previous fetch has finished thus increasing the average read latency.

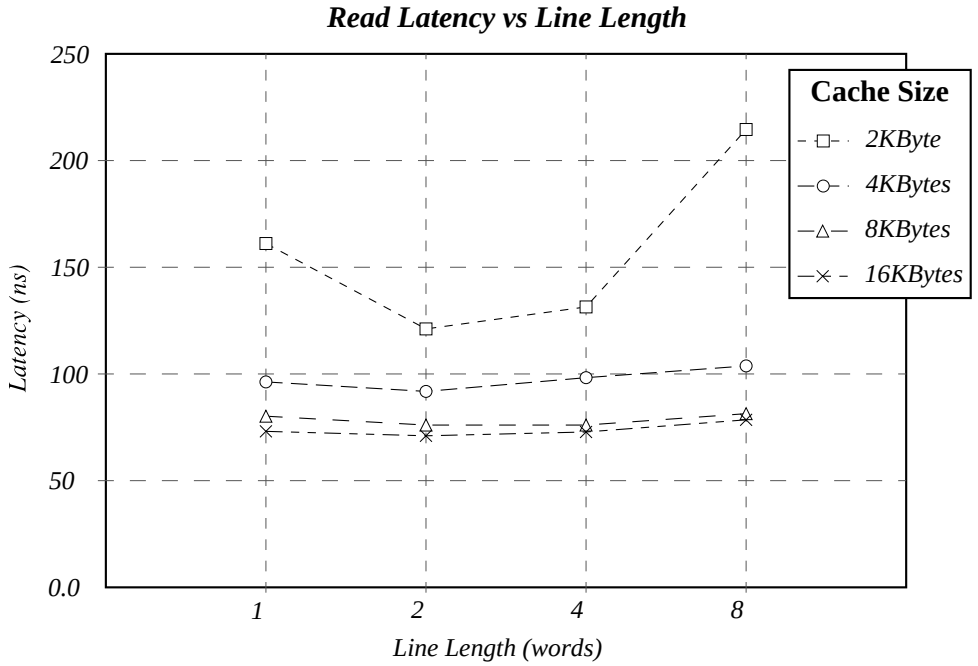

Figure 5.13 Two-Way Set Associative: Read Latency vs. Line Length

Chapter 6

Conclusions

In this chapter the effectiveness of the simulator and micropipelined event-based simulation methods is discussed, paying particular attention to its modularity and implementation. Next the applicability of the metrics used in performance evaluation is addressed. Section three presents recommendations for effective cache structures and strategies based on the results presented in chapter five. The reasoning behind these recommendations is explained. The chapter closes with suggestions for immediate work that should be done and makes suggestions as to longer term goals for micropipelined design.

6.1 Simulation Methods

Using a modular approach for the simulator design allowed for modular cache designs which correspond well with the principles of the micropipelined methodology. Simulated systems are easy to construct by piece-wise addition or replacement of modules using previously written code. C was chosen as the language for implementing the both the basic framework of the simulator and the simulated systems; this resulted in reasonably fast and efficient simulation models. Using C meant that the degree of modularity had to be sacrificed for an increase in efficiency. The use of an object orientated language (eg. C++) may have aided a modular design strategy without compromising efficiency.

On the whole the simulator was adequate for the task of simulating a micropipelined cache. There were however, a few difficulties in modelling the activity within such caches. Concurrency is a major feature of pipelined operation and it was necessary to simulate this behaviour in the single threaded simulation environment developed. Writing such code and debugging faults within it can be problematic. During development large amounts of debugging code was required to monitor the progress of individual modules within the cache. This produced long execution traces within which it was time consuming and difficult to locate logical errors in the design of the modules. Use of a language with built in support for concurrency and with associated debugging tools would be better suited to modelling an extensively micropipelined system. Such languages (eg. Occam) have been designed for multiprocessing and can run on multiprocessor systems; a simulation would therefore also have the opportunity of running faster.

Trace based simulation was a reasonable compromise since a suitable asynchronous micropipelined model of the CPU core did not exist. A complete CPU-cache model would exhibit all the interdependencies found in a real system but would consequently be slow to run. Deterministic memory reference traces

obtained from models of synchronous ARMs are an adequate substitute for better, more realistic traces from asynchronous CPU models. The synchronous traces allow preliminary work to be done until the better traces become available.

The primary motivation for obtaining better (asynchronous) traces would be to allow more accurate examination of the temporal distribution of cache requests. Memory references generated by an asynchronous processor would be distributed over time in a much less regular fashion than those from a synchronous processor. The time interval between references is flexible and is not easily determined, unlike traces obtained from synchronous processors where the requests appear only on clock edges. The interval and any dependencies between requests is significant since free time can be utilised by the cache to perform other tasks − including; resetting latching circuits, precharging CAM/SRAM arrays and completing line fetch or write operations so that the response time for following requests is reduced.

Once obtained, memory reference traces that incorporate these timings can easily be accommodated whilst simulating cache models. A request would not be injected into the pipeline until it actually should occur, i.e. if a request was generated 30ns after the previous one, then it will not be presented to the cache for at least 30ns after the preceding one, even if space becomes available; the current model attempts an immediate request. If space in the pipeline does not become available in this time, then the cache has imposed an extra delay on the processor which must be propagated onto the following requests.

6.2 Performance Evaluation Methods

In lieu of anything resembling a fixed cycle time, evaluation on the basis of actual time to perform operations is justified. The timing of different operations within a micropipelined stage was, whenever possible, based on existing VLSI structures but often had to be estimated. However modularity in the simulator once again allows for the back annotation of more accurate timing figures as they become available. These may arise as the detailed structure and VLSI layout for a circuit is performed. Such exercises in layout would also give an indication of area and power usage.

Assuming that the internal timing of operations was reasonable, obtaining distributions of latencies for a cache organisation gives a good idea of expected performance. How the distribution varies when strategies and structure are altered gives an insight into how a real system behaves and how various techniques affect performance. However it is apparent that there other factors that influence the overall performance of micropipelined caches. The primary one is the average throughput which, when multiplied by the number of memory references indicates the total execution time for a job. Such a figure is of primary importance to the end user. It does not on its own however, provide enough information with which to design and refine cache structures. For such tasks the distribution of read latencies is more applicable since, using it, short comings in the design can be

identified. Once solutions to overcome these have been found a new model can be simulated, the new throughput measured and the results used to refine the cache structures further.

Simulation and evaluation methods are also useful for characterising the effects of different sub-systems. By altering the times for different blocks, perturbations can be observed and the effects of fast/slow sub-systems can be quantified. In this way one can judge the relative benefits of different implementations, trading size and complexity for speed or adding alternative functional units which can be used to minimise power consumption for example.

6.3 Recommended Cache Strategies

A cache structure was partitioned into functional units and the operation of its individual units modelled. These were incorporated into a single executable cache simulation which was then executed using a typical address trace as input. This resulted in production of various metrics and distributions indicating performance. As mentioned, the read latency histogram identifies causes of potential bottlenecks and the throughput rate gives general performance metric.

Using these it was demonstrated that it is desirable to have a cache that is capable of performing a line fetch concurrently with other cache activity. Ideally it should also be able to forward any other required words from the current cache line as required. A cache featuring such functionality was designed and a model reflecting this new behaviour constructed. The refined cache simulation was executed using the same memory reference trace as earlier and the output metrics noted. These show that its throughput had increased indicating a better organisation. Its read average read latency had also increased however, indicating that the extra logic and time required to implement the desirable fetching strategy incurred longer overheads.

Based on the improved cache partitioning, write buffering and several different prefetch strategies were investigated. The results indicated that buffering writes did not produce a significant improvement in this cache model, since a degree of buffering is already provided by the micropipelined nature of the cache-CPU system. An investigation of prefetch strategies was felt to be advantageous because the refinements added to facilitate autonomous line fetch activity meant that prefetch operations could easily be added. Prefetches may occur without affecting regular cache activity excessively, thus gaining maximum benefit from the available resources.

When judging the effectiveness of the different prefetch strategies other, more common metrics such the as miss ratio were used. Different prefetch strategies were iteratively refined, trying to minimise the miss ratio and average read latencies. The effect of employing these can not be fully appreciated at present because current model does not allow for any ‘free’ time when the cache is inactive − not having any requests to satisfy; any prefetch will always affect

throughput and the latency of following operations. The most promising policy investigated, which utilises features and signals present in the ARM architecture, is to instigate a prefetch on ‘any sequential data read from the SRAM array’. This lowers the miss rate whilst not incurring the overhead of excessive prefetch tests that do not actually result in the prefetch occurring. It also avoids contentious use of the fetch engine since the CPU is currently sequentially reading cached data and is likely to continue doing so.

Finally different cache organisations were tried whilst using the same functional units within the cache pipeline. These experiments investigated the effects of using different sizes of cache, faster and more space efficient cacheing structures such as direct mapped and dual-set associative caches. These experiments indicated that, although, at the small sizes previously considered, the highly associative caches make efficient use of resources, better performance can be gained by employing large, set-associative ones. Based on the results of the experiments performed, the best compromise solution would be to use a ‘8Kbyte dual-set associate cache with LRU replacement and four-word cache-lines’. This is fast (SRAM based), a reasonable size − low numbers of tags per data item (employs four-word cache lines) and quite easy to implement. Such a cache should also use the autonomous line fetch engine and synchroniser thus facilitating concurrent activity and allowing fetched words to be forwarded as required.

As a side effect of performing the cache size/structure experiments, possible pathological case behaviour for small cache sizes was noted in the memory reference traces used throughout this thesis. The effects of this will have to be investigated further.

6.4 Future Work

The primary work that still has to be done is the extension of the simulator to incorporate some of the behavioural characteristics of a complete micropipelined CPU-cache system. This can be done in two ways; either by including timing and data dependency information in more accurate memory reference traces or by including rough models of the CPU datapath. Metrics and statistics gathered from this extended simulator with increased functionality would improve the prediction of actual system behaviour. Refinements to the cache system could now be investigated more thoroughly and in greater detail noting how changes affect the throughput of the system as whole.

Coupling the extended simulation model with back annotation of more accurate timing information derived from VLSI layout of cache structures would further refine the details of the results. Trends in power consumption figures could then be observed in more detail thus enabling more informed decisions to be made before committing new designs to silicon. This would also allow more detailed evaluation of power reduction strategies such as specific support for sequential accesses to reduce unnecessary tag store activity (see sections 4.2.4 and 5.2.3).

As a long term goal, incorporation of a micropipelined memory management unit would also significantly benefit an asynchronous microprocessor-cache combination. Pipelining and combining the MMU and cache functions could result in significant performance improvements since it would then be possible to combine address translations with cache hit/miss determination in one or two pipeline stages. The cached data access could then occur in further pipelined stages. This would result in a system that was capable of high throughput for the majority of operations. When page faults or cache misses did occur the micropipelined nature of the system would be capable of buffering some of the effects of these from normal CPU operation. Indeed the concept of a micropipelined cache with dynamically flexible response times could be extended to multi-level cache hierarchies to improve performance further.

If correctly integrated, such a device could provide a complete asynchronous solution to areas where traditional clocked components are usually used. System designers would then be free to utilise these asynchronous sub-components to obtain higher performance per unit power for use in devices that require these characteristics.

Appendix A

References

[AGARWAL87] A. Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* Technical Report Nº. CSL-TR-87-332, Stanford University, May 1987.

[ARM600] *ARM600 Datasheet.* Advanced RISC Machines Ltd. Cambridge England, 1991.

[BERKEL88] C.H. van Berkel and R.W.J.J. Saijs. *Compilation of Communicating Processes into Delay-Insensitive Circuits.* Proceedings of IEEE International Conference on Computer Design, ICCD-1988.

[BORG90] A. Borg, R.E. Kessler and D.W. Wall. *Generation and Analysis of Very Long Address Traces.* ACM SIGARCH Computing Architecture News, Vol 18, Nº. 2, June 1990, pp270-279.

[BRAY91] B.K. Bray and M.J. Flynn. *Write Caches as an Alternative to Write Buffers.* Technical Report Nº. CSL-TR-91-470, Stanford University, April 1991.

[BRUNVAND89] E. Brunvand and R.F. Sproull. *Translating Concurrent Programs into Delay-Insensitive Circuits.* In ICCAD − 1989.

[BRUNVAND92] E. Brunvand, N. Mitchell and K. Smith. *A Comparison of Self-Timed Design Using FPGA, CMOS and GaAs Technologies.* Proceedings of IEEE International Conference on Computer Design: VLSI in Computers & Processors, October 1992, pp76−80.

[CHAPPELL91] T.I. Chappell, B.A. Chappell, S.E. Schuster, J.W. Allan, S.P. Klepner, R.V. Joshi and R.L. Franch. *A 2-ns Cycle, 3.8-ns Acess 512-kb CMOS ECL SRAM with a Fully Pipelined Architecture.* IEEE Journal of Solid State Circuits, Vol. 26, Nº. 11, November 1991, pp1577-1585.

[DEC92a] *DECChip 21064−AA RISC Microprocessor Preliminary Data Sheet.* Digital Equipment Corporation, Maynard, Massachusetts, April 1992.

[DEC92b] D. Meyer. *Alpha Architecture: Hardware Implementation and Software Programming Implications.* Proceedings of IEEE

International Conference on Computer Design: VLSI in Computers & Processors, October 1992 (verbal presentation).

[FURBER89] S.B. Furber. *VLSI RISC Architecture and Organisation.* Marcel Dekker Inc. New York, 1989.

[GARSIDE91a] J.D. Garside. *An Assembly of Building Blocks.* Internal Document: AMULET Group, University of Manchester, Computer Science, June 1991.

[GARSIDE91b] J.D. Garside. *On Asynchronous Pipelines.* Internal Document: AMULET Group, University of Manchester, Computer Science, June 1991.

[GOPAL90] G. Gopalakrishnan and P, Jain. *Some Recent Asynchronous System Design Methodoligies.* Technical Report No. UU-CS-TR-90-016, University of Utah, October 1990.

[HOROWITZ87] M. Horowitz, P. Chow, D. Stark, R.T. Simoni, A. Salz, S. Przybyski, J.L. Hennessy, G. Gulak, A. Agarwal and J.M. Acken. *MIPS-X: A 20-MIPS Peak, 32-bit Microprocessor with On-Chip Cache.* IEEE Journal of Solid State Circuits, Vol 22, No. 5, October 1987, pp 790-799.

[HENNESSY90] J.L. Hennessy and D.A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers Inc, 1990.

[HÅKON90] O.B. Håkon, E.H. Kristiansen and Bjørn O. Bakka. *Trace-Driven Simulation for a Two-Level Cache Design in Open Bus Systems.* ACM SIGARCH Computing Architecture News, Vol 18, No. 2, June 1990, pp250-259.

[JAGGER90] D.V. Jagger. *A Performance Study of the Acorn RISC Machine.* MSc. Thesis, University of Canterbury, Department of Computer Science, 1990.

[MARTIN85] A.J. Martin. *The Design of a Self-Timed Circuit for Distributed Mutual Eclusion.* Proceedings of The Chapel Hill Conference on VLSI, 1985, pp 245−260.

[MARTIN89] A.J. Martin, S. Burns, D. Borkovie, P.J. Hazewindus and T.K. Lee. *The Design of an Asynchronous Microprocessor.* Advanced Research in VLSI: Proceedings of the Decennial CalTech Conference on VLSI, 1989. MIT Press. pp 351−375.

[MEAD80] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison Wesley, 1980.

[PAVER92a] N.C. Paver. *Condition Detection in Asynchronous Pipelines.*

UK Patent Application No. 9114513, October 1991.

[PAVER92b] N.C. Paver, P.C. Day, S.B. Furber, J.D. Garside and J.V. Woods. *Register Locking in an Asynchronous Microprocessor.* Proceedings of IEEE International Conference on Computer Design: VLSI in Computers & Processors, October 1992, pp351−355.

[PRYBYLSKI89] S.A. Prybylski, M. Horowitz and J. Hennessy. *Characteristics of Performance Tradeoffs in Cache Design.* Proc. 15th Annual International Symposium on Computer Architecture.

[PRYBYLSKI90a] S.A. Prybylski. *Cache and Memory Hierarchy Design − A Performance Directed Approach.* Morgan Kaufmann Publishers Inc, 1990.

[PRYBYLSKI90b] S.A. Prybylski. *The Performance Impact of Block Sizes and Fetch Strategies.* ACM SIGARCH Computing Architecture News, Vol 18, No. 2, June 1990, pp160-169.

[SHORT88] R. Short and M. Levy. *A Simulation Study of Two Level Caches*. Proc. 15th Annual International Symposium on Computer Architecture.

[SEITZ80] C. Seitz. Chapter 7 − *System Timing* in Introduction to VLSI Systems. Addison Wesley, 1980.

[SMITH78] Alan Jay Smith, *Sequential Program Prefetching in Memory Hierarchies.* IEEE Computer 11. Dec 1978, 7-12.

[SMITH82] Alan Jay Smith, *Cache Memories.* ACM Computing Surveys. Sept 1982 Vol. 14 No. 3.

[SUTHERLAND89] Ivan Sutherland, *Micropipelines.* Communications of the ACM. June 1989 Vol. 32 No. 6 pp720−738.

[WANG89] W-H Wang, J-L. Baer and H. Levy. *Organisation and Performance of a Two Level Virtual-Real Cache Hierarchy.* Proc. 15th Annual International Symposium on Computer Architecture.

[WESTE85] N. Weste and K. Eshragian. *Principles of CMOS VLSI Design: A System Perspective.* Addison Wesley, 1985.