COPY-BACK CACHE ORGANISATION FOR AN ASYNCHRONOUS MICROPROCESSOR

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy in the
Faculty of Science & Engineering

2002

DARANEE HORMDEE

Department of Computer Science

Contents

List of Figures

List of Tables

Abstract

Over the last decade asynchronous design has re-emerged as a viable alternative to clocked design with mounting evidence of competitive performance, power efficiency and electromagnetic compatibility compared to the more mainstream synchronous style. However, although significant effort has been expended in the design of asynchronous processors, the design of asynchronous caches has been relatively neglected.

This thesis presents an asynchronous cache architecture – the logical choice for use with an asynchronous microprocessor. The design presented here provides the processor with a unified, dual-ported view of its memory subsystem using multiple interleaved blocks. Each block has separate instruction and data line-buffers effectively acting as level-zero (L0) cache, making the cache access time highly variable. The cache employs a copy-back write strategy to support a high-performance embedded processor core.

The other key memory system component required for performance improvement, especially when combined with a copy-back cache, is a victim cache; an asynchronous implementation of a victim cache is presented in the second part of this thesis.

Together, the resultant structure forms an asynchronous cache system for AMULET3, the third generation fully asynchronous implementation of the ARM processor. Although the whole design is optimised for the AMULET3 microprocessor core, the techniques employed are generally applicable to any asynchronous processor.

The proposed cache architecture is extensively evaluated using simulations, and the effectiveness of various alternative configurations is measured to arrive at a suitable trade-off between cost, complexity and performance. The simulations highlight some unusual aspects of the behaviour of asynchronous memory hierarchies.

Declaration

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

Copyright

The Author

Daranee Hormdee graduated from Khon Kaen University, Thailand, in 1996, obtaining a bachelor's degree (B.Eng.) in Computer Engineering. From 1996 to 1997, she worked for the university as a junior lecturer.

In 1998 she was awarded an M.Sc. in Computer Science with a project title "An Analysis of Asynchronous Processor Pipelines" which was carried out in the AMULET group at the department of Computer Science, the University of Manchester, UK.

Since then she has been conducting research on asynchronous cache architecture in the AMULET group, work which has culminated in this thesis.

Acknowledgements

Firstly, I would like to thank my supervisor Prof. Steve Furber for his valuable advice and guidance throughout the time in which the work in this thesis was carried out. I am also especially grateful to my advisor, Dr. Jim Garside, and to Dr. John Bainbridge who both endured much questioning and provided endless support and advice.

Special thanks to my proof readers: Dr. Andrew Bardsley, Peter Riocreux and Dr. Luis Plana for proof reading and commenting on the draft of this thesis.

I would like to thank Dr. Viv Woods for his advice in understanding cache design in an asynchronous environment and Lilian Janin, who came to the rescue with a much faster LARD simulator when I needed it most.

The research presented in this thesis was funded by the Royal Thai Government and Khon Kaen University, Thailand. I gratefully acknowledge this support.

Thanks to my other friends in the AMULET group and in Manchester for keeping me healthy with many enjoyable (indoor) games of badminton and squash during these rainy years in Manchester.

And last, but no means least, my most heartfelt thanks go to my family for encouraging and supporting my education in every way.

This thesis was prepared using Adobe FrameMaker 5.5. Figures and graphs were prepared using Xfig 3.2 and GNUPLOT 3.7 respectively and were imported into FrameMaker as Encapsulated Postscript.

Dedication

To my grandad.

Chapter 1: Introduction

Computer systems are in essence composed of three principal components; a processor – which does the work, a memory – which stores instructions and data and an I/O system – which allows interaction with the system's environment.

With the continued improvements in VLSI technology and processor architecture, the performance of general-purpose processors continues to increase at a relatively rapid pace compared to the memory since the same VLSI technology advances have been primarily used to increase memory capacity. As a result the gap between the speed of processors and the speed of memory systems is widening. The phenomenon of memory access time limiting system performance is well known as the *memory wall* [107].

These trends place increasing importance on the memory hierarchy, typically involving cache memory, to bridge the memory wall and provide the instruction and data bandwidth required by modern processors. In practice, this means that the caches have to be on the same chip as the processor since crossing chip boundaries leads to unacceptable cache access time, but such on-chip caches must necessarily be small. These factors lead to caches with relatively high miss rates and large miss penalties.

Because of its position as the critical component in bridging the processor-memory performance gap, cache memory has been studied extensively and now uses many sophisticated techniques, some of which can be seen in Smith's Second Bibliography on Cache Memories [92]. However, (nearly) all of these developments have been designed around the assumption of a global clock which is used to coordinate activity within the cache systems; yet with a complex memory hierarchy it may be inefficient to coerce all operations into immutable clock periods or multiples thereof.

Conventional synchronous systems are based on global clocking whereby global synchronisation signals control the rate at which different units operate. As the clocks get faster, the systems bigger and the wires finer, it is increasingly difficult to ensure that all parts in the system are operating in lockstep with one another. One solution to this is the use of asynchronous design, which attacks clock-related timing problems by replacing global clock control with some form of agreement on a mutually acceptable protocol of data transmission and acknowledgement which is not regulated by time. This can happen locally within a unit or globally between subsystems. Recent research has also shown that asynchronous microprocessors offer lower power consumption and better electromagnetic emission profiles [32] than their synchronous equivalents.

It is possible to interface a conventional synchronous cache memory to an asynchronous microprocessor but this would subvert many of the possible asynchronous advantages offered by the core. For full benefit an entirely asynchronous solution is preferable. However, there have been very few attempts to construct the supporting asynchronous memory systems needed to exploit these to the full. Such systems, including the one presented here, display data-dependent behaviour which often allows the system to approach average-case performance. Here, this means that when a request is sent from the processor to the cache, the cache response time can be different depending on where in the cache system the data currently resides. Although synchronous caches are well-understood, and comparison techniques to aid their development are well-known, the same is not true of asynchronous caches.

Prior to this work, all asynchronous cache implementations were single-ported and used a write-through strategy. This thesis addresses the added complexity of supporting a Harvard-like processor architecture with a copy-back, unified cache, requiring dual-ported memories capable of handling contention between two independent asynchronous ports. In addition, the implications of fetching and returning cache lines in an asynchronous environment are discussed and new mechanisms developed to address the issues which arise.

The primary focus of this work is a cache architecture for embedded processors, but many of the techniques developed are applicable to larger, high-performance asynchronous caches. The goal of this research was to investigate the potential of such a cache architecture, evaluate its performance and design, and study alternatives. An additional requirement was to seek an architecture that did not have a significant hardware overhead and did not worsen the overall power consumption.

Currently, the most popular way of designing cache systems such as the one studied in this thesis is by the application of decades of experience of synchronous design. An analysis of a few of the differences seen between synchronous techniques and solutions in some cases certainly highlight the most important lessons.

The starting points for the design of an asynchronous cache memory for AMULET3, a third generation asynchronous implementation of the ARM microprocessor, were the memories used on earlier chips incorporating the AMULET2 [32] and AMULET3 [34] processors. The AMULET2e chip included on-chip memory that could be configured as memory-mapped RAM or as a write-through cache; the DRACO chip incorporated an AMULET3 with memory-mapped RAM. The cache memory proposed here adapts features from both of these earlier memories and adds some new features, principally related to the requirement for a copy-back write strategy to support the high processing speed of the AMULET3.

1.1 Thesis organisation

This thesis comprises 9 chapters. The remainder of this chapter summarises the author's contributions and a list of publications, based on the work in this thesis.

Chapter 2 makes a case for an asynchronous copy-back cache. It gives an introduction to the asynchronous design describing the fundamental difference between synchronous and asynchronous design styles and highlighting the pros and cons of asynchronous design. The chapter then presents the nomenclature, structure and operation of caches used in synchronous memory hierarchies, all of which is well-known, and can also be used to describe asynchronous caches.

Chapter 3 looks at the use of cache techniques to enhance the cache system performance and discusses the trade-offs that are normally considered when designing caches. The trade-offs are usually between four competing design requirements: large storage capacity, fast access time, low implementation cost and low power consumption. The chapter also presents a number of commercial synchronous cache implementations in order to illustrate cache techniques and trade-offs that are commonly chosen amongst these practical, recent cache systems.

Chapter 4 presents a summary of earlier processors in the area of asynchronous logic design, together with a survey of a number of asynchronous memories. Special emphasis is placed on the AMULET2e cache and AMULET3i RAM systems which form the basis of this work.

Chapter 5 presents the proposed design of an asynchronous copy-back cache architecture. This poses a number of interesting issues and problems in asynchronous cache design: the line allocation mechanism, write buffering, non-blocking line fetches and out-of-order accesses.

Chapter 6 describes the justification for, and a possible implementation of, a victim cache and write buffer in a totally asynchronous environment. In an asynchronous cache system new implementational problems are introduced as the processor may be completely desynchronised with the bus traffic.

Chapter 7 describes the simulation environments and methodology used to analyse the techniques and designs discussed in chapters 5 and 6. The LARD programming language used in this work is described at the beginning of this chapter.

Chapter 8 presents the evaluation of the proposed cache architecture in the context of the AMULET3 system. Three aspects of the work presented here are evaluated: the effects of varying the multitude of cache parameters; the variability in access times resulting from the systems' asynchronous design; and the effects of adding the victim cache to the cache.

Finally, chapter 9 draws conclusions from the research presented in this thesis and considers how the asynchronous cache architecture and implementation could be further improved.

1.2 Research contributions

The work described in this thesis combines a number of existing architectural features from both synchronous and asynchronous systems and extends them with a number of novel features, particularly an asynchronous copy-back organisation. The result is the first asynchronous dual-ported copy-back cache design. In addition to this, the first asynchronous victim cache is presented, by adapting a technique from the AMULET3 reorder buffer to resolve forwarding in an asynchronous environment. The entire system is evaluated using behavioural models to demonstrate the feasibility of the proposed architecture for the design of a substantial real-world cache. In particular, it is shown how copy-back cache operations can be achieved in an asynchronous context.

The following papers, which have been published or submitted for publication, describe aspects of the work culminating in this thesis:

- *An Asynchronous Copy-Back Cache Architecture* [50]: D. Hormdee, J.D. Garside and S.B. Furber; submitted to Microprocessors and Microsystems Journal.

- *An Asynchronous Victim Cache* [49]: D. Hormdee, J.D. Garside and S.B. Furber; will appear in the proceedings of the International Euromicro Symposium on Digital System Design (DSD’2002).

- *An Asynchronous Copy-Back Cache Architecture* [48]: D. Hormdee and J.D. Garside; appeared in the proceedings of Postgraduate Research in Electronics, Photonics, Communications and Software (PREP 2001).

- *AMULET3i Cache Architecture* [47]: D. Hormdee and J.D. Garside; appeared in the proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC’2001).

- *An Asynchronous Dual-Ported Copy-Back Cache Architecture* [46]: D. Hormdee; appeared in the proceedings of the 8th UK Asynchronous Forum.

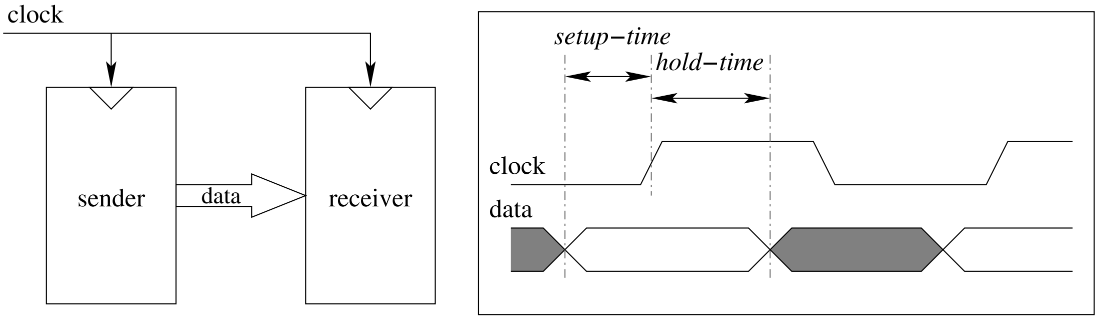
- *A Proposed Asynchronous Dual-Ported Cache Architecture* [45]: D. Hormdee; appeared in the proceedings of the 7th UK Asynchronous Forum.

Chapter 2: Background Material

This chapter provides background information on both asynchronous design and memory hierarchies, especially cache memory systems. Section 2.1 starts with an overview of asynchronous design: it describes both synchronous and asynchronous styles highlighting the differences between them. A number of advantages and disadvantages of asynchronous design are also discussed. Section 2.2 introduces the basic nomenclature, concepts, structures and operation of simple caches.

2.1 Asynchronous design

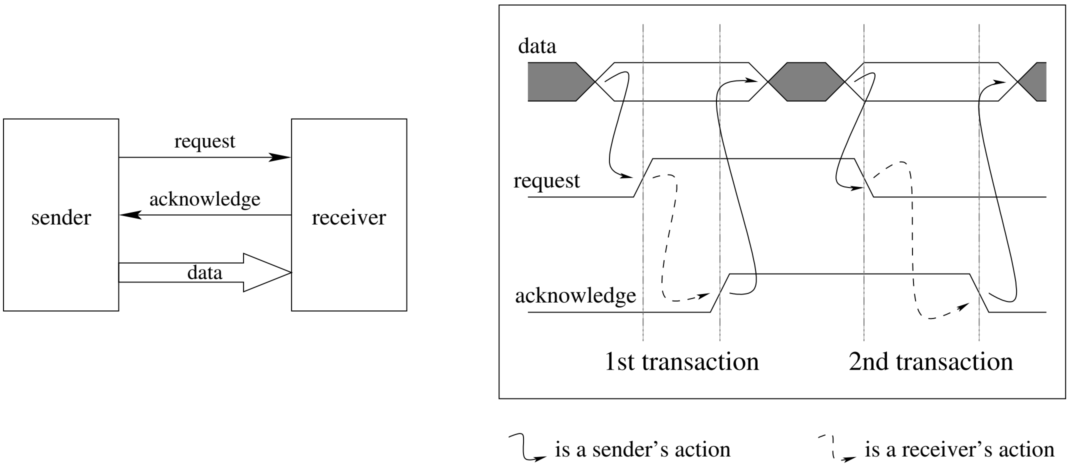
The conventional synchronous, or clocked, design style used today relies upon a globally distributed timing signal known as the clock. All data transfer in the system is regulated by the clock as in figure 2.1a, with *sender*s driving data lines a defined period (the *setup-time*) before the clock edge and *receiver*s having to latch the data within a defined period (the *hold-time*) after the clock edge as illustrated in figure 2.1b. The advantage of this approach is that the timing constraints for correct operation of the circuit (i.e. that the setup and hold times are met) are easily checked with static timing analysis tools.



(a) Connectivity (b) Timing diagram

Figure 2.1: Synchronous design style

Unlike synchronous systems, asynchronous (or as they are widely and more meaningfully known, *self-timed* or *clockless*) systems operate without the use of a central, global clock. Instead they use a distributed control scheme allowing different parts of the system to act independently where there is a lack of interaction between units. A local handshaking protocol is used for communication between these independent modules to indicate when data is available at a sender, and when it has been received for the next module to process, as illustrated in figure 2.2. Data is sent via a group of signal wires, which are normally unidirectional, collectively known as a *channel*. The sender module delivers data onto the channel whilst the receiver module obtains data from the channel. In the bundled-data scheme, two signals, *request* and *acknowledge*, are used to indicate when the sender is ready to send and the receiver has accepted the data respectively. Other schemes may require only the acknowledge signal since request information can be encoded in the data.



(a) Connectivity (b) Timing diagram

Figure 2.2: Asynchronous bundled-data design style

Over the past half century, the majority of computer developers have chosen the clocked design style for its simple-to-validate timing constraints. However, the idea of clockless processors was not entirely abandoned, being kept alive by both academic and commercial organisations. The list of research groups with a major interest in asynchronous logic design can be found at the Asynchronous Logic Homepage [6] and, as can be seen there, efforts over the last decade have brought self-timed logic to the point of commercial readiness but with active research still ongoing. Further details on all aspects of asynchronous design are available elsewhere [7].

2.1.1 Claimed advantages

The continued interest in asynchronous design is stimulated by the potential benefits that it may offer compared to the synchronous design style, as described below.

- **Clock skew avoidance**: Despite the advantages of synchronous design that make it so popular – simplicity in the design, widely available components, settling of system activities by the time a clock event occurs etc. – maintaining global synchrony is becoming increasingly difficult. *Clock skew* is the term used to describe the phenomenon whereby there is a slight difference in propagation time between the clock edges reaching various parts of a design [105]. As systems become larger, an increasing amount of design effort is necessary to guarantee minimal skew in the arrival time of the clock signal at different parts of the system. In an asynchronous circuit, skew in synchronisation signals can be tolerated.

- **Power efficiency**: Conventional synchronous processors are based on global clocking whereby central synchronisation signals control the rate at which the different components of the processor operate. The use of free-running, high-frequency clocks is a source of power inefficiency, causing all parts of the system to consume power whether or not they are doing useful work. This is why most power-conscious synchronous designs use *clock gating* to manage their power consumption. Units inside an asynchronous system negotiate between each other to transfer data allowing parts of such a system to work at their own paces. This improves power-efficiency since functional units use energy only when doing useful work and this comes as a direct product of the asynchronous design methodology [33].

- **Modularity of design**: In an asynchronous framework a design can be constructed from small units within which temporal dependencies are managed locally. This makes the design process more modular, facilitating easy reuse of components as individual stages are independent of each other and can be designed, simulated, evaluated and tested in isolation.

- **Better than worst-case performance**: Whilst a synchronous design operates at a speed dictated by its worst-case timing path, circuits in an asynchronous device operate as fast as they can; infrequent worst-case operations may be allowed to take longer. This increases the overall operating speed of the device, maintaining a high average performance based on typical case operation rather than worst-case operation. In a system including a cache, when a memory request is sent from the processor to the cache, the cache response time can be different depending on where in the cache system the data currently resides. This is true for both synchronous and asynchronous caches, but the asynchronous system can accommodate much finer differences in timing variation because it does not quantise time to multiples of fixed clock periods.

- **Electromagnetic compatibility (EMC)**: In synchronous systems, the system clock synchronises all activity, causing switching actions to happen at the same instant in time everywhere in the clock domain and at regular intervals. In turn this causes sharp spikes in current consumption at each active clock edge. These spikes result in the emission of a large amount of electromagnetic noise, radiated at the harmonics of the global clock frequency. In contrast, activities in an asynchronous system are spread over time and frequency, causing electromagnetic interference (EMI) to be distributed at a lower amplitude and across the electromagnetic spectrum.

2.1.2 Drawbacks

Despite the benefits that asynchronous design can offer, it also has a number of disadvantages which can make it harder to use, such as:

- **Control logic complexity and the deadlock/livelock risks**: Without implicit global clock control, the control logic in asynchronous design is more complex since each module of the design needs hardware to perform synchronisation to wait for data and to trigger other modules when it has produced its data. The use of explicit communications between modules increases the risk of introducing *deadlock*: distributed control through which a circle of unresolvable dependencies causes all activity to cease. There is also an added risk of *livelock*, where units get stuck in loops without exits, resulting in incorrect behaviour. These two problems can be introduced by design errors. Ideally, deadlocks and livelocks should be detected and then avoided at a very early stage in the design process. Unfortunately, current formal validation techniques (e.g. Rainbow [83]) cannot cope with large designs such as the study presented in this thesis, hence the use of extensive simulation to give good confidence in the design functionality.

- **The loss of implied timing related/temporal knowledge**: In a synchronous processor, the positions of an instruction and its result are deterministic, controlled by the clock. However, in an asynchronous system, once a sequence of information is put into a pipeline, there is no way of knowing where in the pipeline the information will be at any later time without explicitly synchronising with the pipeline. Asynchronous design works well where there are few inter-dependencies between blocks, where synchronisation is required a significant penalty is often incurred. Synchronisation should therefore be limited to where it is necessary to allow other units in the system to operate independently at their own rate. An example solution of such a problem in the context of asynchronous cache design is the implementation of a victim cache, providing data forwarding from the write buffer, presented in chapter 6.

- **Design validation difficulties**: Design validation is an extremely important process in order to detect any defects in the design. Because of the inherently non-deterministic activities resulting from arbitration [88] in an asynchronous system, for a particular sequence of inputs there may be various acceptable sequences of outputs. Furthermore, asynchronous systems tend to have more state than comparable synchronous systems and so the set of tests required may be larger.

- **The lack of design tools**: In order to make asynchronous design competitive in a market dominated by clocked synthesis, EDA tools and design methodologies must be developed. When the recent interest in asynchronous design took off in the early 1990s there were very few specific tools available. Most asynchronous systems have been designed by hand, at the cost of a large amount of time and effort. Many research groups and commercial organisations have attempted to address this problem and a range of tools is now available. These range from synthesis systems for constructing small-scale control modules (such as Petrify [22] and Minimalist [27]) to silicon compilers (such as Balsa [10] and Tangram [12]). These silicon compilers have been used for the creation of entire processors: Balsa for SPA [80] and Tangram for an asynchronous low-power 80C51 microcontroller [33]. They remove a lot of the burden of looking after the low-level details of the circuit from the designer and allow designer effort to be concentrated more on data processing and algorithmic issues. Furthermore, a number of high-level modelling/simulation languages exist for asynchronous circuits including LARD [55] (described in more detail in section 7.3), the tool used for modelling in this thesis.

- **Unfamiliarity**: Unfamiliarity with asynchronous design among the digital design community is not a fundamental drawback of the technology but it is nevertheless a factor in preventing its more widespread use since most designers were "brought up" on synchronous techniques. This will only be overcome by the design community seeing real-life applications coming into use which adequately demonstrate a technical and commercial advantage.

A positive characteristic of these drawbacks is that they are the designers' challenges, not the users'. The end users of the system need never be aware that its internal workings are self-timed, although they would hopefully appreciate the advantages gained from the use of asynchronous design.

2.2 Cache and memory hierarchy

Although, as discussed in the previous section, asynchronous and synchronous implementation techniques are very different, similar higher-level architectural planning approaches can be used. For these reasons, this section examines a number of techniques used in the design of synchronous memory subsystems. However, all of the illustrations here are drawn as if they were employed in the asynchronous environment (highlighting required synchronisation), depicting their effects on the asynchronous design.

The performance of a data processing system can be measured in terms of its *latency* and *bandwidth* (also known as *throughput*). The latency of a memory system is the time between the initiation of a memory request from the processor and its completion when the processor receives the result of that request. The bandwidth of a memory system is the rate at which the memory system can satisfy requests or produce results. There are, therefore, four options for increasing memory performance:

- widening the memory bus between processor and memory which increases the bandwidth but does not affect latency;
- making the memory faster which improves both the bandwidth and the latency;
- pipelining the memory which improves bandwidth but usually also slightly increases the latency;
- exploiting locality of accesses to simplify address decoding and reduce latency - see section 5.7.

The cost of memory is proportional to its speed and size. A sufficiently high speed memory of size equal to the address space of a modern processor will be prohibitively expensive. In order to get around this problem, it is possible to use some small amount of high-speed memory to store a portion of the content of a larger amount of lower-speed main memory in a way that approximates the performance of a large high-speed memory with a reasonable cost. This forms the memory system into a hierarchy. Such a small, fast memory in the memory hierarchy is referred to as a *cache.*

The idea of organising memory into a hierarchy (shown in figure 2.3) dates back to the early 1960s. The University of Manchester Atlas Machine's 'One-level storage system' [61] was the first example of a memory hierarchy in the form of *virtual memory*, a mechanism expanding the space available for instructions and data beyond the limits of physical main memory. The concept behind this mechanism is to store the most frequently used data in the high-speed memory, fetching less frequently used data from the disk into the high-speed memory as needed.

In a hierarchical memory system, data replacements are performed between adjacent levels. Upper levels are smaller, faster and closer to the processor. Lower levels are larger, slower and cheaper per bit. Typically an upper level replicates a part of the memory space

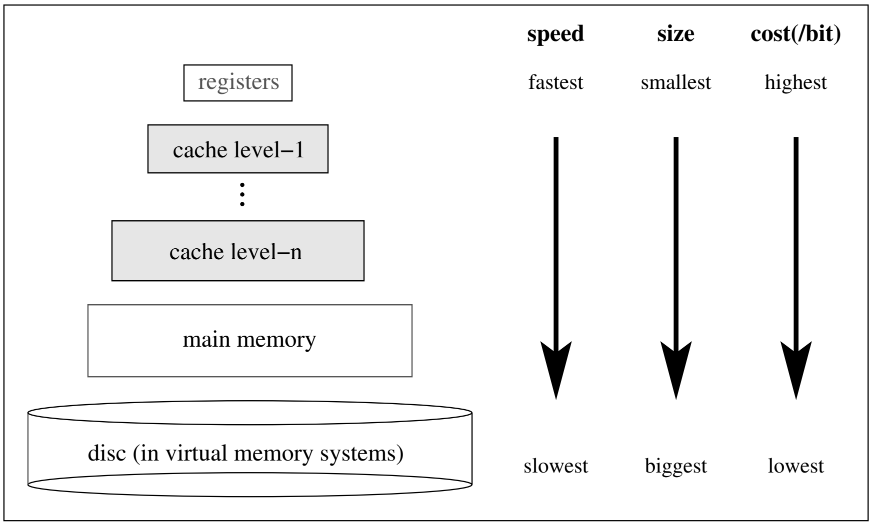


Figure 2.3: Memory hierarchy

of the next lower level. The best-case scenario for the operation of such a memory hierarchy is that the processor can always obtain referenced data from the highest memory level allowing memory access to be completed at the maximum speed of the memory system. However, this is obviously impossible to achieve in the general case. When the processor cannot find the data at the highest memory level, then the next lower level is searched and so on. When the data is found, a *line* – comprising the requested data and possibly some other adjacent data – is copied into the higher levels of the memory system. Of course, some amount of data from the higher levels must be evicted to make room for the new data.

Cache was first introduced by IBM in the System 360 Model 85 in 1968 [56]. It is normally situated between the processor and the main memory and decouples these two components. It can be either on-chip or off-chip. Accessing a cache involves two operations:

- **address tag look-up**: to check whether (and where) the required data is in the cache;
- **data access**: to read from or write into the cache.

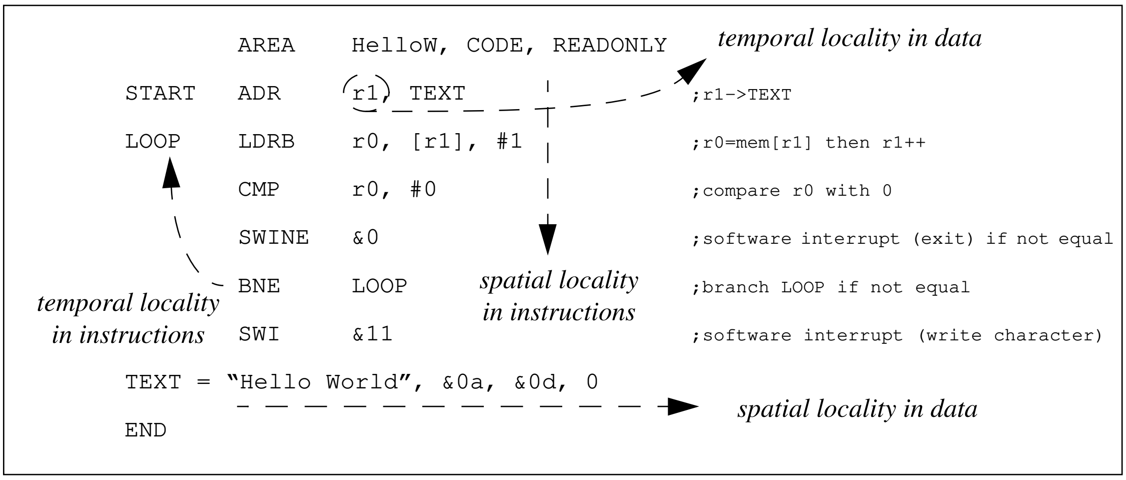
By exploiting the sequentiality of memory accesses, the frequency of the tag look-up operations can be reduced. A solution to this is the line-buffering technique, described briefly in section 3.4.3 and in greater detail in section 4.3.2.

The cache literature spans a number of decades, however the terminology used is inconsistent in many places. It is not possible to identify a definitive terminology to describe caches. Therefore, for consistency, the remainder of this section describes the basic cache terminology that will be used in the rest of this thesis.

2.2.1 Locality of reference

Memory hierarchy relies on two properties of the access patterns of most programs: the *temporal locality* (locality in time) – if something is accessed once, it is likely to be accessed again soon, and the *spatial locality* (locality in space) – if one memory location is accessed then nearby memory locations are also likely to be accessed.

Figure 2.4 shows a fragment of ARM assembly code exhibiting both of these properties. Temporal locality of code occurs here because the loop is executed many times. Spatial locality of code also occurs from the loop and from the fact that the code is largely sequential, so after fetching the `LDRB` instruction the `CMP` instruction is fetched. Temporal locality of data access is shown by the operations on the data `r1`, stored in a register here, and data locality is shown by the in-order accesses to the text string.



```
        AREA    HelloW, CODE, READONLY
START   ADR     r1, TEXT              ;r1->TEXT
LOOP    LDRB    r0, [r1], #1          ;r0=mem[r1] then r1++
        CMP     r0, #0                ;compare r0 with 0
        SWINE   &0                    ;software interrupt (exit) if not equal
        BNE     LOOP                  ;branch LOOP if not equal
        SWI     &11                   ;software interrupt (write character)
TEXT = "Hello World", &0a, &0d, 0
END
```

Figure 2.4: Code example of locality

2.2.2 Hit or miss

A cache *hit* results when a memory request can be satisfied from the contents of a given cache as the required data is present in the cache. A cache *miss* results when the required data is not present in the cache and causes an access to the next level down in the memory hierarchy. In some situations, the cache continually misses because, for a given piece of code or data (e.g. a loop or array), the fetching of required lines is causing the eviction of other required lines. This pathological case of evictions causes serious performance degradation and is known as *thrashing*.

There are three kinds of cache miss, also known as the three C's [41]. Firstly, a *compulsory miss* occurs when the data resides in a previously unreferenced cache line. This kind of miss is unavoidable when loading a new program into the cache. Secondly, a *capacity miss* occurs when the required data is amongst the cache lines that have been discarded (because the cache is too small) to make room for new cache lines. Thirdly, a *conflict miss* occurs whenever a cache must discard a cache line to allocate another line on a miss, even if other sets have unused lines. (Hence there is no conflict miss in fully associative caches – see section 2.2.5.)

The *hit rate* is the fraction of memory accesses found in the cache with respect to the total number of memory accesses whilst the *miss rate* (1 – *hit rate*) is the fraction of memory accesses not found in the cache with respect to the total number of memory accesses. Miss rates that are measured when starting with an empty cache are called *cold-start*[1] miss rates. Those that are measured from the time the cache becomes initially full are called *warm-start* miss rates. The warm-start miss rate is obviously equal to or lower than the cold-start rate when running the same program. The difference is the number of compulsory misses, because the warm-start miss rate omits all cache misses at the start when filling the empty cache.

---

[1] Note that this definition is different from Hennessy and Patterson [41] where the cold-start miss rate is called the compulsory miss rate.

2.2.3 Cache line fetch

When a cache miss occurs, resulting from an attempt to access a cacheable location that is not replicated in the cache, a cache line, containing the required data, is copied from the memory into the cache. This process is called a *line fetch*.

The simplest line fetch procedure is to halt the processor for the entire period of fetching the cache line, which starts with the lowest address. This method is known as *stall-on-miss* [41]. It requires the processor to stall for a period of time which may be considerably longer than is strictly necessary. Other, more efficient, line fetch techniques are presented in section 3.4.2.

The *miss penalty* is the additional time required to service the data access for a miss, which is the time to access the main memory and forward the value to the processor. In some designs the miss penalty includes the time taken to replace a line in the cache with the corresponding line from the main memory.

2.2.4 Cache (physical) organisation

In the 1940s, researchers at Harvard University built the ‘Mark’ series of computing machines [41]. The Mark-III and the Mark-IV were stored-program machines with separate memories for the instructions and data. The name *Harvard architecture* was then coined for this type of memory organisation. Whilst this type of architecture is rare today in conventional microprocessors (though still common in DSPs [25]), it is common for a machine to have a shared main memory but separate instruction and data caches; this is called a *modified Harvard architecture* [28]. The alternative, a single, shared memory system invented around the same time, is named a *unified architecture* [41].

In a modified Harvard architecture an advantage of separating the data and instruction caches is that it makes it simpler for instructions and data to be fetched from memory simultaneously. In systems in which the processor is pipelined and can fetch instructions and data simultaneously, it is a great advantage to have separate buses to memory to fetch these items, rather than forcing a stall and fetching them one at a time. Furthermore, in an asynchronous environment a unified cache would require arbitration between these two request sources because the timing relationship of the two requests cannot be controlled.

Although arbitration is not particularly expensive, it is undesirable for deterministic behaviour. Shifting the point of arbitration to when the cache cannot satisfy a request locally and requires access to the lower levels in the memory hierarchy, a much rarer situation, results in higher average overall performance and fewer arbitration events.

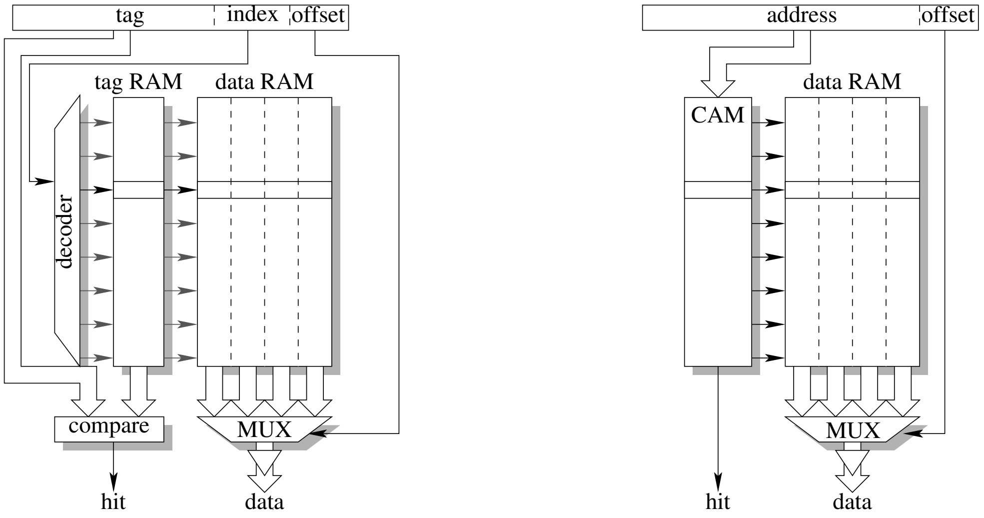
With a split cache architecture it is usually impossible to modify the instruction stream since, typically, the instruction cache is capable of reading memory but not writing to it; therefore self-modifying code becomes much more difficult to implement. In addition, since the two caches are physically separate, the cache hierarchy is less able to adapt to changing conditions by modifying the partitioning of cache resources between instructions and data. On the other hand, the benefit of a modified Harvard memory architecture is that it is possible to optimise each cache individually to meet its bandwidth, locality and power requirements since they need not be identical.

An effective architectural approach that can provide most of the benefits of a Harvard architecture combined with those of a unified approach is described in chapter 5. It is a form of unified memory system that has been used in a number of cache/memory designs including the AMULET3i macrocell on the DRACO chip [34], and is also used in this work.

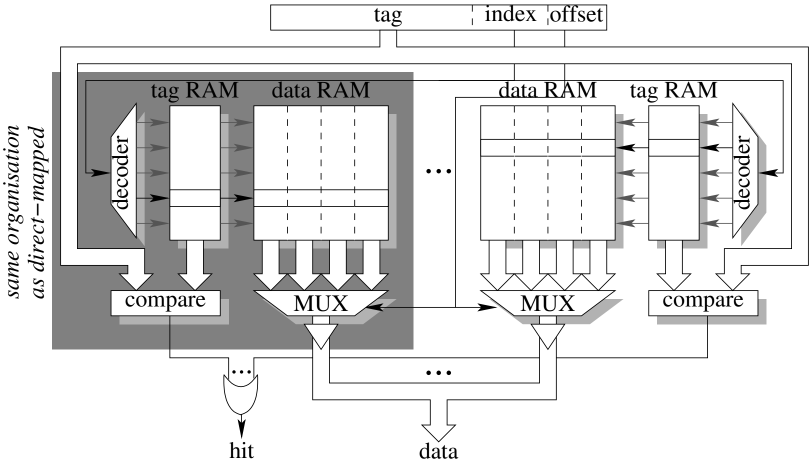
2.2.5 Degree of associativity

Caches are frequently classified by the manner in which the access address to cache line mapping is accomplished. The simplest way to allocate the cache to the system memory is a *direct-mapped cache* (figure 2.5a) where each main memory address maps to a single fixed location within the cache. Multiple main memory addresses may map to a single cache address, since the cache's address space is smaller than the main memory address space.

At the other extreme a *fully-associative cache* (figure 2.5b) is one where any main memory address can be stored in any part of the cache. Lastly, the *set-associative cache* (figure 2.5c) is a compromise between the two extremes of the direct-mapped and fully-associative styles where, for example, in a 2-way set-associative cache each



(a) Direct−mapped cache organisation (b) Fully−associative cache organisation



(c) Set−associative (2−way) cache organisation

Figure 2.5: Cache organisations (after [28])

memory address can be stored in either of a set of two locations, the set being determined by some part of the address.

As seen in all organisations in figure 2.5, information held in each cache line is stored in two parts: the data is held in a data store and its address is held in a *tag* store. In direct-mapped and set-associative caches, to check whether the content of particular memory locations are held in the cache, the high-order part of the address (the ‘tag’ in figure 2.5a and figure 2.5c) is compared with the stored tag in the tag RAM, accessed by the relevant address bits (the ‘index’). If they are the same, the information is in the cache. The last few low-order bits of the address, the ‘offset’, are used to indicate the required data from the retrieved cache line.

On the other hand, the tag store in fully-associative caches is designed differently using CAM (Content Addressable Memory), a RAM cell with integrated comparator so that all tag look-ups can be processed in parallel. If there is a hit, the *offset* can be used to indicate the required data from the cache entry as before.

An advantage of a fully-associative cache is that as long as there is a free entry in the cache, then there is no need to replace the cache contents with other addresses. If all of the entries are full and a miss occurs then ideally the line which has not been needed for the longest period of time would be evicted. Since this is difficult to accomplish, alternative methods, *cache replacement strategies* (described in the next section), must be used.

2.2.6 Cache replacement strategies

There are several strategies used to determine which entry in an associative cache to use to store a newly fetched line. This can be an important choice because – except at startup – this involves overwriting/replacing an existing cache line. The most common strategies are [41]:

- **Cyclic replacement**: When a line fetch occurs, the next line in sequence is replaced. Upon reaching the last line in a set, the cycle restarts at the first line. This type of algorithm is also well known as FIFO (First-In-First-Out), round-robin and LRA (Least-Recently-Allocated).

- **Random replacement**: The decision is based on a random or pseudo-random value to spread allocation uniformly, hence there is no ‘pathological’ reference stream causing worst-case results.

- **Least Recently Used (LRU) replacement**: The two algorithms described above come at low cost under the assumption that all addresses are equally probable. In most applications, however, long address sequences are far from random, and so the history of the address sequence can be used to increase memory throughput [90]. With the LRU algorithm, the cache keeps a record of accesses to each cache line. Temporal locality means that the recently-used cache lines tend to be used again soon, and so the new line is placed into the cache line that has been unused for the longest time, which is likely to be the best candidate for disposal. This is, however, more expensive to implement.

In empirical studies [41] there was little performance difference between the LRU and random replacement strategies and this difference becomes less obvious as the cache becomes larger.

2.2.7 Memory burst access

A typical memory, as shown in figure 2.6, is constructed as a dense matrix of storage cells whose contents are accessed via a two-stage process. Firstly, part of the address forms the row address used to read a whole row from the matrix. Then the remainder of the address forms the column address used to select which part (e.g. word) of the row to deliver at the data output pins. If two consecutive accesses address the same row, then the row retrieved from the matrix for the first access can be reused for the second without having to retrieve it again. With dynamic memories, this also avoids an additional precharge stage that must precede the row access.

This is exactly what is done in modern computer systems and is known as *burst mode access* or *bursting* [81], a rapid data-transfer technique automatically generating a series of consecutive addresses each time the processor requests a single address with the assumption that the following requests will be sequential to the previous one. Bursting can be applied both to read (from memory) and write (to memory) operations.

In a synchronous system, the timing of burst mode access is generally stated using shorthand: ‘*x-y-y-y*’ referring to the number of clock cycles for each access of a quad-word burst. The first number *(x)* represents the number of clock cycles to do the first access. The other numbers *(y)* are how many clock cycles are used for the second, third and fourth accesses. An example would be ‘2-1-1-1’, which means 5 clock cycles to do the whole burst. Using conventional random access mode, this access sequence would

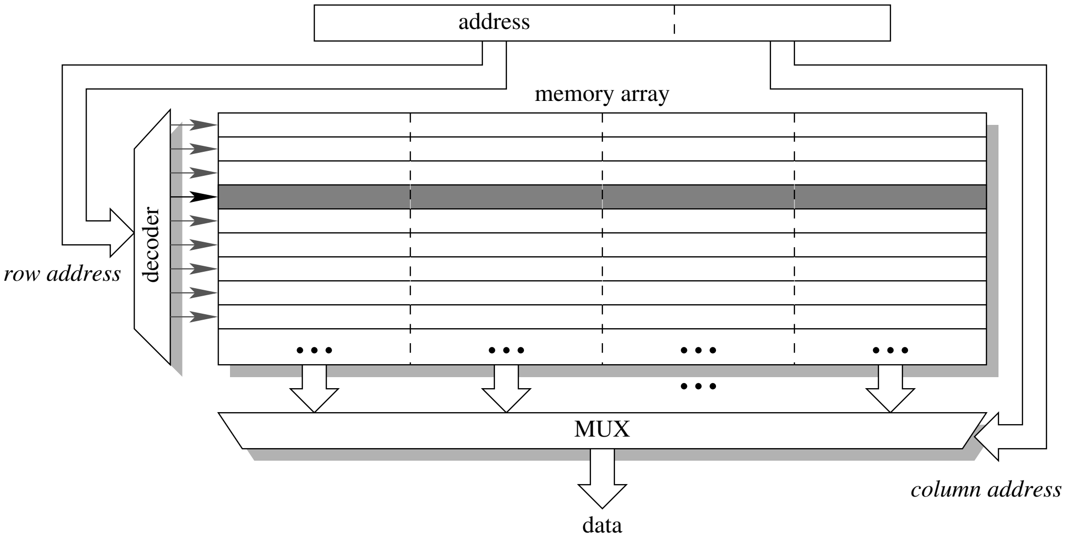


Figure 2.6: A memory array

take 8 clock cycles: ‘2-2-2-2’. Figure 2.7 illustrates how performance can be improved by burst mode memory access. In an asynchronous environment x and y would represent the delay to complete those tasks.

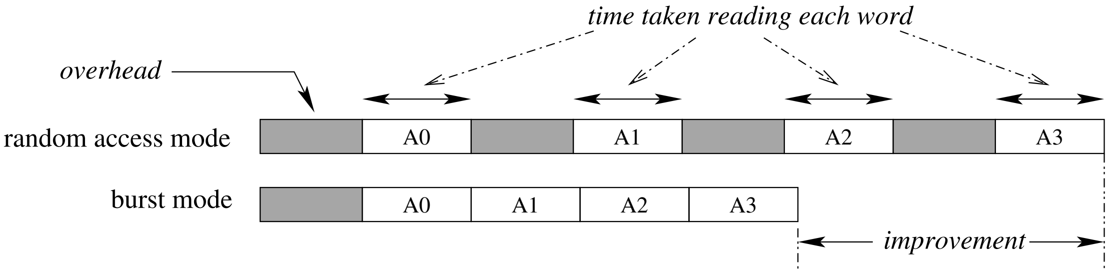


Figure 2.7: Memory access modes

2.2.8 Write policies

In addition to caching reads from memory there are different types of policies determining how a cache handles writes:

- **Write hit policies**: When writing into the cache, there are two possible options. In a *write-through* cache, data is always sent to the external memory as well as updating the cached copy. All levels of the memory hierarchy are said to be consistent. In a *copy-back* (also known as *write-back*) cache, data is written only to the cache. Only modified data, indicated by a *dirty bit*, is sent to update the memory when the line is reallocated or the cache is *flushed* (i.e. the main memory is made consistent with the cache). This type of cache provides better performance than a write-through cache because it reduces write traffic to the memory.

- **Write miss policies**: Another issue in handling memory writes is what to do in the event of a write miss. In a cache using the *write-around* policy (also known as *no-fetch-on-write*) no cache allocation is performed for write operations whilst in a *write-allocate* (or *fetch-on-write*) cache, cache allocation is performed on a write miss and is then followed by a write action into the cache.

Usually, copy-back caches use the write-allocate policy and write-through caches use the write-around policy. This first choice is reasonable since it is hoped that subsequent writes to that line will be captured by the cache. In the second case, subsequent writes still need to go to the main memory. However, write-through caches may benefit from applying the write-allocate scheme as well under the reasonable assumption that the line which has just been written is likely to be read in the near future.

2.2.9 Write buffering

Writes to main memory or the next cache down can occur because a line is evicted from a copy-back cache or a write into a write-through cache. In both cases the write data can be buffered, in a FIFO structure called a *write buffer* [41], so that it does not impede fast cache operation whilst waiting for the main memory.

Write buffering is desirable in both a write-through and a copy-back cache to improve performance but for slightly different reasons. In a copy-back cache, a write buffer containing a single slot is desirable so that an evicted line needing to be copied back can be held temporarily to avoid interfering with the more urgent line fetch process. In a write-through cache several slots in a write buffer are normally required so that the processor does not have to stall for writes (which normally occur in clusters) to be completed. However, a large cluster of writes may still fill the buffer causing performance to be degraded.

Except when the write buffer is full, or when it contains the most up-to-date data needed by a read, buffered writes are usually sent out to the next level down when the bus is idle, so hiding any further memory access penalty. In these two exceptional cases, there is no choice but to drain some writes from the buffer. In most cases therefore, only reads are critical, affecting the overall performance when the processor has to stall for a read request.

2.3 Summary

This chapter has introduced the basic principles of asynchronous design and the reasons for its gaining popularity. Instead of a single global clock, parts of an asynchronous system can work at their own pace, negotiating with each other whenever data needs to be passed between them. The basic terminology used in caches and memory hierarchies has also been introduced, with a basic grounding in cache techniques. The next chapter extends this by detailing how the memory hierarchy performance can be improved.

Chapter 3: Tuning Memory Hierarchy Performance

The previous chapter described background material essential for this thesis. Basic asynchronous design concepts were discussed and arguments for and against the use of this design style in the context of a memory hierarchy were introduced. The latter part of chapter 2 summarised the cache terminology used herein.

This chapter surveys techniques used to improve memory hierarchy performance in the mainstream, synchronous design. A number of commercial cache implementations are also described. The chapter concludes by discussing candidate techniques that need to be studied in more detail for use in an asynchronous environment. Those studies are described in later chapters.

3.1 Measuring performance

Memory hierarchy performance, represented by the average memory access time (T_{avg}), can be calculated as [41]:

$$T_{avg} = T_{hit} + (M \times T_{penalty}) \qquad \text{Equation 3.1}$$

Where T_{hit} is the cache access (or hit) time, M is the miss rate and $T_{penalty}$ is the cost incurred on a miss (the miss penalty).

Improving memory hierarchy performance is all about reducing the average memory access time ($\mathrm{T_{avg}}$) which can be achieved by:

- reducing the cache access time T_{hit};

- reducing the cache miss rate M, thus changing the ratio of low-cost (T_{hit}) hits to expensive ($T_{penalty}$) misses;
- reducing the miss penalty $T_{penalty}$;
- hiding read-write latency, usually through increased concurrency;
- reducing memory traffic, causing less contention for main memory between the processor and other peripherals (e.g. DMA controller) and giving the added bonus of reduced power consumption. Clearly the first obvious technique for reducing processor-memory traffic is caching itself.

Both hardware and software approaches can be explored to optimise improvements in the above areas. However, the scope of this chapter is restricted to uniprocessor, non-software-related improvements concerning only the cache memory level. The chapter introduces a number of hardware techniques for improving performance in general-purpose systems. Some of these techniques deliver dramatic improvements on specific, possibly infrequent, access patterns whilst others provide only small local improvements whose cumulative effect is appreciable. Note that such small or variable improvements can be much better explored by an asynchronous system since system activity is not quantised in time.

3.2 Reducing cache hit time

Hit time is typically critical to performance since it affects the majority of memory references and memory latency is often the limiting factor on system performance (and so clock frequency).

An integrated cache on the same die as the processor can support high bandwidth and low latency memory accesses by using a wide interface and eliminating the delay of pads and buses that arises with off-chip accesses. Furthermore, on-chip caching also decreases energy consumption in the memory system due to the reduction in off-chip accesses. However, on-chip area is often limited and so the cache has to be small. The less control involved in a cache's implementation, the shorter the delay in the critical path through the hardware. Small and simple caches are ideal in this respect and a direct-mapped cache is suggested to reduce hit time, since tag checking can be overlapped with data access. Hill [42] has confirmed this through studies of cache miss rates and access times. However the major disadvantage of small and simple caches is that they are more likely to suffer from higher miss rates, i.e. T_{hit} is reduced but M increases.

3.3 Reducing cache miss rate

To reduce the miss rate, some of the misses due to the three C's (compulsory, capacity and conflict) must be eliminated. Compulsory misses are caused by loading data into an empty cache and are therefore unavoidable. It is possible to change the mixture of miss types, for example: compulsory misses can be converted to conflict or capacity misses by changing cache size or other parameters. There are many parameters that can be adjusted to control the mixture of conflict, capacity and compulsory misses and a rather awkward trade-off must often be made to obtain a comfortable balance.

3.3.1 Larger cache size

The most straightforward approach to reduce the number of capacity misses is to increase the size of the cache. Nevertheless, changing the cache size also affects the number of conflict misses since references are spread differently possibly allowing misses to be moved from capacity to conflict and vice versa. The total cache size can be increased by having more cache lines (of the same size), by enlarging the cache line whilst fixing the number of lines, or by increasing both parameters. However, these changes come at a cost, primarily in silicon area. Furthermore, adding more cache lines results in a larger tag store which slows the look-up process requiring larger tracks on chip and larger address decoders (increasing the hit time) and increases the power consumption in high associativity caches.

3.3.2 Longer cache line

For a given cache size, using short cache lines can provide a lower miss penalty (assuming that the processor is stalled for the whole line fetch duration) since less data is required to be fetched into the cache for each line fetch. Longer lines take better advantage of spatial locality, decreasing the number of compulsory misses since subsequent requests could become hits in the previously fetched, long cache line. However, this technique might increase other types of misses when references do not follow locality rules. Also, fewer cache lines can be stored in such a cache. Larger cache lines are also more likely to contain unwanted items since – for example – branches occur quite frequently in the code.

3.3.3 Higher degree of associativity

Direct-mapped caches usually suffer from a large number of conflict misses. According to Hennessy and Patterson's *2:1 cache rule of thumb* [41], "a direct-mapped cache of size *N* has about the same miss rate as a 2-way set associative cache of size *N/2*"; higher associativity with the same cache size can improve the cache hit rate. However, set-associative caches have slower tag comparison stages than direct-mapped caches. Since this stage is in the critical path for a cache access, the hit time is increased.

3.3.4 Better replacement strategies

More effective replacement strategies allow associative caches to obtain further reductions in the number of conflict misses. The perfect replacement strategy is to reject the line that will be needed furthest forward in time. This cannot usually be achieved. The three most common practical replacement strategies are described in section 2.2.6. The weight of research evidence has found that LRU generally performs better than FIFO or random replacement [11,90,93]. Nevertheless, LRU is often expensive to implement and so the almost as 'effective' and much cheaper pseudo-random algorithm is commonly used.

3.3.5 Victim cache

Jouppi [57] proposed the *victim cache*, a small fully-associative cache, as a technique to reduce the number of misses in a direct-mapped main cache (figure 3.1). The victim cache holds lines ejected from the main cache along with their corresponding addresses; these lines are buffered to be written into memory. On a hit in the victim cache, the hit line can be forwarded directly from the victim cache into the main cache. The required data can also be presented to the processor faster than if reading it from the memory. Alternatively, this technique can be thought of as reducing the miss penalty if the hit in the victim cache is categorised as a cache miss due to its absence in the main cache. Moreover, the forwarding mechanism clearly reduces main memory traffic on a line fetch. On each request in Jouppi's victim cache, for high performance purposes (low hit time), the main cache tag look-up and the victim cache tag look-up proceed in parallel.

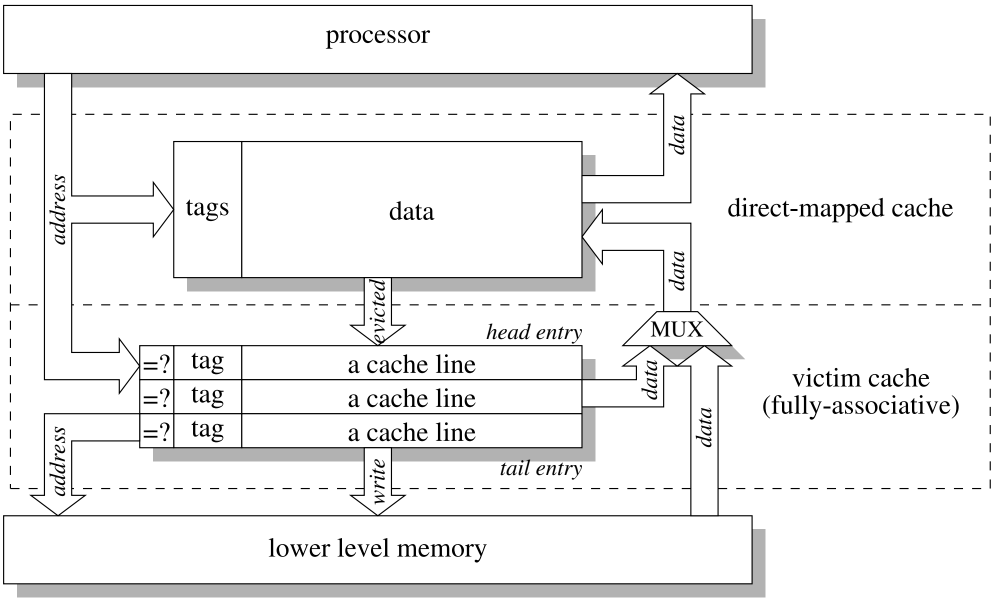


Figure 3.1: Jouppi's victim cache organisation

3.4 Reducing cache miss penalty

A number of approaches to improve the first two parameters from equation 3.1 (cache hit time and cache miss rate) have been discussed. This section examines techniques to reduce the last parameter in equation 3.1, the cache miss penalty. Cache misses usually occur rather infrequently compared to cache hits and, furthermore, some cache misses can be eliminated by the techniques described earlier. However, the miss penalty still has a major effect on memory hierarchy performance as it is directly related to main memory speed, which is likely to be slow compared to the speed of the cache or processor.

For example, with a miss rate of 10% in a 4-word line size cache with 10ns access time connected to a 50ns access time main memory, the average memory hierarchy performance is $T_{avg} = 10ns + (0.1 \times (50ns \times 4)) = 30ns$ according to equation 3.1. T_{avg} is here dominated by the miss penalty as is usually, but not always, the case.

3.4.1 Giving read misses priority over writes

The first miss penalty reduction technique is to give priority to read misses over write operations (also known as *read-overtake-write*) [41]. To implement this technique a write buffer is required to hold write data whilst reads are allowed to proceed. Although this technique can alleviate the processor stall for the requested data, allowing a read to overtake any write can introduce a data hazard (i.e. fetching the wrong data from the main memory whilst the most up-to-date data is in the write buffer). Obviously when implementing this scheme some precautions must be observed to avoid reading that stale data. A detailed discussion of read-overtaking-write techniques, particularly in a self-timed implementation environment, is presented in section 5.10.

3.4.2 Line fetch mechanism

The conventional stall-on-miss line fetch scheme shown in figure 3.2a (described in section 2.2.3) can be improved upon in a number of ways as illustrated in figure 3.2b and figure 3.2c and described below.

Early-restart

Early-restart [41], as in figure 3.2b, allows the processor to obtain the required word as soon as the requested word is fetched. However, the processor still has to wait for an appreciable time for the required data – the worst-case is when the required word is the last word of the fetched line.

Requested-word-first

With the commonly used *requested-word-first* technique [56] (also known as *critical-word-first* or *wrapped fetch*), that has been in use for over 30 years, the required word is retrieved from main memory first followed by the other words in the line.

Although employing the early-restart method with the requested-word-first technique shown in figure 3.2c shortens the processor stall for the requested data, the processor still has to wait until the entire line fetch is completed before continuing with other read/write

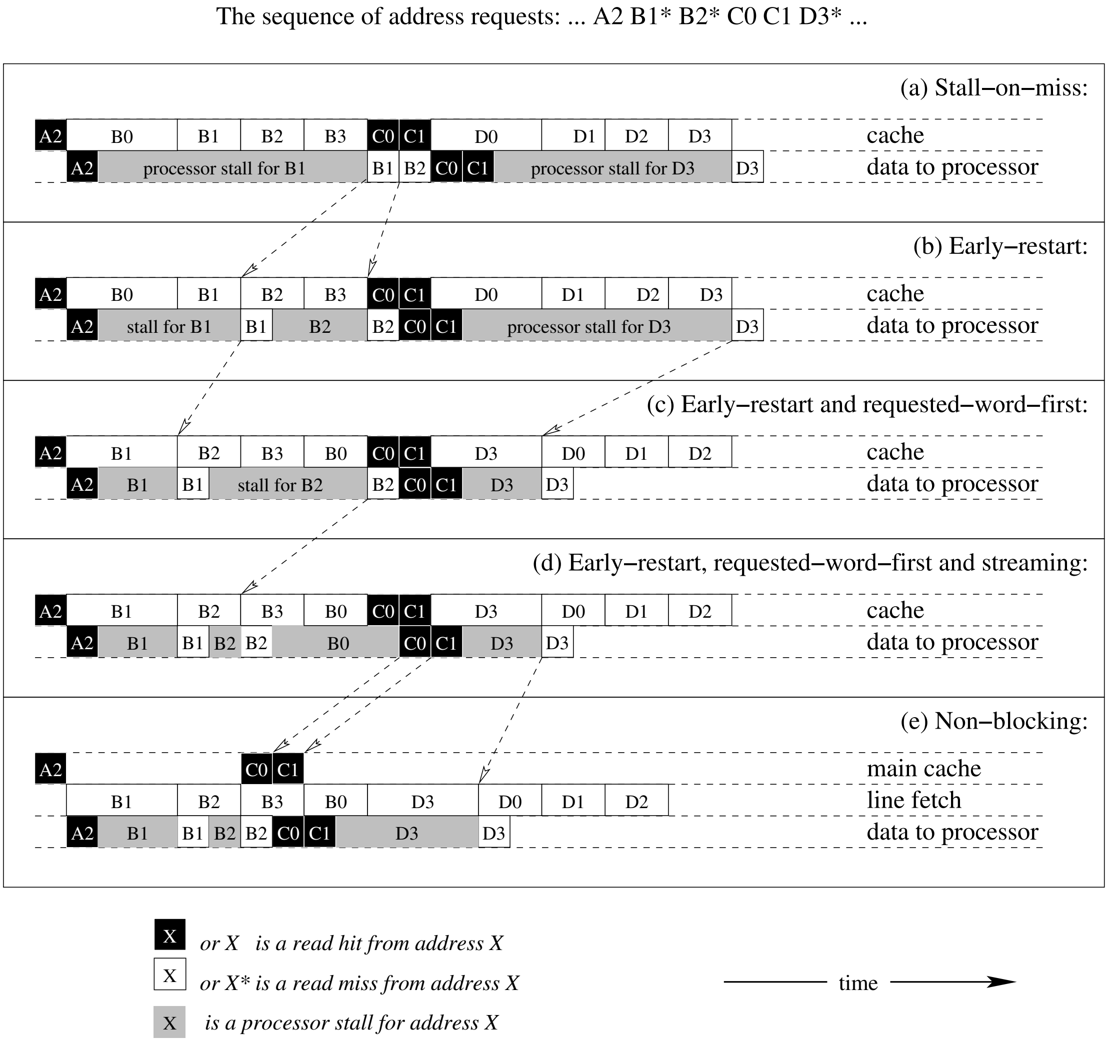


Figure 3.2: Comparison of line fetch schemes

operations. Write requests may be special cases for some write hit policies. In a write-through cache, writes are ‘fire-and-forget’ operations which are unaffected by line fetches. However, writes in a copy-back cache (often used with the write-allocate scheme) and reads with either write hit policy may:

i. cause a cache miss requiring a new line fetch;

ii. be to a word in a line that is still being fetched;

iii. access a line that is already in the cache.

Since there is contention accessing main memory in case (i) either the current line fetch can be abandoned to start fetching this new line into the cache, or the cache waits until the current line fetch process has completed before proceeding with the new line fetch, or the cache could support the concept of partially fetched lines. Although it causes a longer processor stall for the subsequent memory accesses, the second approach (the simplest) guarantees that only completely fetched lines are stored in the cache. Two techniques are described below to reduce the duration of stalls due to subsequent requests falling into the last two cases.

Streaming

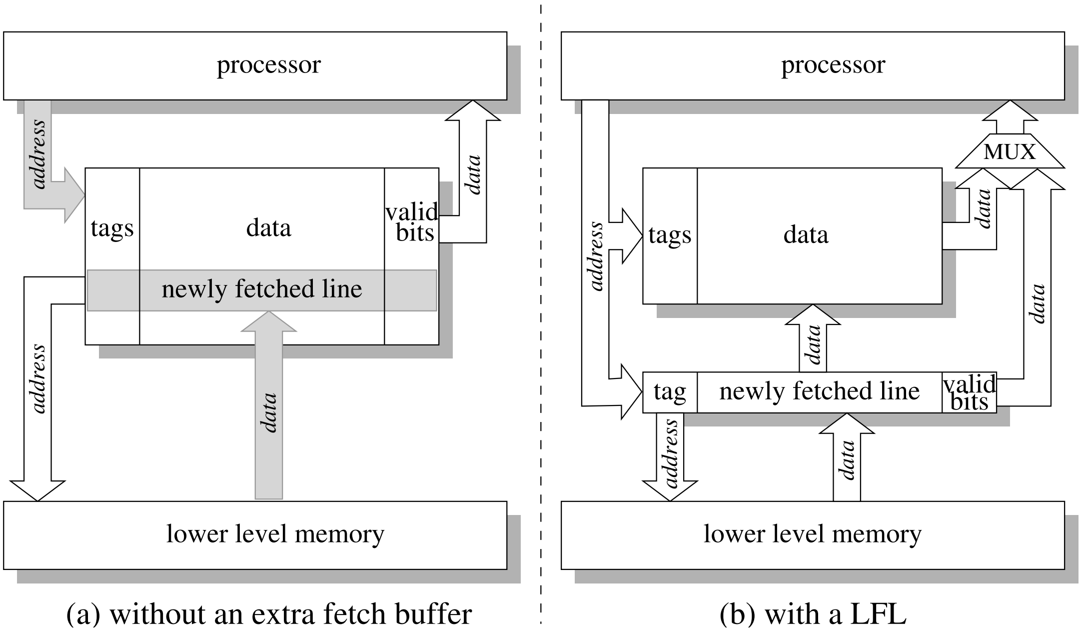
Streaming [72] allows concurrency between the current line fetch process and processor accesses to other words of the same cache line being fetched. This behaviour is useful in many cases such as long instruction fetch sequences. In case (ii) above, the processor can obtain the required data as soon as it arrives in the cache. However, subsequent requests may result in case (iii). With this method, accesses to words other than those in the fetched line still have to wait for the line fetch process to complete before proceeding. Combining the early-restart and requested-word-first approaches with streaming is shown in figure 3.2d.

Non-blocking

If, on a cache miss, the cache cannot continue to serve the processor until the required word was received from the lower-level in the memory hierarchy, then this cache is a *blocking cache*. The blocking cache can be thought of as an in-order cache design; data arrives at the processor in the same order that it was requested.

Kroft's scheme, first known as *lockup-free* [67] and also known as *non-blocking* [79] (figure 3.2e) combines the requested-word first, early-restart and streaming techniques to allow the processor to continue concurrently with a line fetch process proceeding in the background. The situation, described in case (iii) above, known as *hit-under-miss* [41] can be exploited here where the cache has the ability to work on other hit requests, waiting only for memory to supply further misses. A non-blocking cache can be thought of as an out-of-order cache design, by analogy with an out-of-order processor design where the processor does not have to execute instructions in the same order that they were fetched.

When implementing streaming or non-blocking in an asynchronous design environment, processor requests must be synchronised with the incoming fetch data to guarantee that the required data is present in the cache. This is simply done by having an extra ‘valid’ bit (in figure 3.3) for each data word. At the start of a line fetch process, all of these valid bits are cleared, ready for the fetched data. As soon as a word of the data arrives, the corresponding valid bit is set to indicate the presence of data. Subsequent accesses then check for the presence of the required data with these bits. If the data is not present at that time, the processor has to wait. Clearly, implementing this method requires arbitration between the line fetch process and processor requests for access to the cache RAM. figure 3.3a illustrates this simple implementation of a non-blocking cache. The two shaded arrows represent processes requiring arbitration to access the cache. This implementation was used in the TITAC-2 cache system [101].



(a) without an extra fetch buffer

(b) with a LFL

Figure 3.3: Non-blocking caches

Alternatively, instead of fetching a new line directly into the cache, the new fetched line can be held in a dedicated line-length set of latches, a *Line Fetch Latch (LFL)*. Situated outside the cache RAM, the LFL has its own tag and comparator and acts like other cache lines, but unlike those in the RAM block, it is easily made to have multiple ports.

The implementation of a non-blocking cache with an LFL, shown in figure 3.3b, was used in the AMULET2e cache system [35]. With the same 'valid bits' technique, this implementation method needs no arbitration. Both the AMULET2e and TITAC-2 cache systems are described in more detail in the next chapter.

3.4.3 Using multiple levels of cache

The increasing performance gap between processors and memories leaves the designer with a choice between making the cache faster (to keep pace with the processor) or bigger (to keep pace with main memory, to reduce miss rate). This gap has become large enough that a single level of cache is often insufficient to overcome the wildly different main memory and processor speeds. Multiple levels of cache offer a cost-effective solution to providing an improved average memory performance [8]. Furthermore, such a system may have a number of potential advantages over a single-level cache including design independency for each cache level, improvement in cache miss penalty and lower power consumption.

Generally, the first level cache (L1) is small and therefore fast enough (for both tag look-up and data access) to be clocked at same speed as the processor in a synchronous system or be able to keep up with the processor in an asynchronous system. This L1 cache often is split into separate instruction and data caches to support the instruction and data fetch bandwidth of processors. This also allows each cache to be designed independently to serve its own purposes.

A second (and maybe even third) level (L2, L3, ...) is often included in high-performance systems to catch accesses that miss in the primary cache. Lower levels of the cache are large enough to intercept many accesses that would otherwise go to main memory. They are usually built with lower associativity (or even as direct-mapped caches) and are unified so that cache lines are dynamically chosen to hold instructions or data depending on the requirements of the program, as opposed to the static partitioning given by separate L1 caches. These lower-level caches reduce the miss penalty when data is not found in the L1 cache.

In the case of a multiple-level cache the calculation for the memory hierarchy performance from equation 3.1 would be [41]:

$$T_{avg} = T_{hitL(1)} + (M_{L(1)} \times T_{penaltyL(1)}) \qquad \text{Equation 3.2}$$

Where $T_{hitL(1)}$ and $M_{L(1)}$ represent the access time and the miss rate respectively for the first-level of the *N*-level cache and its miss penalty, $T_{penaltyL(1)}$, can be derived from [41]:

$$T_{penaltyL(i)} = T_{avgL(i+1)} = T_{hitL(i+1)} + (M_{L(i+1)} \times T_{penaltyL(i+1)}) \qquad \text{Equation 3.3}$$

Where $T_{penaltyL(i)}$ and $T_{hitL(i)}$ represent the miss penalty and the access time respectively for the i^{th}-level cache. Typically $T_{penaltyL(i)}$ is shorter than $T_{penaltyL(i+1)}$ and $T_{hitL(i)}$ is faster than $T_{hitL(i+1)}$. $M_{L(i)}$ is, using Hennessy's and Patterson's terminology [41], a *local miss rate* for the i^{th}-level cache, L(i). A local miss rate of a particular cache level is calculated from the number of misses incurred in this cache alone divided by the total number of incoming memory accesses to this cache *only*, as opposed to a *global miss rate* of a cache which is the number of misses incurred in the cache as a fraction of all memory accesses made by the processor.

The earlier example from the beginning of section 3.4 is re-evaluated for a multiple-level cache architecture as illustrated in figure 3.4. To keep this example simple, and in keeping with the previous techniques, a non-blocking scheme as described in section 3.4.2 is included. Therefore, the average memory hierarchy performance for the single-level cache (figure 3.4a) can be re-calculated as $T_{avg} = 10ns + (0.1 \times 50ns) = 15ns$. In figure 3.4b a second-level (L2) cache with 20ns access time connected between the L1 cache and the main memory, is added to this hierarchy. The miss rate of this L2 cache is $M_{L2} = \frac{4}{10} = 0.4$ and so the miss penalty of the L1 cache is $T_{penaltyL1} = 20ns + (0.4 \times 50ns) = 40ns$ i.e. somewhat reduced from 50ns. Lastly the average two-level cache memory hierarchy performance can be calculated as $T_{avg} = 10ns + (0.1 \times 40ns) = 14ns$.

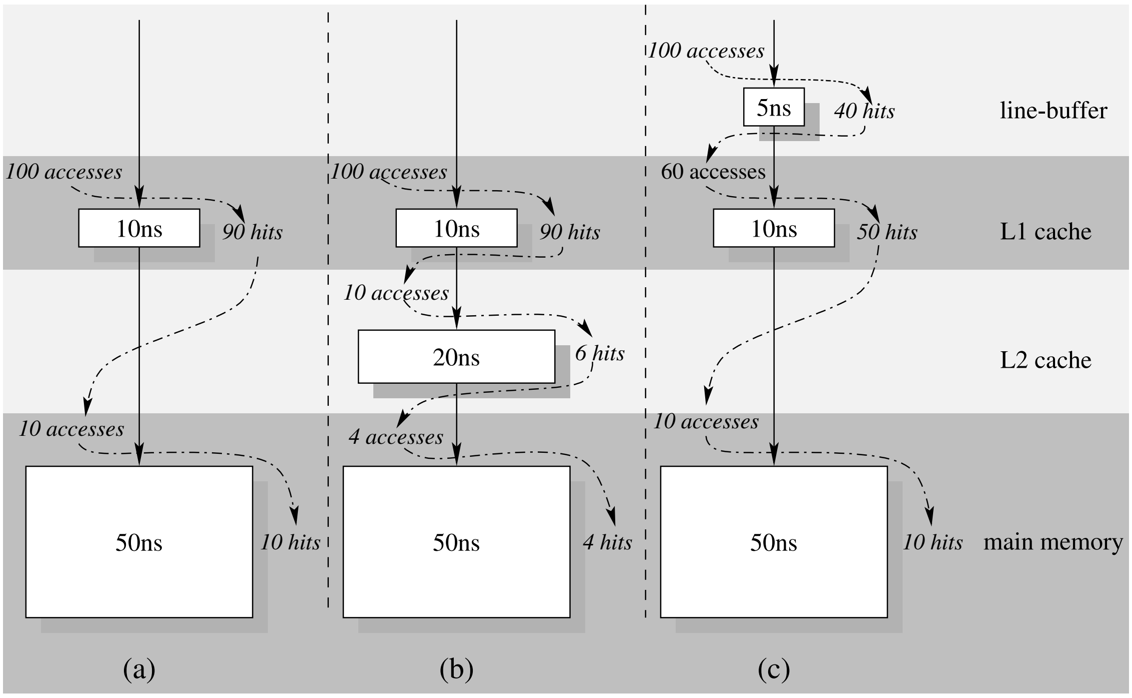


Figure 3.4: Illustration of multi-level cache behaviours

More detailed studies of cache access times and miss rates for various cache parameters focusing on multi-level cache organisations can be found elsewhere [58,82].

Figure 3.4c illustrates another structure of two-level cache. Instead of having a large conventional L2 cache, the fact that most of the code is sequential (spatial locality) can be exploited. A few read-only, line-length latches known as *line-buffers* or *block-buffers* [15,60,62,99], holding recently-read lines, are integrated to reduce the number of accesses to the main L1 cache. Unlike an L2 cache that lies between the L1 cache and the main memory, these line-buffers are placed between the processor and the L1 cache in the manner of an L0 cache. A basic line-buffer internal structure is shown in figure 3.5.

The benefits of the line-buffering technique are not only power consumption reduction and a decreased number of main cache accesses but, because the size of these line-buffers is small, their tag look-up is just a simple, fast address comparison and, because they can be implemented in the same way as the LFL in the previous section, their data access can be much faster than a normal RAM access. The nature of asynchronous systems is such

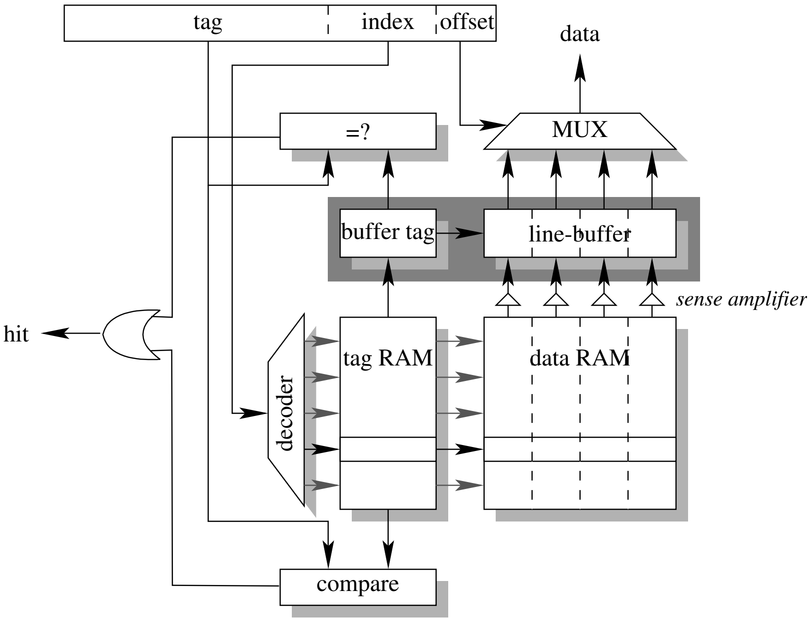


Figure 3.5: Line-buffering

that a wider variation of delay characteristic can be explored, hence this difference in access time can be exploited.

Figure 3.4c illustrates a two-level cache memory hierarchy with the same L1 cache as in the previous example, and a 5 ns access time line-buffer acting as the L0 cache with a 60% miss rate. The average memory system performance for this system can then be re-calculated as $T_{avg} = (5ns \times 0.4) + (0.6 \times (10ns + (0.1 \times 50ns))) = 14.5ns$ which is slightly longer than the ‘proper’ two-level cache structure. This line-buffering method has been exploited in several architectures such as: the SPUR [43], the IBM PowerPC 405 instruction cache [51] and the AMULET3i memory system (although this is not a cache) [34]. Details of the line-buffering used in the AMULET3i memory system will be discussed in depth in chapter 4.

3.5 Hiding latency

Techniques for coping with memory latency are essential to achieve high processor utilisation. Such techniques will become increasingly important in the future as the gap between processor and memory speeds continues to widen. Although latency hiding does not (directly) decrease the time taken for a hit or a miss, it potentially increases overall system throughput.

The latency of writes can be hidden by buffering write accesses with a write buffer as discussed in section 2.2.9. This technique exploits the fact that a processor does not have to wait for a write to complete as long as it observes the effect of future written data. Therefore the processor can perform a write by simply issuing it to the write buffer, provided that future reads check the write buffer for matching addresses. The advantage of a write buffer is not only that the processor does not stall when executing a write, but also that multiple writes can be overlapped to exploit pipelining.

Buffering read accesses is more difficult because, unlike writes, the processor cannot proceed until the read access completes since it needs the data that is being read. With a non-blocking cache it is, however, possible to buffer and pipeline reads as discussed in section 3.4.2.

This section discusses two basic hardware techniques for tolerating memory latency: prefetching and pipelining. These techniques complement the non-blocking cache by allowing concurrency, thus reducing average read latency.

3.5.1 Prefetching

Prefetching involves fetching data from the memory before it is actually needed by the processor. This technique hides the line fetch latency, reducing the miss penalty, if subsequent accesses can be serviced with this prefetched data. Prefetching can be done by using either software or hardware approaches. Software prefetching [16,63] uses the compiler to transform the code, usually by adding extra explicit *fetch* instructions to instruct the hardware which information is to be prefetched. This approach is not considered here.

Hardware-based prefetching, on the other hand, relies on either simple prefetch techniques that fetch a fixed pattern of data or more sophisticated techniques that approximate memory access patterns dynamically. Chen and Baer [19] proposed that this approach is more advantageous than software prefetching since it does not require the use of explicit fetch instruction and operates dynamically at runtime.

All caches, even the simplest, employ some prefetching in that, on a cache miss, a whole line is fetched containing the required data along with some prefetched data in the same line. This can be extended further with a simple sequential prefetch technique proposed by Smith [90], *prefetch-on-miss*, where the cache fetches the next consecutive cache line(s) after the requested line in the hope that this will avoid subsequent miss(es). This has a similar effect to using a larger cache line size. Having a larger cache line is beneficial for consecutive (often called *unit-stride*) code but brings problems of increased memory traffic as discussed in section 3.3.2.

Both instructions and data can be prefetched either directly into the cache or into a buffer outside the cache. This buffer, known by Jouppi as a *stream buffer* [57] (figure 3.6), holds prefetched data and provides it when a subsequent reference requests it. This reduces the miss penalty since accessing the stream buffer is faster than accessing the memory and it does not pollute the cache with non-requested (prefetched) data. The data slot in the stream buffer will be overwritten if the subsequent reference does not demand its contents. The stream buffer completely hides the read latency for unit-stride access code; however it still causes an increase in the total memory traffic and does not help with non-unit-stride access code. With additional *stride detection* hardware, non-unit-stride code can be better handled.

3.5.2 Pipelining

Pipelining is an implementation technique that exploits parallelism and is one of the most common techniques used to improve the performance of processors. It comes from the observation that instruction execution can be split into a number of independent stages chained into a *pipeline*, allowing a number of instructions to be operated upon concurrently, one in each different stage of execution. Pipelined processing is beneficial when all of the following are true [65]:

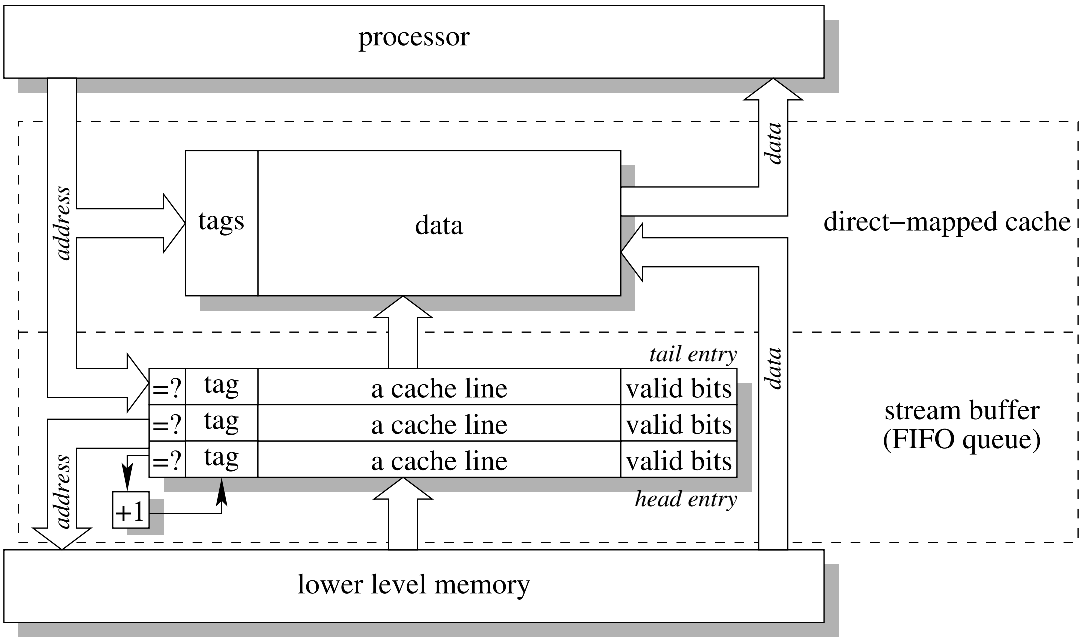


Figure 3.6: Jouppi's stream buffer organisation

- Each task is relatively independent from the previous one.
- Each task requires approximately the same sequence of stages.
- The durations of time required by each of the different stages are approximately equal. (For asynchronous pipelining, the time per stage may not be constant but rather a function of both the stage and the data passing through it.)

In the same way that modern processors are pipelined to allow overlapping of memory accesses with instruction execution, thus hiding memory latency, the multiple stages of a cache access can be pipelined for similar effect. This requires the processor to support either multiple outstanding memory accesses or multi-port memory (e.g. a Harvard architecture) or both.

Pipelining is actually complementary to other techniques used to improve memory hierarchy performance, including prefetching and non-blocking operation. It improves performance by increasing the number of outputs in a given time (throughput), as opposed to decreasing the time taken for an individual element to traverse the pipeline (latency).

In synchronous systems data is stepped through the pipeline by the clock. Each stage is constructed so that its processing is finished within a time slot dictated by the clock signal. However, since a memory access is, relatively, much slower than processor execution of an instruction, a memory access is usually designed to use multiple clock cycles in order not to degrade the performance of other parts of the processor.

An example timing diagram of a simple synchronous pipeline in a memory hierarchy is shown in figure 3.7a. The clock period of the synchronous pipeline is limited to the minimum time taken by the slowest pipeline stage (RAM read rather than tag look-up) to complete its processing.

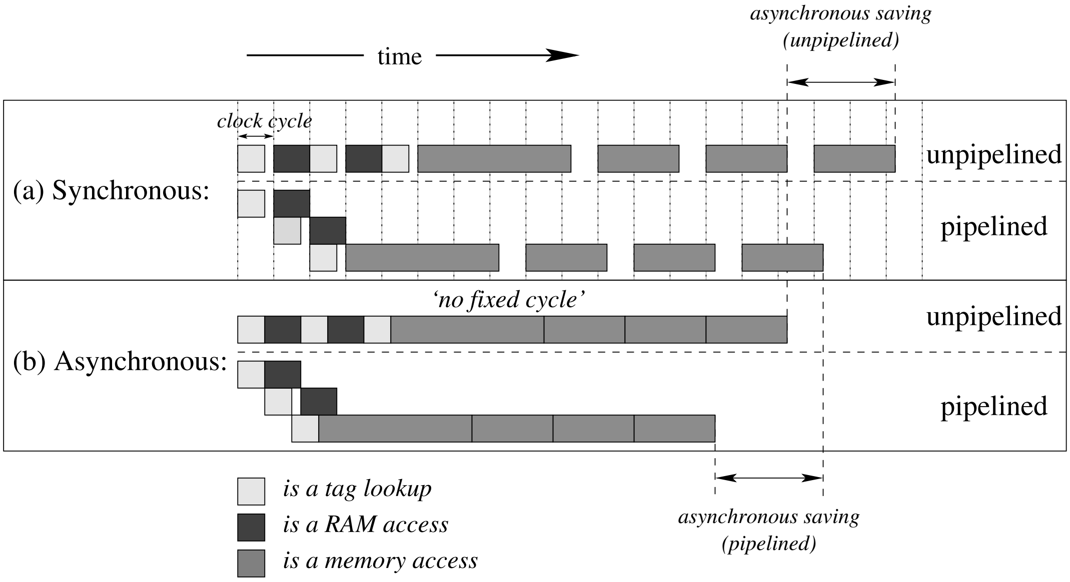


Figure 3.7: Asynchronous vs synchronous cache pipelining

By contrast, an asynchronous pipeline does not have global clock, hence every stage can take a variable time to finish and can work independently. Therefore the next stage can begin as soon as the previous stage has finished which theoretically allows an asynchronous pipeline to be faster than a synchronous one. Figure 3.7b gives an example of how the timing of an asynchronous pipeline may look.

3.6 Reducing memory traffic

This section describes techniques for improving memory hierarchy performance by reducing the total memory traffic. All of the earlier techniques reduce latency related stalls but also increase the traffic between main memory and the processor. The main benefit of reducing the traffic is the power saved by not going off-chip to access external main memory, but lessening the traffic could also aid in reducing the miss penalty. Two common techniques, *write merging* [41] and copy-back, are discussed in this section.

3.6.1 Write merging

Cache lines are usually larger than the size of any single piece of write data. Many modern write buffers have the ability to merge memory writes to save both write buffer space and memory traffic. This can be done by bringing together a new write operation with a previous write operation already resident in the write buffer. The new write is placed in the same write buffer entry as an existing write when the address of the new store falls inside the line address range of the existing entry. By this means two or more writes to the same location can be collapsed into one write or two or more writes to sequential locations in the same cache line can be merged into a single buffer entry and then written out using a high speed memory burst of the type introduced in section 2.2.7.

3.6.2 Copy-back write policy

The fundamental cache activities affecting write policies are reviewed in figure 3.8. At the beginning of each access is a comparison to determine whether the request is to a cacheable location. Uncacheable instruction or data accesses are passed on directly to the system bus and the operation (read-write) performed on the main memory.

The least complicated operation is a read hit in the cache when the data is simply read out straight from the cache and sent to the processor. However, in some (multi-level) cache systems, including the one described later in chapter 5, an extra activity might be required to update the higher level in the cache system. A read miss is slightly more complicated: the line fetch process fetches the required data from the main memory (or a lower level in the memory hierarchy) along with data close to it.

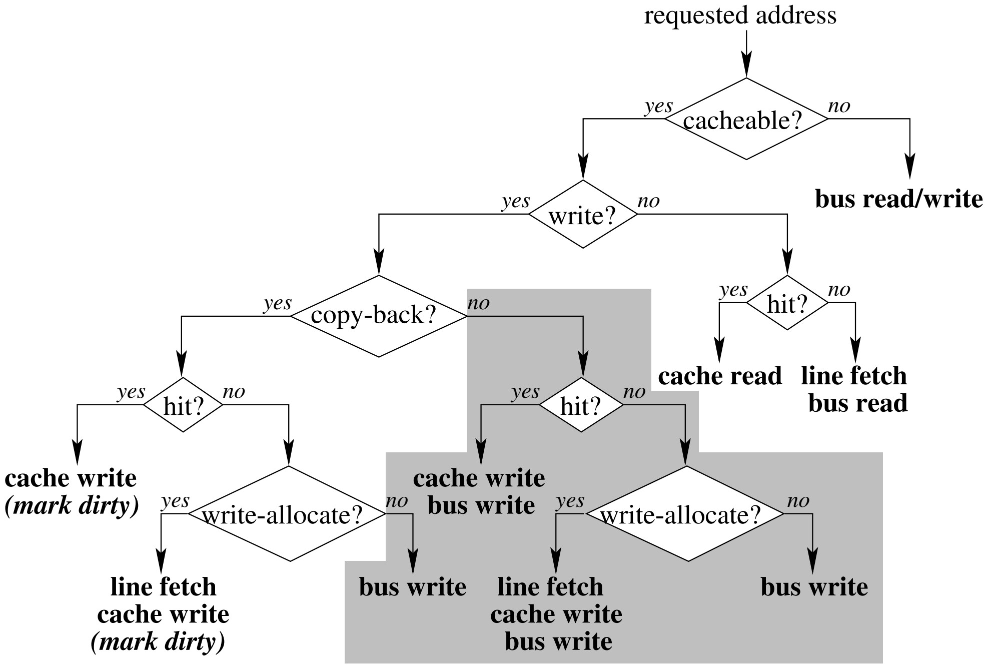


Figure 3.8: Basic cache operations

Whilst activities on a read access are fairly simple, there are several issues (policies, and techniques) involved in a write, some of which need to be decided at a very early design stage.

Note here that the cache architecture developed in this thesis is a copy-back cache with write-allocate policy, hence those activities in figure 3.8 shown on a shaded background are not applied to this particular cache. However, for comparison purposes, operations in a simple write-through cache with write-around policy (some in shade) are also described here.

In a write-through cache, a write is always performed in the main memory as well as in the cache if it is a hit. This clearly ensures that the data in the main memory is kept up-to-date. Whilst any choice of write miss policy could be used here, the obvious (simplest) choice is the write-around policy because this avoids any additional fetch activities on a write miss (as discussed earlier in section 2.2.8).

Although a write-through cache is less complicated, a copy-back scheme provides better performance, especially for writes. This is principally because memory bandwidth requirements are reduced by avoiding redundant write cycles. On a write hit, the write is performed into the cache only. To maintain data coherence between the cache and the memory, the dirty data is written into the main memory only when ejected from the cache. Applying the write-allocate policy to the cache makes the design slightly more complicated because, on a write miss, instead of just writing data into main memory a line fetch is also triggered and a whole line is fetched into the cache so that the write operation can take place locally. Furthermore, subsequent reads and writes can then be serviced from the cache. The write-allocate approach also eases the process of forwarding from the write buffer since there are only complete cacheable valid lines stored in the write buffer.

A short theoretical analysis shows how dramatic reductions are generated by a copy-back policy:

$$T_{avg} = T_{RH} + T_{RM} + T_{WH} + T_{WM} \qquad \text{Equation 3.4}$$

Where T_{RH}, T_{RM}, T_{WH} and T_{WM} are the contributions of cache read hit, read miss, write hit and write miss consecutively. The read hit contribution in both write-through and copy-back caches is:

$$T_{RH} = R \times H \times T_{hit} \qquad \text{Equation 3.5}$$

Where R represents the fraction of total read accesses over all accesses (for instruction and data) and H represents the hit rate.

Whilst a write hit in a copy-back cache only has to proceed in the cache, in a write-through cache a write operation must be performed in the main memory. The write hit contribution for write-through and copy-back caches can then be derived respectively as follows:

$$T_{WH(write-through)} = W \times H \times T_{MEMwrite} \qquad \text{Equation 3.6}$$

$$T_{WH(copy-back)} = W \times H \times T_{hit} \qquad \text{Equation 3.7}$$

Where W represents the percentage of total write accesses, $T_{MEMwrite}$ is the time taken to update the main memory with the assumption that $T_{MEMwrite} >> T_{hit}$. Since a line fetch occurs on a read miss, the read miss contribution in a write-through cache is simply:

$$T_{RM(write-through)} = R \times M \times (T_{Rpenalty} + T_{hit}) \qquad \text{Equation 3.8}$$

Where $T_{Rpenalty}$ is the miss penalty for fetching a line. A line allocation in a copy-back cache involves a line eviction (needing to write dirty data back to the memory). Therefore, in the absence of a write buffer (either for decoupling the processor and the main memory or for decoupling copy-back allocation) the read miss contribution is given by:

$$T_{RM(copy-back)} = R \times M[(T_{Rpenalty} + T_{hit}) + (D \times T_{Wpenalty})] \qquad \text{Equation 3.9}$$

Where D represents the percentage of dirty data amongst the evicted lines and $T_{Wpenalty}$ is a miss penalty for updating the main memory with a dirty evicted line and is directly related to $T_{MEMwrite}$.

In a write-through cache (assuming a write-around policy) a write miss contribution is similar to that of the write hit in the same cache, however, in a copy-back cache (with a write-allocate policy), it is similar to that of the read miss in the same cache. Write miss contributions are thus:

$$T_{WM(write-through)} = W \times M \times T_{MEMwrite} \qquad \text{Equation 3.10}$$

$$T_{WM(copy-back)} = W \times M[(T_{Rpenalty} + T_{hit}) + (D \times T_{Wpenalty})] \qquad \text{Equation 3.11}$$

Simplifying the above formulae gives the memory hierarchy performance of write-through and copy-back caches respectively as:

$$T_{avg(write-through)} = R[T_{hit} + (M \times T_{Rpenalty})] + (W \times T_{MEMwrite}) \qquad \text{Equation 3.12}$$

$$T_{avg(writhe-through)} = T_{hit} + M[T_{Rpenalty} + (D \times T_{Wpenalty})] \qquad \text{Equation 3.13}$$

For a read:write accesses ratio of 9:1 with a miss rate of 5% in a 4-word line size cache that has a 10ns access time and is connected to a 50ns access time main memory, and assuming that 10% of evicted lines are dirty in the copy-back cache, with no write buffer, these give:

$$T_{avg(write-through)} = 0.9 \times [10ns + (0.05 \times (50ns \times 4))] + (0.1 \times 50ns) = 23ns$$

$$T_{avg(copy-back)} = 10ns + 0.05 \times [(50ns \times 4) + (0.1 \times (50ns \times 4))] = 21ns$$

When a non-blocking scheme is applied, $T_{Rpenalty}$ can be reduced from $50ns \times 4$ =200ns to 50ns, a miss penalty only for the required word thus reducing both $T_{avg(write\text{-}through)}$ and $T_{avg(copy\text{-}back)}$ to 16.25ns and 13.5ns respectively. Furthermore, when a memory bursting mode (2-1-1-1) is applied, $T_{Wpenalty}$ can also be then reduced from 200ns to (50ns+25ns+25ns+25ns)=125ns. Overall $T_{avg(copy\text{-}back)}$ is reduced to 13.1ns with ~20% improvement over the write-through cache.

3.7 Other Notable Techniques

Two other techniques are commonly encountered in cache systems. Neither of these has a direct impact on a cache's general-purpose average performance but each offers specific benefits of common interest.

3.7.1 Sub-blocking

An architecturally different cache organisation strategy to reduce cache power dissipation is to break a cache data array into multiple sub-blocks [15,60,99]. Only the cache sub-block where the requested data may be located is addressed for each cache access. This technique saves power by making each access across a smaller cache. The proportion of power saved depends on the number of cache sub-blocks. *Sub-banking* as it is also known, is very attractive to computer architects designing energy-efficient microprocessors. A basic structure for cache data array sub-blocking is presented in figure 3.9.

3.7.2 Cache lock-down

Since caches are transparent to user software, predicting the exact performance of a program in a system with a cache is difficult. This is an undesirable effect in many embedded systems which require real-time response. A technique commonly used in embedded systems to ensure deterministic behaviour is to load critical code into the cache under supervisor software control and then, via special hardware support, prevent it from being evicted. This process is known as *cache lock-down* [4]. Clearly, locking down most

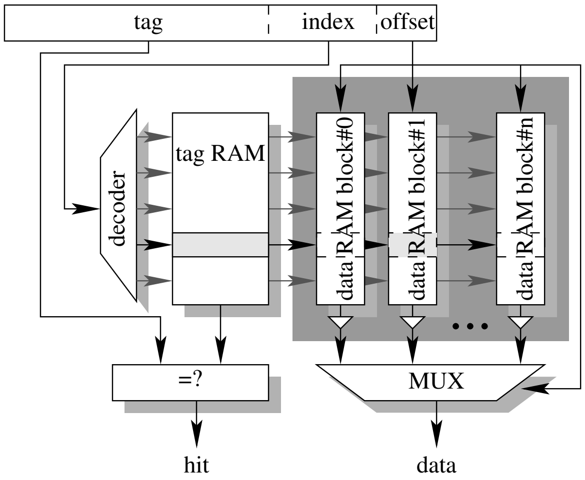


Figure 3.9: Cache RAM array sub-blocking

of the cache compromises its ability to accelerate the general performance of the machine, so it is important to have control of the lock-down mechanism at a fine granularity.

3.8 Commercial Cache Implementations

Many of the techniques described in this chapter are not new – they have been used in both high-performance and low-power cache designs in the past. To illustrate a range of trade-offs that have been made in choosing parameter values and various cache techniques, a few recent examples of practical cache systems are described below.

3.8.1 The AMD-K6-III cache system

The cache system for the AMD-K6-III [2] is composed of on-chip 64kilobyte L1 and 256kilobyte L2 caches and an optional L3 off-chip cache. The L1 cache is organised as separate 32kilobyte instruction and 32kilobyte dual-ported data caches, each with two-way set associativity. The L2 cache is organised as a 4-way set associative, unified cache. Both the L1 and the L2 caches exploit copy-back with write-allocate policies. The L1 can be filled from either the L2 cache or the external memory. Whilst the L1 instruction and the L2 caches use the LRU replacement strategy, the L1 data cache uses the LRA (Least Recently Allocated), another name for cyclic eviction.

3.8.2 The Intel Pentium 4 cache system

The Intel Pentium 4 architecture [52] includes a split L1 cache (a 12 kilobyte *Execution Trace Cache* and an 8 kilobyte data cache) and an on-chip unified L2 cache.

The difference between the execution trace cache and a conventional instruction cache is that instead of storing x86 instruction bytes, the trace cache stores decoded *micro-operations (micro-ops)* which removes the decoder from the main execution path, thereby increasing performance (by storing the micro-ops in the cache, cache hits can begin execution sooner because they have already been decoded). The data cache is write-through, 4-way set-associative and dual-ported to allow one load and one store per clock cycle. The *Advanced Transfer Cache* is Intel's new name for the 256 kilobyte, 8-way associative, non-blocking, unified L2 cache used in the Pentium 4 architecture.

3.8.3 The Intel StrongARM SA-1110 cache system

The Intel StrongARM SA-1110 processor [53] implements the ARM V4 architecture. It contains a 16 kilobyte instruction cache and an 8 kilobyte data cache. Both caches have 32 byte lines and provide 32-way set-associativity with a round-robin replacement style. In addition to this, a 16-entry, 2-way set associative mini-cache with the LRU algorithm is provided to prevent periodic large data transfers from thrashing the main data cache. The processor also provides a write buffer and a separate read buffer. The write buffer has eight entries and allows each entry to contain between 1 to 16 bytes. The read buffer, allowing critical data to be prefetched under software control to prevent pipeline stalls from occurring during external memory reads, has four entries and allows each entry to contain 1, 4 or 8 words.

3.8.4 The ARM940T cache system

The ARM940T [5] has separate data and instruction caches. Each is four kilobytes in size and comprises four 64-way associative CAM-RAM blocks with 4-word lines. An 8-word non-merging write buffer is also included. The data cache supports both write-through and copy-back modes with the write-around policy. A *lock-down mechanism* is also exploited to lock critical or frequently-accessed references (either instruction or data) in the cache. This lock-down mechanism has a granularity of a single 128-bit cache line across each of the four cache blocks of the same cache, hence the smallest area that can be locked down is 16 words.

3.8.5 The Sun UltraSPARC III cache system

The UltraSPARC III [106] features 100kilobytes of on-chip L1 caches. These are organised as 32kilobyte instruction and 64kilobyte data caches plus two kilobyte prefetch and two kilobyte write caches, each with 4-way set associativity. The two latter caches work in conjunction with memory and register addressing to buffer read-write processor data within a high-speed, low latency cache memory area. The prefetch cache is independent from the data cache and can load data when this is deemed appropriate. The write cache acts like a write buffer by deferring writes to the L2 cache and also exploits write merging by evading unnecessary writes of individual bytes until entire cache lines have to be updated.

Also integrated on the chip are the tag RAM and controller supporting a 1, 2 or 8 MB, 2-way set-associative off-chip L2 cache.

3.8.6 The IBM PowerPC 405 cache system

The PowerPC 405 [51] 32-bit RISC embedded processor implements separate instruction and data caches. Each has configurable size (the PPC405B3 has a 16 kilobyte instruction cache and an 8 kilobyte data cache), is two-way set-associative, and operates using 8-word (32 byte) cache lines. The caches are non-blocking to allow the PowerPC 405 to overlap instruction execution with reads over the processor local bus. The LRU replacement policy is used to replace cache lines. An instruction line-buffer is included on the instruction cache access, four instructions are read from the appropriate cache line and placed temporarily here. Subsequent instruction cache accesses can then check this line-buffer for the requested instruction prior to accessing the cache array. The data cache functions in either write-through or copy-back mode with the write-around scheme.

3.9 Discussion

The above cache systems exemplify the cache architectural techniques presented here in synchronous implementations. This section considers the possibility of using these techniques in the context of an asynchronous framework.

All of the examples in section 3.8 have a number of notable features in common: separate instruction and data L1 caches, each using set-associativity and (with the exception of the AMD K6-III data cache) using the LRU replacement strategy to choose victim lines.

The optimal choice of set associativity among the previous examples is undecided since they vary from low (4-way) up to full associativity in each cache block (64-way). Although the tag check for CAM-tag is expensive because the tag is broadcast to the CAM in order to find the proper line for the data, high associativity provides good support for lock-down mechanisms because locking down cache lines causes a more noticeable degradation in performance for low-associativity caches. For example, in a 4-way associative cache, locking down one line reduces the associativity by 25% whereas locking down one line in a 64-way associative cache cuts the choice by only 1.5%, giving a much more gradual degradation in performance as lines are locked down.

The idea of multiple levels of cache is attractive in the context of asynchronous design where a wider variation in access time can be exploited in a manner that would prove expensive and difficult in a synchronous framework. This is because each unit in a synchronous system must complete its task in an integer number of clock cycles. However, the idea of having multiple-levels of cache on-die is questionable since this study is aimed at small low-power systems, like those built around the ARM processors where the L1 cache size tends to be small. Alternatively, L0 caches such as the line-buffer in the IBM PowerPC 405 cache system are more desirable. As covered in section 3.4.3, line-buffering not only reduces the number of L1 cache accesses resulting in dramatic power-savings but also, due to the nature of asynchronous design, the fast(er) access time of the line-buffer equates to an improved average performance of the cache system as a whole.

To combine line-buffering and cache sub-blocking (two of the most common memory techniques in energy-efficient design) effectively, the cache should be sub-blocked differently from the scheme presented in section 3.7.1, e.g. as in the Multiple-Divided Module (MDM) cache [64] or as shown for two blocks in figure 3.10. This style of sub-blocking is regularly used in embedded systems such as those of interest in this thesis. Each block/bank caches a different region of the address map, often using an interleaved mapping allowing consecutive references to be spread over a number of cache blocks by using low-order bits to select the block.

Zhang and Asanovic [108] reported that when applying sub-blocking, aside from a 10% area overhead, there is no significant performance penalty in terms of either the access latency or the energy efficiency associated with the choice of a tag CAM over a tag RAM. The favoured choice here is to use a sub-blocked L1 cache architecture with full-associativity in each cache block and a line-buffer as a L0 cache.

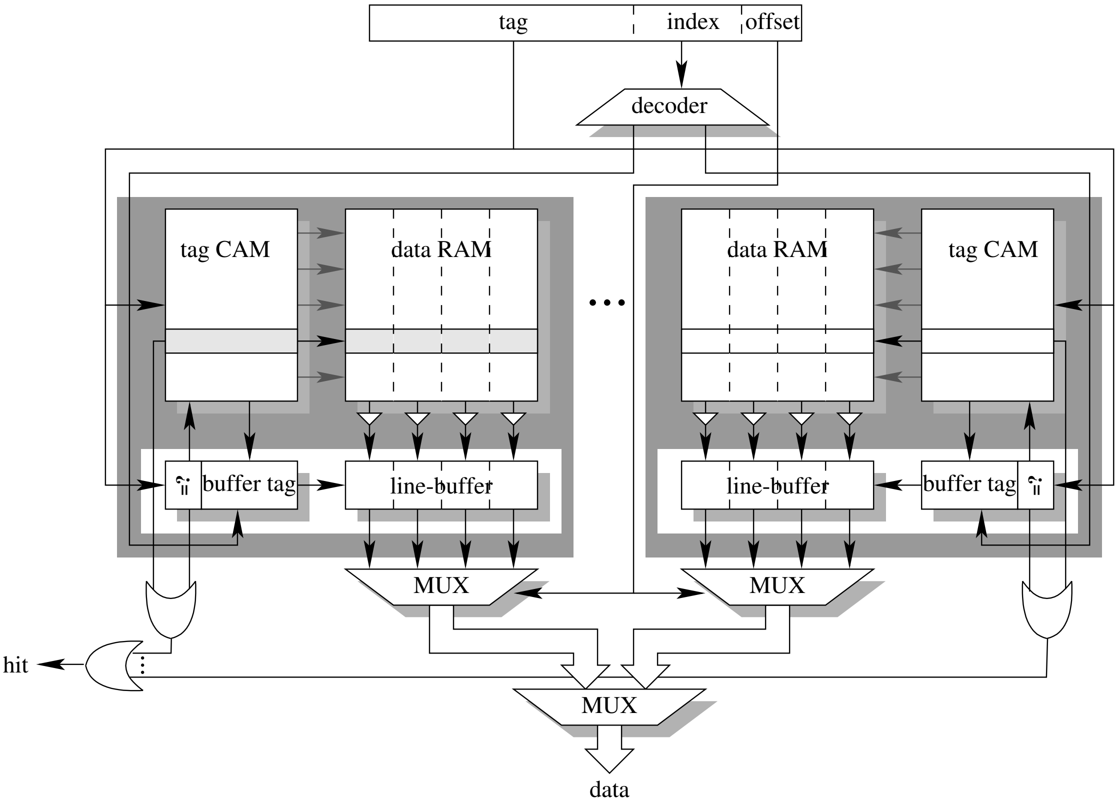


Figure 3.10: Combining line-buffering and cache sub-blocking

By exploiting general parallelism (e.g. pipelining) or the overlap of processor computations with data accesses within one process (e.g. write buffering, non-blocking and prefetching), memory latency can be improved. Since prefetching and non-blocking are not mutually exclusive (exploiting *pre*-miss and *post*-miss operations respectively), Chen and Baer [20] proposed a hybrid design, combining these two approaches to reduce the memory latency penalty. The non-blocking technique has become popular and easy to build because most caches today have a pipelined structure and use prefetching. Furthermore, previous work [35,101] has shown that a non-blocking architecture can be implemented in an asynchronous design. Chen and Baer [20] also confirmed previous studies [57] indicating that write buffering can remove a significant miss penalty when reads are allowed to overtake writes.

In a copy-back cache system, the write buffer (for cacheable references) can be modified into a victim cache. In a write-through cache system the write buffer and the victim cache must be two separate buffers; for storing write data (usually at a single word granularity) to the memory and storing evicted lines for forwarding purpose respectively. Write-merging is an effective technique for reducing write traffic in a write-through cache. However, since a copy-back cache absorbs writes in the cache, and so can offer a significant reduction in write traffic, write-merging is then less necessary.

3.10 Summary

The literature surveyed in this chapter concentrated mostly on the architectural level hardware-based techniques that use alternative cache organisations for improving memory hierarchy performance. Improving one aspect of the cache performance usually comes at the expense of others. A number of recent commercial cache implementations were also described in this chapter to give a broader view of current trends in cache design.

Most of these techniques can be easily applied in an asynchronous system. However, non-blocking, read-overtaking-writes, forwarding and write-merging would present problems since they all require some degree of undesirable synchronisation. The next chapter describes notable existing self-timed memory systems in depth. Chapter 5 then describes an approach to apply the techniques presented in this chapter in an asynchronous framework.

Chapter 4: Asynchronous Memories

The previous chapter contained a survey of relevant hardware techniques for improving memory hierarchy performance. All of the techniques discussed were originally introduced in synchronous designs. However, they may all be used in asynchronous designs as well, although the costs, benefits and difficulties encountered may differ.

Asynchronous systems promise a number of advantages over synchronous systems. Efficient asynchronous memory systems are, therefore critical to the success of asynchronous systems.

Much literature has been published in the past two decades describing synchronous cache organisations that exploit a whole range of architectures, strategies and mechanisms with varying levels of complexity and development, but very little work has been presented in the area of asynchronous caches.

This chapter describes existing previous asynchronous memory systems with the primary focus on low power embedded systems where asynchronous design would appear to offer the most advantages.

4.1 Asynchronous processor survey

Asynchronous design was used in the 60s and 70s in high performance mainframe systems such as the MU5, constructed at the University of Manchester [74]. However, with the development of integrated circuit techniques the synchronous design style became dominant because of the simple, global timing constraint that it imposes.

Asynchronous design was reborn in the late 80s with the world's first entirely asynchronous VLSI microprocessor [68] built at the California Institute of Technology

(Caltech) by a team under the leadership of Martin. Since then, a number of asynchronous processors have been proposed or built by other academic research groups and industrial laboratories. A detailed description and comparison of some notable asynchronous processors (designed between 1989 and 1996) was published by Werner and Akella [104]. The widespread nature of this continued research into asynchronous systems can be seen in the following sections. Emphasis here is placed on the memory architectures employed which often restrict the core performance.

Caltech asynchronous microprocessor (1989)

This microprocessor [68] has a 16-bit RISC-like instruction set. The processor was constructed using an approach based upon Hoare's Communicating Sequential Processes (CSP) [44] with eight processes, each of which can be thought of as a separate pipeline stage. Instruction and data memories are separated.

STRiP: Self-Timed RISC Processor (1992)

STRiP [24], built by Dean at Stanford University, was based on the MIPS-X processor architecture. Even though the processor has a global clock signal and could be considered synchronous, it is unusual in that the speed of the global clock is self-adjusting to sequence the pipeline structure. This provides much of the advantage of an asynchronous system whilst avoiding the complexity and overhead of fully asynchronous structures.The processor was still susceptible to clock skew problems and did not provide any reduction in power dissipation. STRiP incorporates small, separate instruction and data prefetch buffers lying between the processor and the separate L1 caches.

FAM: Fully Asynchronous Microprocessor (1992)

FAM [21] was developed by Cho from the Korean Institute of Science and Technology and Okura and Asada from the Tokyo Institute of Technology. The processor has a load/ store four-stage (fetch, memory, decode and execution) pipelined RISC architecture with unified external cache memory requiring arbitration between instruction fetch and execution stages (for load/store instruction completion) to access the cache.

NSR: Non-Synchronous RISC processor (1993)

A simple 16-bit processor [14] was developed using FPGA technology by Brunvand at the University of Utah. The processor was structured as a collection of asynchronous blocks operating concurrently. In addition to being internally self-timed, the pipeline stages are decoupled through self-timed FIFO queues, allowing a high degree of instruction execution overlap. The memory system was implemented as simple, separate memories for instructions and data.

CFPP: Counter-Flow Pipeline Processor (1994)

An innovative architecture [97] was proposed by Robert Sproull et al. at Sun Microsystems Labs. The name of the processor came from its fundamental feature: instructions and data results propagate in opposite directions in a bidirectional pipeline and interact as they pass. This interesting approach neatly solves the problem of result forwarding in an asynchronous pipeline. The CFPP executes SPARC instructions with separate memory ports for instructions and data.

Fred (1996)

A self-timed decoupled, pipelined computer architecture [85] was extended from NSR by Richardson and Brunvand at the University of Utah. It is a 32-bit processor based on the Motorola 88100 RISC instruction set. The memory system was designed as simple, separate memories for instructions and data.

MiniMIPS (1997)

An asynchronous microprocessor [69] executing a reduced MIPS instruction set (hence the name) similar in architecture to the MIPS R3000 was developed by Martin et al. at

Caltech. The architecture was based on very fine pipelining offering high throughput. The system also included an on-chip cache system which is described in Section 4.2.3.

TITAC: Tokyo Institute of Technology Asynchronous Chip (1994)

A simple asynchronous 8-bit processor [77] was built at the Tokyo Institute of Technology. The architecture is non-pipelined with a simple accumulator-based instruction set. TITAC was optimised for delay-insensitivity rather than performance. All of its memory requirements were met by using RAM.

ECSTAC (1995)

A simple 8-bit asynchronous (deeply) pipelined microprocessor [75] was designed at the University of Adelaide. It has been reported that its variable length instruction format and the mismatch between the datapath width (8-bit) and the address size (24-bit) caused some complex design problems and also reduced the system performance. The system incorporated an on-chip fully asynchronous cache system [3] which is described further in section 4.2.1.

Hades: Hatfield Asynchronous DESign (1995)

The design [26] was proposed as an asynchronous superscalar processor to act as a test bed for assessing alternative asynchronous processor organisations at the University of Hertfordshire, UK. It was designed with four pipeline stages (fetch, decode, execute and writeback) and a decoupled result forwarding mechanism. It has its own RISC-like instruction set and a complex ‘multiple-instruction-issue’ design. The proposed architecture included separate instruction and data caches, however the cache designs are not described in the literature.

TITAC-2 (1997)

An asynchronous 32-bit microprocessor [101] with the MIPS five-stage pipelined architecture was developed at the University of Tokyo. It is based on the MIPS R2000 instruction set. It included a cache system which is described in section 4.2.2.

ASPRO-216 (1998)

A 16-bit RISC standard-cell asynchronous microprocessor [84] was developed for embedded applications at E.N.S.T. in Bretagne, France. Instructions are issued in-order but are allowed to complete out-of-order and a register locking mechanism is adopted to solve data dependencies. The processor includes standard synchronous on-chip 48 kilobyte instruction and 64 kilobyte (byte or word addressed) data memories.

Kin (1998)

A high performance asynchronous superscalar processor architecture [66] was proposed by Kol and Ginosar at Technion-Israel Institute of Technology. The architecture was designed at the microarchitectural level to allow for future technologies (predicted for the year 2012) enabling more than one billion transistors per chip with extremely fast processing obtained by aggressively exploiting massive out-of-order execution and parallelism to speed processing and bypass both control and data dependencies. A high performance cache memory is required to support such architectures. The proposed high performance memory system for Kin is described in section 4.2.4.

AMULET series (1993, 1996, 2000)

The AMULET group, a part of the Computer Science Department at the University of Manchester, was established late in 1990 in order to investigate the claimed advantages and the feasibility of designing large asynchronous systems. One aim of this group is to realise asynchronous microprocessors with lower power consumption than are currently available using synchronous design techniques. Since then the group has developed and fabricated three asynchronous RISC processors capable of executing ARM code [4].

The AMULET1 microprocessor [29], the first asynchronous implementation of the commercially popular ARM instruction set, showed the feasibility of implementing a whole system asynchronously. All of the design effort was put into the processor including the difficult areas of interrupts and exceptions in a self-timed environment. It did not incorporate an on-chip memory system.

The AMULET2e system [32] is an *embedded* asynchronous system based on the AMULET2 processor core. It proved that asynchronous design could achieve competitive performance with good EMC results compared to an equivalent synchronous system. The AMULET2e chip included on-chip memory that could be configured as either memory-mapped RAM or as a write-through cache.

Recently the DRACO (DECT RAdio Communications Controller) chip [34], a telecommunications controller intended for ISDN (Integrated Services Digital Network) DECT (Digital European Cordless Telephone) base station applications, was developed as a commercial collaboration. It comprised the AMULET3i asynchronous processing subsystem and a synchronous telecommunications peripheral subsystem. This showed that asynchronous technology is becoming commercially viable and is competitive in terms of performance, area and power efficiency with synchronous design. The AMULET3i incorporated a memory-mapped RAM system that could not be configured to operate as a cache.

The cache memory proposed in this thesis adapts features from the AMULET2 and AMULET3 memories and adds some new features, principally related to the requirement for a copy-back write strategy to support the higher processing speed of AMULET3.

This long list describing many existing self-timed processor designs provides the rough state of the emergence of this design style. However, only a few attempts have been made to construct efficient, suitable memory systems to support these processors.

The remainder of this chapter describes previous asynchronous memory systems. Then, a more detailed coverage of the asynchronous memory systems built by the AMULET group for use with AMULET processor series is given in section 4.3. The chapter then concludes by presenting observations regarding noteworthy techniques and design styles from these existing asynchronous memory systems.

4.2 Asynchronous cache systems

A number of the above asynchronous processor cores were accompanied by dedicated, asynchronous cache systems. The two most recent AMULET memory systems are covered separately, in greater detail in section 4.3 because being already tailored for the ARM architecture, they form a basis of the work in this thesis.

4.2.1 The ECSTAC cache system

The ECSTAC cache system [3] contains separate instruction and write-through data caches. These communicate with the external memory via a synchronous external bus. An on-chip memory unit is constructed to arbitrate, synchronise and sequence accesses from the on-chip caches to the external memory or I/O devices.

The instruction cache is one of the primary determinants of the performance of the processor as a whole and so it is carefully designed. It is 2 kilobytes in size with a 16-byte long cache line. The data cache has much lower bandwidth requirements hence it is not designed as aggressively as the instruction cache. It is 1 kilobyte with the same line size (16 bytes). Although the data cache is not pipelined, two stages (tag decode and data access) can be performed concurrently in the instruction cache. Both caches are two-way associative. To reduce stalls due to write operations, a three-deep write buffer is added for the data cache.

4.2.2 The TITAC-2 cache system

The TITAC-2 cache system consists of an 8 kilobyte on-chip instruction cache designed as a direct-mapped cache with an eight-word line size. A line fetch process reads data from the main memory a word at a time (taking 8 cycles to fill the whole line). Although this cache system was described as using early-start with streaming [101], from the remainder of the paper describing its operation it would appear that it also uses the non-blocking scheme since the line fetch process can proceed in parallel with other cache accesses. However, unlike the AMULET2e cache system, every cache access (either via the line fetch process or any other processor request) requires arbitration, leading to undesirable stalls. A data cache was not implemented.

4.2.3 The Caltech MiniMIPS cache system

The MiniMIPS cache system [78] included both a four-kilobyte instruction cache and a separate four-kilobyte direct-mapped data cache. Each cache is divided into four interleaved blocks. The instruction cache has support for branch prediction and prefetch whilst the data cache has support for writing using a write-through policy with a write buffer. Each cache line is one word long (32 bits) with its own tag (16 bits). Cache fetches are carried out with a 128-bit (4-word) line fetch block. The entire cache system is deeply pipelined for high throughput which introduces structural hazards as follows.

Assume that a cache read miss causing a line fetch is immediately followed by a write at the same address. If allowed to run to completion, the write process would update the cached copy and then sometime later the fetched data would arrive and update the cache. Ultimately and incorrectly, the cache location would contain the old value. The solution to this problem used in MiniMIPS was to add the ability to repeat certain operations in the cache, in this case the `STORE` is repeated. Although this can resolve the problem, it was certainly not an energy-efficient approach.

Another problem in this (deeply) pipelined cache is the double line fetch problem where multiple consecutive reads occur in the same line fetch block. If the first read misses, the second is likely to miss as well, unnecessarily causing another line fetch of the same refill block. Although there is no harm in this type of problem, it increases both memory traffic and power consumption unnecessarily.

The MiniMIPS cache system is the only known asynchronous cache system to date to include a write buffer which is capable of merging writes.

4.2.4 The Kin memory system

The on-chip cache proposed for the Kin processor comprises separate caches for instructions and data. Instructions are decoded into simple micro-operations and can then be stored in a *Decoded Instruction Cache (DIC)* – similar to the Execute Trace Cache in the Intel architecture described in section 3.8.2 – resulting in a fast cache access time. The DIC is multiported allowing simultaneous multiple fetches. The write policy for the data cache is unspecified.

4.3 AMULET memory systems

This section describes the on-chip memory systems incorporated in the AMULET2 and AMULET3.

4.3.1 The AMULET2e cache system

The organisation of the AMULET2e chip is shown in figure 4.1. The AMULET2e contains an AMULET2 processor, four kilobytes of memory and a flexible memory interface (the *funnel*), allowing external devices e.g. DRAM to be connected directly. The on-chip memory can be configured as either a cache or a fixed RAM area.

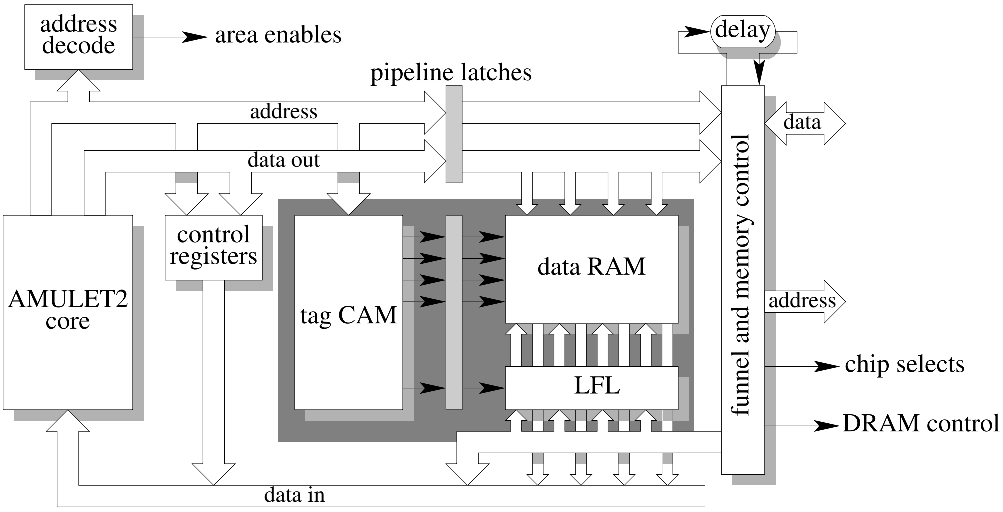


Figure 4.1: The organisation of the AMULET2e chip (after [32])

The key features of the on-chip cache [35] relevant to the work described in this thesis are:

- The cache (shown shaded) is constructed from four independent one-kilobyte (interleaved) blocks, each of which is fully associative, giving 64-way associativity for the whole cache system. It has a pipelined CAM-RAM structure;
- The system includes an arbitration-free, non-blocking cache line fetch mechanism employing a write-through scheme (every write is sent to the main memory), write-around policy (no cache allocation on writes) and random replacement;
- Each cache block includes a *Line Fetch Latch* (LFL) which is used to hold the new line fetched on a cache miss. An additional CAM entry holds the address tag for the LFL and a hit on the LFL (which is loaded addressed-word first) can happen any time from after the first word has been loaded to when the LFL contents are copied into the main cache. The contents of the LFL are copied into the main cache only when the next miss occurs or on a write hit in the LFL;
- A read hit in the LFL can be serviced from the LFL as soon as the required data arrives in the LFL. A write hit in the LFL is slightly more complicated since the LFL is *read-only*. Writes have to wait until the LFL is full. The required write location (which could be 1, 2 or 4 bytes) is masked out. Then the line is copied into the cache RAM merging with the write data;
- The cache supports a form of hit-under-miss which means that cache hits can continue to be serviced while a line fetch is still in progress, though only after the originally requested data that caused the miss has been supplied;
- Since many memory accesses are sequential, a sequential signal is generated to indicate whether a reference is to the address following the previous address. This can be used to make memory accesses more efficient in terms of speed and energy by avoiding some stages of the memory access mechanism. In particular, unnecessary CAM look-up is averted and RAM precharge is postponed, providing a shorter access time. This variation, which can result in access times shorter than a normal ‘cycle’, can be exploited automatically by an asynchronous processor [94].

4.3.2 The AMULET3i dual-port RAM system

The AMULET3 is a Harvard-like processor architecture as shown in figure 4.2. It has separate instruction and data memory interfaces which provide for the supply of instructions and data to the processor independently from a unified memory system. The data port is used for memory accesses by load/store instructions. The instruction port is used for instruction fetches. In addition, the instruction port is also used for all data loads to the program counter.

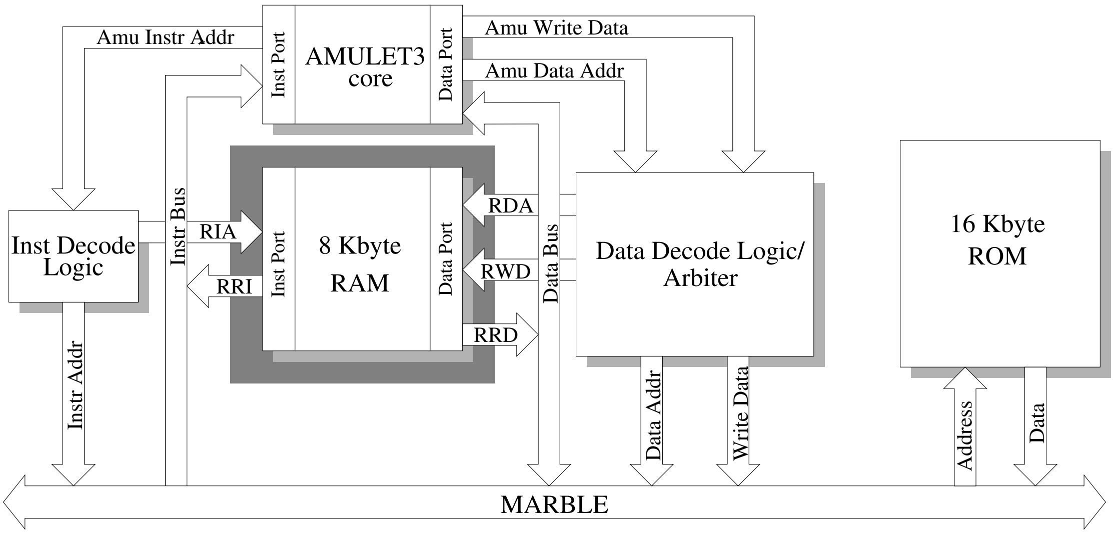


Figure 4.2: The organisation of the AMULET3i subsystem (after [34])

The consequence of this design is that there are two separate memory buses from the processor, but both local RAM on the DRACO chip (shaded in figure 4.2) and the external ROM are unified. These memory buses are required to ‘merge’ at two places outside the processor core, one for accessing the local RAM system and the other for accessing the external memory via the MARBLE on-chip bus.

Instruction accesses are read only and are handled by the local RAM or passed to the MARBLE bus [9]. Data accesses are read/write and may also be passed to the MARBLE bus. The RAM may also be accessed by remote MARBLE masters through the data port, so use of the data bus must be arbitrated for AMULET3 and MARBLE accesses.

Two decode logic blocks are required to monitor accesses to these two ports. The main functions of the instruction and data decode logic blocks are:

- to detect aborts (memory access exceptions) for the instruction and data requests;
- to perform memory management functions;
- to direct instruction and data requests to the required targets (either MARBLE or RAM).

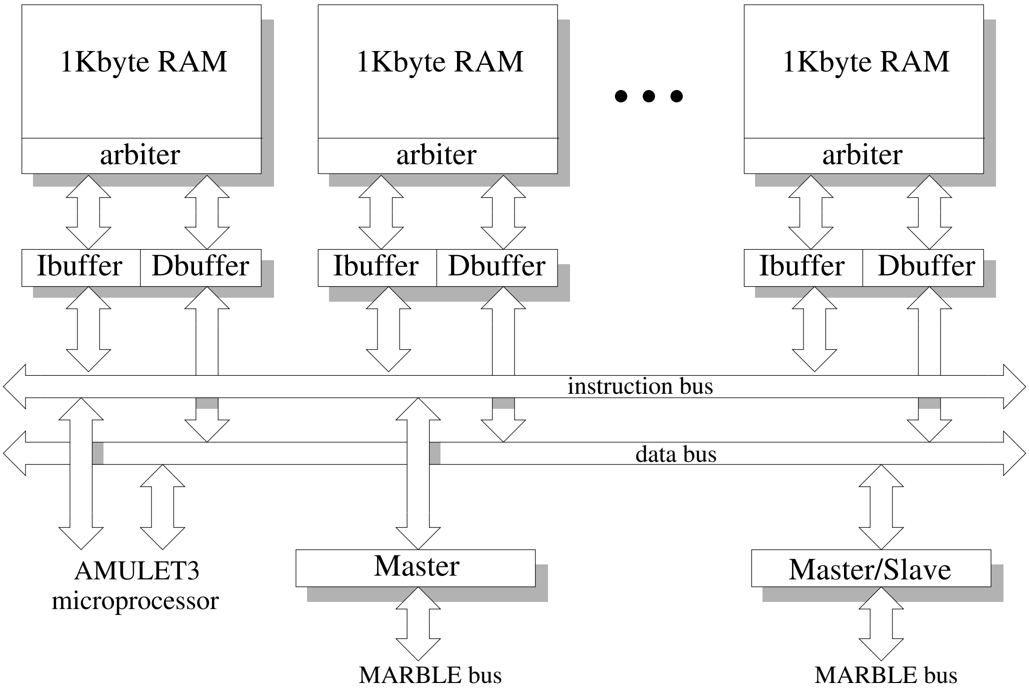


Figure 4.3: The AMULET3i RAM block organisation (after [34])

The organisation of the memory system used with AMULET3 on the DRACO chip is shown in figure 4.3. It is an 8 kilobyte static RAM satisfying both instruction and data requests. This is achieved by dual-porting the memory which is divided into eight one-kilobyte blocks, each block being a conventional (single-ported) RAM.

Splitting the RAM into eight separate, interleaved blocks has the following benefits:

- reducing the power consumption of the AMULET3i RAM, since only a single one-kilobyte RAM block is accessed by an instruction or data reference;
- reducing the probability of clashes between instruction and data accesses.

Each block has two *line-buffers*, one holding the line of four words (one word is four bytes) containing the last instruction supplied by the block and the other holding the line containing the last data item read by the processor. These line-buffers are placed after the RAM sense amplifiers. Data is read from the RAM a line at a time and latched in the appropriate line-buffer; future accesses can then read data from the line-buffer without cycling the RAM. This saves power and decreases the average RAM access time. However, there are more sense amplifiers in AMULET3 than in AMULET2e (128 rather than 32) and so power consumption when the RAM block is accessed will be higher.

The benefits of line-buffering include:

- improving the overall performance of the RAM, since the majority of the instruction references are sequential which allows the address decode phase to be bypassed in a sequential cycle;
- reducing the total power consumption by minimising the number of accesses to the RAM itself.

Having separate instruction and data line-buffers in each RAM block in a dual-ported system provides benefits by:

- avoiding a large proportion of the occurrences of contention of instruction and data accesses needing the same RAM block;
- avoiding interrupting the sequentiality of fetches especially in the instruction stream;
- providing a larger line-buffer level to the memory system.

When a memory access does not find the data it needs in a line-buffer it must access the RAM. Only when the instruction and data accesses require data from the same RAM block at the same time (in both cases not finding the required data in the line-buffers) is there contention that must be resolved. Because there is no clock on which to make a decision, access to the block is controlled through asynchronous arbitration on a ‘first-come-first-served’ basis. Hence each block has an internal arbiter [88] to resolve this contention as shown in figure 4.3.

To maintain coherency, the contents of the line-buffers are invalidated whenever there is a write hit in the buffered line on that memory port. Thus the line-buffers can be considered as a form of limited ‘level-zero’ (L0) split (but coherent) cache.

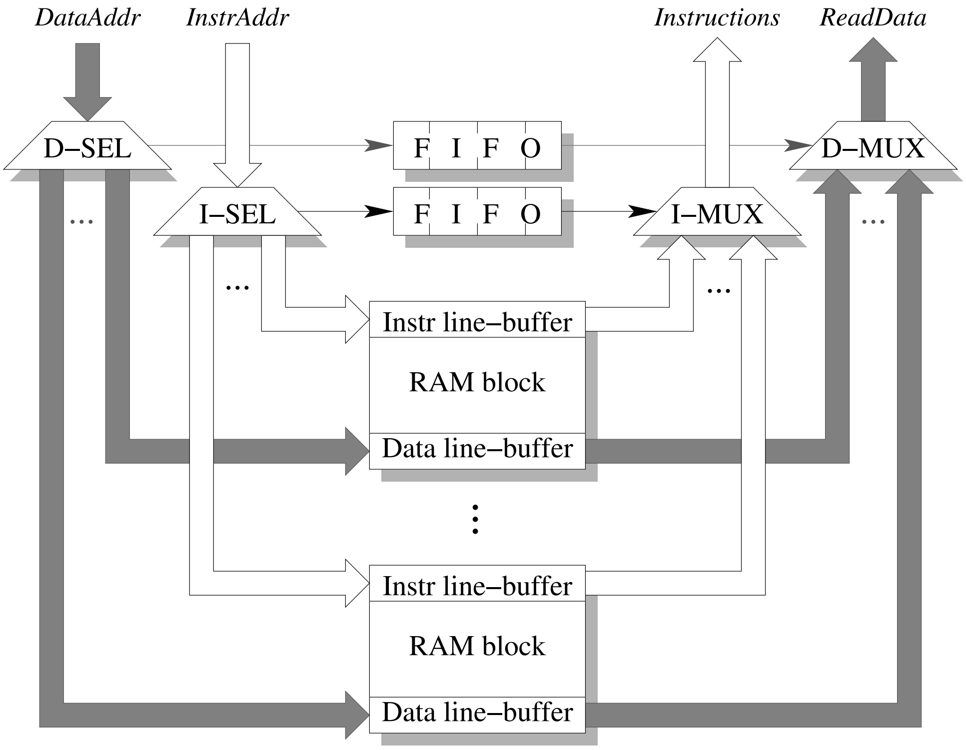


Figure 4.4: Controlling ordering with the FIFO

Although there are two separate locations in the local RAM system that can service the data, either from the line-buffer or the RAM, all fetched data is sent to the processor via the line-buffer in the RAM block concerned. A FIFO buffer (as shown in figure 4.4) is used on each port to hold control information sent from the selector (SEL) to switch the multiplexer (MUX) to the correct path for the returning instruction/data. The depth of the FIFO imposes a maximum limit on the number of outstanding memory accesses on each

)

memory port. This number of outstanding memory accesses limits the number of RAM blocks that can be in use at any instance.

To exploit pipelining in the memory subsystem, the AMULET3 processor core must be able to issue multiple outstanding memory requests. To avoid causing deadlock, the memory system must be designed such that an arbiter input may not be serviced until it is known that the service will complete in finite time. One way to achieve this is to guarantee that there is space to hold all of the requested values after the critical region by inserting latches. Such latches, forming a FIFO, have to be carefully positioned so that they are after the critical region shared between both the instruction and data access paths. If insufficient latches are used, or they are located in the wrong places, then a possible deadlock can occur when the processor fills the critical part of the memory with fetch requests and so blocks a data access upon which there is a data-dependency.

Figure 4.5 illustrates this showing where the latches must be placed. For a throttle set to give *N* outstanding memory accesses, *N* latches are required.

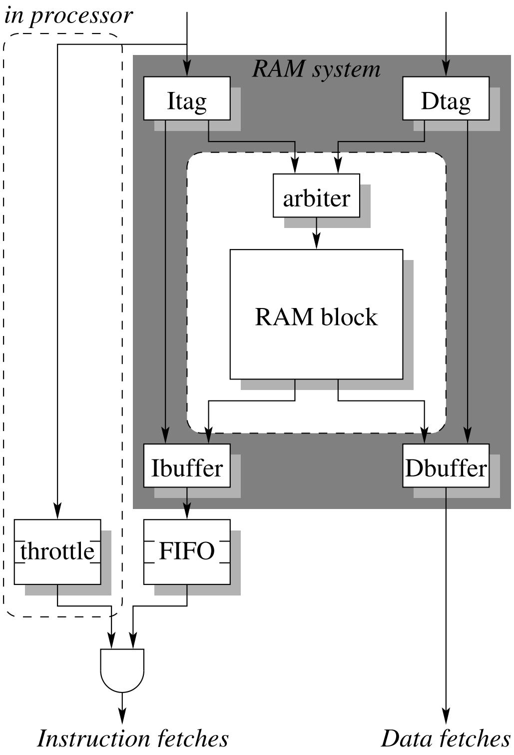


Figure 4.5: AMULET3 memory throttling (after [94])

The same sequential optimisation used in AMULET2e was also adopted in the AMULET3 processor to provide a faster subsequent RAM access time from the external DRAM, but no use was made of sequential information on-chip.

4.4 Observations

A number of observations can be drawn from the memory organisations described in this chapter. These include:

- Line-buffering can offer a great power consumption reduction. The variation in access time between the line-buffer and normal L1 cache could be much less than a whole ‘cycle’ and therefore would require additional design in a synchronous system to gain the benefit from a fast line-buffer whereas it is automatically exploited in an asynchronous system.

- The combination of line-buffering and sub-blocking not only decreases the power consumption but also increases the overall performance by exploiting concurrency.

- A modular approach, based around sub-blocking, allows cache size configuration to be decided later.

- The sequential signal technique, used in the AMULET2 and AMULET3, is a good, cheap approach to reduce the cache power due to CAM comparisons.

- A deeply pipelined structure (like the MiniMIPS cache system) can offer high system performance, however they come with disproportionately large energy cost. Another (probably better) solution for both types of problems described in section 4.2.3 is to stall subsequent accesses to the same address as on-going accesses. This solution not only removes unnecessary operations, providing better energy-efficiency, but could also improve the overall performance e.g. the second `LOAD` in the double line fetch problem would become a hit, hence reducing access time.

- The high performance features like those found in the Kin design (caching decoded instructions and allowing massively out-of-order operations) are usually too expensive to use in small, low-power embedded systems.

- All existing asynchronous caches adopt the write-through policy; it should be possible to improve upon this using a copy-back scheme to reduce the intensity of memory write traffic.

The work described in this thesis presents the application of a copy-back policy to an asynchronous cache system which also simplifies the implementation of other effective features such as write-allocation and victim caches.

4.5 Summary

A number of processor design proposals and successfully fabricated chips have been produced by various research groups, some of which were described in this chapter. Some of these are now beginning to appear in commercial applications. However, there is still a large knowledge gap to bridge before satisfactory asynchronous memory systems can be constructed as is apparent from the observations on notable previous asynchronous memory systems presented in this chapter.

The following chapters build upon the existing knowledge base to narrow this gap, with the existing designs used as the basis of an improved asynchronous memory system in the form of a cache system for the latest processor in the AMULET series, AMULET3.

Chapter 5: An Asynchronous Copy-back Cache

Chapter 4 reviewed earlier asynchronous memory systems. One common feature of all of these existing cache systems is their use of a write-through policy. This chapter describes the design of an asynchronous, copy-back cache architecture for use with a Harvard-like architecture processor core. Issues addressed here include the line allocation mechanism, write buffering, non-blocking line fetches and out-of-order accesses. Detailed examples of a variety of cache operations are provided showing the flexible timing behaviour that can be supported by the cache.

5.1 Environment

The cache architecture presented in this thesis is intended to work with the AMULET3 microprocessor [34], a third generation asynchronous ARM implementation. Although the first AMULET3 system had no cache, it featured eight kilobytes of memory mapped RAM, as described in the previous chapter. The organisation of this first system will affect the design of the cache in this thesis. The top-level organisation of a possible processor and cache subsystem is shown in figure 5.1. The major units in this figure are:

- **an AMULET3 core**: this is an implementation of the v4T ARM architecture [4]. It is compatible with both the ARM instruction set and its compressed form, the Thumb instruction set whose purpose is to increase the program code density. With its Harvard-like architecture, the closest equivalent synchronous ARM is the ARM9 [87]. AMULET3 has two 32-bit memory ports, the instruction port, which is read-only, and the data port, which supports both read and write operations.

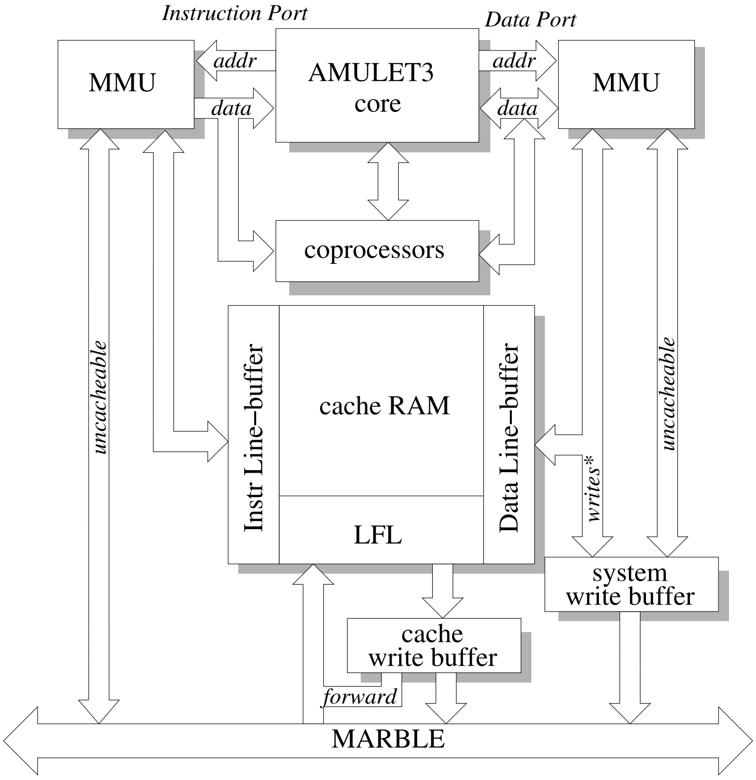


Figure 5.1: AMULET3 cache system

- **two MMUs** (Memory Management Units): located next to each of the instruction and data ports, these check whether a memory location is cacheable[1]. Uncacheable memory accesses bypass the cache. The MMUs also detect memory access permission violation and page faults, signalling these to the microprocessor. (MMUs were not included in the initial AMULET3i system.)

- **coprocessors**: these are used to extend the ARM architecture and are the mechanism used to support system management tasks such as programming the MMUs, enabling cache features, locking down cache regions and flushing the cache

---

[1] Typically, MMUs have a range of functions, not all of which may be used in any given system. Common functions include: virtual to physical address translation, address alignment checking, access permission checking to ensure that user mode code cannot modify region of memory restricted to supervisor mode only access etc. The MMUs would normally be configured through a coprocessor in ARM systems with information such as the locations in the memory map that correspond to read sensitive peripherals, which should not be cached.

and write buffers. Many of these operations are not supported in the system proposed here.

- **MARBLE**: an on-chip asynchronous system bus [9] connects the MMUs and cache to other system components and the off-chip memory interface.

- **write buffers**: a significant penalty is associated with write operations that slow the processor to the speed of the main memory. Write buffers can accept write data at a higher speed than the main memory and allow the processor to continue whilst the buffer writes the data back to main memory. The *system write buffer* [41] in figure 5.1 buffers uncacheable writes; it would hold all writes in a write-through cache system. It is desirable for processor-memory speed decoupling. The *cache write buffer* decouples ‘copy-back’ operations; it is unnecessary with a write-through cache.

5.2 Basic architecture

The processor architecture and its usage place a number of constraints on the cache. The cache presented in this thesis is to be unified but dual-ported to accommodate the AMULET3 Harvard-style memory interface. A number of features from earlier designs can be adapted:

- The cache is divided into (interleaved) blocks as shown in figure 5.2, as were the memory systems of the earlier processors in the AMULET series [32,34]. This gives dual-port access with the additional benefit of reduced power consumption since only one cache block is active for each memory access. Just as with the AMULET3 RAM, arbitration is necessary only when both ports request access to the same block at the same time. This happens transparently in the asynchronous system; no clock gating or wait signals are required. If the cache were a single block such collisions would be quite frequent; however with (provisionally) eight independent cache blocks it is unusual for a collision to occur and most cycles will proceed at full speed. This (typically) gives a performance close to that of a split cache but guarantees cache coherence and is much cheaper than a dual-ported SRAM.

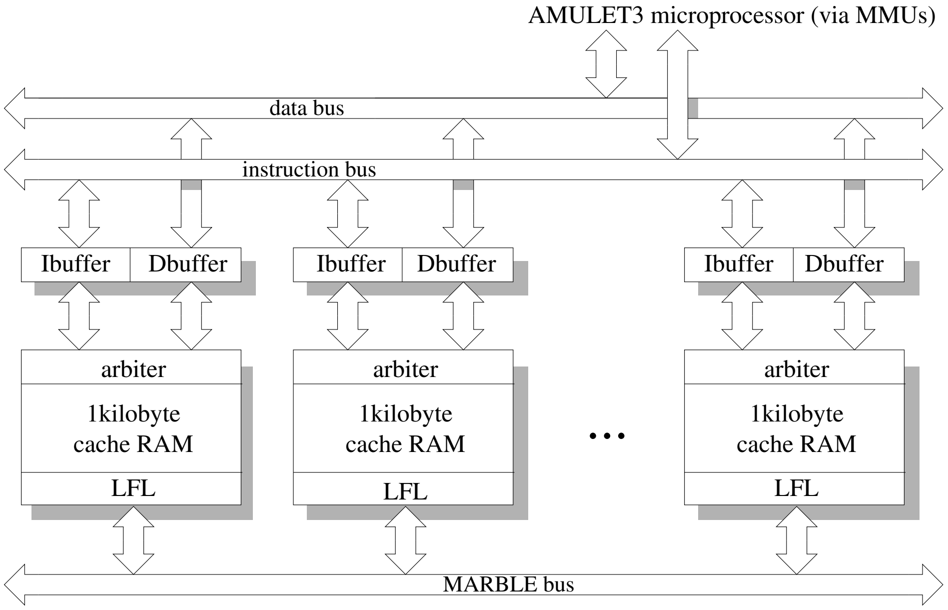


Figure 5.2: AMULET3 cache block organisation

- As can be seen in figure 5.2, the dual (read-only) line-buffering technique employed in the AMULET3 RAM system is reused here. One line-buffer for each port is used to store the last read line which can then be re-read quickly for consecutive accesses. This can be considered as an implicit level-0 cache system for reads. The reasons for not implementing a write capability in the line-buffers are presented in section 5.6.2.

- Additional techniques are adopted from AMULET2e [32], notably the line fetch latch (LFL) mechanism [72]. This offers the possibility of a non-blocking scheme permitting hit-under-miss operations. To allow high degrees of associativity, combined with adequate speed, a pipelined CAM-RAM structure is used for each cache block (as in the AMULET2e cache). Pipelining allows tag look-up from one port to proceed in parallel with the data access on the other port. Also from AMULET2e comes a fully-associative CAM-RAM structured block with adequate pipelining within the block. Fully-associative pipelined structures are used as they provide higher performance and a simple lock-down mechanism as discussed in section 3.9.

Figure 5.3 illustrates the structure of an individual cache block bringing together two line-buffers, one each for the instruction and data ports, an arbiter, a pipelined CAM-RAM structure and an LFL.

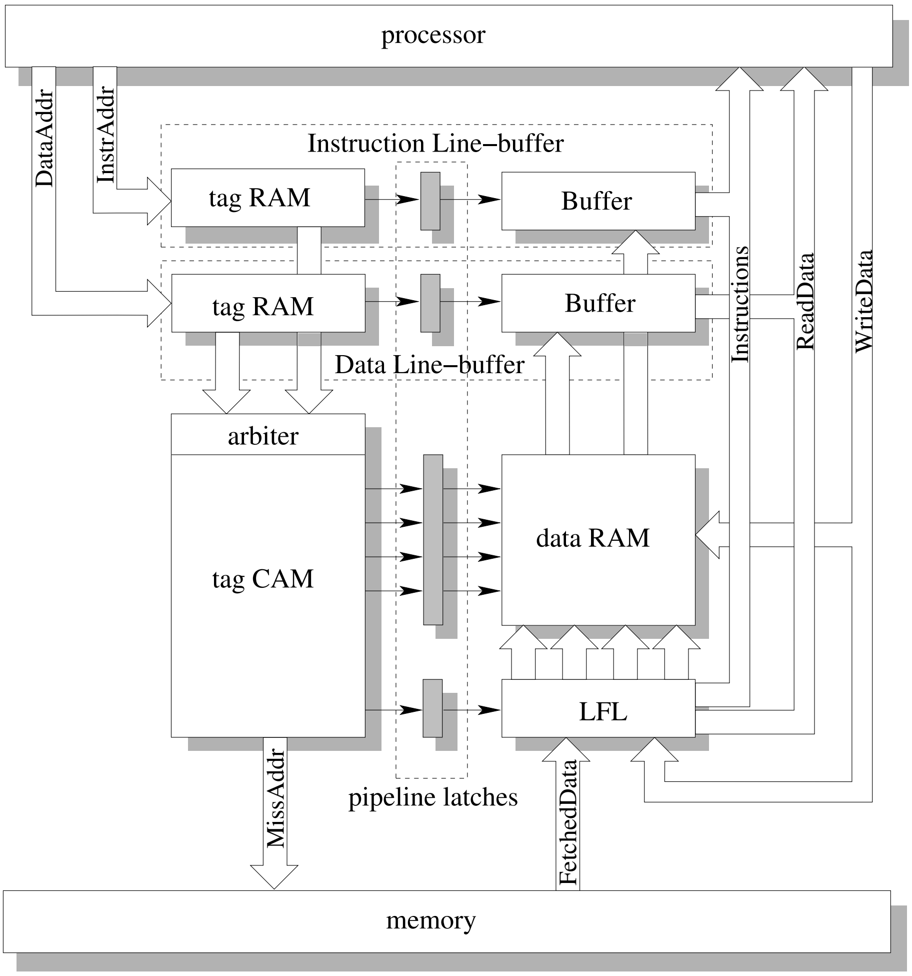


Figure 5.3: Dual-ported asynchronous cache block

In addition to combining these elements a number of new features have been developed. The most significant is the design of an asynchronous copy-back mechanism [47]. This adds significant complexity because data written to cached memory locations is retained locally. The cache has to remember that the affected cache line is ‘dirty’ in order to write it back to memory later, when the line is reallocated. The advantage that this provides is that memory bandwidth requirements are reduced, giving an overall increase in system performance.

The other important new feature is the extension of the cache write buffer to support forwarding, whereupon it becomes a form of victim cache [57]. In doing so memory bandwidth is better utilised, giving improved overall performance. The design of an asynchronous victim cache [49] is addressed in depth in the next chapter.

The remainder of this chapter describes in detail the cache operations and techniques that are adopted in the asynchronous copy-back cache for AMULET3 developed by the author. Throughout this cache description, it is assumed that the evicted cache lines are presented to the memory via the cache write buffer.

5.3 Pseudo two-level cache structure

In this cache architecture there are a number of places in each cache block from which data can be fetched.

5.3.1 'Cache hit'

A cache 'hit' can be considered to occur when the required data can be retrieved from any of the units shown in the upper half of figure 5.4 resulting in a pseudo two-level structure.

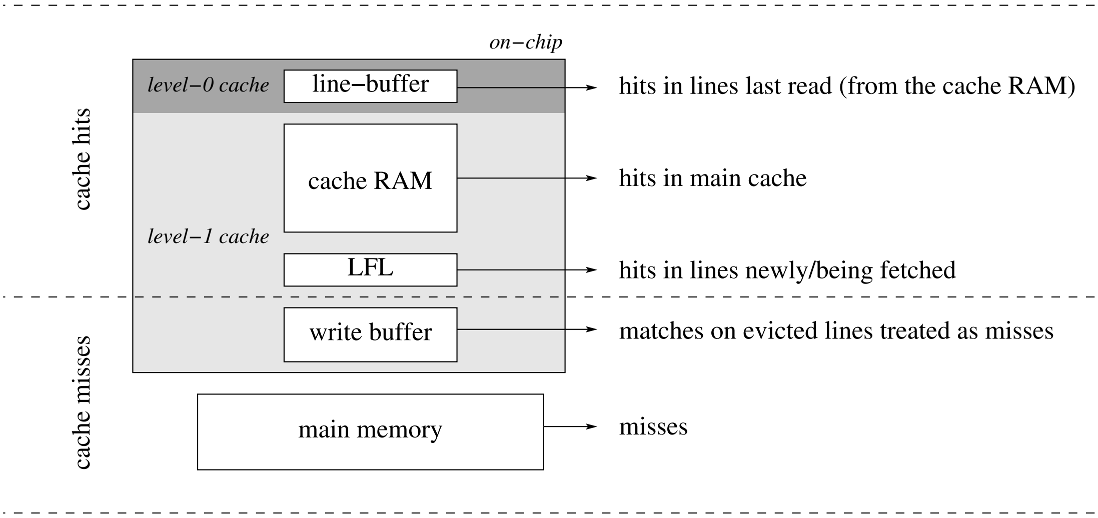


Figure 5.4: 'Nearly' two-level cache structure

Level-0 cache

The term L1 cache is usually used to refer to the smallest, fastest cache closest to the processor. However, the line-buffer level is only thought of as L0 because it has limited functionality and is very small. It buffers the last data *read* from the cache RAM on each port. Subsequent reads from the same line can then be satisfied quickly from this line-buffer. Each line-buffer has its own corresponding tag address, checked during the tag comparison which is performed at the start of each cache access. It does not have a write policy; it is simply invalidated on a write miss and the write request is passed down to the core of the cache system. The activities upon a write hit in the line-buffers are described in depth in section 5.6.2.

Level-1 cache

The true L1 cache is formed by parts shared between both instruction and data ports (after the arbiter in figure 5.3):

- **the main cache RAM**: storing most of the cached data. The tag addresses corresponding to data in the cache RAM are held in a CAM providing a fast parallel look-up mechanism which is performed only when the access is not a read hit in the line-buffer.

- **the LFL**: buffering the most recently fetched data line. Although this is a separate latch from the cache RAM (like the line-buffer), the LFL tag address is stored as part of the same CAM used by the main cache data store. Both the cache RAM tags and LFL tag are checked in parallel because the cache access time is critical. The reason for not counting the LFL as another level in the cache is because it holds the newly fetched lines which are the only copy of the data in the cache system and behave as an extension of the L1 cache.

5.3.2 ‘Cache miss’

The term ‘cache miss’ describes an access to an address for which the only copy of the data is in the main memory or in the write buffer. Figure 5.4 also reinforces this point.

The write buffer never generates a cache hit since, without forwarding, the only way to get data from it is to drain it to the main memory and retrieve the data from there. Obviously, this can cause a significant stall on reads after line rejections. A technique for overcoming this, the victim cache, is considered in the next chapter.

5.4 Line fetch engine

Figure 5.5 illustrates the control flow of a cache read request in this architecture. The top part shows the line-buffering adapted from the AMULET3i memory system. The bottom part shows the line fetch technique adapted from the AMULET2e cache system [72].

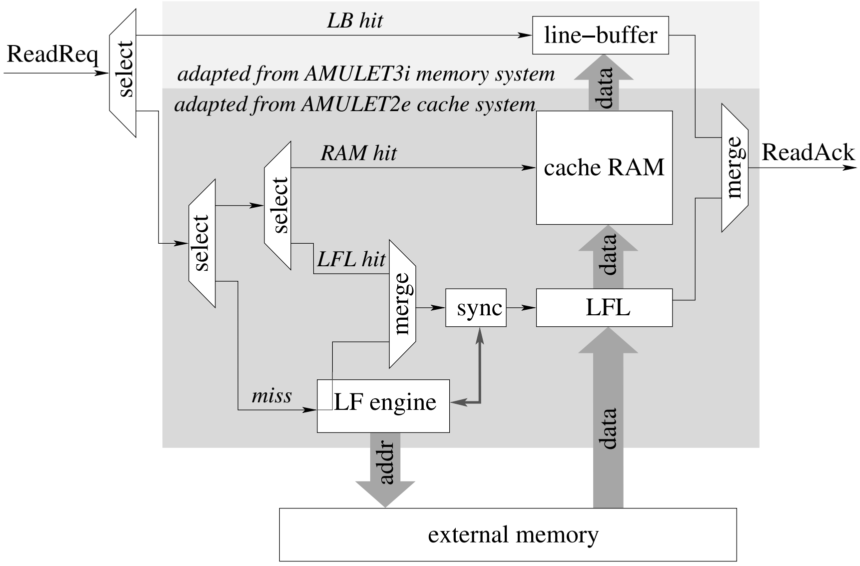


Figure 5.5: Cache request steering control logic

The line fetch engine in figure 5.5 is a separate unit from the cache. It takes a line fetch address and interacts with the memory/system bus to fetch the line (for example issuing four addresses to retrieve four words of data). These (four) accesses could be in any order; the most efficient approach being to fetch the required word first as described in section 3.4.2. The *word-synchroniser*, shown later in figure 5.7 and discussed in section 5.5, ensures that the cache waits for the requested word to be valid in the LFL, allowing any ordering to be used by the fetch engine.

5.5 Line allocation mechanism

The line allocation mechanism is similar to that used in AMULET2e, although its complexity is increased somewhat due to copy-back operation. When a line fetch is needed the first activity triggered is a request for a read burst from the next level in the memory hierarchy. The key internal cache activities of the line allocation, which run partially concurrently with the memory access, are shown in figure 5.6. The control of these operations is illustrated in a 2-phase control style [100] in figure 5.7 which shows how a non-blocking line fetch engine can be supported in an asynchronous environment without requiring any arbitration. The activities in the cache line allocation as numbered in figure 5.6 are:

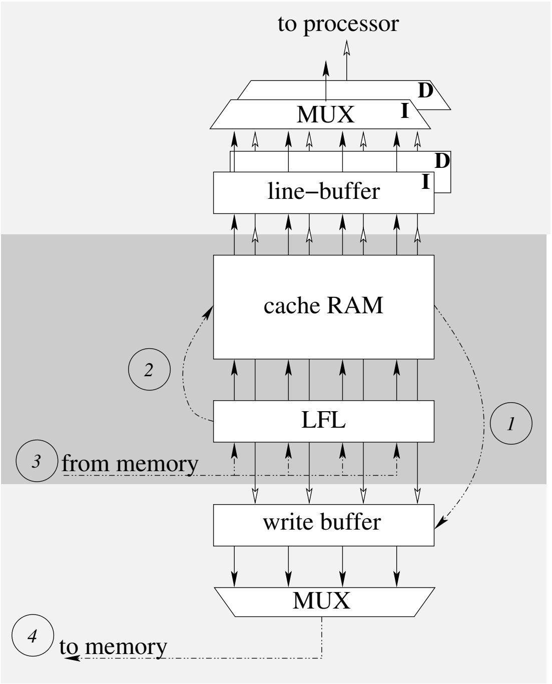


Figure 5.6: Cache line allocation data flow

- **Activity 1**: Select a victim line and eject it (from the RAM) to the write buffer, regardless of whether that line is dirty or not.
- **Activity 2**: Copy the complete, previously fetched line, stored in the LFL, into the RAM, ((`2b`) in figure 5.7), 'emptying' the LFL. This can only happen after every

word of the previous line fetch has arrived (ensured by a Muller-C element (`2a`)). The request also resets the exclusive-OR gates' outputs (`2c`), making the transparent latches (TL) opaque ready for the next line fetch.

- **Activity 3**: When the LFL is empty, data from the new line fetch is streamed in, each word opening the appropriate latch to indicate its arrival. The same word synchronisation logic is also used when an LFL hit waits until the requested word is present.

- **Activity 4**: After receiving the ejected line, the cache write buffer tests to see whether it is dirty. If it is not dirty it is marked as 'written'; otherwise a request is made to the bus to perform the appropriate write(s) ((`4`) in figure 5.7) which will be granted *after* the read burst has completed. The bus interface is a separate unit which may defer writes if it has more urgent read requests to service. The cache write buffer is described in section 5.10.

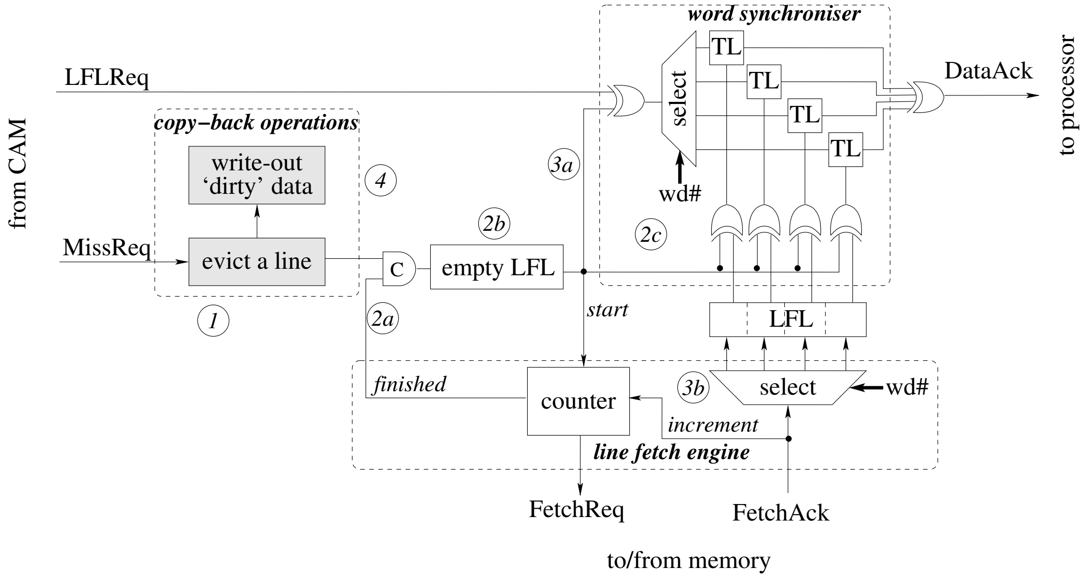


Figure 5.7: Line fetch engine (after [71])

There are some resource use conflicts amongst these activities which preclude all of the activities running completely in parallel. Activities `1` and `2` both need to access the cache RAM whilst processes `3` and `4` both require the memory bus. Activities in each set have to proceed sequentially. Dependencies such as these have to be considered in an asynchronous environment to avoid misoperation and potential deadlocks.

A write-through cache (such as that used in AMULET2e) does not have to deal with evicting possibly dirty data and so performs only steps 2 and 3 as it is known that the RAM contents are 'clean' and can be overwritten.

5.6 Cache operations

This section describes major activities that may occur for any cache access in the proposed asynchronous cache system. Though these operations are described in the context of the copy-back cache, consideration of the requirements for a write-through policy are also included for comparison purposes. In this section, all possible activities are also illustrated in figures with their numbering usually showing the ordering of operation; in cases where concurrency is possible, the numbering is used only to differentiate the activities. The key markings used to describe the cache activities are given in table 5.1.

marking	**activity**
R	a read operation
W	a write operation
wt	a write-through cache activity
cb	a copy-back cache activity
–	write operation performed only when the data is dirty
*	a special case specific to the action described

Table 5.1: Key markings describing cache activities

5.6.1 Line-buffer read hit

A read hit in the line-buffer can be satisfied quickly from the appropriate fast asynchronous line-buffer shown in figure 5.8. The requested word is sent to the processor without performing a full CAM look-up nor cycling the RAM of the main cache. This allows requests from the other port that may need to access the same cache block (including the other line-buffer) to proceed concurrently.

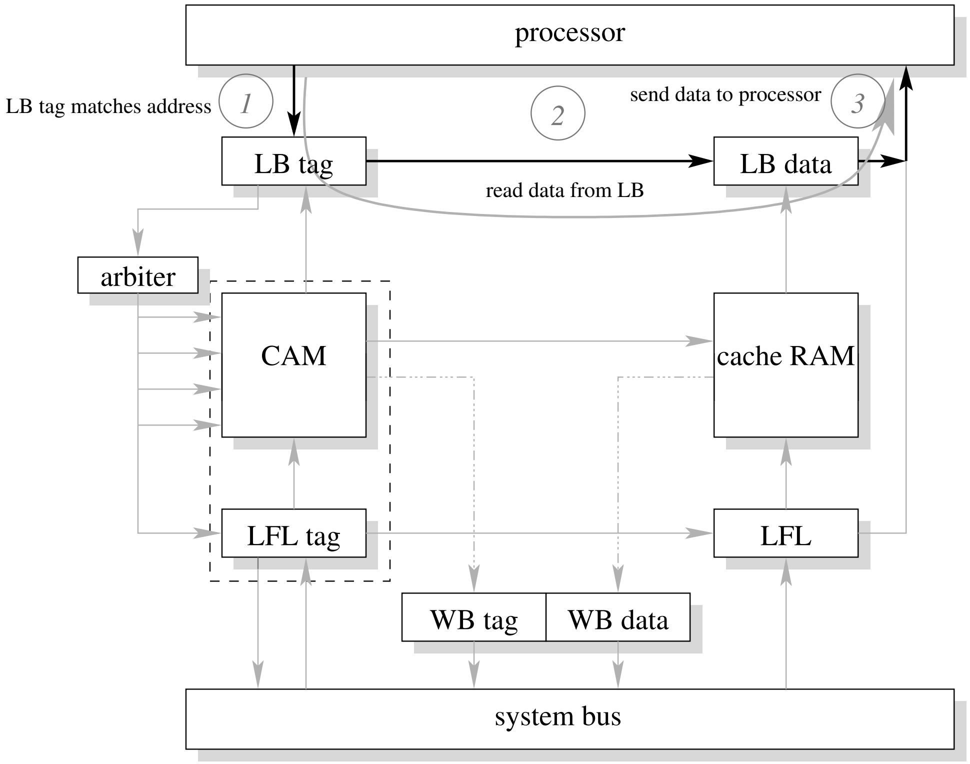


Figure 5.8: A line-buffer read hit

5.6.2 Line-buffer write hit

The major difference between dealing with a line-buffer write hit in this architecture and in the AMULET3i RAM system is that a line-buffer write hit in the RAM system is guaranteed also to match a location somewhere in the main RAM, whilst in the proposed cache system a line-buffer write hit could match in any level of the cache or not at all.

The possible scenarios are matches:

- in both the main memory and the cache RAM – the usual case, described in Section 5.6.4;
- in both the main memory and the LFL – quite rare but can happen if the line has been evicted from the main cache and then subsequently fetched into the cache again;
- only in the main memory – also rare if the line has been evicted from the cache.

In a write-through cache, allowing a line-buffer write hit to update the line-buffer would be relatively simple to implement, requiring also an update of the buffered line in the main cache so that all locations holding the referenced information are kept coherent. However, the scenarios described above for a copy-back cache make it much more complex to allow updating of the line-buffer in a copy-back cache system. For example: What should be done if the line-buffer is the only part of the cache containing the data? How would an updated line-buffer's contents be evicted later without losing the dirty data? What should happen if both line-buffers contain the same line?

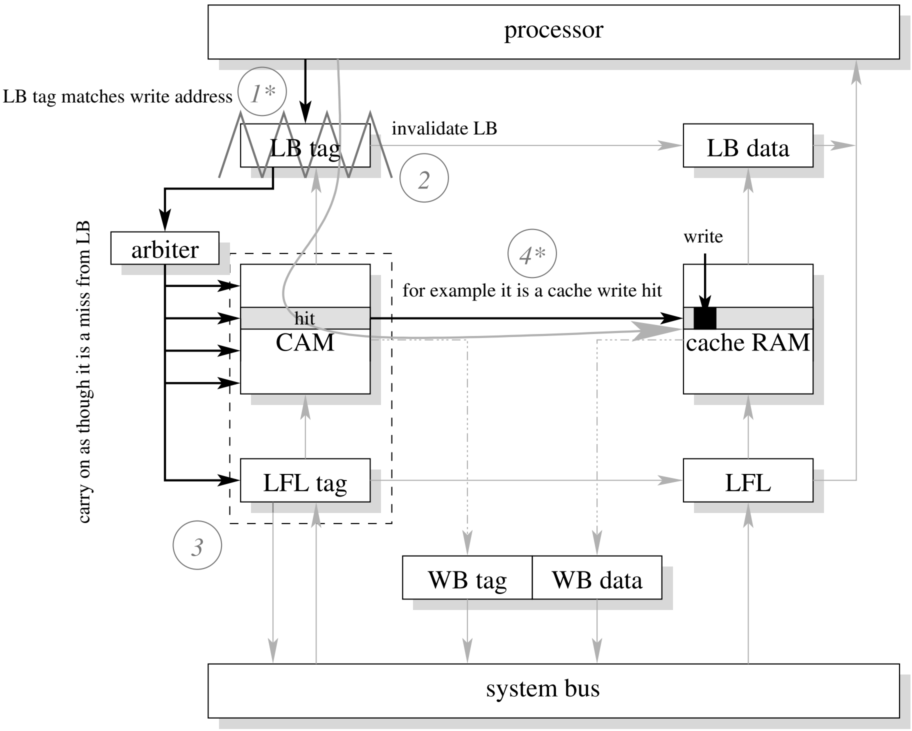


Figure 5.9: A line-buffer write hit

To avoid these many complex scenarios, the line-buffers have been made 'read-only' for the processor and are simply invalidated if a line-buffer write hit occurs (as shown in figure 5.9). This choice has little effect on performance since the line-buffers are small and therefore, with sequential access patterns often accessing more than four words consecutively, the line-buffer contents are updated fairly regularly. Note that although the instruction port is read-only, if the code is self-modifying the instruction line might contain the written location and must therefore be invalidated to maintain coherency. This must only occur when the instruction port is idle, so the data write may have to wait (*1**). Apart from invalidating the line-buffer (*2* and indicated by the zigzag), the other actions performed are the same as those when a write misses in the line-buffer. Since the arbitration is already done, the write request is then passed straight down for a full CAM look-up (*3*).

5.6.3 Cache RAM read hit

If an access does not hit in the line-buffer it is allowed, via an arbiter to serialise activity from the two ports, to access the 'main cache' system. Line-buffer write hits must also follow this path.

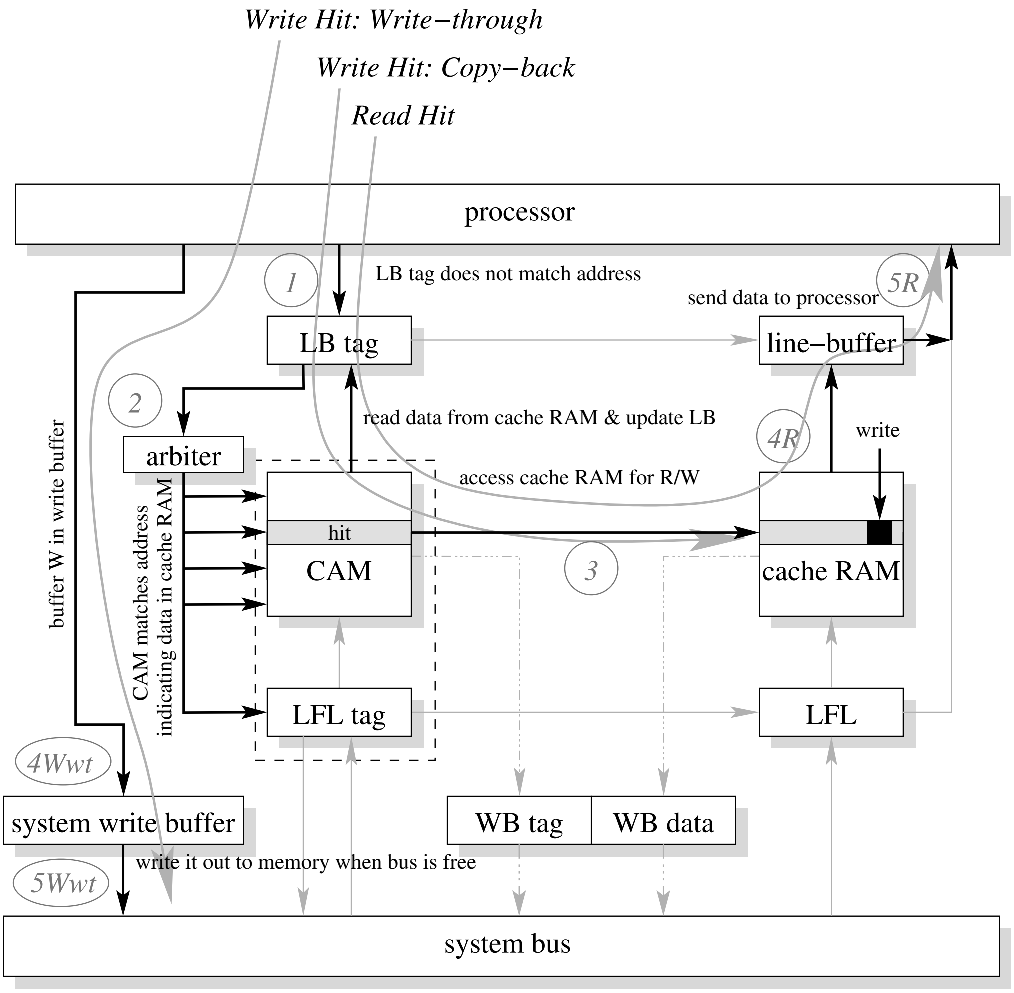


Figure 5.10: A cache RAM read/write hit

The operations for a cache hit in the main cache RAM, as shown in figure 5.10, are very straightforward and much like hit operations in other caches. First, the appropriate line-buffer tag look-up has to be performed because of the existence of the dual line-buffers. In this case the request is not a line-buffer read hit, so another look-up has to be performed. Hopefully then a tag in the CAM matches the requested address indicating a hit in either the main cache RAM or the LFL. (LFL hits are considered in section 5.6.6).

For a cache RAM read hit, the behaviour (annotated on figure 5.10) is similar to other multi-level cache systems where the cache level closer to the processor is updated; data is read from the cache RAM with a whole line being copied into the appropriate line-buffer (*4R*) and the requested word is sent to the processor (*5R*).

5.6.4 Cache RAM write hit

In the copy-back cache, whilst performing the write operation in the cache RAM for a write hit (indicated by the grey path, labelled *Write Hit: Copy-back* in figure 5.10), the dirty bit corresponding to the line entry is set to indicate that the line has been modified.

In the write-through cache, a write request also joins the system write buffer queue (*4Wwt*) and then, when the bus is idle, the write request (in the queue) updates the main memory (*5Wwt*). This activity is shown along the grey path, labelled *Write Hit: Write-through* in figure 5.10.

5.6.5 LFL read hit

The operations performed on an LFL hit (shown in figure 5.11) are much the same as those for a cache RAM hit. However, they are not identical since the LFL buffers the fetched data which arrives a word at a time.

An access again starts with the appropriate line-buffer tag look-up followed by the CAM look-up which indicates that there is/will be a copy of the data in the LFL. With the non-blocking line fetch scheme, access to the LFL is possible even if a line fetch is still in progress. However, before any operation can proceed, the data to be read must be valid

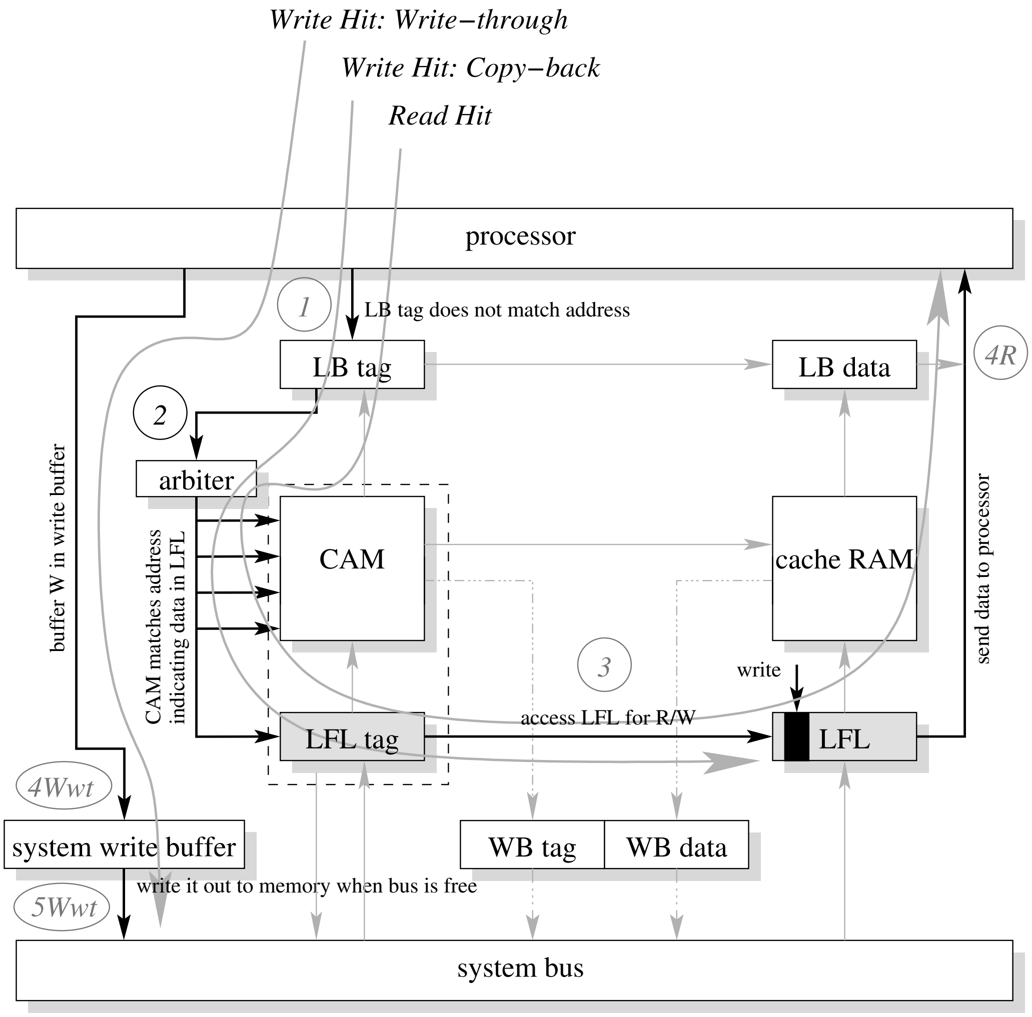


Figure 5.11: An LFL read/write hit

in the LFL, at which point it can be read out directly. This can be considered as a snooping operation since no cache updating occurs at any other cache level. The operations performed in an LFL read hit are indicated by the grey loop, labelled *Read Hit* in figure 5.11.

5.6.6 LFL write hit

There are a number of different ways an LFL write hit could be handled during or after a fetch has completed:

- **Option 1**: wait until the whole line is fetched into the LFL, copy it into the cache RAM and then overwrite the affected bytes in the cache RAM;
- **Option 2**: wait until the whole line is fetched into the LFL then combine it with the affected bytes as the line is written in the cache RAM. This approach is used in the AMULET2e cache system;
- **Option 3**: wait for the word needing to be modified to be fetched into the LFL then overwrite the affected bytes in the LFL;
- **Option4**: wait for the word needing to be modified to arrive at the LFL then combine it with the affected bytes as the word is written in the LFL;
- **Option 5**: store the write-data in the LFL, recording which bytes have been written so that if those bytes have not yet been fetched, the fetch process will not overwrite them;
- **Option 6**: have a separate, parallel latch for the processor to write to. This preempts the LFL at any read attempts and, once written, invalidates that particular part of the LFL.

Table 5.2 summarises the stall duration required for each option on an LFL write hit and a write miss. The first two options both incur an expensive 4 memory-fetch-cycle stall in the worst case (a write miss) and also the RAM overwrite time for option `1`. Options `3` and `4` do much better in that they only wait until the word to be overwritten has been fetched, incurring a stall of between 1 and 4 memory-fetch cycles in duration. With a write-allocate and requested-word-first approach options `3` and `4` are, of course, much better on a write miss than options `1` and `2`, only incurring 1 cycle stall as opposed to 4 cycles.

	stall duration	
option	**write hit in LFL**	**write miss**
option 1	upto 4 memory cycles + a cache write	4 memory cycles + a cache write
option 2	upto 4 memory cycles	4 memory cycles
option 3	until the word fetched + an LFL write	1 memory cycle + an LFL write
option 4	until the word fetched	1 memory cycle
option 5	an LFL write	an LFL write
option 6	an LFL write (arbitration-free)	an LFL write (arbitration-free)

Table 5.2: Stall duration during LFL write

As with the difference between options `1` and `2`, option `4` saves the 'overwrite' time by merging data on the way into the LFL, but options `5` and `6` offer the highest potential performance. Writes only have to wait at most the time required to store a word that has just been fetched into the LFL before they can themselves be stored for option `5`. Such conflicts should be rare since fetches are slow. However, this scheme requires arbitration and a flag for each byte in the LFL to indicate if it contains data from a 'write' that should not be overwritten by a subsequent fetch. Option `6` achieves the same results but is arbitration-free. It does, however, complicate the LFL reading process.

Option `4` is used here as it offers a balance between complexity and performance.

Therefore, in the proposed architecture, an LFL write hit (shown in the grey path, labelled *Write Hit: Copy-back* in figure 5.11) is simply performed in the LFL (*3*); the dirty bit corresponding to the line entry is set to indicate that the data has been modified.

As is the trend in this chapter, figure 5.11 also shows, for comparison purposes, the additional activity required in a write-through cache (shown by grey path, labelled *Write Hit: Write-through)*, including sending all writes into the write buffer (*4Wwt*) and then later draining them to the main memory (*5Wwt*).

5.6.7 Read miss

A cache miss is the most complicated and slowest scenario that can happen in any cache system. This is because the request requires access, via the system bus, to the slow main memory. Various policies can be applied on either a read miss, a write miss or both. In this architecture a write-through cache uses a write-around scheme – no cache allocation is performed for write operations – whilst the copy-back cache uses a write-allocate scheme – cache allocation is performed on a write miss and is then followed by a write action into the cache – as discussed in section 2.2.8.

The operations for a cache miss start with the usual (appropriate) line-buffer tag look-up (*1*) followed by the CAM look-up (*2*). These tag comparisons indicate that the request requires access to the memory for a line fetch (and writing back dirty data to the memory for a copy-back cache). A cache read miss (shown in figure 5.12) incurs the cost of a line fetch. However, prior to starting the line fetch, space must be created for the previously fetched line from the LFL to be inserted in the main cache; requiring an existing line to be ejected. If the previous line fetch has not yet completed emptying the LFL process must wait until all the previous data is present in the LFL.

Line eviction is potentially a complicated process in a copy-back cache since the evicted line might have been modified – in which case it needs to be written back into the main memory. The line eviction approach adopted here is to copy the victim line from the cache RAM into the write buffer regardless of whether the line is dirty or not (*3cb*). In the write-through cache, the ejected line is simply discarded.

Then (for both write-through and copy-back caches) the requested cache line is fetched from the memory and latched in the LFL. As soon as the requested word arrives in the LFL it is sent to the processor whilst the remainder of the cache line is fetched. Lastly, in a copy-back cache, when the memory bus is idle any dirty data from the write buffer can be copied out to the main memory (*8-*) to maintain coherency.

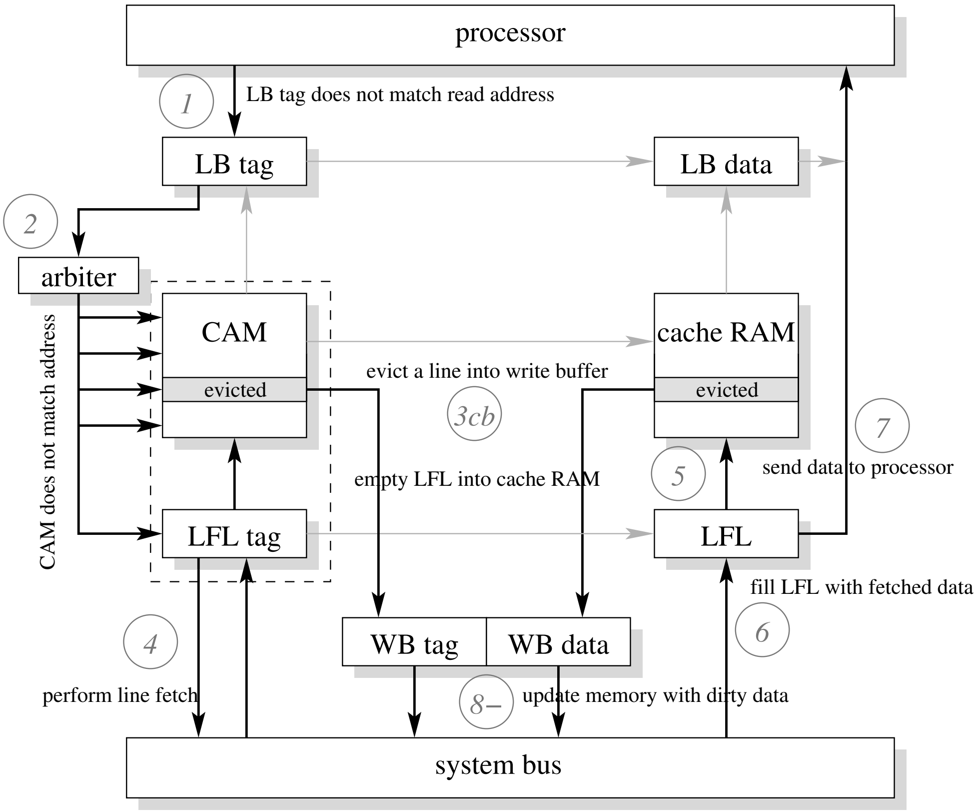


Figure 5.12: A cache read miss

5.6.8 Write miss

As can be seen from figure 5.13, the operations that occur for a cache write miss vary considerably with the choice of write policy. In a write-through cache, a write miss is a rather simple operation. Since every write has to update the memory, the write request is sent to the system write buffer (*3wt*) and is emptied out when the memory bus is free (*4wt*). In a copy-back cache with write-allocation, a write miss requires a few additional operations, some of which (those concerning cache allocation) were described in section 5.5. A line fetch is also performed on a write miss then the write modification proceeds as if it were a write hit in the LFL (*7cb*). Again, when the bus becomes idle the first dirty data in the cache write buffer queue is copied to the memory to maintain coherency (*8cb-*).

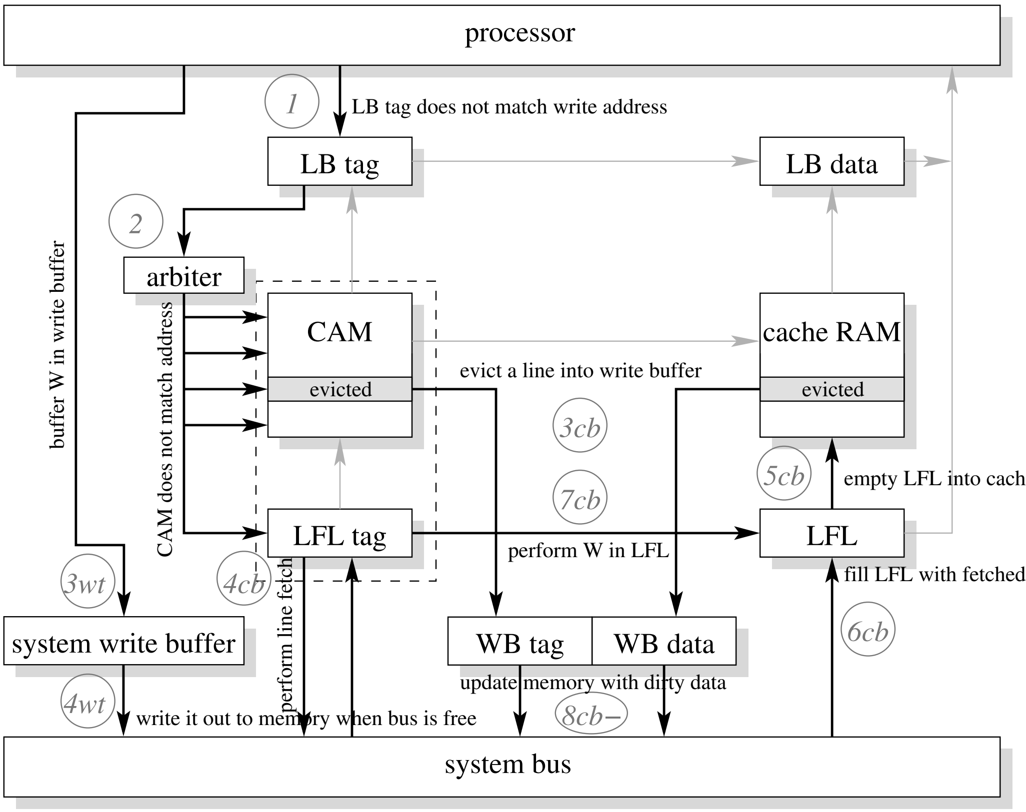


Figure 5.13: A cache write miss

5.7 Exploiting sequentiality

The AMULET2e cache exploited the sequential relationship between many of the memory accesses performed by the core. This improved both power and performance in two ways: by reducing the number of CAM look-ups and by only precharging the RAM when necessary for an access.

These improvements are viable since the special cases when such optimisations cannot be applied, non-sequential accesses or the first access to a line, are indicated by the processor’s sequential flag and by the ‘all zero’ pattern of the line’s word select bits (bits 2-3 here) as shown in figure 5.14.

Using line-buffers as a L0 cache means that most sequential accesses will hit in the line-buffer and, therefore, not cause any CAM activity. The sequential optimisation can, however, be used to improve performance by avoiding a line-buffer tag check when

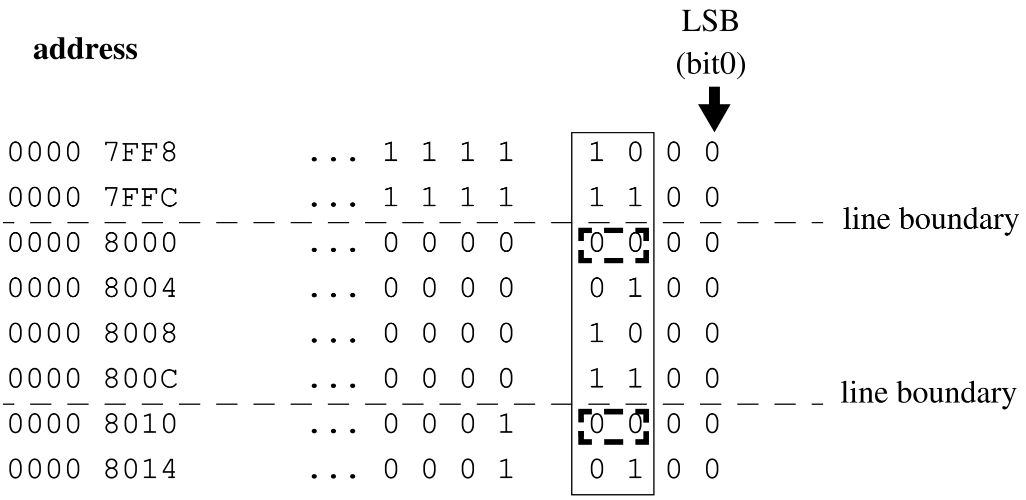


Figure 5.14: Identifying when not to perform sequential optimisation

detecting sequential line-buffer hits. Furthermore, this technique also allows the CAM check to be avoided for sequential LFL accesses, but slight additional complexity is required to manage correctly cases where the arbiter within the block interrupts a stream of sequential accesses from one port to service the other port.

5.8 Timing in a non-blocking line fetch mechanism

In asynchronous systems, the timing of individual components is much more flexible than in synchronous systems since there is no global clock signal to control all of the activities. Instead of having to finish in a fixed number of clock cycles, each component in this asynchronous cache system has its own ‘intrinsic’ timing and delivers results as soon as it is ready, rather than waiting until the next clock edge to do so.

This section illustrates the timing of the different activities in the cache system, especially those concerning a non-blocking line fetch mechanism and a copy-back scheme. For simplicity only a single memory port is described. AMULET3 has a Harvard-like architecture and may make parallel or overlapping memory accesses, but this does not affect the fundamental picture given here.

5.8.1 Hits and misses in a non-blocking scheme

Figure 5.15 depicts a number of activities that can happen in a cache system where a non-blocking line fetch mechanism is used. The upper part of the figure shows the contents of the cache starting from the left i.e. initially line `X` is in the line-buffer, line `Y` is in the cache and line `A` is in the LFL.

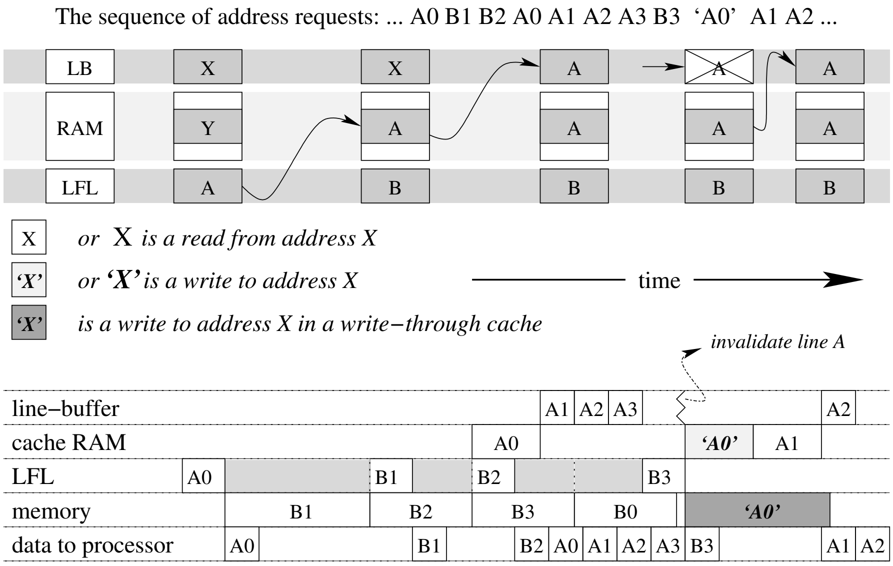


Figure 5.15: Hit timing

The first operation is a read request from address `A0` which is a hit in the LFL. This can be serviced directly from the LFL. The next request is another read, `B1`, which is a cache miss requiring a line fetch for line `B`. The contents of the LFL (`A`) are copied into the cache. As soon as the required word, `B1`, arrives in the LFL, it is sent to the processor.

The next operation is another read, `B2`, which is in the currently fetched line. Because of the streaming method (part of the non-blocking scheme) the processor only has to stall until the word is valid in the LFL. As the line fetch process continues, further cache requests are also processed in the hope that they will be serviced by other parts of the cache (e.g. the line-buffer).

The next request is another read from address `A0` which is now in the main cache RAM. Since the majority of the memory accesses are sequential, the assumption that “the words subsequent to the current request are usually requested” holds. Therefore, the whole line A is read from the RAM and buffered in the line-buffer and `A0` is sent to the processor. Then the following three read requests (`A1,` `A2` and `A3`) are serviced quickly from the line-buffer.

The next request is a read from address `B3` which has just arrived in the LFL, and so there is no extra stall for the data. The subsequent request is a write to address `A0` which is in the line-buffer. The request is passed on to the main cache after the line-buffer is invalidated to prevent any subsequent requests from reading the wrong data as described in section 5.6.2. In this case the write can be performed in the main cache and, if this uses a write-through policy, the write also proceeds to main memory.

The next request is a read from line `A` which has just been invalidated in the line-buffer. Therefore the request has to access the main cache RAM which again retrieves a whole line and updates the line-buffer. The last request here is a read from `A2` which now can be read out directly from the fast asynchronous line-buffer.

5.8.2 Handling writes

Whilst the previous section described the cache activities focusing on read requests, this section presents cache activities related to write requests and how these writes are handled in different write policies; the choice is between write-through with a write-around policy and copy-back with a write-allocate policy.

Figure 5.16 illustrates the benefit of using a copy-back scheme. The cache contains line `X` in the line-buffer, line `A` in the main cache and line `B` in the LFL at start-up. The request sequence consists of writes to: `A0, B2, B3, C1, C2` and then a read from `C1.` The first operation, `A0`, is a cache write hit whereas the next two writes, `B2` and `B3`, are LFL write hits. In a copy-back cache (figure 5.16b) these writes are handled entirely in the cache unlike in a write-through cache (figure 5.16a) where write operations to the main memory are also required. Clearly using a copy-back scheme reduces write traffic to the memory though there will still be some later write operations needed for dirty evicted data.

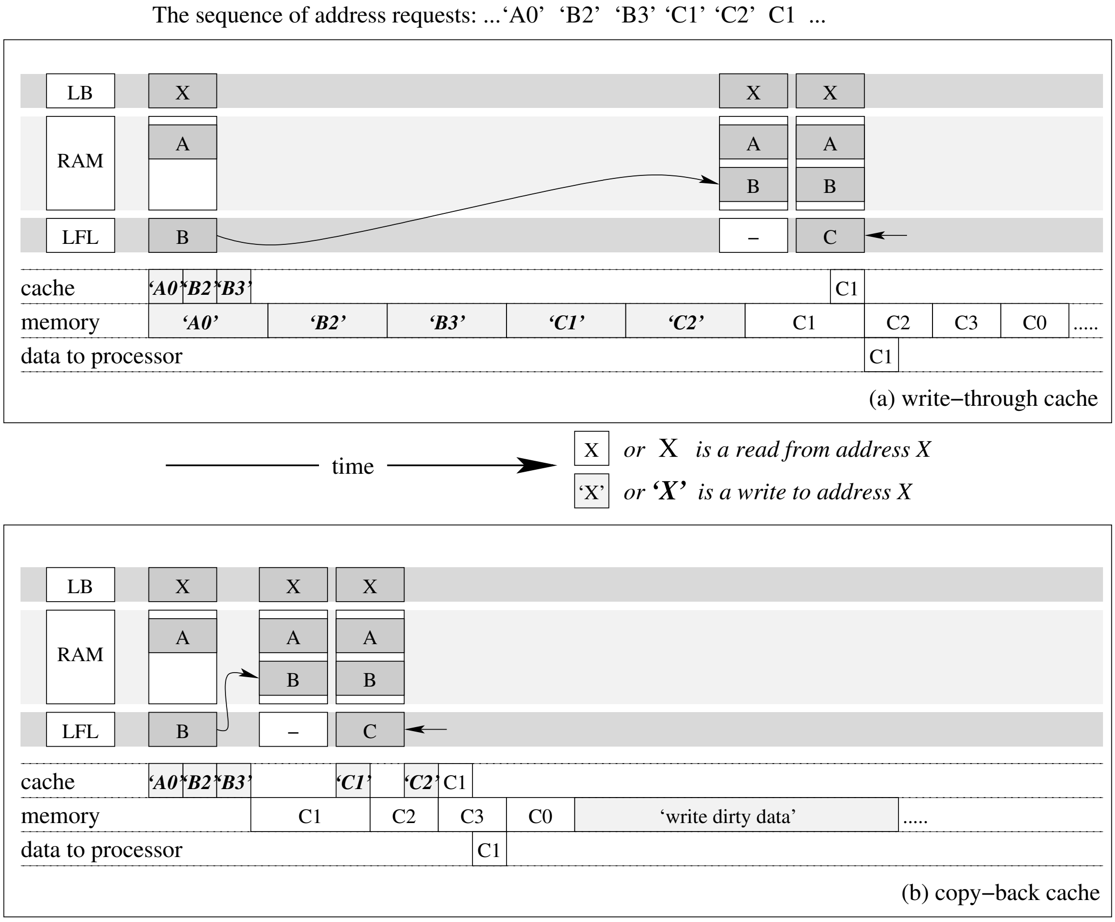


Figure 5.16: Timing for a sequence of writes

The writes `C1` and `C2` are cache write misses which are fairly simple write operations to the memory in a write-through cache but in a copy-back (with allocate-on-write) cache the first write `C1` sets off a line fetch and then the second write `C2` is performed in the LFL.

The last operation is a read from `C1`. It is a reasonable assumption that data that has been written is likely to be read again in the near future. In a copy-back cache this operation is an LFL read hit, showing how fetching on a write miss can actually 'increase' performance by effectively prefetching ahead of subsequent reads from the same line.

As can be seen in figure 5.16, burst mode memory access can be more beneficial in a copy-back cache than in a write-through cache. This is because both a line fetch and a data writeback can (easily) take advantage of a burst-mode memory access in the copy-back cache whereas only the line fetch can benefit in a write-through cache unless the write buffer is extended to coalesce individual writes into a burst.

5.9 Resolving ordering problems

To simplify the description of the cache's basic operation, section 5.8 assumed that the processor issues a request from only one port and the cache system has only a single cache block. The real system is somewhat more complex. This section describes the combined effects of having a dual-ported processor and multiple-cache blocks and allowing more than one outstanding memory accesses per port. In practice, this requires multiple tokens in the processor throttle system of the AMULET3 core as described in section 4.3.2.

Since the cache system is divided into (provisionally, eight) cache blocks, all of these could provide fast memory accesses concurrently. This would clearly yield higher throughput and better performance than a single block. Furthermore, with (nearly) two-levels of cache in each cache block where each location (line-buffer, main cache RAM and LFL) in the cache has intrinsic timing delay and a pipelined structure, there could potentially be more than one memory accesses in progress at any one time in each block.

Unfortunately, with the allowance of the multiple-outstanding memory accesses required to support this, there is a risk of read data being presented to the processor's memory port in a different order from which it was issued. The following subsections describe two major out-of-order scenarios that could occur in this cache architecture and the solutions that are used to support out-of-order memory completion. This is a commonly encountered problem in asynchronous design with well-known solutions including result reordering [37], ordered result collection [94] and avoidance of the problem through single outstanding activities [9].

5.9.1 Inter-block data ordering

The first scenario where ordering problems can occur is when consecutive cache accesses are handled by different cache blocks by virtue of the interleaving of the cache blocks as described in section 5.2. In this situation if, for example, the first access hits in the main cache-RAM and the second access hits a line-buffer (of a different block) then the second requested data may be ready before the first. A similar situation arises with any fast access immediately following a slow access (e.g. a miss) as can be seen in figure 5.17.

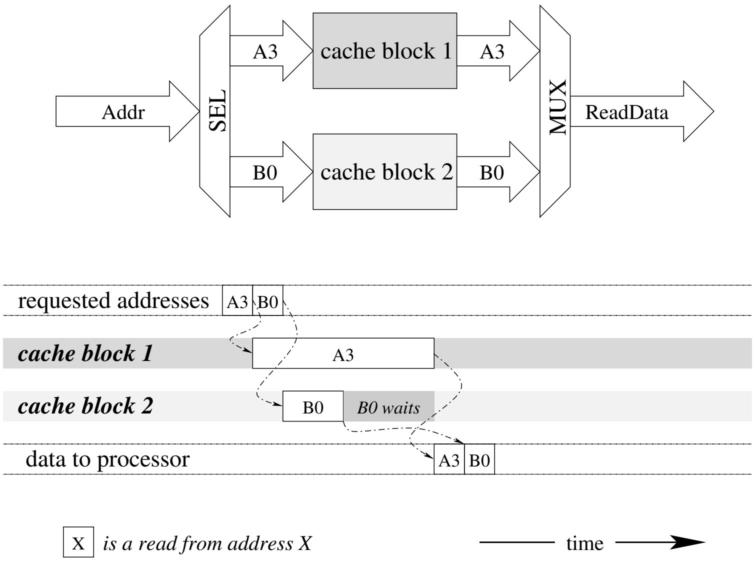


Figure 5.17: Order problem due to concurrent activities of different durations

This type of situation, where ordered activities of different durations are allowed to run concurrently, can be accommodated either by enforcing order by delaying the start of the second activity or by reordering the returned data so that it arrives at its final destination in the expected sequence or by collecting the results in the correct order.

Here the last approach is used – the problem is managed through the use of a control FIFO at each memory port to control the collection of data from different blocks prior to its presentation to the processor. This is the same approach as was used with the AMULET3 RAM (described in section 4.3.2), and is only viable if the blocks are constructed such that within each block, requests and their corresponding data enter and leave in the same order.

5.9.2 Intra-block data ordering

The second scenario where data ordering may pose a problem is when consecutive requests are handled via different paths within the same cache block. There are two such cases where this could happen: a fast line-buffer access racing against a full cache access involving arbitration, and a cache access racing against a preceding miss.

Ordering across the arbiter

For improved performance, each cache block is pipelined, allowing the tag of one access to be compared whilst the data for the previous access is retrieved. Additionally, each level of the cache is separately pipelined internally, therefore when a line-buffer read hit occurs after an access that does not hit in the line-buffer and has to pass through the arbiter to the L1 cache or main memory, the line-buffer hit is likely to produce its result before the preceding access. Again this ordering is enforced through the use of another control FIFO (similar to the one used to solve inter-block data ordering in section 5.9.1) to collect accessed words from the right place. In fact, two separate FIFOs are used, one for each port, as shown in figure 5.18.

Ordering after the arbiter

The structure of the L1 cache proposed here is similar to the AMULET2e cache, however, the pipelined stages are now shared between ports using an arbiter. With the simple AMULET2e pipelining, a stall on one port due to a miss would unnecessarily block the other port to access the main cache as well. To avoid this problem, the pipeline latch between the main cache RAM and the LFL is split, along the zigzag in figure 5.19, allowing LFL access to be sidelined so that the main cache RAM is still accessible whilst a line fetch is performed.

Unfortunately, in allowing this another potential race situation is introduced: when a cache access occurs after a miss, the hit is likely to produce its results before the preceding miss. As before, this ordering is managed through the control FIFOs, described above.

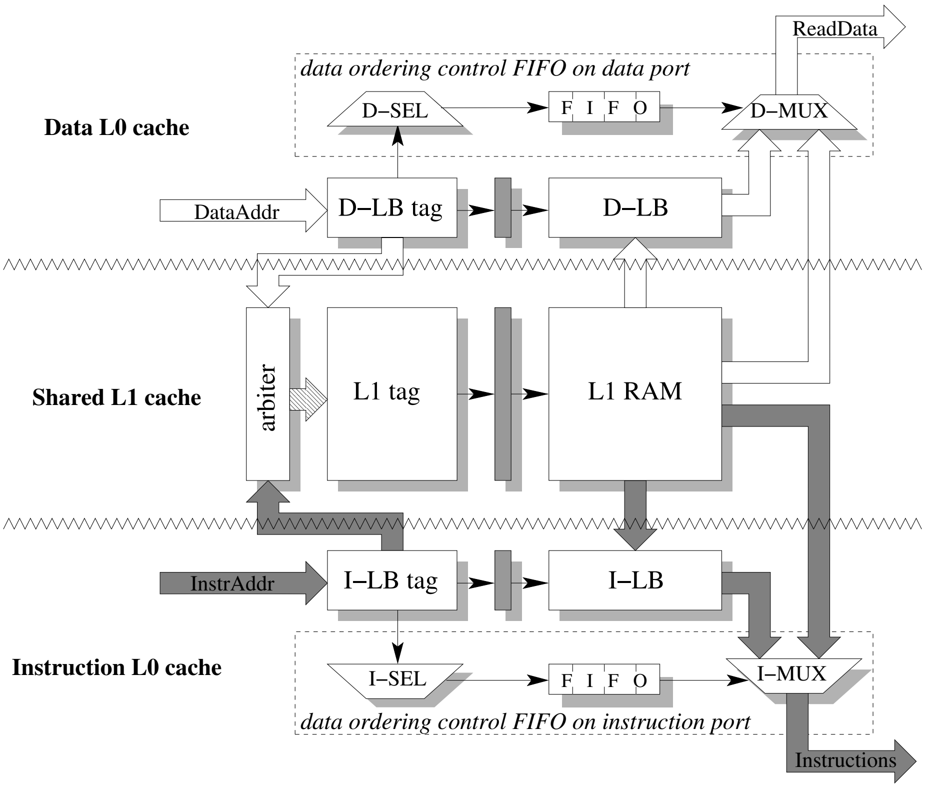


Figure 5.18: Managing ordering between L0 and L1 caches

5.10 Write buffering

Although there are two write buffers, each of which buffers different things as described in section 5.1, this section discusses only buffering in the cache write buffer.

Apart from a cache flush, a copy-back cache only writes data to the main memory when a cache miss occurs, which requires a (possibly dirty) line to be emptied in order to make room for the newly fetched data. A cache miss can then trigger (up to) two different processes which require access to the memory bus: a line fetch (`R`) to retrieve the required data and, possibly, the writing back of the dirty evicted line (`W`). Figure 5.20 illustrates three sequences of memory bus activity possible when two misses (`A2` and `B1`) occur in close succession. Each of these misses causes both a line fetch and a write back process where the first line miss `A` evicts line `B` which is required for the immediately subsequent miss (`B1`). Markings are used to indicate either the first miss (`-1`) or the second miss(`-2`). Without buffering the order of bus accesses is `W-1 R-1 W-2 R-2`. This is shown in figure 5.20a.

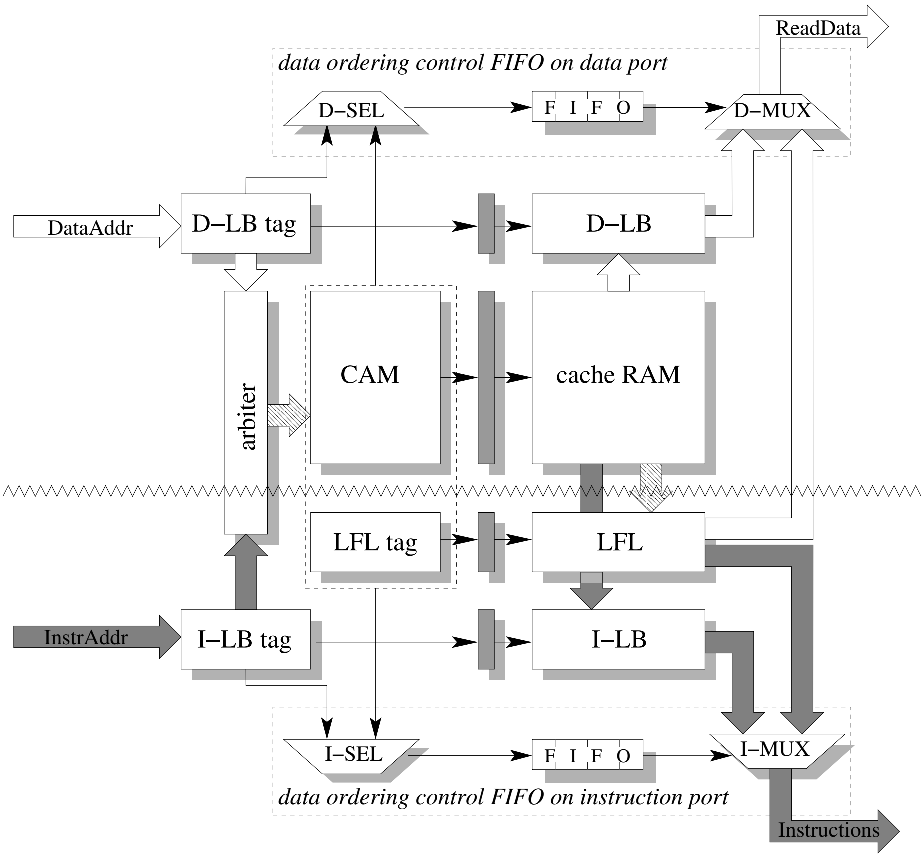


Figure 5.19: Control FIFO resolving intra-block data ordering

For obvious performance reasons, the write should be performed after the read corresponding to the same miss since the processor is waiting for the new data. This requires additional storage to hold the dirty data in the meantime. The mechanism described above introduces a write buffer [41] (the cache write buffer) as a place to which to move out a potentially dirty line from the cache RAM in order to make space for newly fetched data to allow the read (which is urgent) to precede the write of the evicted line to memory.

In general, reordering state-changing operations is liable to cause hazards, especially as in this case data may be read before it has been altered by an earlier write. However, in the case of reordering the data read and evicted line write for only a ‘single’ cache miss a hazard cannot arise because the line being fetched caused a cache miss and so cannot be aliased to the rejected line.

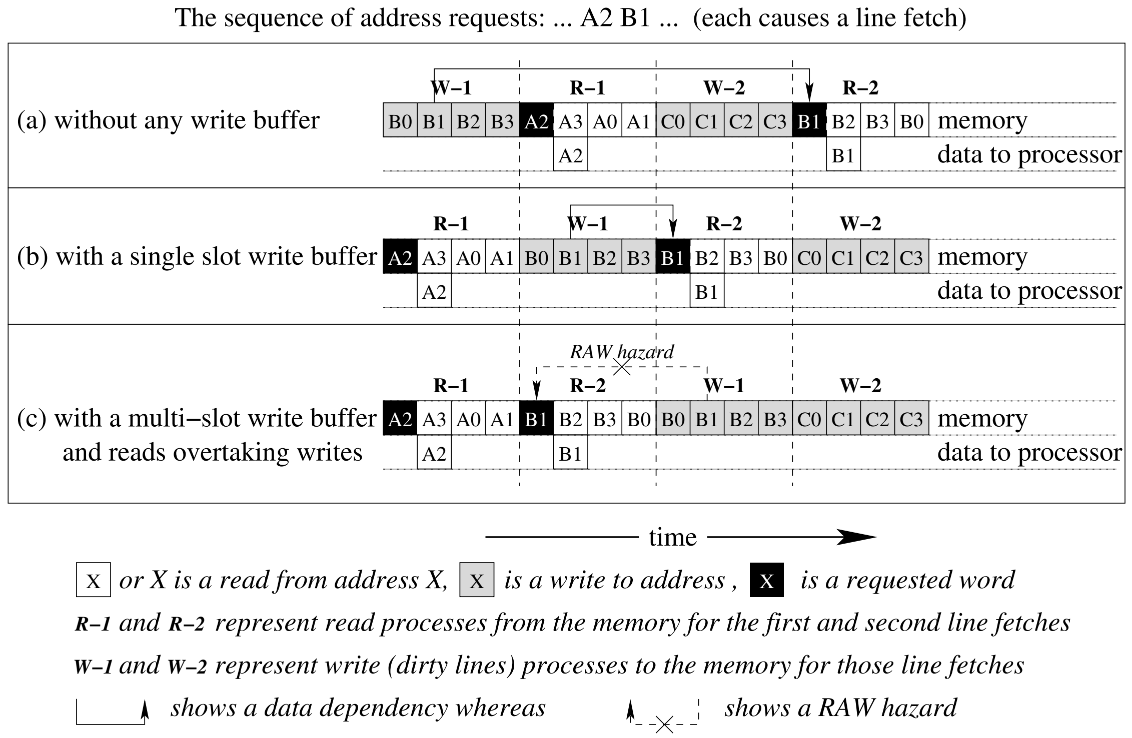


Figure 5.20: Write buffering

The simplest write buffer scheme has sufficient storage for a single cache line. Only one slot in the write buffer is required for the write to be deferred until after the read has completed, as in figure 5.20b. Each time there is a cache miss the buffer is updated whilst a new line is fetched; it is subsequently emptied into memory if it is ‘dirty’ or simply marked as empty if the write would be superfluous. If a second line fetch is required then that must wait until the write buffer is empty before it can begin, giving the ordering `R-1 W-1 R-2 W-2`. If `W-1` is a true write (from a dirty line), this could delay the performance-critical `R-2` operation.

In order to reduce processor stalls further when two or more line fetches are required in close succession, memory accesses can be reordered so that all outstanding reads are performed before the writes begin (a *read-overtake-write* scheme). For the above example, two line fetches which both cause write operations, the memory accesses could be performed in the order `R-1 R-2 W-1 W-2`, resulting in a significant latency reduction for `R-2`. Clearly this requires more than one slot in the write buffer as shown in figure 5.20c.

Whilst fairly straightforward in the synchronous domain, overtaking can cause problems in an asynchronous implementation where it can be difficult to determine if a read operation has been requested before a write burst begins because of the lack of synchronisation between the input and output units of the write buffer. Because the write and a subsequently requested read are asynchronous, arbitration is required to make the decision between the line-fetch process and the write buffer write-out process, leading to non-deterministic behaviour. As shown in figure 5.20c this mechanism can also lead to *Read-After-Write (RAW) hazards* [41], which must be resolved for correct operation.

5.10.1 Arbitration for the system bus

If a line fetch has evicted a dirty line there will be data in the write buffer waiting to be written into main memory. In a simple system the write out operation could be queued to be the next main memory bus transaction after the line fetch and the system would be wholly deterministic (i.e. arbitration free). However, with hit-under-miss system support, it is plausible that a second cache miss could occur before the first line fetch is finished. In these circumstances it is desirable for the second line fetch to overtake the pending write to reduce read latency.

In an asynchronous system, it is possible for the second fetch to arrive at the instant the previous fetch completes, requiring an arbiter to decide whether it preempted the write starting. Because most standard asynchronous arbiters work on a ‘first-come-first-served’ basis, and the write is likely to arrive first, this circuit needs to be specially biased to grant a read if at all possible i.e. if the write buffer is not full.

A suitable circuit for this, similar to the one used to sample interrupts in AMULET3, is shown in figure 5.21. This arbiter is the only point of non-determinism introduced in this scheme. Its behaviour is such that: when the system bus is not busy, and there is a fetch pending or the write buffer is not empty, the mutex [88] R2 input is asserted. When the arbitration is won by the R2 input, G2 is asserted activating either ‘do read’ or ‘do write’, depending on the value held in the latch (TL). When the transfer begins, the bus becomes busy and a return-to-zero sequence of events releases the mutex R2 input, and G2 is deasserted. The value held in the latch, which determines whether to ‘do read’ or ‘do write’ can only change when R1 is asserted and wins the arbitration (within the mutex),

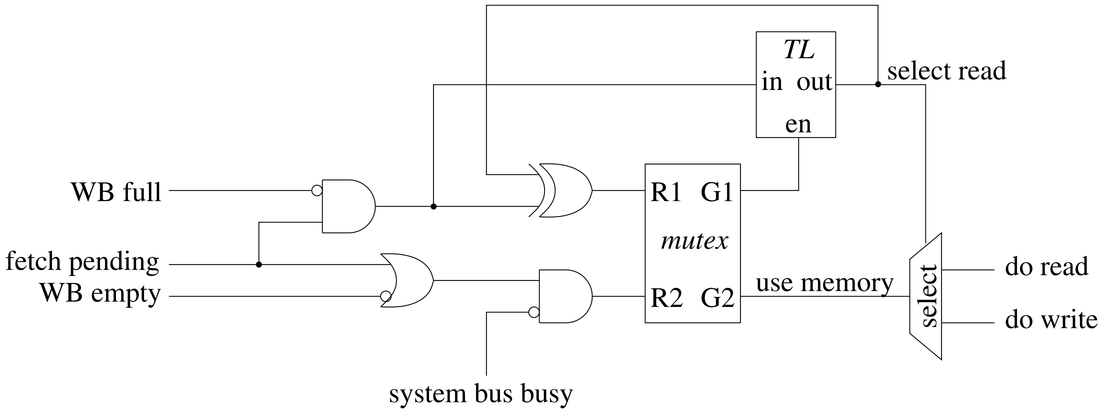


Figure 5.21: Next memory transfer decision logic

which guarantees a new transfer will not start whilst the latch is open. When the latch input and output are the same, the XOR gate drives low, releasing the mutex and closing the latch. Thus a read is selected whenever a fetch is required, unless the write buffer is already full.

5.10.2 Read-After-Write hazards

Allowing a read to overtake writes – other than a corresponding evicted line – introduces potential memory coherency hazards, i.e. RAW hazards. This is not a problem with a single evicted line because, by definition, the outgoing line cannot conflict with the line being fetched to replace it, but if more than one entry is allowed in the write buffer this protection is no longer assured and must be provided explicitly. A write buffer with more than one entry could lead to a situation where `R-2` clashes with `W-1` as in figure 5.20c.

Solutions to this problem include:

- Do *not* reorder. The write buffer must be drained before a read is performed. This would not allow the advantage of read-overtake-write.

- *Forward* the required data to the processor directly from the write buffer if it is fetched again. Forwarding not only solves the coherency problem but, by virtue of storing and returning recently ejected lines locally, turns the write buffer into a victim cache [57].

Clearly the second option is preferable if some mechanism of forwarding can be provided without introducing hazards in the asynchronous environment. Implementing the forwarding mechanism and victim cache is the subject of chapter 6.

5.11 Summary

This chapter has detailed how a number of novel and existing techniques can be combined to create an asynchronous copy-back cache for the AMULET3 microprocessor.

The major techniques used here include:

- dividing the cache into a number of independent blocks in order to reduce the power consumption and the probability of clashes between instructions and data;
- internal pipelining in each block allowing tag look-up and data access to proceed concurrently;
- separate instruction and data line-buffers which effectively behave as a (fast) L0 split cache;
- a writable line fetch latch (LFL) with a non-blocking line fetch mechanism to reduce processor stalls on a write miss and support hit-under-miss;
- a copy-back write policy to reduce memory bandwidth;
- a write buffer with read-overtake-write support to reduce processor stall during for requested data.

The benefits of these techniques in a model of the fully asynchronous cache are analysed in chapter 8. However, prior to this, chapter 6 describes an asynchronous victim cache that can be used with the cache described here to resolve RAW hazards in the write buffer.

Chapter 6: Victim Caches

Chapter 5 presented an asynchronous copy-back cache architecture designed to work with the AMULET3 processor. That chapter ended with the problem of a RAW hazard in basic write buffering[1] using the read-overtake-write technique where the line fetch data conflicts with the buffered writes in the write buffer. This could also happen in a synchronous environment, where one well known solution is to forward directly from the write buffer. There is no obvious reason why the same technique should not be applied in an asynchronous environment although implementing a forwarding mechanism in an asynchronous system, as addressed in this chapter, is more difficult because the data to be forwarded is flowing in an unsynchronised manner to the process which requires it.

6.1 Forwarding

A possible solution to forwarding in an asynchronous environment was introduced by Gilbert [37], an asynchronous implementation of a *reorder buffer* intended for use in a processor register bank. The reorder buffer accepts input data with arbitrary ordering and outputs them in a pre-assigned order. Forwarding of any entry is allowed from the time it is written until it is overwritten by new data. A similar technique can be used here. This allows memory writeback to proceed unimpeded, but leaves valid data in the write buffer until it is overwritten.

Forwarding not only solves the coherency problem, but can also reduce the number of memory cycles by intercepting line fetches to recently ejected addresses (due to mismatch between system behaviour and the replacement algorithm). Evicted lines which are still required will then be returned to the main cache before they are lost from the local system.

---

[1] a cache write buffer in the previous chapter

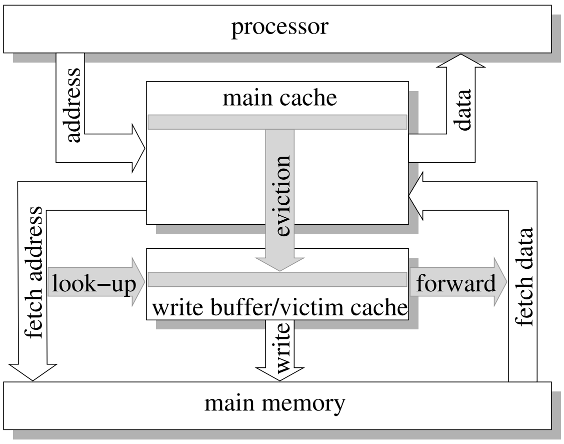


Figure 6.1: Write buffer/victim cache position

In this situation the write buffer is now performing the function of a *victim cache*. The position in a memory system of a write buffer/victim cache is shown in figure 6.1.

Unlike the victim cache first proposed by Jouppi [57], where the victim cache tag look-up was performed in parallel with the main cache tag check, thereby reducing the miss penalty, in this architecture, the victim cache tag look-up is triggered only on a cache miss. This gives better power efficiency since most of the accesses can be satisfied in the main cache.

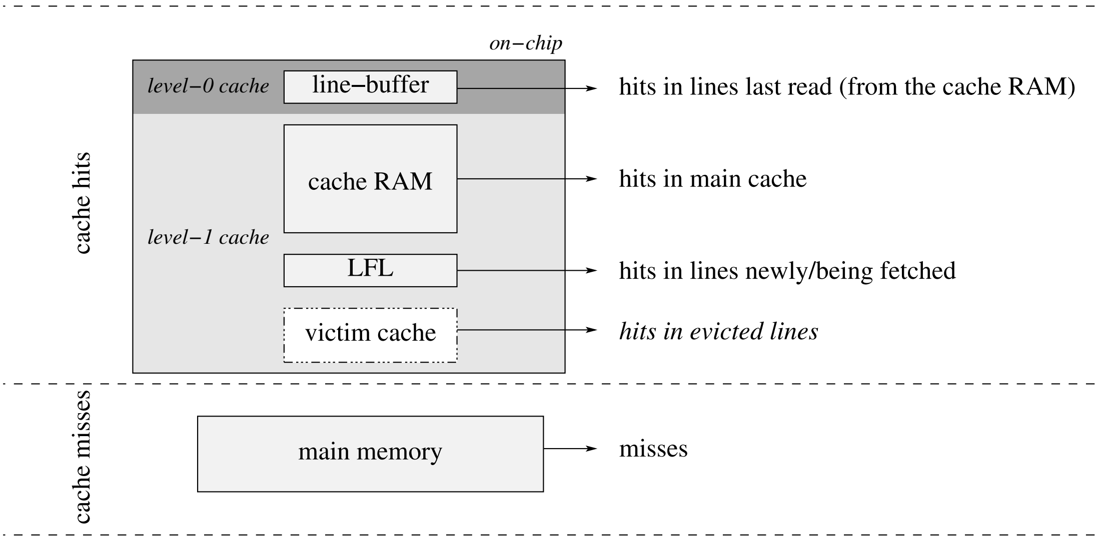


Figure 6.2: ‘Nearly’ two-level cache structure incorporating a victim cache

Referring back to figure 5.4, the inclusion of a victim cache provides a further ‘hit scenario’, as clarified in figure 6.2, since the victim cache holds recently evicted lines. This has its own address tags which are checked in a cache look-up after the request misses in all of the above locations but before the request can trigger an access to the main memory.

When a cache miss occurs, the line which is being ejected to the victim cache need not be considered in the address comparison for forwarding purposes since it will never contain the required line. It must be excluded because the fetch (and, possibly, forward) and the write buffer insertion processes are asynchronous so the contents of this location may be changing during the comparison process. Therefore the victim cache holds *one fewer lines* than it has storage locations in the write buffer.

Figure 6.3 illustrates the different sizes of data transfer from/to the cache system presented here. Whilst cache communications with the main memory are always word-transfers (32 bits), communication with the processor can be done at various granularities up to a word long (indicated using ‘*’); i.e. a byte, half-word or a word. All internal communications within the cache system transfer a whole cache line at a time. The transfer with ‘#’ indicates the forwarding path (for both the line address and content) from the victim cache. Because the victim cache contains only complete lines, as opposed to a mixture of bytes, half-words and words as in the system write buffer shown in figure 5.1, forwarding is a viable option in a copy-back cache.

6.2 Victim cache processes

The victim cache was proposed by Jouppi [57] as a method to reduce the impact of conflict misses in direct-mapped cache structures, but is easy to generalise to any cache architecture. It is loaded only with items ejected from the main cache. In the case of a cache miss that hits in the victim cache the LFL can therefore be filled without the penalty of a memory read burst.

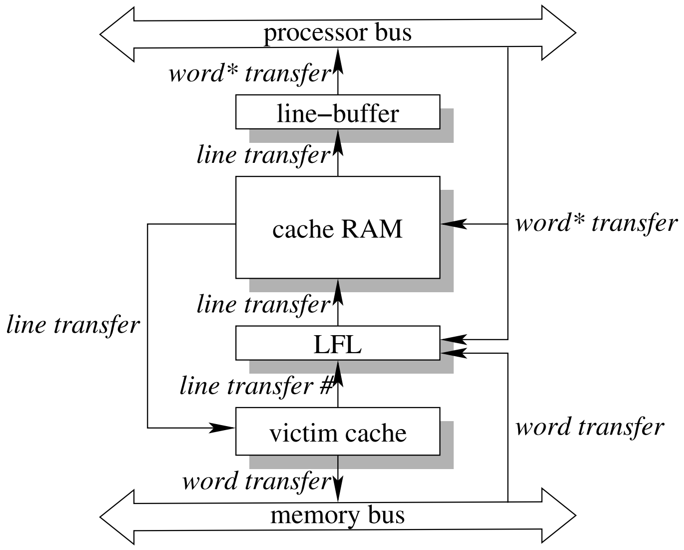


Figure 6.3: Data transfer granularity

Figure 6.4 illustrates the control flow of the victim cache operation. The victim cache itself is a fully associative cache composed of two main parts. Addresses are held in a tag store (CAM) and their corresponding data is held in the data store (RAM). However, operationally, the victim cache can be considered as a memory with three different functions indicated by the grey loops (clockwise starting from the top left) acting upon it:

- **Line-fetch and forwarding**: A main cache miss occurs so the miss address is passed to the victim cache, which must supply (forward) the requested line if it can. Again a Muller-C element ensures that the LFL is emptied before refilling it with newly fetched data.

- **Cache eviction**: A cache miss occurs and the main cache empties a line into the victim cache (shown in figure 6.4 labelled ‘fill VC’). The victim cache has to provide an empty storage location for the line at this time.

- **Buffered writes**: The victim cache autonomously copies ‘dirty’ lines into the main system memory (shown in figure 6.4 labelled ‘drain VC’), freeing space for re-use.

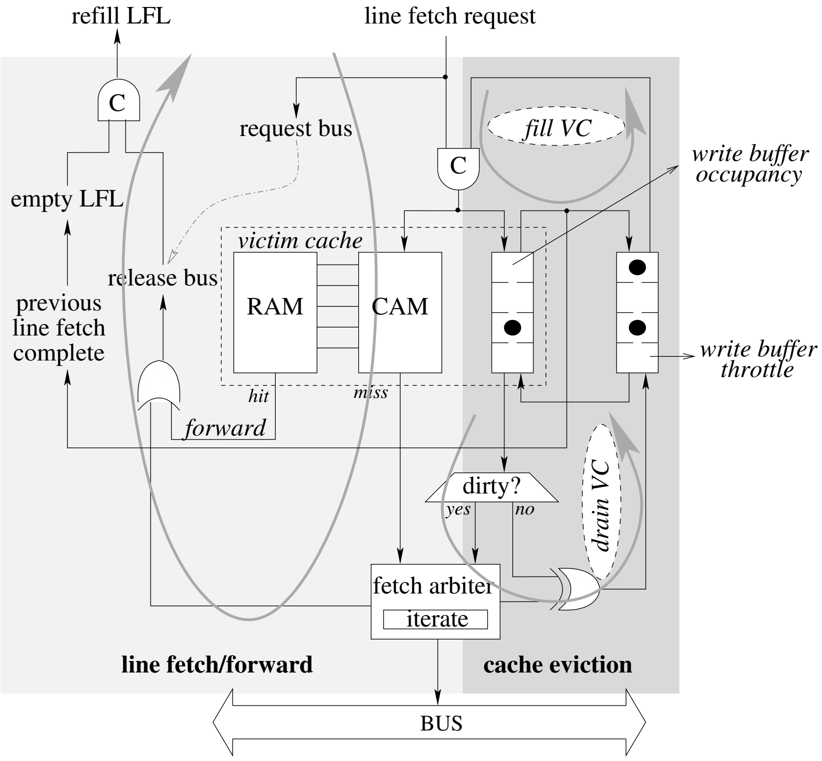


Figure 6.4: Control flow in the victim cache

However, there are only two independent, concurrent processes among these activities: filling (the first two functions) and draining the victim cache (the last function), since a line fetch causes a cache eviction. The difficulty in an asynchronous implementation is that the data flowing into/out of the victim cache is entering/leaving in an unsynchronised manner from the line-fetch/forwarding process that may require it.

6.3 Victim cache implementation

A similar approach to the one used in the reorder buffer in AMULET3 [37] which forwards register values is used here, with the simplification that inputs and outputs are always in the same order.

The write buffer is a 'circular buffer' (which is a way of implementing a FIFO). Write operations are made to the `in` pointer of the buffer and the write process strips entries from the `out` pointer whenever the bus goes idle. (The `in` and `out` pointers are shown later in figure 6.5.) A useful property of circular buffers is that data does not move within the buffers' storage elements and so can be read and forwarded despite the fact that another asynchronous process may be writing the other data concurrently. The lifetime of the 'forwardable' data is fixed by the number of write buffer entries and is entirely independent of the copy-back process.

Although the mechanism used here is similar to Gilbert's, there are some differences in the details. In particular, the possibility of forwarding is determined by a CAM look-up in both cases. In the original this was maintained by the instruction decoder, remote from read-out process and in a different timing domain, but here the CAM contains the ejected lines' addresses and therefore must be local. This is feasible because the write buffer is only modified when a line fetch is needed and thus the write and the forwarding processes are inherently synchronised.

In practice even this synchronisation is not necessary and the two processes may be run in parallel. This is because, as observed earlier, there cannot be a match between the requested and evicted lines. If the implementor desires to exploit this extra concurrency the `in` pointer of the queue must be excluded from the CAM look-up comparison because the contents of this location may be changing during the comparison process. Note that, in either case, the victim cache holds one fewer valid lines than it has storage locations.

6.4 Victim cache storage

Three types of information are stored in each line of the victim cache: the address – held in a tag CAM allowing fast parallel look-up; the data – held in RAM; and a number of additional control markers must also be kept. There are also global `in` and `out` pointers (as in figure 6.5) steering the writing into and emptying out from the victim cache respectively. Three extra bits for each data entry describe the data held (also shown in figure 6.5):

- **full** – the entry has been filled but not copied-out;
- **dirty** – the entry should be copied into the memory since it has been written to whilst in the main cache;
- **valid** – the entry may be considered for forwarding.

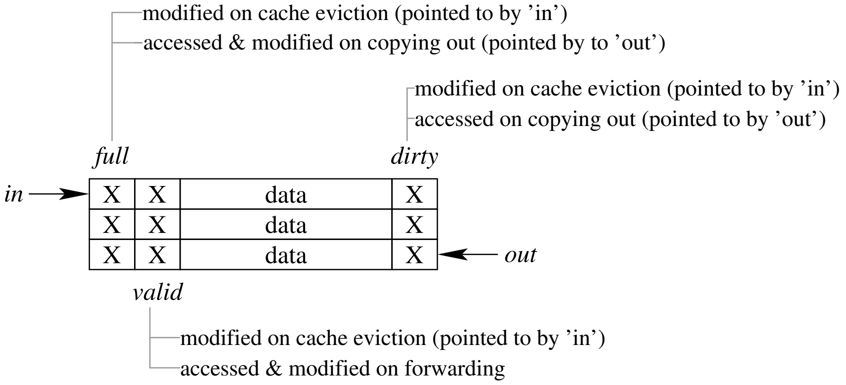


Figure 6.5: Victim cache RAM structure

When a line of data, along with its 'dirtiness', arrives it is stored in the next empty slot as indicated by the `in` pointer and the `valid` and `full` bits for the entry are set. The `dirty` bit for the entry is also set *if* the entry is dirty. The `in` pointer then moves forward to the next slot.

The concurrent process pointed to by the `out` pointer waits for an entry to be full and then checks its 'dirtiness'. If it is dirty, the process competes for the bus and performs a set of writes to the memory, otherwise these writes can be skipped. Lastly, the `full` bit is cleared to indicate that the write phase is complete and the `out` pointer moves forward to the next entry. Note that this process proceeds regardless of any, possibly concurrent, forwarding activity.

The function of the `valid` bits is to prevent the wrong data being forwarded. They are cleared at start-up when the victim cache is empty and the tag fields are undefined. However, the `valid` bit for a line is also cleared when the line is forwarded; this prevents different versions of the same cache line being valid in the victim cache at the same time, so that there can be at most one forwardable line matching any address. This removes the need for prioritisation logic to guard against the (unlikely, but possible) chance that a line is evicted, forwarded and evicted again in close succession. The forwarding process can safely clear the `valid` bit because forwarding is not possible from the entry currently used for eviction (when the `valid` bit is set).

This approach still retains the independence between forwarding (accessing and modifying the `valid` bit) and copying data out (accessing and modifying only the `full` bit). This means the forwarding scheme always returns *clean* data to the cache whilst the copying out process has to be performed regardless of whether the data has been forwarded (depending on the `dirty` bit).

There is an important difference between this forwarding scheme and a conventional register forwarding scheme. In the victim cache forwarding *moves* the data back to the cache rather than *copying* it, thus forwarding can occur only once per entry. A register forwarding scheme may duplicate the data an unlimited number of times.

The eviction and copy back processes are independent and largely decoupled, although the `in` pointer must not *lap* the `out` pointer. In practice, the constraint is slightly more strict as is illustrated in section 6.7.

6.5 Victim cache operations

The cache operations involved in forwarding are illustrated in figure 6.6. Addresses (*VC tag*) are held in the victim cache along with their data (*VC data*). Before reading external memory, a line fetch address can be compared with these address tags (*5**) and, if a match occurs, the data can be ‘forwarded’ directly from the victim cache (*6**) instead of fetching the line from memory. This does not interfere with the (asynchronous) process of writing to the memory (*8-*) which may not yet have started, may be in progress, or may have completed at this time. In the cache, the forwarded line is marked as ‘clean’ in the process of being forwarded as it is already coherent with that in the main memory or will be so after it is drained from the victim cache.

With this forwarding mechanism, the control flow for a cache read request from figure 5.5 can be extended as illustrated with the shaded region in figure 6.7. The extra complexity only has an effect on a cache miss where it will hopefully be able to forward the required data directly from the victim cache into the main cache avoiding a full line fetch.

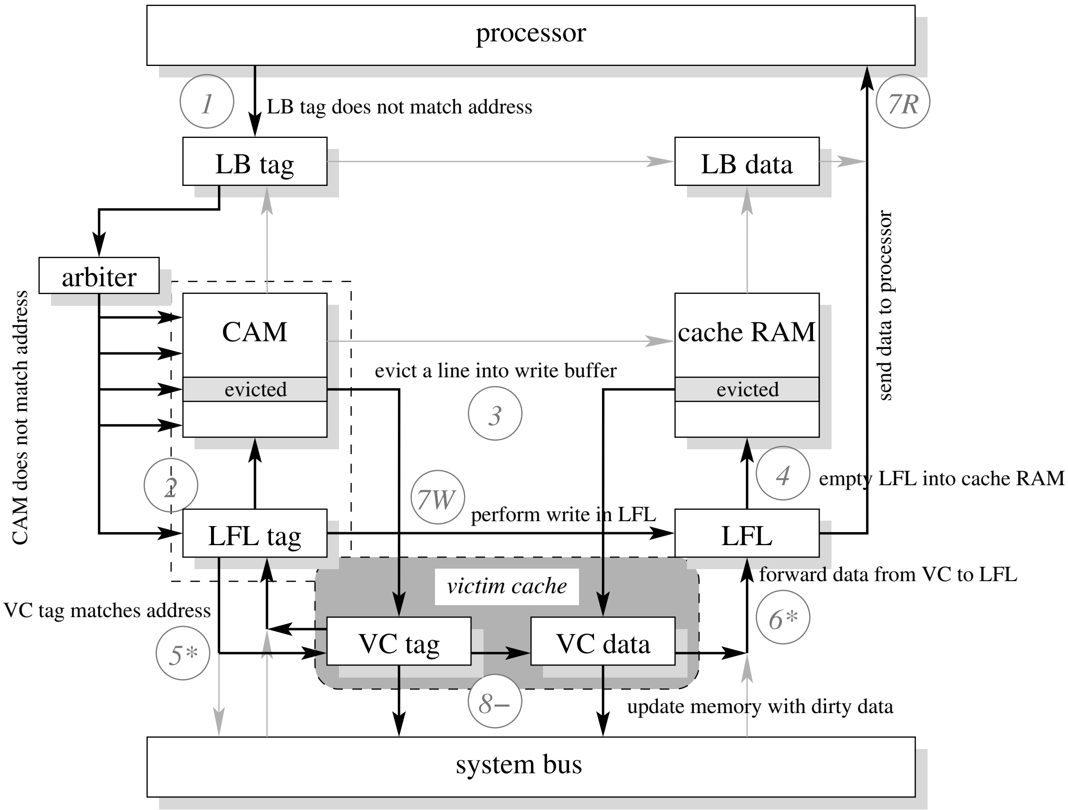


Figure 6.6: Cache forwarding operations

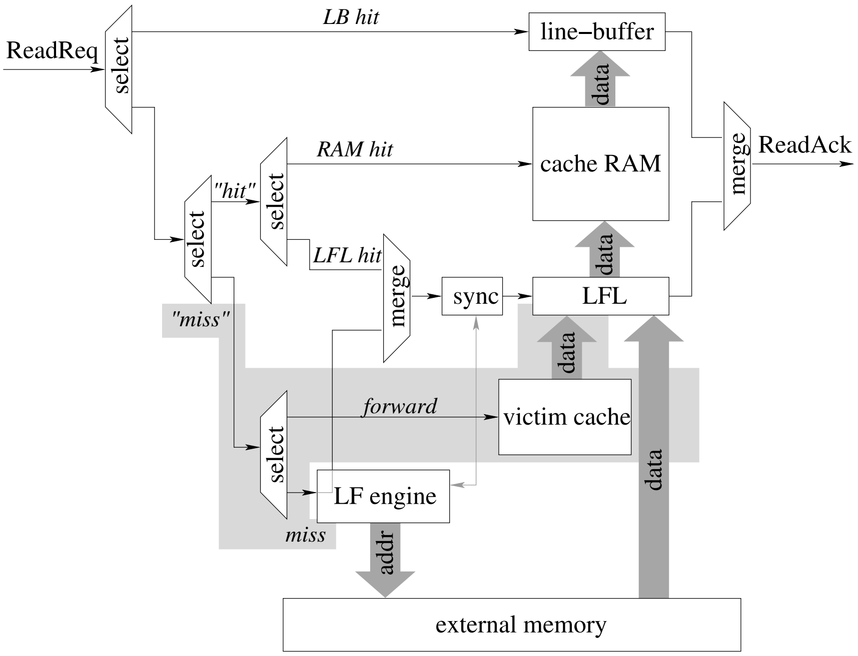


Figure 6.7: Cache read request control flow with forwarding

6.6 Victim cache benefits illustrated

Figure 6.8 illustrates the benefit of the forwarding mechanism. In this example, the system's state is that two lines (`A` and `B`) have been recently rejected from the main cache into the write buffer and the main memory has been updated with line `B`. Then these are required again with the sequence of address requests `A2` followed by `B1` each of which is a cache miss (and would originally require a line fetch). In this architecture line fetch data retrieved from the main memory enters the main cache RAM via the LFL as described in chapter 5.

The victim lines that are ejected from the cache on these line fetches are not shown in the figure since they are not directly involved in this example but it is assumed that they are all buffered in the write buffer.

The sequence of address requests: ...A2 B1... (each causes a line fetch)

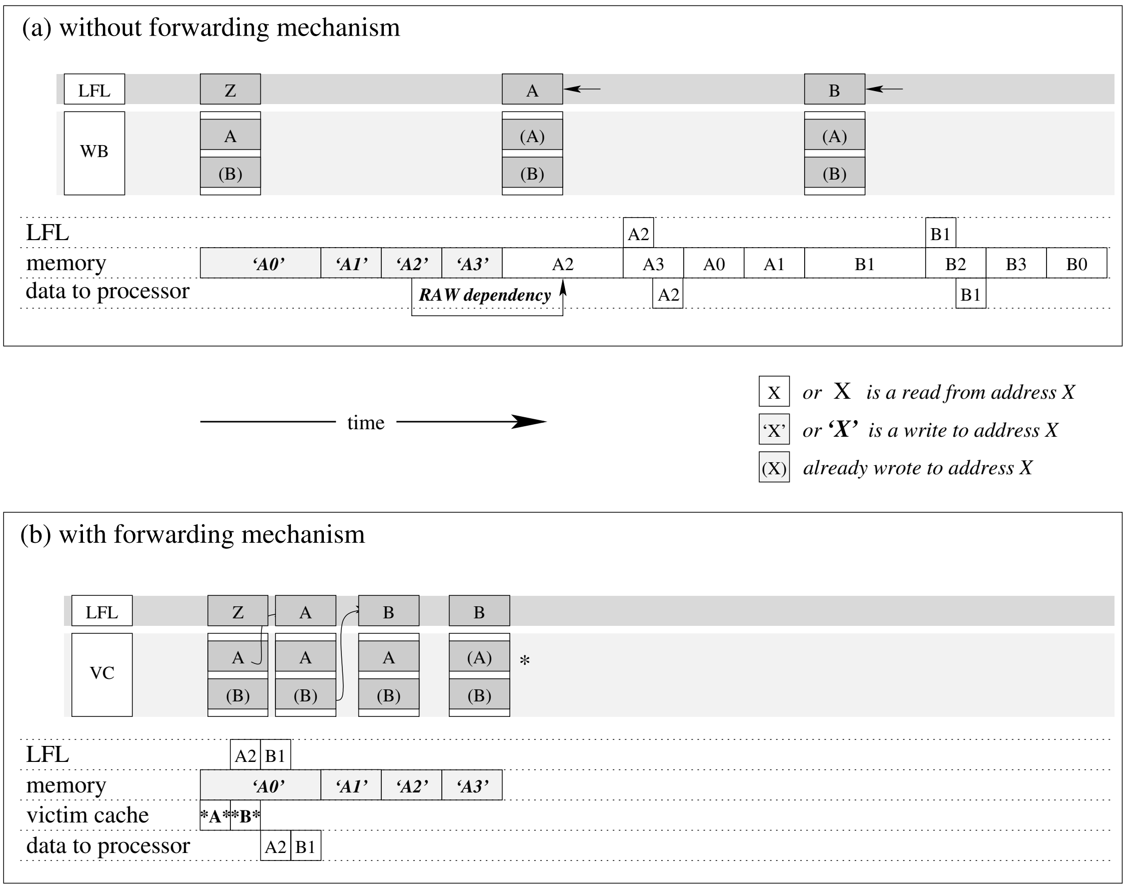


Figure 6.8: Illustration of benefits of forwarding

Figure 6.8a illustrates the activities of this request sequence without a forwarding mechanism. To solve the RAW problem, the first request (`A2`) must wait for the up-to-date value of line `A` to be written into the main memory. Then the whole of line `A` can be fetched from the memory into the cache system and the required word `A2` can be sent to the processor. The second request (`B1`) can invoke the line fetch process for line `B` directly since the copy in the write buffer is already updated into the main memory.

With the forwarding mechanism as shown in figure 6.8b, instead of stalling to resolve the dependency, `A2` can be forwarded directly from the write buffer even if that line is being written into the main memory or is waiting in the writing out queue. In this model the line fetch process is ‘short circuited’ and can occur in a single, on-chip cycle rather than four, slow bus cycles. This leads to an asynchronous process with a highly variable delay.

Furthermore, `B1` can also be forwarded from the write buffer even though it has been written into the memory. This is because the writing out process *leaves* a *copy* of `B` in the write buffer.

In this approach, forwarding reduces the processor stall period and avoids a full line fetch from the memory but does not reduce the write traffic. It is possible to cancel the copy-back process if a victim cache line is salvaged; this is discussed briefly in section 6.8.

6.7 Avoiding deadlock by using a token queue

If reads are allowed to overtake writes, there is a potential for deadlock during the cache line allocation process in a copy-back cache if the victim cache become full. This is illustrated in figure 6.9. When the line fetch engine asks for data from the memory, the memory tries to send the data to the LFL (1). However, the LFL must be emptied before it can store the newly fetched line (2). To empty the LFL requires allocation of a line in the cache RAM which must first be emptied into the victim cache (3), before the LFL can be read. If the victim cache is full, a line must be written from it into the main memory, (4), requiring the memory bus. This results in deadlock if the memory is busy performing the read (and cannot service the memory writes).

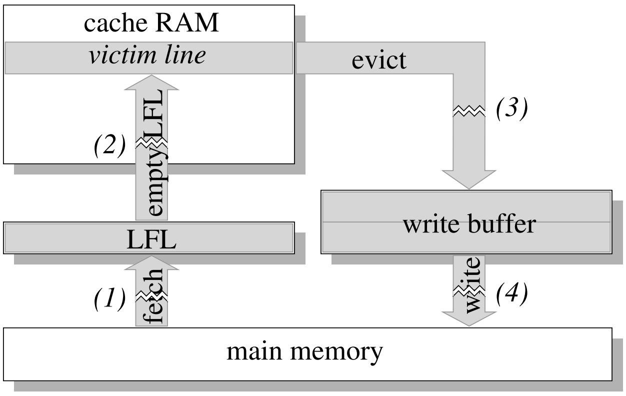


Figure 6.9: Illustration of a deadlock situation

The solution to this problem is to keep at least one slot in the victim cache empty. In an asynchronous environment, a standard way to implement this solution is to use a token queue [37] where tokens corresponding to the victim cache locations are circulated (figure 6.4). Initially, the allowed number of tokens are placed in a pool (*write buffer throttle* in figure 6.4) and then one is claimed before each eviction can begin. The tokens then reside in the victim cache (in *write buffer occupancy* in figure 6.4) until the copy out process returns them to the write buffer throttle. As there is one fewer token than victim cache locations, eviction will always stall before the last victim cache entry is filled.

6.8 Extending the victim cache to reduce write traffic

Figure 6.8 showed that forwarding can both reduce the processor stall period by avoiding a full line fetch from the external memory and (as a by-product) reduce the read traffic. However, the write traffic remains unaffected. This is because, in the approach described, the forwarding mechanism does not interact with the process copying data out to the memory. Therefore all dirty data must be written back to the main memory regardless of whether it has been forwarded.

It is possible to avoid the data copying out process if a victim cache line is salvaged. This can be achieved by detecting that forwarding has been performed before a write out (copy-back) has begun. In this case it would be possible to abort the write and instead return a (possibly dirty) line to the cache. This could reduce the bus traffic a little further, but the cost in added complexity is considerable. The additional complexity mainly involves some form of synchronisation of the forwarding and copy-back processes before forwarding is performed for any data. Unfortunately, this synchronisation may result in a long stall duration if a write out (which may possibly be irrelevant to the forwarding) is under way. The exact benefits such a scheme would offer have not been thoroughly investigated because the extra cost involved is unlikely to be justifiable.

6.9 Victim cache distribution

As described in chapter 5, the cache is partitioned into blocks although there is only a single memory bus upon which evicted data can be written. This means that there are two alternative positions for the victim cache: centralised and shared, or distributed amongst the blocks. The following subsections discuss the advantages and disadvantages of each of these two styles of victim cache for a cache system divided into N cache blocks with total victim cache size of V entries.

6.9.1 Centralised victim cache

Having a centralised and shared victim cache for the whole cache system means that V can be any size, with a minimum of 1 line. However, for forwarding V must be at least 2 lines. This is because, as described earlier, there will be one entry in the victim cache that must not be considered for forwarding, leaving $V–1$ forwardable entries.

In this style of victim cache, stalls due to filling up the victim cache are rare compared to the distributed scheme as the victim cache is less likely to be full of entries waiting for copying to the main memory. Moreover, this stalling can be easily recovered from by writing out a data entry from the victim cache. This is because the multiplexers in such a system, one required to multiplex write-out data from the N cache blocks and the other required for distributing forwarded data back to the *N* cache blocks, are placed before the victim cache, which is actually the critical path from the processor's and the main cache's perspective.

Figure 6.10 illustrates the organisation of a centralised victim cache scheme. It also depicts the wiring problem that this organisation causes due to the cost of large, wide buses (128 bits) connecting the cache blocks to the shared victim cache.

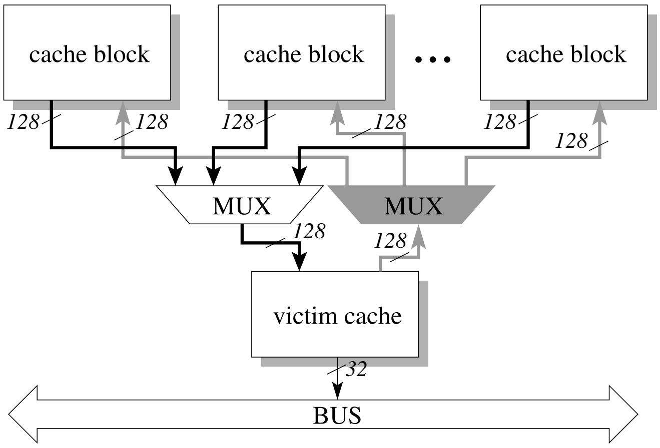


Figure 6.10: Centralised and shared victim cache

6.9.2 Distributed victim cache

For a cache system divided into *N* blocks, to provide the same total storage as the centralised scheme, each cache block has a local victim cache of *V*/*N* lines. To allow forwarding, *V* must be an integer multiple (≥ 2) of *N* where the same rule of forwarding ability is applied as for the centralised scheme.

However, since the size of each distributed victim cache is small(er), the tag comparison is either faster (for tag RAM) or cheaper in power consumption (for tag CAM). Furthermore, having a victim cache locally by each cache block, as illustrated in figure 6.11, offers two further advantages over the centralised victim cache scheme. The first is cheap wiring using short, narrow (32 bit) local copy-back and forwarding

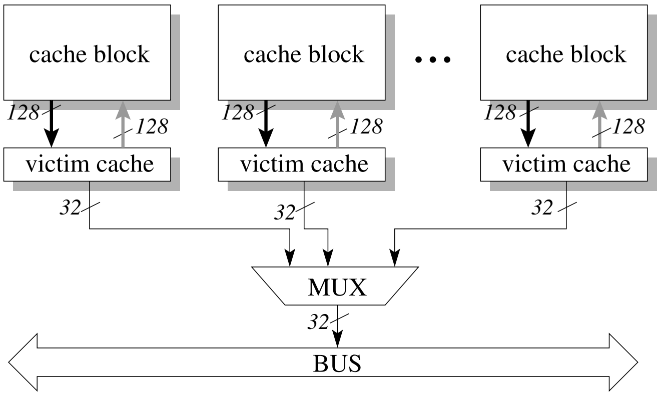


Figure 6.11: Distributed and localised victim cache

paths. The second is that the multiplexing process becomes non-critical to performance. However, with small local victim caches, long duration stalls due to filling up a victim cache are more likely to occur as the main memory arbiter may be in use draining dirty data from a different (non-critical) victim cache.

The choice of which victim cache implementation is best is not an obvious one; both schemes have advantages (unshaded) and disadvantages (shaded) summarised in table 6.1, some of which will only be quantifiable when layout is produced.

	Centralised victim cache	**Distributed victim cache**
tag comparison	bigger, hence slower tag array	faster
restriction on *V*	any size, minimum of 2 lines	integer multiple (≥ 2) of *N*
wiring cost	expensive 128-bit buses connecting blocks to victim cache	much cheaper short local forwarding paths
forwarding ability	(*V* -1) lines can be considered for forwarding	(*V* - *N*) lines
stalls due to filling victim cache	very rare as victim cache unlikely to be full of entries waiting for copying to main memory, and easily recovered.	likely, and possibly of long duration as the main memory arbiter may be servicing a different block's (non-critical) victim cache drain
multiplexing	in critical path	everything is local

Table 6.1: Benefits of distributing the victim cache

6.10 Summary

Forwarding not only solves the coherency problem introduced by using a write buffer (with read-overtake-write) but, by virtue of storing and returning recently ejected lines locally, turns the write buffer into a victim cache providing a reduced processor stall period and avoiding a full line fetch from the memory. However, it does not reduce the write traffic since this seems to require unjustifiable additional cost.

This chapter not only described how to implement a victim cache in an asynchronous framework, it also provided a suitable victim cache storage structure to guarantee that the correct data is forwarded even in the presence of multiple entries at the same line address in the victim cache. Furthermore, the token queue technique from the AMULET3 reorder buffer is reused to avoid deadlock in the copy-back process.

Finally, two schemes for implementing a victim cache for the cache architecture described in chapter 5 have been proposed and the advantages and disadvantages for each scheme have been discussed in depth. Results and evaluations of the victim cache and the alternative implementations discussed here are presented in section 8.3.

Chapter 7: Simulation Methodology

Models of the cache architecture and the victim cache described in chapters 5 and 6 were created and extensively simulated in order to verify the architectural design, validate the design decisions and evaluate the benefits of the new features. This chapter describes the simulation environment and the tools used, benchmark programs, reference cache parameters and the choices of configuration and parameter values which were evaluated. The results of these simulations are presented in chapter 8.

7.1 Synchronous cache evaluation

The three most often used methods for the study and evaluation of cache memory systems in a synchronous environment are: hardware measurement, analytical models [1] and simulation. Obviously the first technique is of no use in the early stages of cache design, but using it on existing designs allows calibration of the latter two approaches. The principal advantage of models and simulations is their flexibility which allows evaluation of a range of cache parameters in a short time and without having to build any real caches.

Analytical models can be rapidly constructed by representing only the key factors that affect cache performance, providing a quick, rough estimate of cache performance for a wide range of programs and cache configurations. Models found in the literature vary greatly in their complexity and applicability from straightforward, probabilistic models using only a few metrics and parameters to more sophisticated models including measured and calculated parameter values to corroborate results. Usually the more complex the metrics and equations involved, the more accurate, reliable and applicable the result.

There are a number of approaches for generating inputs to drive a cache simulator; probability distribution-driven, trace-driven, execution-driven and application driven simulations.

Probability distribution-driven simulation is a good choice when detailed information about the workload (program behaviour) is not available. This is the most efficient method of generating simulation inputs. However, because the need to specify a distribution, this method suffers from the same drawback as the analytical modelling discussed above.

Smith's survey [90] shows that trace-driven simulation has been used to evaluate memory systems in a synchronous framework for decades. It is a form of event-driven simulation, in which the events are supplied from a *trace*, collected from a real system. For cache memory studies, traces consist of lengthy address-reference sequences gathered by one of a variety of hardware or software methods, e.g. collecting from actual hardware in real time or capturing from simulations. The drawbacks of this technique are the storage space required to record the traces and the simulation time requirements. As cache sizes are growing, these problems are getting worse because very large traces are required to obtain accurate estimates of cache performance.

A solution to this is to select only a subset, or sample, of the complete data population. This technique, known as trace sampling, either samples in address space or in time to reduce both disk space and simulation time requirements for simulating large caches. When properly constructed, a sample can be used to derive estimates for some feature of interest without having to process the entire data set. Chen [18] reported that these sampling techniques can be very effective in reducing simulation time at the cost of a small amount of errors.

Execution-driven simulation avoids the need to store the trace by generating the inputs to the simulation on-the-fly by running program code on a real processor. This gives real-world inputs, and fast simulations but it difficult to implement.

Application-driven simulation overcomes the difficulties of the execution driven approach by extending the simulation to couple a model of the processor to the cache model. This however increases the simulation complexity and time.

7.2 Asynchronous cache evaluation

Both trace-driven and probability distribution-driven simulations are difficult to use for evalutating asynchronous caches because, in addition to representing the data-values they also have to represent the variable time between address requests. This is important because: firstly, the processor's behaviour varies depending on what action it is performing and, secondly, conflicts will not be properly represented without accurate fine-grained timing. Incorrectly modelling such conflicts gives poor results because the non-determinism of asynchronous systems means that their behaviour is closely timing dependent.

Execution-driven simulation was not possible at the time this work was performed because the only available AMULET3 processosr was embedded in a full SoC design.

As discussed above the next best method for generating addresses to the cache is to couple a model of the cache to a model of a real processor. This can then be used to evaluate early decisions in cache design. Such a model normally consists of behavioural descriptions of each unit in the design and a structural description of the interconnection between units. This type of model was used in conjunction with the already available AMULET3 behavioural models to verify the behaviour of the proposed cache architecture.

Ideally, all real timing information should be included in such a model but it is often not feasible to do so. The main purpose of the model is to study architectural trade-offs e.g. to compare central versus local victim cache designs. For these reasons, a very detailed model with a large number of parameters may be unnecessary.

7.3 Choice of modelling language

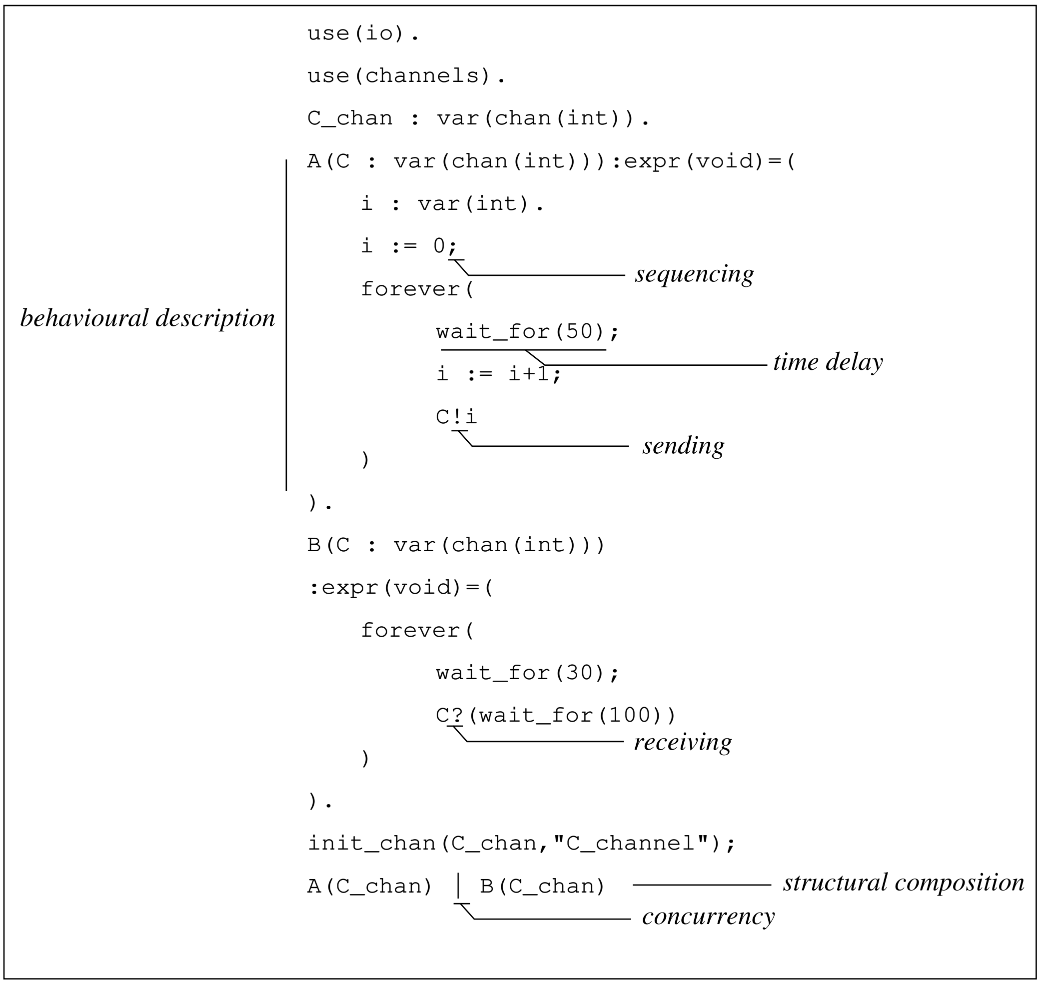
The significant advantage of conventional hardware description languages (HDLs) for modelling an asynchronous system over a conventional programming language (such as C) is concurrent programming capability. The model of concurrency used in HDLs such as VHDL is not, however, good enough to allow a simple description of the fine grain parallelism (in particular, the mixture of parallel and sequential operations) necessary to model asynchronous channel communications, resulting in a cumbersome, complicated model [36]. Frankild's VHDL++ [102] extends VHDL with new language constructs supporting the design of asynchronous circuits; however it is not yet a fully mature tool.

Different caching strategies and designs were modelled and simulated using a functional model written in LARD (Language for Asynchronous Research and Development) [55]. LARD is an HDL developed for describing asynchronous systems although little is specific to that purpose and it can be used to describe synchronous systems or even as a general-purpose programming language.

The key difference between LARD and conventional hardware description languages is its provision of CSP-like [44] channels whereby abstract communication can easily be modelled using the notations '**!**' and '**?**' for sending and receiving respectively. In addition, LARD has fine-grained concurrency with statements composed either sequentially (using '**;**') or concurrently (using '**|**'). An example LARD program illustrating this channel communication and a diagram showing its 'structure' can be seen in figure 7.1a and figure 7.1b respectively.

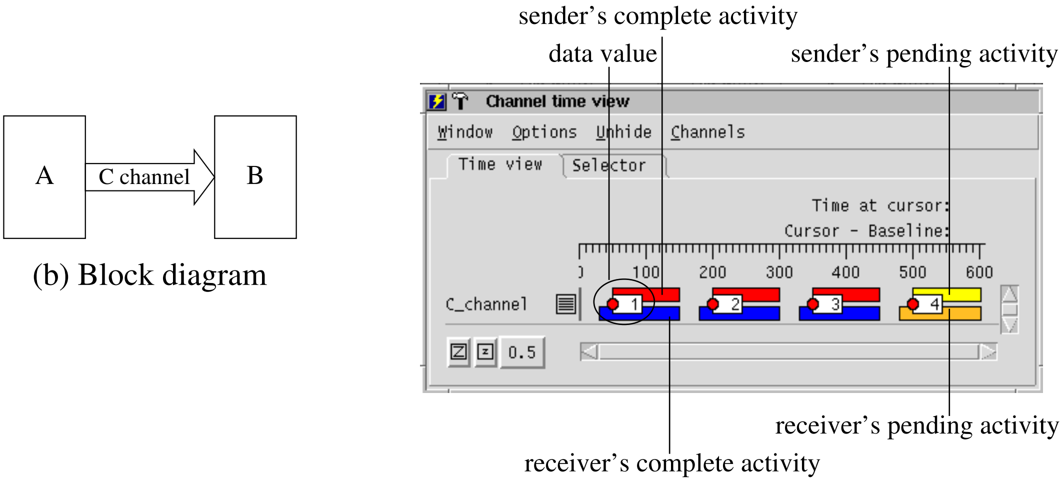
Balsa [10] was considered for this work because it, too, uses channel-based communication but it is primarily a synthesis system with limited ability for abstract behavioural modelling. Balsa in fact uses LARD as its simulation engine.

A final point in favour of using LARD was the existence of an AMULET3 LARD behavioural model.



```
use(io).
use(channels).
C_chan : var(chan(int)).
A(C : var(chan(int))):expr(void)=(
    i : var(int).
    i := 0;
    forever(
        wait_for(50);
        i := i+1;
        C!i
    )
).
B(C : var(chan(int)))
:expr(void)=(
    forever(
        wait_for(30);
        C?(wait_for(100))
    )
).
init_chan(C_chan,"C_channel");
A(C_chan) | B(C_chan)
```

(a) LARD code

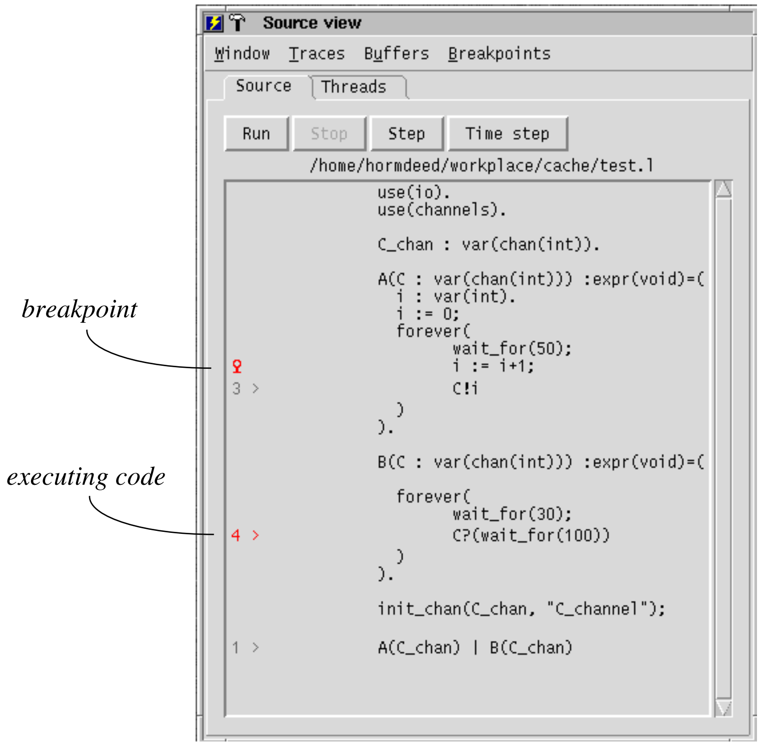


(b) Block diagram

(c) Screenshot of Channel time viewer windows

Figure 7.1: An example in LARD

An example of the use of Endecott's original LARD tool suite is shown in the upper box in figure 7.2. The output of the simulation can be either textual or graphical. The Channel Time Viewer, shown in figure 7.1c, is a graphical interface in LARD which displays the sender and receiver's activities and data values associated with channels allowing visual inspection of the causes of bottlenecks, conflicts and deadlocks as they occur in the simulation. Standard debugging facilities such as breakpoints and single-stepping are provided in Source View, the LARD source debugger, a screenshot of which can been seen in figure 7.1d.



```
/home/hormdeed/workplace/cache/test.l
          use(io).
          use(channels).

          C_chan : var(chan(int)).

          A(C : var(chan(int))) :expr(void)=(
            i : var(int).
            i := 0;
            forever(
                  wait_for(50);
                  i := i+1;
3 >               C!i
            )
          ).

          B(C : var(chan(int))) :expr(void)=(

            forever(
                  wait_for(30);
4 >               C?(wait_for(100))
            )
          ).

          init_chan(C_chan, "C_channel");

1 >       A(C_chan) | B(C_chan)
```

(d) Screenshot of Source view window

Figure 7.1: An example in LARD (cont.)

Around 80% of the cache model verification was performed using this LARD tool suite. With LARD debugging tools (source debugging and the channel activity display) bugs/errors (e.g. deadlock conditions) that are encountered can be traced and diagnosed. For example, a deadlock situation will cause the simulation to terminate with some indications of pending communications where no more progress was possible in both the Source View and the Channel Time Viewer.

The major drawback of this LARD tool suit is its low simulation speed especially when running in graphical mode, therefore running a very long program a number of times with varying configurations is unacceptably slow. Recently, however, Janin introduced a replacement LARD simulation framework, *Lard2C* [54], which compiles LARD code into C code. By simply running an executable file instead of interpreting the bytecode, the simulation is much faster than the original LARD, a significant advantage when many different cache configurations are to be analysed by simulation. Development of Lard2C is on-going having moved from 10x to 60x performance improvement over the original LARD in the time taken to write this thesis. However, without any graphical or debugging tools, lard2C is only useful for running models in textual mode once they are functioning correctly. Lard2C hooks into the design flow as shown in the bottom box in figure 7.2.

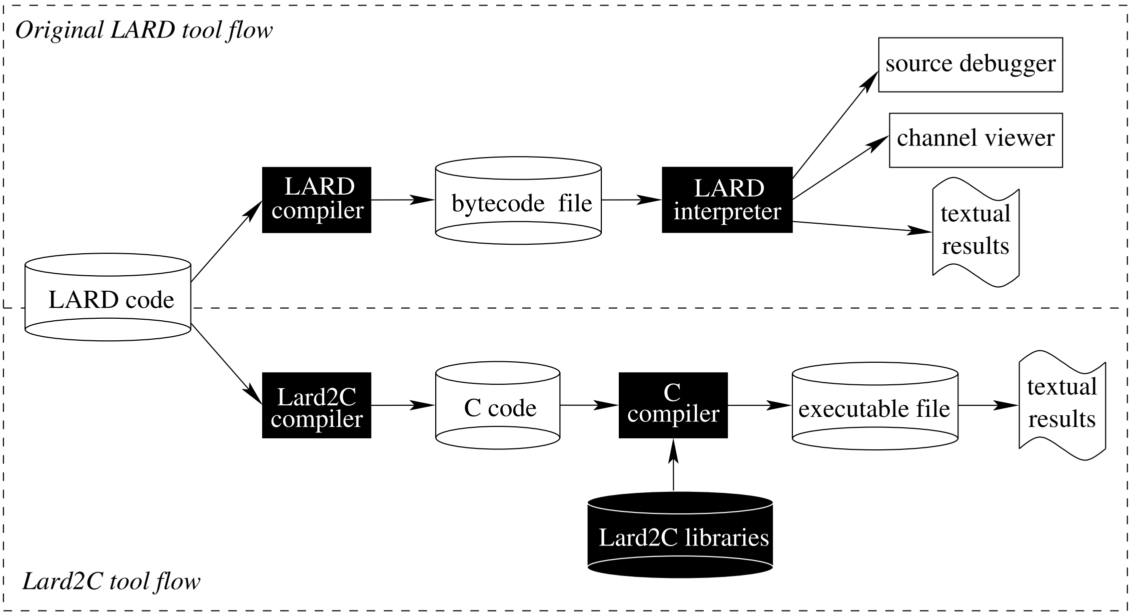


Figure 7.2: The simulation process in LARD

7.4 Benchmark programs

Given the limited execution speed of the simulator even when using Lard2C, large benchmarks such as the current SPEC suites [96] were not suitable. Instead the following nine benchmark programs were used:

- **Dhrystone 2.1** [103]: a short, synthetic C benchmark program intended to be representative of processor/compiler integer performance. Ten loops of the benchmark itself are executed.

- **Espresso** [95]: a logic minimisation program that takes as input a two-level representation of a two-valued (or a multiple-valued) Boolean function, and produces a minimal equivalent representation.

- **ST compiler** [86]: a freely-distributable C compiler released by Sozobon Ltd. The program here compiles the small 'subs_c.c' program.

- **Sim** [89]: an integer program that finds the *k* best non-intersecting alignments between two sequences or within one sequence. Using dynamic programming techniques, SIM is guaranteed to find optimal alignments.

- **Da** [40]: an integer program which uses the 'heap sort' method of sorting a random array of long integers up to 2MBytes in size. Each type of heap (Binary heap, Fibonacci heap and 2-3 heap) has been implemented using Dijkstra's algorithm[1]. The number of samples used in this work is 100.

- **DES**: a fast and portable DES (Data Encryption Standard) encryption and decryption program written by How.

- **Blackjack** [13]: a program to evaluate playing and betting strategies in the game of Blackjack. A single deck is used in this work instead of the default number, 4.

- **Whetstone 1.2** [17]: a C converted Whetstone double-precision benchmark, the first major synthetic benchmark program, intended to be representative of numerical (floating-point intensive) programs. One million Whetstone instructions are executed in each major loop of which 10 are usually run, but only a single run is used in this evaluation.

---

[1] an efficient algorithm for finding shortest paths in graphs

- **MM** [70]: a collection of nine different algorithms for doing matrix multiplication. The standard size of the matrix is (500×500), however due to the simulation time required a smaller array of (24×24) is used in this work.

These programs are all written in the C programming language. They were compiled into ARM code to use as trace file input to the model as described in section 7.5. The reasons why these programs were chosen were:

- they are fairly simple programs that can easily be compiled with existing ARM libraries;
- they are small enough to be representative of embedded applications (and not too big for use in LARD simulations);
- they illustrate a range of cache behaviours and miss rates.

Figure 7.3 shows a breakdown by access type of the cacheable memory accesses, showing how the number of cache misses due to writes varies dramatically depending on the benchmark used.

Though it may not meet all these criteria, the Dhrystone program was also used in this work because, with its small size, it could be used to test the model swiftly. Apart from Dhrystone, all of the benchmarks shown perform accesses to locations that must not be cached as they correspond to memory-mapped I/O ports. These accesses are thus excluded from the results shown here. The number of such accesses is also indicated in figure 7.3.

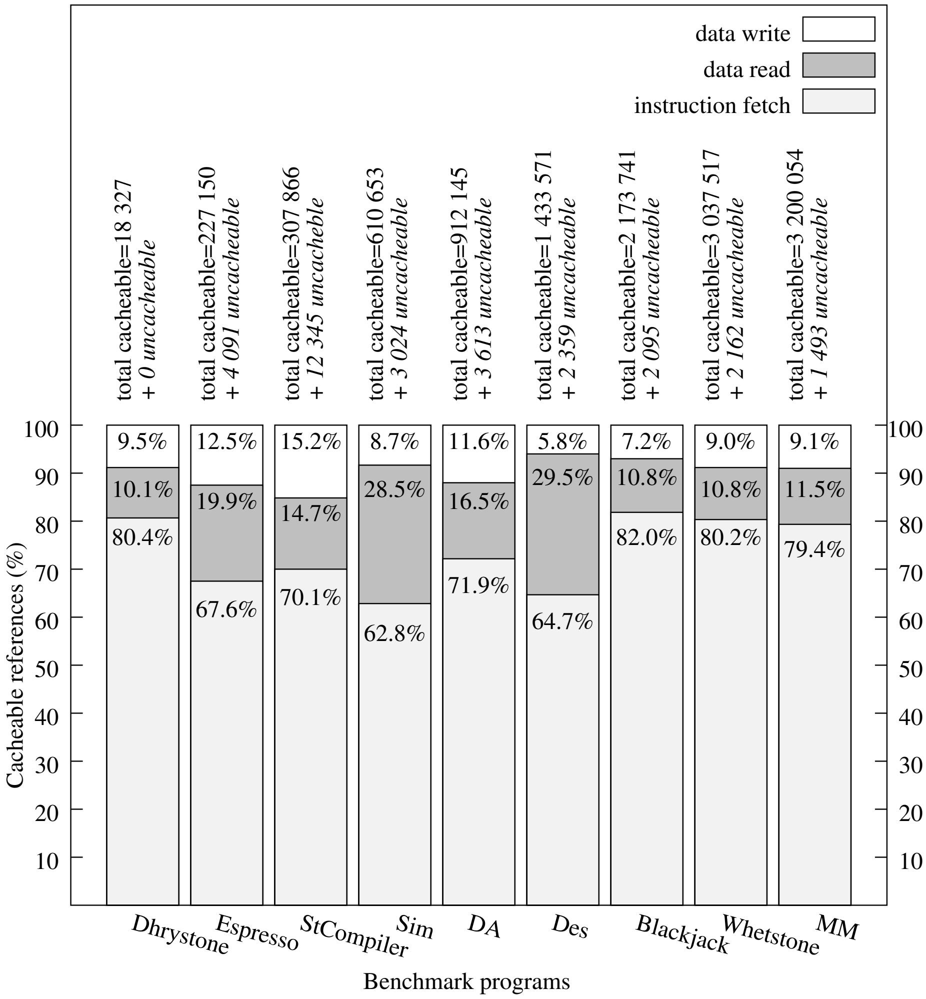


Figure 7.3: Benchmark memory access details

7.5 Simulation flow

The simulation flow used for the analysis of the techniques and architecture presented in this thesis is shown in figure 7.4. There are multiple stages to the simulation process:

- **Step 1**: The benchmark C source code is compiled using version 3.0.2 of the `GNU` C compiler and executed on a Sun Ultra 5 workstation with a 333MHz Ultra-SPARC IIi processor, its output being captured to file.

- **Step 2**: The benchmark C source code is compiled using the ARM development toolkit version 2.11 with the ARM ‘Demon’ library and executed on ARMulator, an ARM emulator. The captured output is checked against the output of step `1` to check that the cross-compiled binary functions correctly. A run-time disassembly of executed instructions is also captured.

- **Step 3**: The ARM binary from step `2` is used as input for a LARD model (in the dashed box in figure 7.4) containing the AMULET3 core and memory system as were assembled for modelling the AMULET3i system, i.e. not containing a cache. The behaviour was validated against the output and disassembly from step `2`.
- **Step 4**: The ARM binary from step `2` is used as input to a LARD model of the AMULET3 core and the proposed cache system, as shown in the shaded box in figure 7.4. Correct behaviour is validated against step `3` with logs of instruction and data accesses from the AMULET3 core in step `3` used to aid debugging where necessary.

For each benchmark, step `4` was repeated many times for differing cache configurations for both the write-through and copy-back variants of the cache. The main simulation path is shown by the grey curve in figure 7.4.

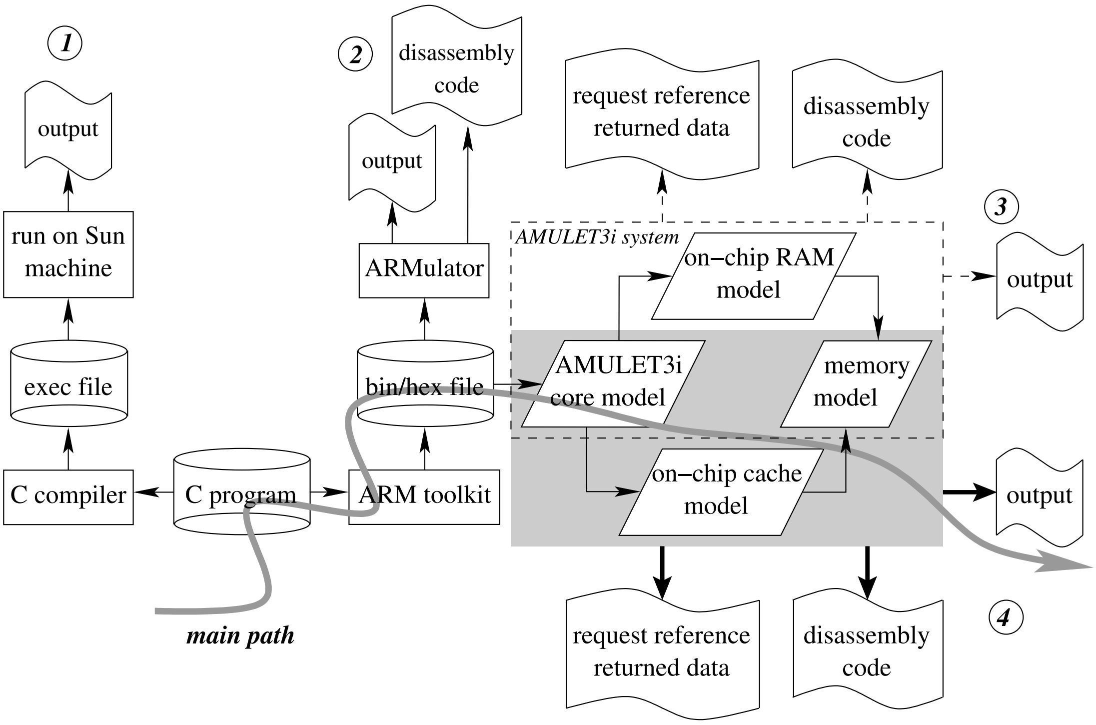


Figure 7.4: Cache model simulation process

7.6 Simulation base-level parameters

The initial cache parameters used as the starting point for the simulations were based upon reasonable assumptions and are summarised in table 7.1. Simulation runs investigate the effects of changing various parameters individually and in groups.

parameter	value
cache size	8 kilobytes
number of cache blocks	8
cache line size	4 words (16 bytes)
associativity	64-way (fully-associative in each cache block)
replacement strategy	random
write policy	copy-back with write-allocate
dirty bits	1 per line
line-buffers	2 per block (1 for instruction and 1 for data)
line-fetch latches	1 per block
write buffers	2 lines per block
outstanding memory accesses	1
memory random access time	100 ns
memory sequential access time	50 ns
cache access time	10 ns

Table 7.1: Base-level cache parameters

7.7 Simulation parameter variations

Even with the Lard2C simulation method that became available toward the end of this work, some of the cache simulations can take upto 6 hours of CPU time on a 1.4 GHz AMD Athlon machine or around 2.5 times longer on a Sun Blade 100 with an Ultra-SPARC IIe processor. Simulating all cache metric combinations for all 9 benchmarks is obviously impractical. A number of selective groups of simulations were therefore performed to understand how the underlying cache operation influences the performance of the system and the effects of changing the cache parameters.

Table 7.2 lists the simulation parameter variations used in these experiments. The `SIZE` groups of simulations, the only group that varies the cache size, varies the number of 1 kilobyte cache blocks from 1 to 32, also changing the total cache size and the total number of line-buffers and LFLs. The `BLOCK` group divides a fixed-size (8kilobyte) cache into various numbers of blocks (from 1 to 32). This changes the number of line-buffers and LFLs and also changes the set associativity whilst holding the cache size constant. The `ASSOC` group varies the associativity in each cache block from direct-mapped to fully associative (64-way). The `LINE` group varies the size of the cache line from 1 to 16 words. This also affects the size of the line-buffer and the LFL.

Various groups of simulations were performed to explore the effects of the line-buffer: the `LB` group varies the number of lines held in the line-buffer from 1 to 3, the `WTnoLB` group models write-through caches without line-buffers. The `CPparLB` groups model a copy-back cache with parallel line-buffer tag and main CAM look up.

Four `BURSTING` groups of simulations: `WTnoBURST`, `WTwithBURST`, `CBnoBURST` and `CBwithBURST`, were performed with the cache to main memory delay ratio varied from 2 to 20 to investigate the effects of write-through vs copy-back with and without memory bursting.

Finally, the `DPW` group uses a dirty bit per word (potentially) to reduce the number of writes performed.

<table>
<tr><th>groups of simulations
parameter</th><th>SIZE</th><th>BLOCK</th><th>LINE</th><th>ASSOC_RAN</th><th>ASSOC_LRU</th><th>ASSOC_CYCLIC</th><th>MEM</th><th>LB</th><th>WTnoLB</th><th>WTnoBURST</th><th>WTwithBURST</th><th>CBwithBURST</th><th>CBnoBURST</th><th>CBparLB</th><th>DPW</th><th>FW</th></tr>
<tr><td>cache size</td><td>1-32</td><td colspan="15">8 kilobytes</td></tr>
<tr><td>cache blocks</td><td>1-32</td><td>1-32</td><td colspan="14">8 cache blocks</td></tr>
<tr><td>cache line size</td><td colspan="2">4 words</td><td>1-16</td><td colspan="13">4 words</td></tr>
<tr><td>cache lines</td><td>64</td><td>#[a]</td><td>#[b]</td><td colspan="13">64 cache lines in each block</td></tr>
<tr><td>associativity</td><td colspan="3">fully each block</td><td colspan="3">1-64 way</td><td colspan="10">fully each block(64-way)</td></tr>
<tr><td>replacement strategy</td><td colspan="4">random</td><td>LRU</td><td>cyclic</td><td colspan="10">random</td></tr>
<tr><td>write policy</td><td colspan="8">copy-back with write-allocate</td><td colspan="3">write-through</td><td colspan="5">copy-back with write-allocate</td></tr>
<tr><td>dirty bits per line</td><td colspan="14">1 dirty bit per cache line</td><td>4 bits</td><td>1 dirty bit</td></tr>
<tr><td>line-buffers</td><td colspan="7">1 line-buffer entry on each port</td><td>1-3</td><td>none</td><td colspan="7">1 line-buffer entry on each instruction and data port</td></tr>
<tr><td>tag comparison</td><td colspan="13">sequentially line-buffer tag and CAM comparison</td><td>parallel</td><td colspan="2">sequential</td></tr>
<tr><td>memory accesses</td><td colspan="6">1 outstanding memory access</td><td>1-5</td><td colspan="9">1 outstanding memory access</td></tr>
<tr><td>$T_{\text{mem random access}}$</td><td colspan="9">100 ns.</td><td colspan="4">20-200 ns.</td><td colspan="3">100 ns.</td></tr>
<tr><td>$T_{\text{mem sequential access}}$</td><td colspan="9">50 ns.</td><td>20-200 ns.</td><td colspan="2">10-100 ns.</td><td>20-200 ns.</td><td colspan="3">50 ns.</td></tr>
<tr><td>write buffer</td><td colspan="15">local write buffer/victim cache scheme (2 lines for each cache block)</td><td>1-5 lines for each block</td></tr>
</table>

Table 7.2: Cache simulation parameter variations

[a] Since they can be calculated from the block size and the cache line size, the number of cache lines in each block varies depending on the changing block sizes.

[b] Similarly, they vary depending on the changing cache line sizes.

experiment	**VC_CentralAssoc_1**	**VC_CentralAssoc_8**	**VC_CentralAssoc_64**	**VC_Extra_Direct**	**VC_Extra_Fully**	**VC_LocalAssoc_1**	**VC_LocalAssoc_8**	**VC_LocalAssoc_64**
cache size	8 kilobytes			8 kilobytes+(1-5 line in each block)		8 kilobytes		
cache lines	64 lines			#[a]		64 lines		
associativity	1	8	64	direct-mapped	65-69[b]	1	8	64
victim cache	central (8-40 lines)			none	none	local (1-5 lines for each block)		

Table 7.3: Cache simulation parameter variation for victim cache experiments

a. The number of cache lines depends on the total cache size. In this case there are 65-69 lines in each cache block.

b. Fully-associative in each cache block

Similarly, table 7.3 summarises the simulation parameter variations used to analyse the victim cache. The details of each victim cache simulation are as follows:

The `VC_Assoc` groups are used to explore the associativity with either a direct-mapped (`Assoc_1`), an 8-way associative (`Assoc_8`) or a 64-way associative (`Assoc_64`) cache system, for both central (`Central)` and distributed (`Local`) victim caches. The `Central` group was performed with a central victim cache of 8, 16, 24, 32 or 40 entries. The `Local` group was performed with the total size of distributed victim cache as 8, 16, 24, 32 or 40 lines, which is equivalent to 1 to 5 entries in each local victim cache.

The two `VC_Extra` groups are for cache systems with extra storage of 8, 16, 24, 32 or 40 lines distributed evenly between the cache blocks making each cache block slightly bigger. Whilst the `VC_Extra_Direct` group is with a direct-mapped cache system, the `VC_Extra_Fully` group uses full associativity in each cache block.

Important observations from the results obtained from the simulations listed above are described in chapter 8.

7.8 Summary

Trace simulation is an effective method for evaluating the behaviour of a memory hierarchy in a synchronous environment, but unfortunately is not suitable for use with an asynchronous system. Probability distribution-driven simulation is unsuitable for the same reasons and no processor was available to drive an execution-driven simulation therefore an application driven simulation was performed using an AMULET3 model.

A plan of suitable parameter values for a meaningful series of simulations has been presented. These simulations were executed on a LARD model of both the cache and the AMULET3 processor running ‘typical’ benchmark programs; the result are presented in the next chapter.

Chapter 8: Results and Evaluation

This thesis has presented a dual-ported asynchronous block-based copy-back cache architecture. Theoretical analysis of an asynchronous cache is practically impossible because of the many different delays in the system. Even calibrating a model of such a system is difficult since one has to choose the level of granularity at which to model and calibrate the system. However, whilst this makes it hard to give realistic performance estimates, a model is adequate for the analysis of different architectures as presented in this chapter.

There are two aspects to the performance of a cache memory: access time and miss rate. Both of these metrics are difficult to use; the exact effect of design changes on access time is hard to specify without involving a circuit technology, and translating the miss rate into a measurement of speed is tricky and depends on implementation details. This should be borne in mind when examining the experimental cache results presented in this thesis (and in other model based cache studies).

Throughout this chapter, miss rates are given in terms of both read requests (data loads and instruction fetches) and write requests (data stores) without including uncacheable requests. Combining read and write requests can lead to confusing results since the mechanisms by which reads and writes affect the overall performance are quite different and the boundaries of both mechanisms are not obvious. For example, one major feature evaluated here is for a copy-back write policy with write-allocation which would affect both read and write miss rates.

The results presented in this chapter, derived from the simulations described in section 7.7, are subdivided into three categories, each addressing a different issue. First, section 8.1 evaluates the basic cache features. These features include the cache size, sub-blocking, write policies, forwarding, set associativity and replacement strategies. Then section 8.2 describes asynchronous issues in designing a cache system including observations on the distribution of cache hit locations, delay characteristics and the asynchronous line-buffer (a L0 cache). Lastly, the behaviour of the victim cache is analysed in section 8.3.

8.1 Evaluation of cache features

In order to design a cache, several architectural decisions must be made. There are also a number of adjustable parameters including the dimensions of the cache (width, depth and size), organisation of the cache, write behaviour and the method of interaction between different memory levels. The evaluation begins with an analysis of the effects of varying a few of the basic cache parameters. These results are similar to those that would be expected from a synchronous cache system of similar architecture [41].

8.1.1 Cache size and sub-blocking

The parameter most fundamental to the performance of a cache is its size. This is not the most essential issue in this work since the cache size is heavily dependent on the application, especially in an embedded system. However, it still remains one of the first parameters that must be fixed in any cache design. Using a block-based architecture allows the cache size to be varied in a number of ways including changing the block size or changing the number of blocks of a fixed size. Figure 8.1 (obtained using simulation set `SIZE`) shows the effect on the miss rate of varying the cache size by changing the number of 1 kilobyte blocks.

The slope of each line in figure 8.1 depends on the individual characteristics of each benchmark and, as expected, larger caches provide lower *capacity* miss rates, up to a point. ST Compiler confirms this expectation, although it does not reach this point here. The program has the largest binary in the benchmark suite used here, requiring a large cache (8 kilobytes) to obtain a reasonable hit rate (>93%). The reason for the low hit rates, particularly in the data stream, is a very large binary and data set resulting in very high compulsory misses. Blackjack's anomalous results are because it thrashes the cache for sizes below 4 kilobytes.

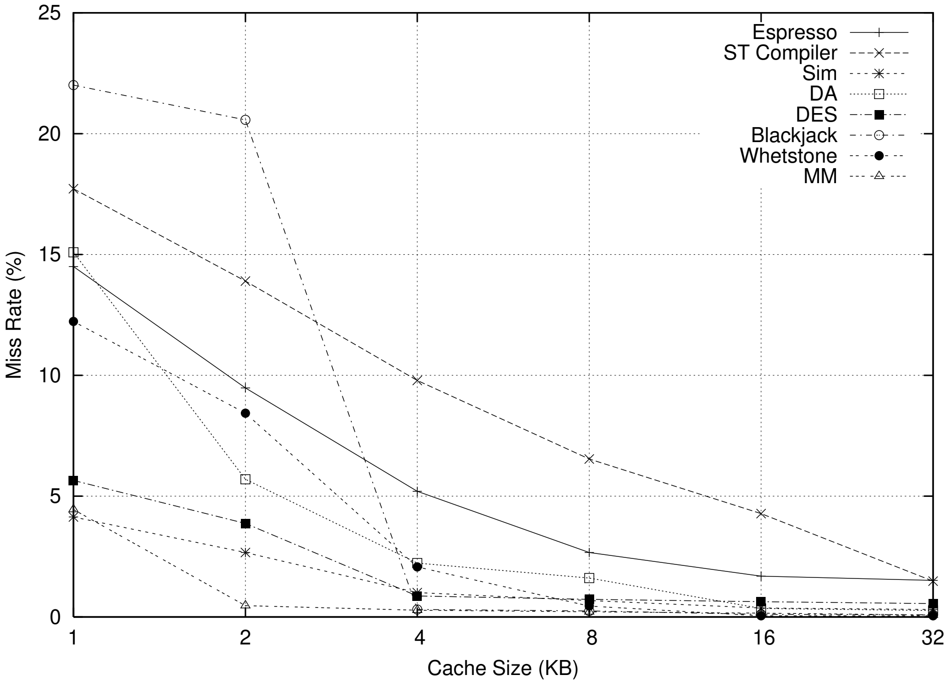


Figure 8.1: Effects of cache size on miss rate

However, the majority of these benchmarks show that going beyond 8 or 16 kilobytes gives diminishing returns. Therefore, for the subsequent simulations the cache size has been fixed at 8 kilobytes, a reasonable size for embedded systems.

Figure 8.2, obtained from the same set of simulations, shows how the normalised overall run time improves with a larger cache. This is not merely because of the lower miss rate of the bigger cache, but also because of the structure of the multiple cache blocks where more blocks means more (fast) dual line-buffers, more LFLs, and reduced conflict between instruction and data accesses. Blackjack's behaviour changes dramatically from 2 to 4 kilobytes for reasons described later in section 8.2.1.

All of these benchmarks give better performance as illustrated in figure 8.3, obtained from set `BLOCK`, which shows the effects on the average system access time of varying the number of blocks whilst keeping the cache size constant (8 kilobytes). The greatest improvement is provided in programs with the highest proportion of data accesses which allow greater parallelism.

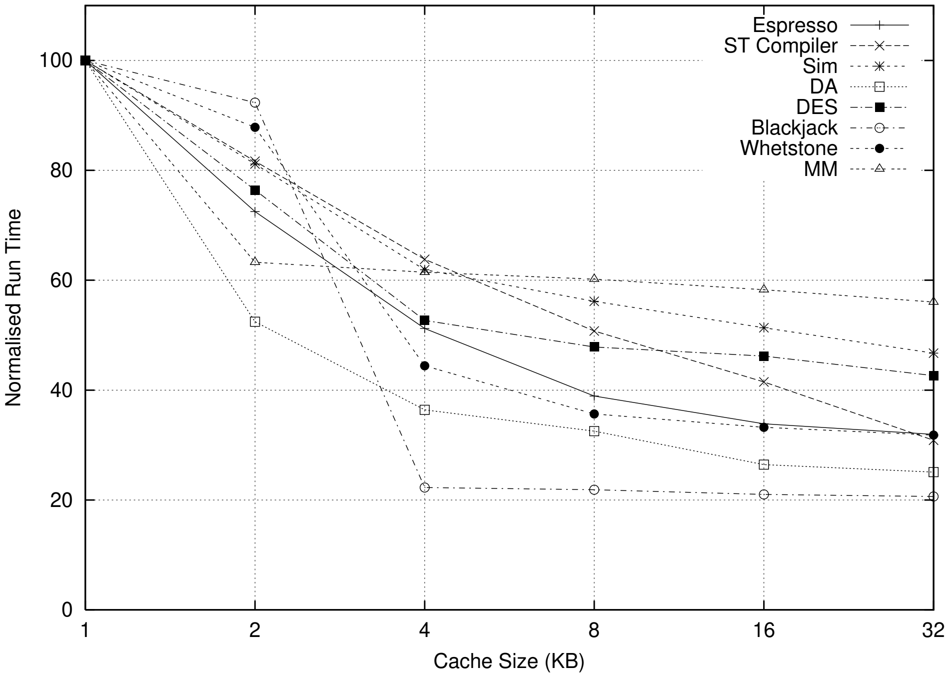


Figure 8.2: Effects of cache size on run time

Sub-blocking into more blocks also gives a reduction in power consumption since more accesses can then be satisfied by the line-buffers, avoiding full cache accesses and, on a line-buffer miss, a smaller cache block is active.

8.1.2 Cache line size

Increasing the cache line size means increasing the number of words fetched on a cache miss and, with this architecture, longer cache lines result in larger line-buffers and LFLs. The number of hits in these locations increases because of sequential access patterns, giving the reductions in miss rate shown in figure 8.4 (obtained from set `LINE`).

The advantages of small line size are:

- reduced stall duration on a miss that occurs whilst a previous miss is still being fetched;
- lower probability of the line containing unnecessary data. Only a few extra bytes are transferred along with the actually required data, reducing the overhead costs of fetching useful data.

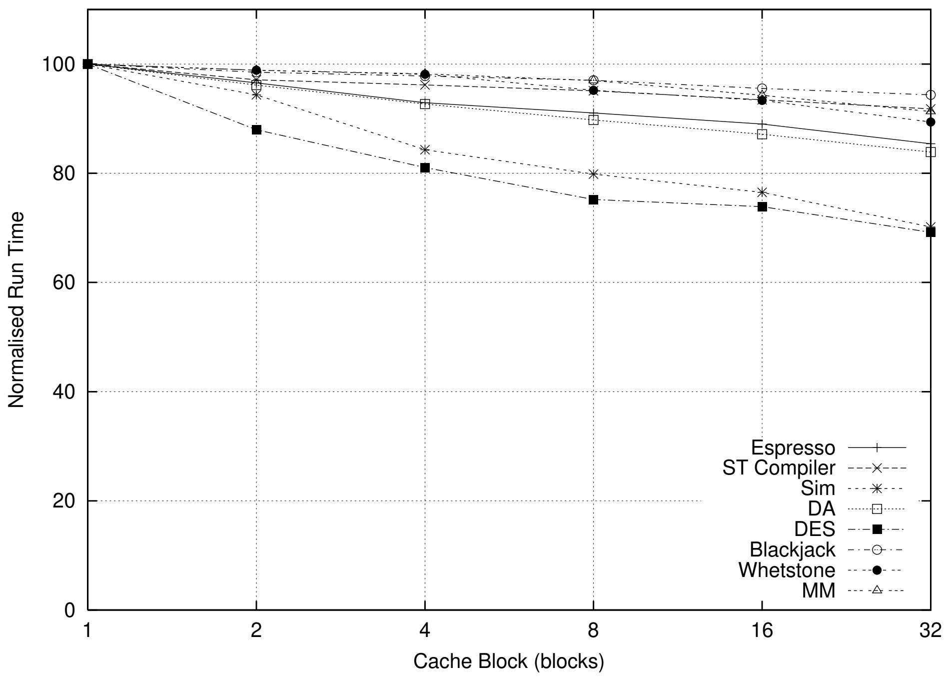


Figure 8.3: Effects of cache sub-blocking on run time

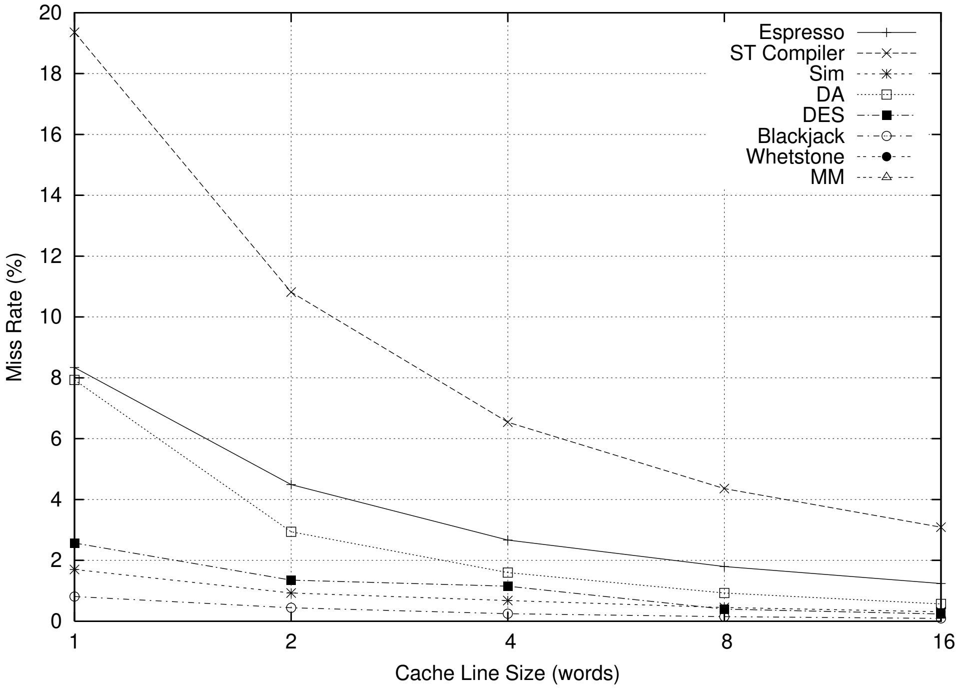


Figure 8.4: Effects of cache line size on miss rate

However, large lines also have advantages in that they:

- are more effective if most of the data in a line is actually used;
- require fewer lines for the same cache size, hence the tag comparison logic is smaller and fewer storage bits are needed to hold the tag and replacement status.

The line size also affects the miss rate. However, judging the best line size from the miss rate alone gives an incomplete view since this also depends on the behaviour of each benchmark.The relationship between line size and performance is complex and no definitive optimum value has been found [98].

All of the above discussion about line sizes is generally true for any cache system. Further detailed analysis of the effect of varying the cache line size can be found elsewhere [91]. Typically, the effects of varying the line size on the normalised run time would be expected to give a ‘V-shaped’ graph turning at the optimal line size, above which the increase in miss cost is greater than the reduction in miss rate. In these simulations, this is clearly visible in the top two lines (from the ST Compiler and Espresso benchmarks) of figure 8.5 which is obtained from set `LINE`. However, most of the benchmarks do continue to show a very small benefit from enlarging the line size beyond 4 words. This is because those programs can still benefit from a ‘long’ dual-port line-buffer in this architecture. Overall, the results show that the most cost-effective line size is 4 words as the benefits of longer lines are small at best and the cost is not insignificant.

8.1.3 Set associativity and replacement strategy

In the `ASSOC` simulation sets, the cache associativity is varied from direct-mapped to fully associative within each block. Three replacement strategies were tested – LRU, cyclic and random – but there was very little difference in their performance as shown in figure 8.6 (LRU was the best and random the worst overall, but by small margins). Whilst the random and the cyclic replacement algorithms are fairly easy to implement, the LRU strategy can be expensive, especially for large caches and/or high degrees of associativity. Typically the cyclic scheme would be expected to perform noticeably worse than the others. However, here the cyclic strategy is proving to be quite competitive. This is simply

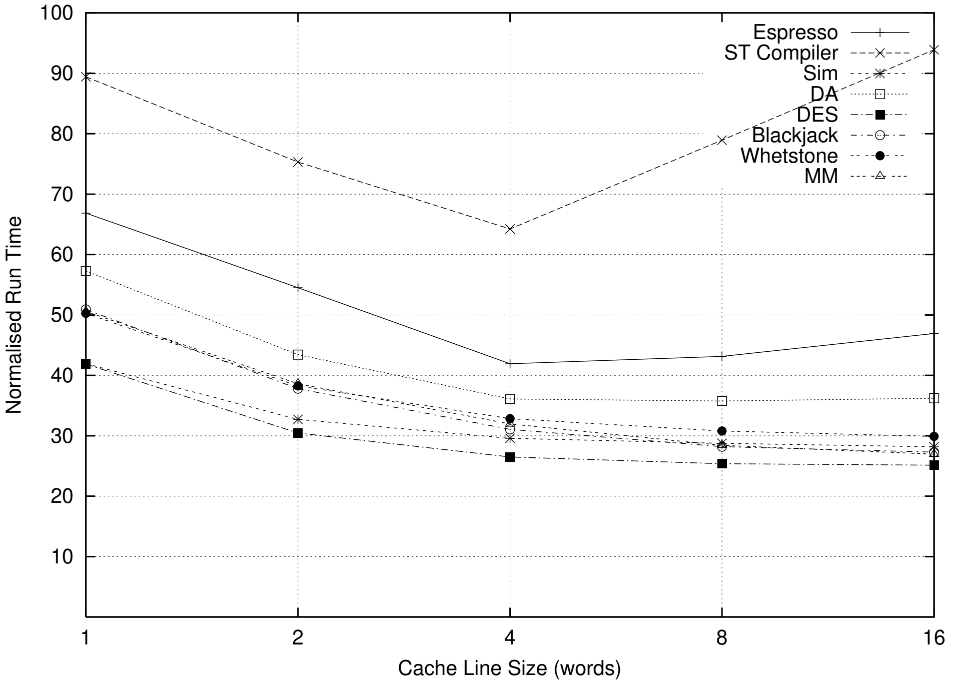
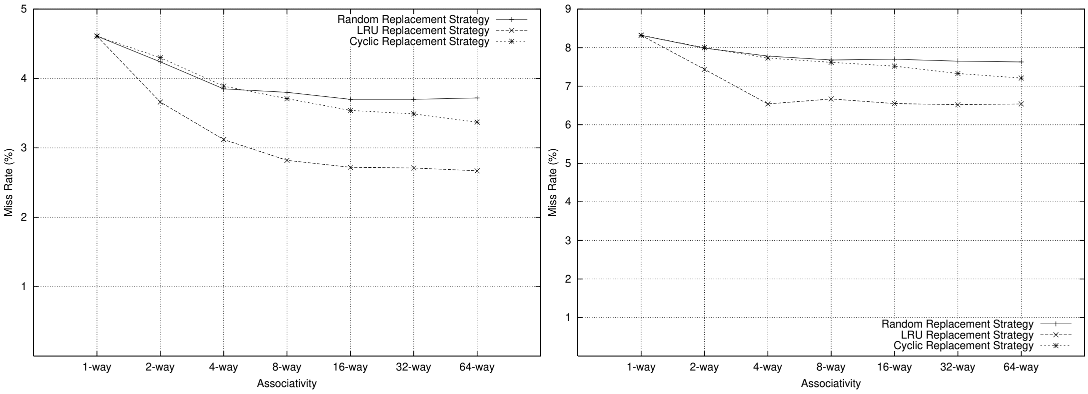


Figure 8.5: Effects of cache line size on run time

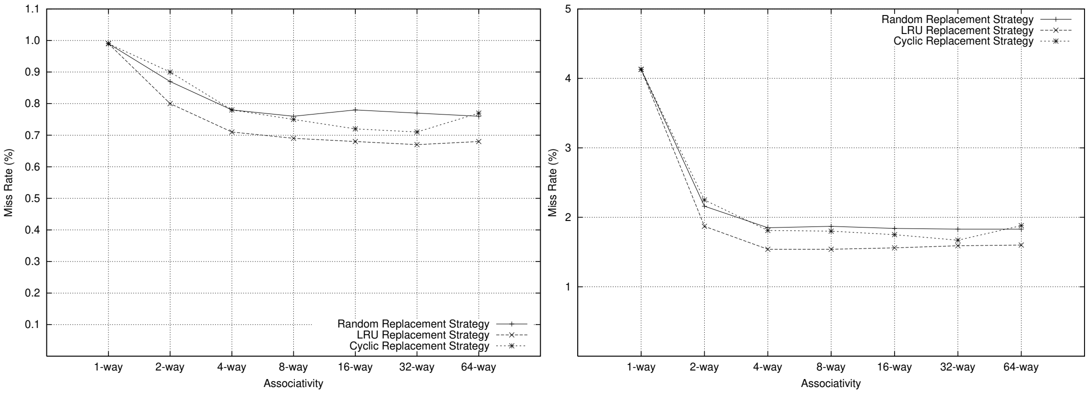
because, with the chosen parameters, none of these programs coincides with the pathological case for the cyclic replacement. But in order to avoid this possibility, the random replacement strategy remains the favourite choice here.

Although the graph in figure 8.7, obtained from set `ASSOC_RAN`, confirms that going beyond 4- or 8-way associativity [41] gives negligible benefit in terms of miss rate, full associativity (within each cache block) is still favourable on two counts: with careful design it can be more power-efficient than a 4-way set-associative cache, and it provides better support for *lock-down* mechanisms. This is because locking-down cache lines causes a more noticeable degradation in performance for low-associativity cache blocks.



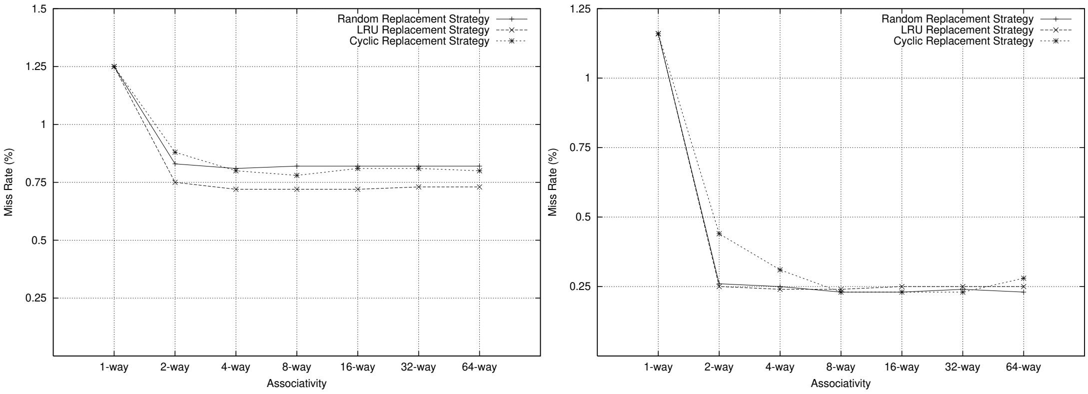
(a) Espresso

(b) ST Compiler



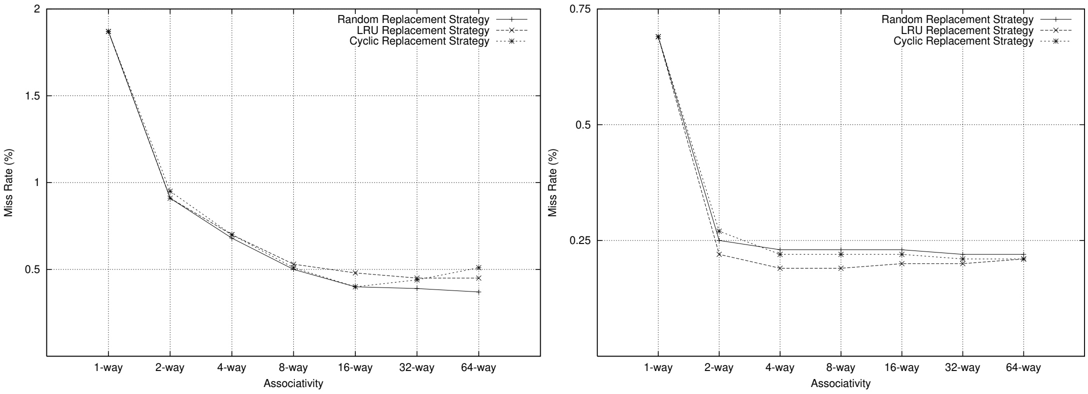
(c) Sim

(d) DA



(e) DES

(f) Blackjack



(g) Whetstone

(h) MM

Figure 8.6: Replacement strategy vs associativity

NB. vertical axis scales not identical

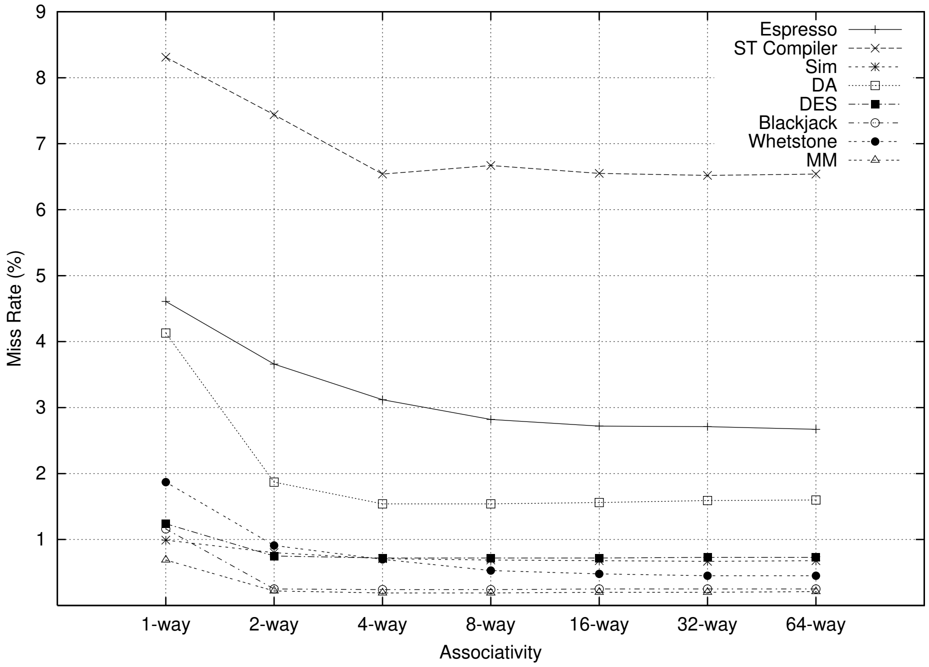


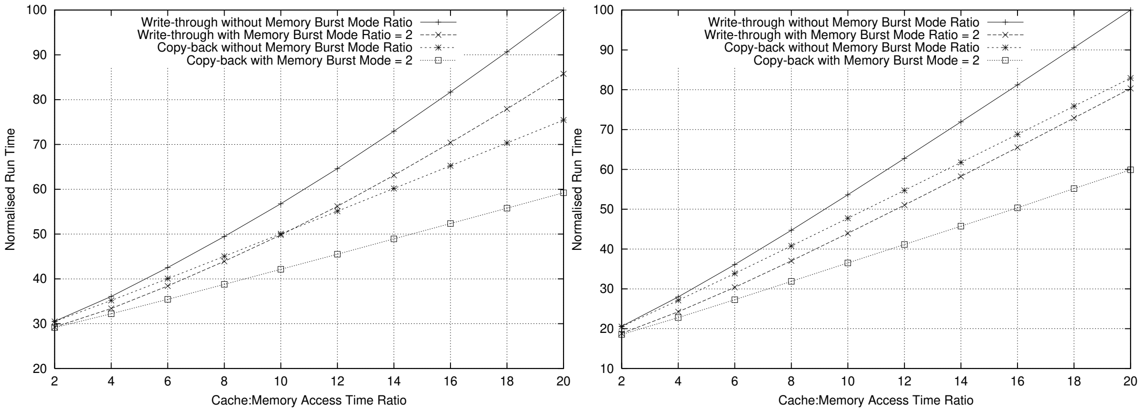
Figure 8.7: Effects of associativity on miss rate

8.1.4 Memory burst-mode access

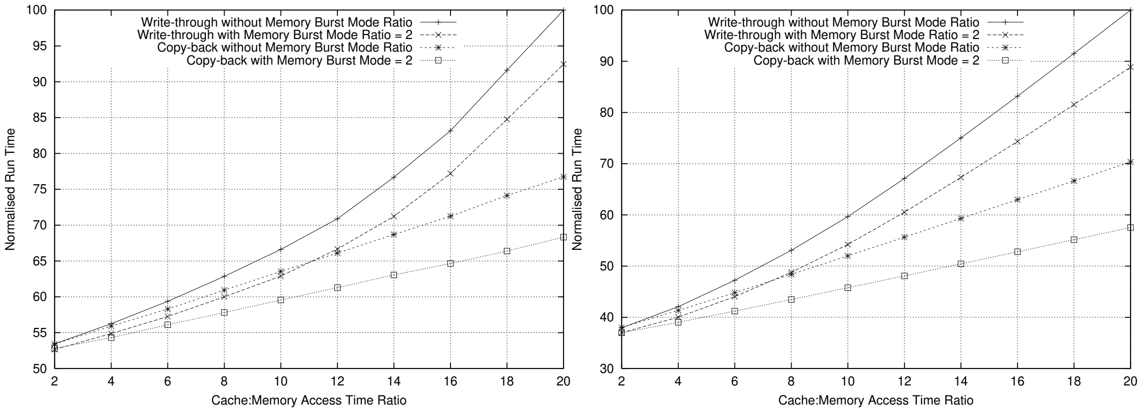
Most memories support a burst-mode access where the first access in a consecutive sequence is slow but subsequent accesses in the sequence can occur much faster. Figure 8.8 illustrates the benefits of supporting burst-mode when performing consecutive memory accesses with both write-through (obtained from sets `WTnoBURST` and `WTwithBURST`) and copy-back (obtained from sets `CBnoBURST` and `CBwithBURST`) caching styles. Unfortunately, the write-through caches can only benefit from burst-mode access for line fetches. In contrast, the copy-back caches can benefit both for line fetches and for writing dirty lines back to the main memory. This further widens the performance gap between write-through and copy-back as is apparent in section 8.1.5.

For this reason, the copy-back scheme is the better choice, especially when it is used with burst-mode memory.

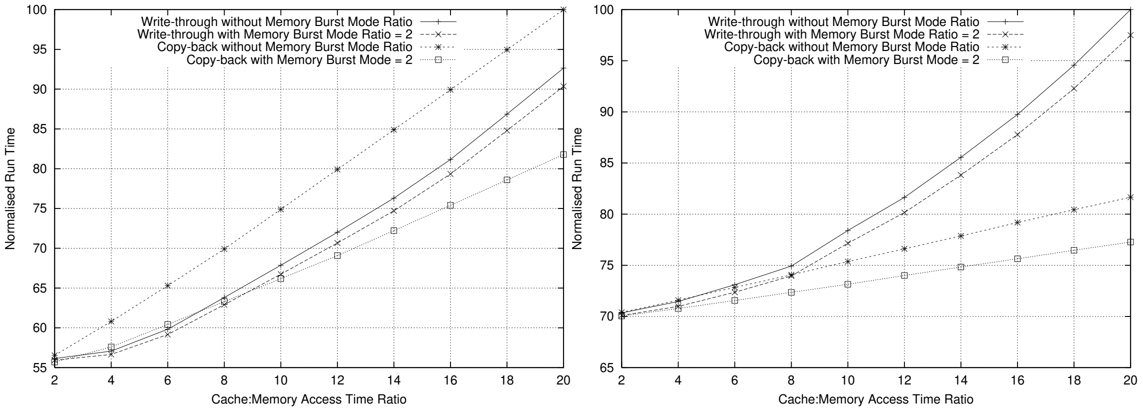
(a) Espresso (b) ST Compiler



(c) Sim (d) DA



(e) DES (f) Blackjack



(g) Whetstone (h) MM

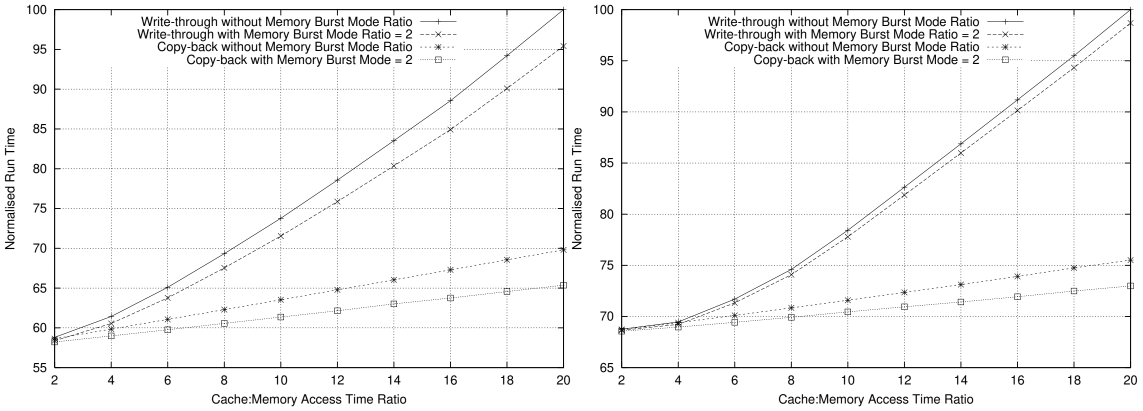


Figure 8.8: Effects of memory burst-mode access

NB. vertical axis scales not identical

8.1.5 Copy-back vs. write-through

Figure 8.9 illustrates the effect of the disparity between the external memory and the cache access times (averaged amongst the benchmarks). As this ratio increases the performance gap between the write-through and the copy-back cache widens. This is because the write-through cache sends each data write to the main memory so, as the memory becomes slower compared to the cache (and processor), its write bandwidth will eventually become saturated. The effect of write-through and copy-back caching on each individual benchmark program can also be seen in figure 8.8 which is used to illustrate the effect of memory bursting.

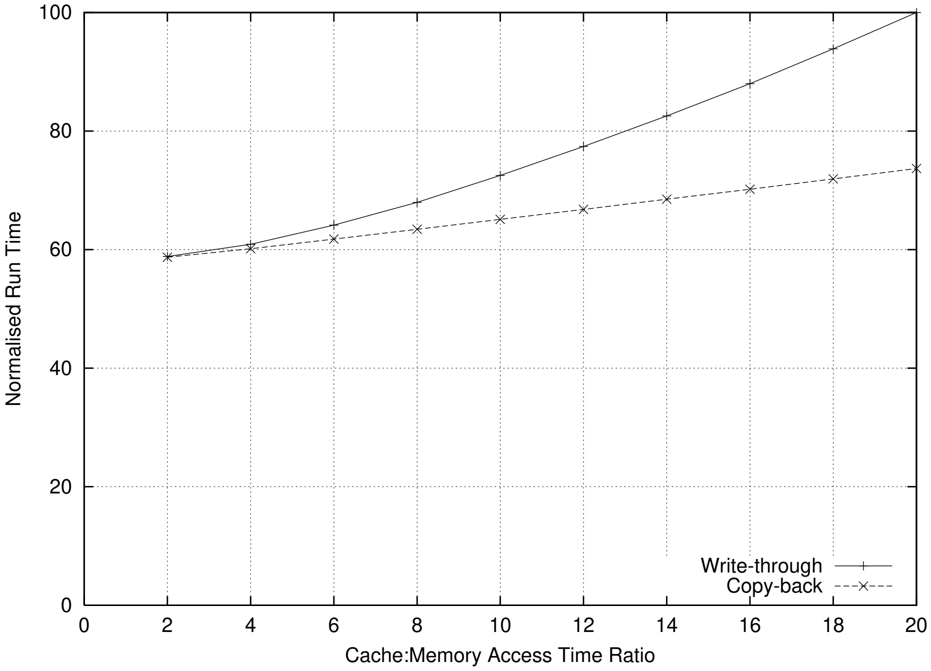


Figure 8.9: Write-through vs. copy-back

A write-allocate strategy assumes that in the near future the processor will access a line that has been recently written; normally a reasonable assumption. Applying write allocation in a copy-back cache also simplifies the victim-cache forwarding mechanism requiring less control logic to check whether the lines in the victim cache is a complete valid line (a write from the processor will not be a complete *line* of data).

Figure 8.10 (obtained from set `LINE`) shows the proportion of writes to already dirty lines with increasing cache line size. This obviously decreases write traffic in the copy-back cache since there are fewer write operations from the main cache; if a write affects an already dirty line this means that a memory write operation has been averted. All of these programs experience a 75% or greater reduction in write traffic through the use of the copy-back cache.

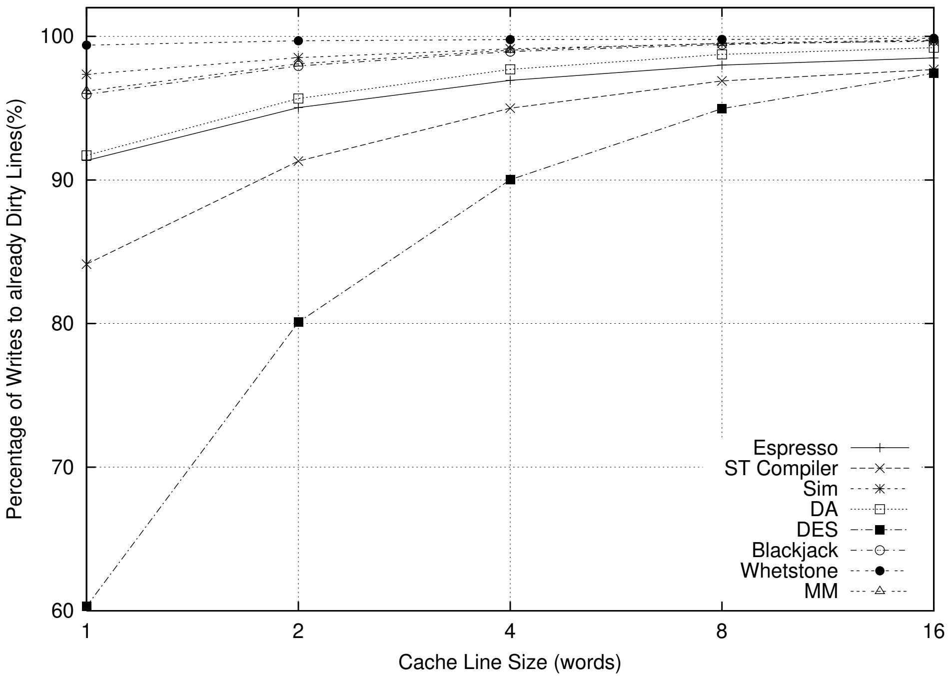


Figure 8.10: Proportion of writes to dirty lines

Simulation set `DPW` was used to investigate the dirtiness granularity of the dirty lines. Having a dirty bit per word means only the minimum necessary writes are performed, however they will likely be performed in non-sequential order.

Table 8.1 shows that over 80% of dirty evicted lines require the entire line to be written back to the memory. A single dirty bit per line therefore remains the choice for the architecture, and it also allows exploitation of memory burst-modes.

program	1 word	2 words	3 words	4 words
Espresso	15.95%	10.06%	16.07%	57.92%
STcompiler	13.33%	8.25%	4.32%	74.21%
Sim	10.75%	5.81%	4.09%	79.35%
Da	6.11%	4.17%	6.03%	83.68%
DES	0.15%	0.01%	0.16%	99.68%
Blackjack	2.71%	1.74%	2.11%	93.44%
Whetstone	5.08%	14.58%	21.19%	59.15%
MM	0.46%	0.33%	0.46%	98.75%
average	*6.82%*	*5.62%*	*6.80%*	*80.76%*

Table 8.1: Dirtiness of evicted dirty lines

As with conventional clocked caches [59], these results again confirm that a copy-back cache provides better overall performance due to the significant reduction in memory traffic. For the results presented here, the average breakdown of the access types is shown in table 8.2. Here only 9.9% of cacheable accesses were data writes and significant improvements with copy-back caching can already be seen. These improvements would be even greater if the mix contained a higher percentage of writes.

instruction fetches	data reads	data writes
73.2%	16.9%	9.9%

Table 8.2: Average cacheable memory access details

8.1.6 Write buffering and forwarding

Another issue in the comparison of copy-back and write-through caching is the size of the write buffers. Both write-through and copy-back caches gain significant performance from a write buffer, and a copy-back cache *needs* at least one buffer entry to hold an evicted line.

The required depth of the write buffer is directly related to the number of pending writes, which in turn depends on the clustering of write operations and the processor speed to memory speed ratio. It is likely to be fairly small; 2-4 write buffer entries were suggested in a study of a (synchronous) write-through cache [41]. Statistically a copy-back cache will produce less clustered write operations than a write-through cache, so it should not need as large a write buffer. In this architecture, the write buffer/victim cache is implemented as part of each cache block. The main reason for having it local to the block is to reduce wiring. Furthermore, arbitration can then occur in a non-critical path, at the interface between the write buffer and the external bus, rather than introducing delay between the cache and the write buffer.

With only a single line buffer in each block, forwarding from the write buffer cannot be applied. Moreover, a write buffer with more than one entry with a copy-back cache would be useful where multiple misses with dirty victim lines occur in series in each block, hence a dual-line write buffer which now acts as victim cache is used in each block.

Further increasing the size of the victim cache is likely to be beneficial. However, there is a trade-off (hardware resource vs. performance) here since more lines in the victim cache allow more data to be forwarded back to the cache, the main cost being in silicon area.

Although having the victim cache locally potentially provides significant advantages over having it centrally, the drawback is the larger minimal requirement on the victim cache size. For example, instead of only 2-entries – one for the evicted line and the other in case of serial line fetches and for forwarding purposes (50% of those lines are forwardable) – to perform the same functions with an 8-block cache system, sixteen entries are needed. Even though this gives an increase in silicon area, having a few more buffers is considered inexpensive for the forwarding benefits that can be gained here.

Figure 8.11 (obtained from set `FW`) illustrates the effects of varying the number of entries in the victim cache in each block on the selected benchmarks. These results were obtained after a *warm start*.

Further investigation of the victim cache is described later in section 8.3.

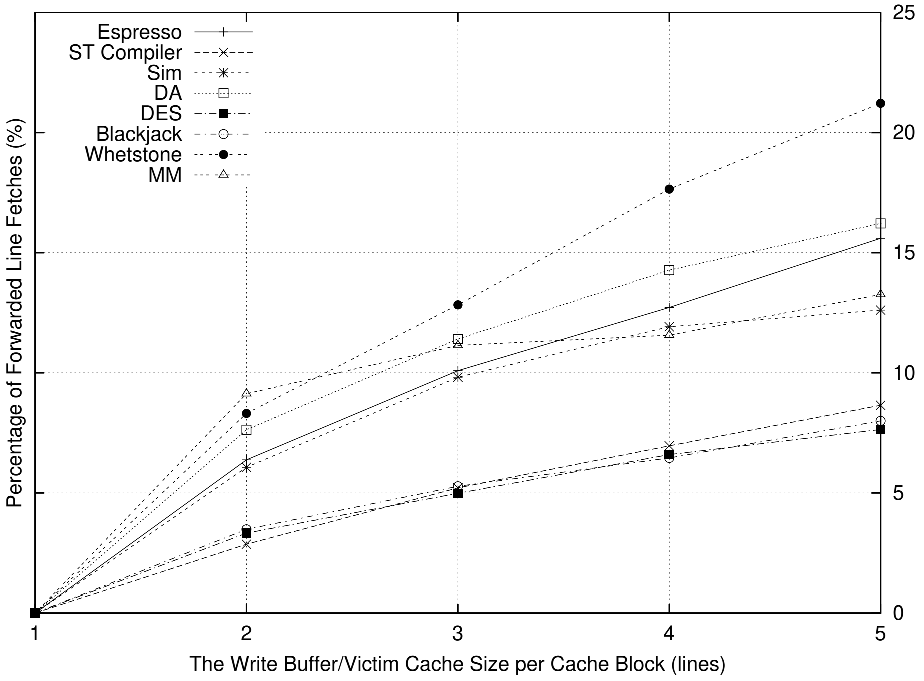


Figure 8.11: Effects of the victim cache size

8.1.7 Number of outstanding memory accesses

The cache architecture has been designed to allow its different stages to be pipelined so that there is the potential for multiple cache accesses to proceed in parallel. The occupancy of the pipeline is regulated by the number of outstanding memory transactions that the processor can perform and affects the performance, as investigated using the `MEM` simulation set.

Figure 8.12 illustrates how increasing the number of outstanding memory accesses (and hence the pipeline occupancy/parallelism) gives a reduction in the normalised run time. Increasing the number of outstanding memory accesses to two gives a significant reduction because both pipeline stages in this memory system can proceed concurrently. Increasing further, in some cases gives a slight performance improvement because parallelism between blocks is still allowed.

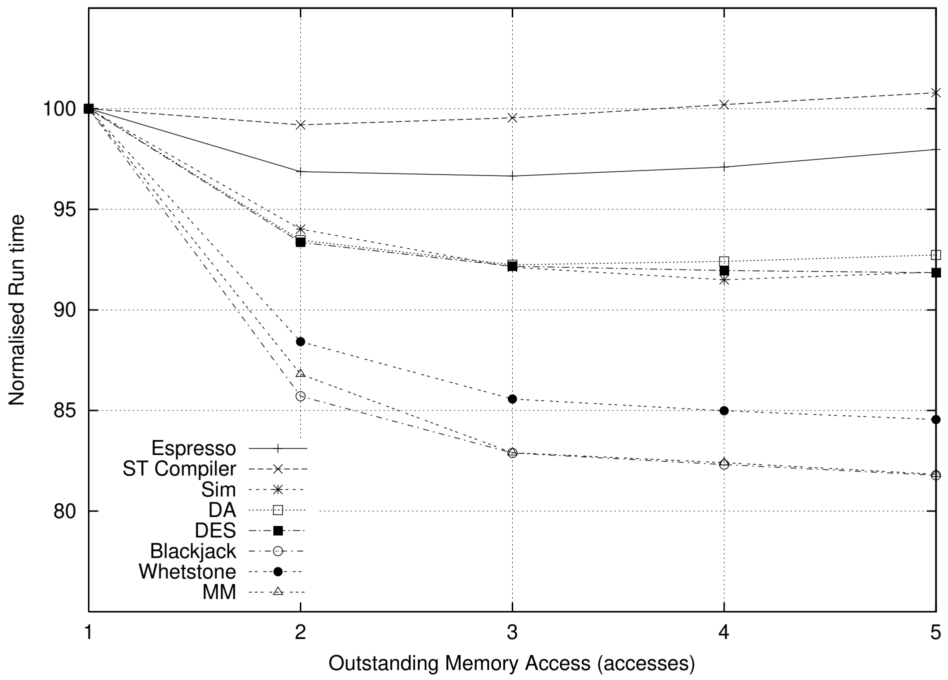


Figure 8.12: Effect of varying the number of outstanding memory accesses

8.2 Asynchronous issues

The previous section showed the feasibility of a dual-ported asynchronous block-based copy-back cache architecture, and explored the basic parameters of such a system to set the ‘ground level’ for further study of the unique aspects of the architecture and its asynchronous advantages. These are described below.

8.2.1 Distribution of cache hit locations

In this system, a cache hit could occur in the line-buffer, the main cache RAM or the LFL. Since these all have different speeds, the unit in which the hit occurs affects the access time.

Generally the distribution of hits is roughly 9:7:3 (LB:RAM:LFL) as can be seen in figure 8.13 (obtained from set `SIZE`). A few of the benchmarks deviate from this distribution with small caches; the Blackjack program is a good example, with very high LFL hit rates (over 50%) for caches smaller than 2 kilobytes. This is because a miss occurs causing a line fetch, and then the words in that line are accessed shortly thereafter (from the LFL). Because the cache is small, resulting in cache thrashing, the line only remains in the main cache RAM for a short time before being evicted and so it has little chance to get into the fast line-buffers. This also results in poor overall run time.

Interestingly, increasing the size of the cache (with the current architecture) can be counter-productive in some situations. This is illustrated in figure 8.13 by the (small) Dhrystone program which fits entirely in a few kilobytes of cache. A cache size of 4 kilobytes is sufficient to ensure a hit rate of nearly 99%. When line fetches effectively cease, the last data fetched is left in the LFLs, access to which requires a full CAM look-up. Because accessing the LFL requires passing through the arbiter between the instruction and data ports it results in an increased access time. However, Dhrystone, while quick to run, is a fairly poor benchmark for cache performance and although it does highlight a situation that could arise with any program small enough to fit entirely in the cache, the adverse effect is fortunately small. This problem could be cured by copying the LFL to the line-buffer but, unfortunately this would require extra synchronisation to check if the line fetch was complete.

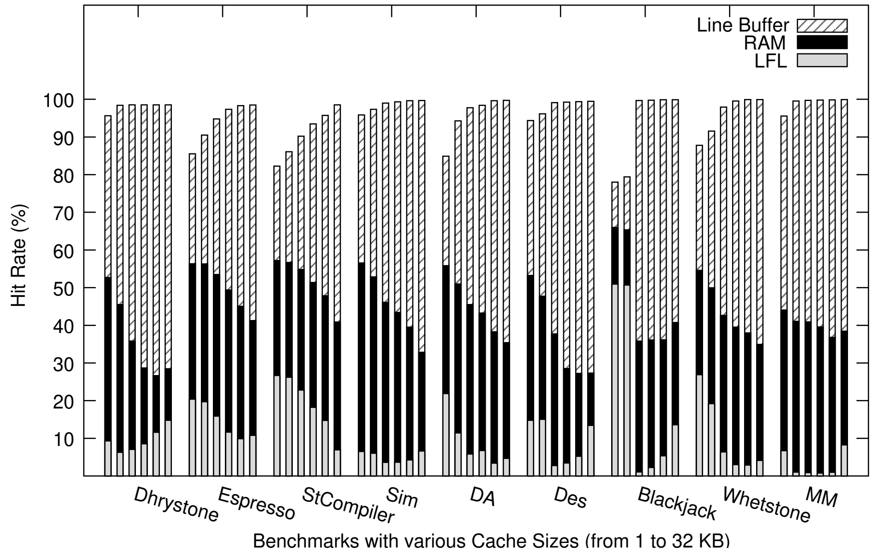


Figure 8.13: Effects of cache size on distribution of cache hit locations

Figure 8.14 shows the hit distribution obtained from the `BLOCK` simulation set. The majority of the benchmark programs demonstrate what would have been expected – more blocks means more accesses hit in the fast line-buffers and LFLs. However, there is also a cross-effect between the hit locations. With a fixed cache size, more LFLs cause more LFL hits since they contain the most recently fetched memory locations. This can be seen clearly in most of the cases presented here, especially Dhrystone.

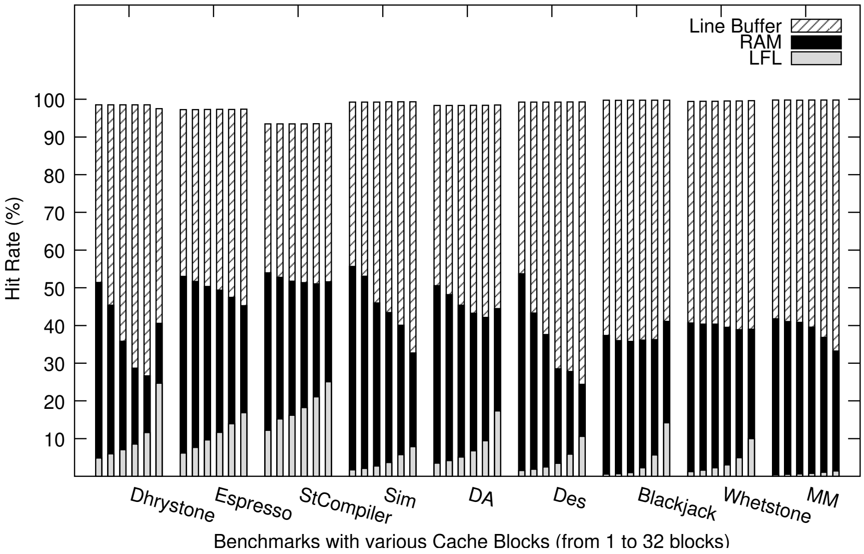


Figure 8.14: Effects of sub-blocking on distribution of cache hit locations

To improve the overall performance, accessing asynchronous fast line-buffers is preferred to accessing LFLs (and main RAMs). In some cases increased line-buffer hits can be achieved by reducing the number of LFLs since data in the LFLs cannot be moved to the main RAM, hence nor to the line-buffer. A better scheme for emptying the contents of the LFLs into the RAM would improve this aspect of performance. However, the basic line fetch mechanism is still favoured here because it results in no unnecessary activity, ideal behaviour for a low power system.

8.2.2 Asynchronous delay characteristics

Figure 8.15 (obtained from simulations with the base-level parameters described in section 7.6) shows a histogram of both instruction and data (combined cacheable and uncacheable) request latencies. Latency is timed from the point at which the request is presented to the memory system to the time the data emerges at the output. It can be seen that there is a wide distribution in latencies, the handling of which can easily be accommodated in an asynchronous design.

The two dominant sets of peaks (at ~6 ns and ~28 ns) coincide with the majority of the memory requests: line-buffer hits (which are serviced quickly from the fast asynchronous line-buffers) and main cache RAM hits. Amongst the peaks representing RAM hits, a small peak at ~40 ns is from the operations forwarded from the victim cache. The two sets of peaks at ~17 ns and 52 ns both coincide with LFL hits. Whereas the 52 ns set is due to ‘hits’ on words that have not yet been fetched, the faster 17 ns set are for hits where the words requested are already there in the LFL, hence there is no extra stall. The smaller spikes around these dominant sets occur when there are conflicts between instruction and data accesses to the same cache block.

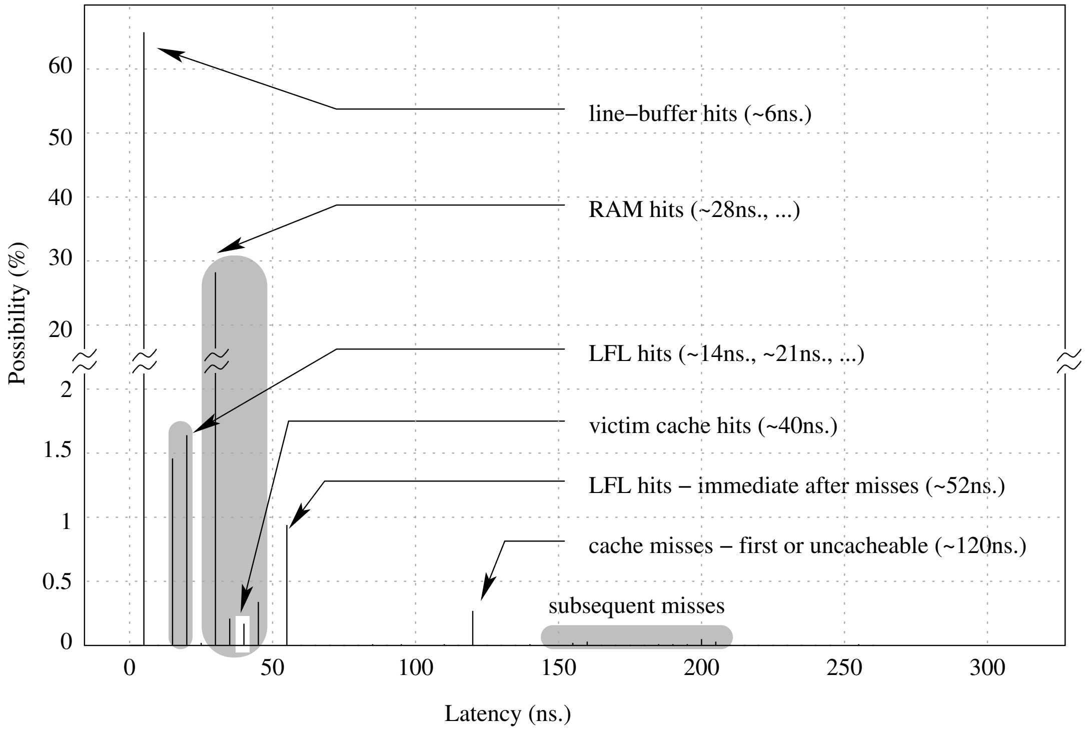


Figure 8.15: Latency distribution

Further to the right, the peak at ~120 ns coincides with either cache misses or uncacheable accesses which then cause slow external memory activities. The shaded area at the far right indicates misses that are queued behind other misses. These memory requests have to wait until the entire previous cache line is fetched from the memory.

8.2.3 Line-buffering

Without line-buffers there will be a full CAM look-up for every cache access, leading to high power consumption. If there were not separate line-buffers between the instruction and data ports this would also mean increased arbitrations for instruction and data accesses to the same block, leading to performance reduction. Furthermore, the (single) line-buffer contents would get changed when a data access comes between instruction accesses (or *vice versa*). This would then lead to fewer line-buffer hits and more line-buffer updating.

Figure 8.16 (obtained from the `WTnoLB` set and partly from sets `WTwithBURST` and `CBwithBURST`) shows a comparison of the (normalised) total run times of the chosen benchmarks using a copy-back cache and a write-through cache with and without line buffers. The top regions in the histogram came from experiments which can be considered as an 8 kilobyte AMULET2e cache system. The middle regions can be thought of as combining techniques from the previous AMULET memory systems and the bottom of each bar represents the results from the proposed cache architecture.

The major effect which can be seen is the impact of the line-buffer which both acts as a fast ‘level-0’ cache and helps alleviate the problems of data and instruction fetch conflicts. These simulations suggest that dual line-buffers should reduce the frequency of accesses to the cache RAM and LFL by around 40%, with a resulting decrease in power – since this also prevents a full CAM look-up – when compared to having no line-buffers. The graph also shows the (5-15%) difference in the performance of the write-through and copy-back caches.

The line-buffer tag comparison and the CAM look-up could be performed either sequentially – as assumed previously – or in parallel. Doing them in parallel (set `CBparLB`) would seem to provide higher performance but, in fact, it does not because the

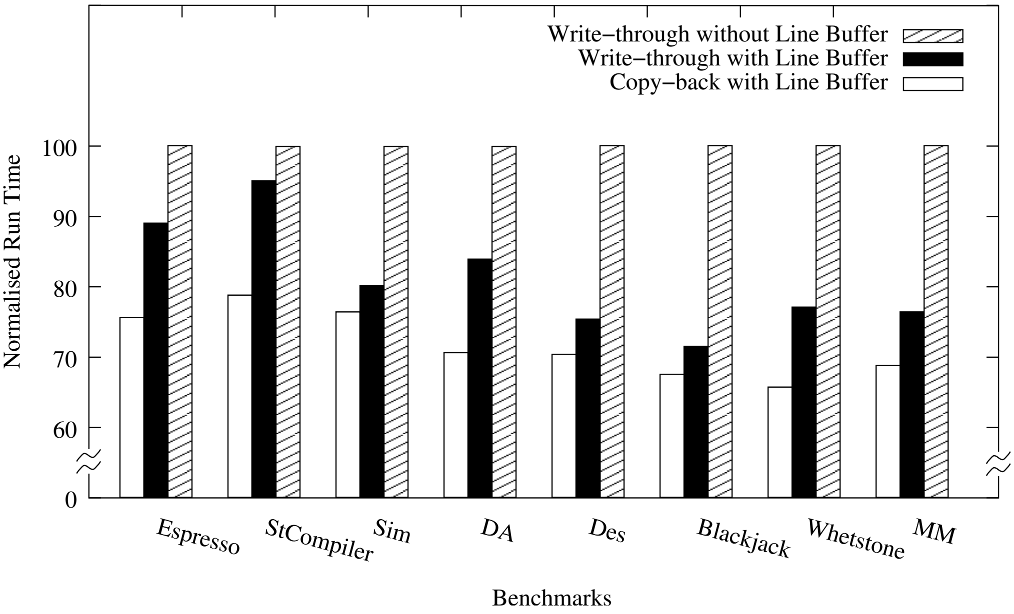


Figure 8.16: Line-buffering and copy-back styles

CAM look-up process follows the arbitration of the two cache buses, and as the hit rates on the line-buffers are relatively high the arbitration and CAM look-up is frequently unnecessary. Omitting these operations can therefore enhance performance and save considerable power. Therefore whilst parallelising these operations would make sense in a single-ported cache, in this ‘dual-ported’ architecture checking the line-buffer tags before activating the CAM is noticeably beneficial. This leads to a very variable association time for the cache. In an asynchronous implementation this does not impose much performance penalty; in a synchronous implementation it could impose a clock cycle overhead.

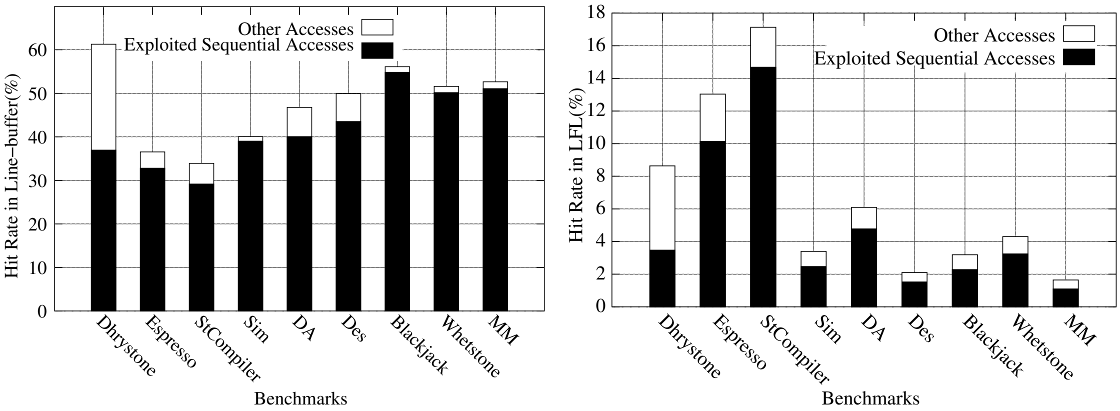
Since the line-buffers are heavily used (by over 40% of accesses) and have a very fast access time, further investigation was performed (set `LB`) to verify that larger line-buffers (e.g. double or three times the size) give only a marginal performance increase. This is because most of the benefit of the line-buffer is due to the sequential accesses to the words it contains, not because it is holding its contents for later reuse. In fact the contents are typically evicted from the buffer by another block access long before they would be reused.

8.2.4 Address sequentiality

As expected, sequential optimisation provides significant benefit in terms of a reduced number of line-buffer tag look-ups and CAM look-ups. The effects are less noticeable in data accesses as implied by figure 8.17, which shows the proportion of accesses for which sequential optimisation can be exploited, i.e. accesses marked sequential by the processor which are not the first access to a line.

(a) Instruction Hits in Line-buffer

(b) Instruction Hits in LFL



(c) Data Hits in Line-buffer

(d) Data Hits in LFL

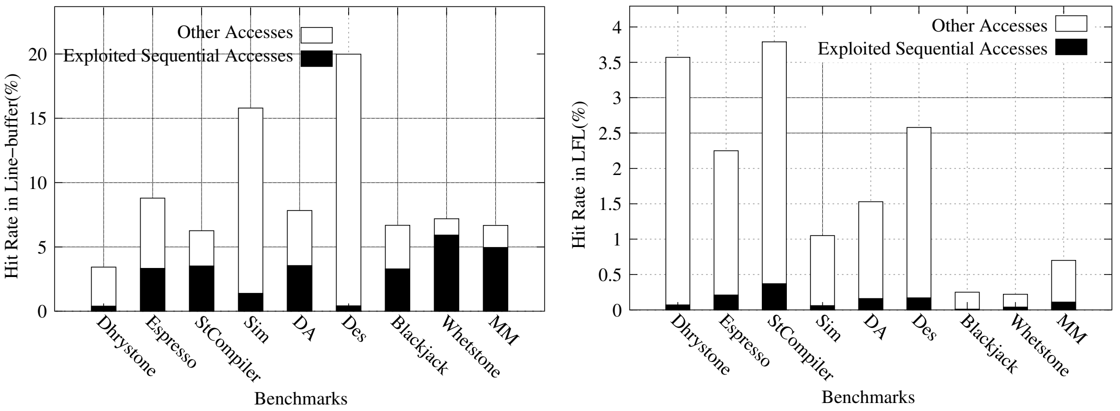


Figure 8.17: Breakdown of exploitable sequential accesses

NB. vertical axis scales not identical

8.3 Victim cache

The advantages of having a victim cache depend on several issues including the main cache miss rate, the organisation of the victim cache itself and the behaviour of the application programs. Moreover, the miss rate depends on the main cache size, organisation and replacement strategy. Since the total cache size here is fixed at 8 kilobytes as described earlier, and a range of programs are to be run, the focus of this section is on how the effects of the victim cache vary with different set associativity in the main cache and on the effects of distributing the victim cache.

Figure 8.18 contains five sets of results for a variety of total victim cache sizes. The leftmost and the rightmost bars in each set illustrate the behaviour if instead of having a victim cache, the storage it would have used is distributed evenly between the blocks to make each block slightly larger. They show the miss rate of a direct-mapped cache and of the fully-associative (in each block) cache respectively.

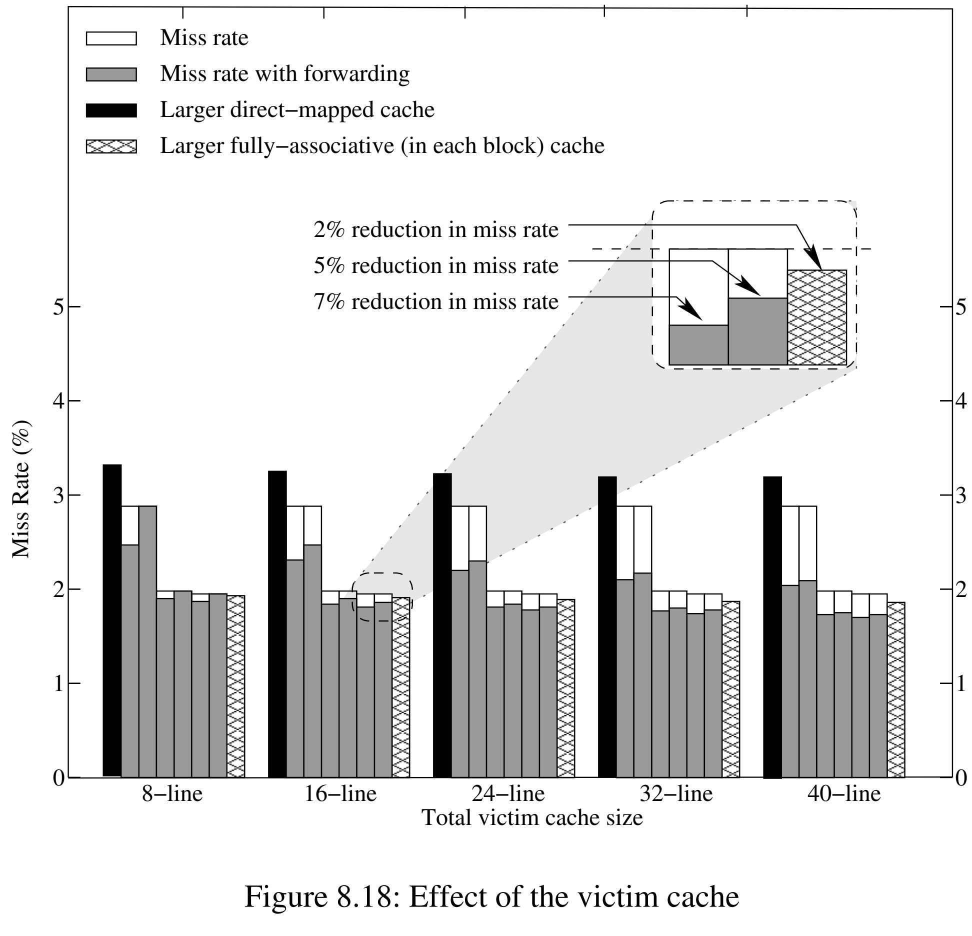


Figure 8.18: Effect of the victim cache

The middle six bars in each set show (from left to right) the results for the following formats:

- direct-mapped cache with central victim cache
- direct-mapped cache with local victim caches
- 8-way cache with a central victim cache
- 8-way cache with local victim caches
- 64-way cache[1] with a central victim cache
- 64-way cache[1] with local victim caches.

Each bar shows the miss rate of the bare cache, and the overall miss rate of the combination of the cache and the victim cache.

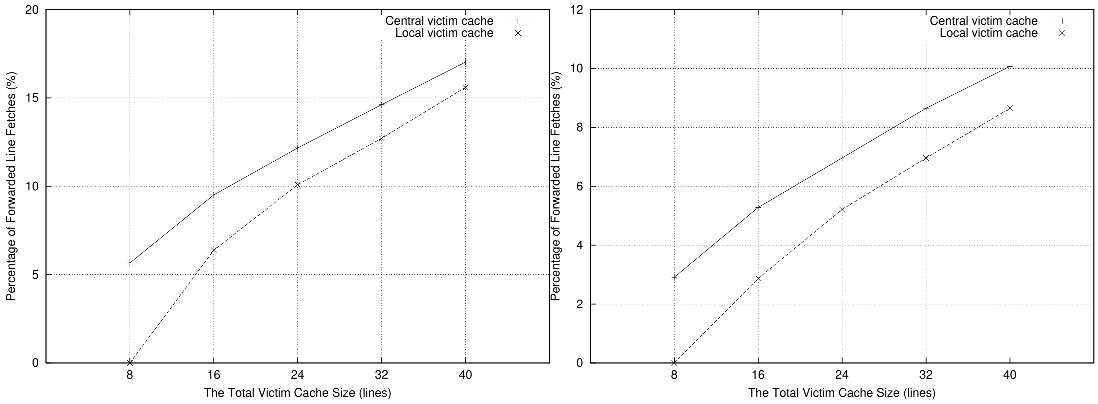
All of the set-associative caches in these simulations use a pseudo-random replacement strategy to choose victim lines.

8.3.1 Direct-mapped vs set-associative caches

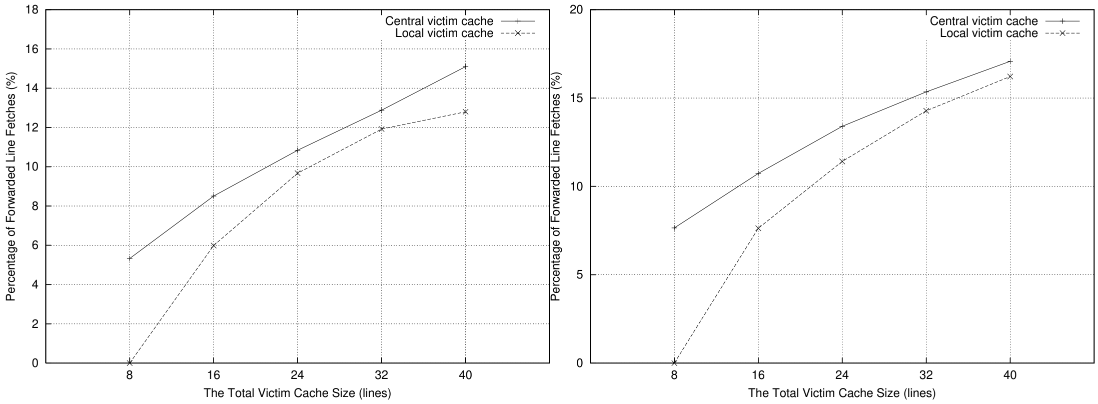
It is obvious that high associativity caches provide a much better (original) miss rate compared to direct-mapped caches with the same victim cache style since there are fewer conflict misses, however, the effect of having a victim cache is more dramatic on direct-mapped caches where ejection of lines that are subsequently required again is more of a problem. Nonetheless, as shown in figure 8.18, the advantage of the victim cache is that it also improves the miss rate of the 64-way caches by up to 0.25%. This, combined with the additional simplicity of implementing cache *lock-down* (where critical code is loaded into the cache and prevented from being rejected), makes the block-based associative cache the preferred choice.

---

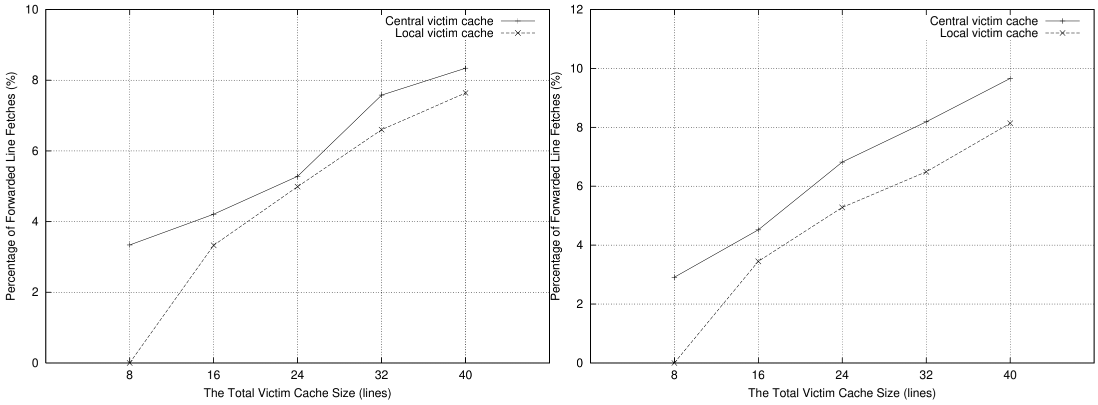
1. 64-way cache comprised of 8 memory interleaved fully-associative 64-line blocks


(a) Espresso

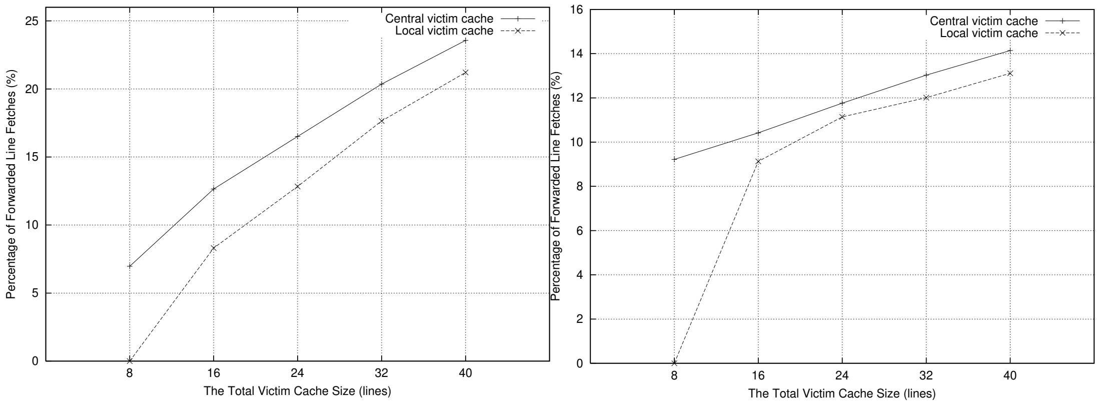
(b) ST Compiler


(c) Sim

(d) DA


(e) DES

(f) Blackjack


(g) Whetstone

(h) MM

Figure 8.19: Distribution of the victim cache

NB. vertical axis scales not identical

8.3.2 Victim cache distribution

The results in figure 8.19 show what would have been expected from table 6.1. With only a single-line victim cache in each block (8-line local victim cache), no forwarding can occur, hence the benefit of having the central victim cache over local victim cache is at its maximum at this size. Away from this extreme situation, a small difference in miss rate between these two styles is observed, with the local victim cache scheme marginally lagging (in performance terms) the centralised approach. This should be traded against the performance benefit of using short, localised forwarding paths giving a faster (lower latency) forwarding route, which can be fully exploited in an asynchronous environment.

8.3.3 Efficient use of resource

In fully-associative caches, using a few extra lines as a victim cache gives more benefit than extending the cache by the same amount. Although the difference appears small the effect of the victim cache is magnified when considered as a proportion of the miss rate (inset, figure 8.18). Using a few extra lines of store as a victim cache rather than adding them to the main cache can reduce the miss rate by an extra ~5%, the benefit increasing as the victim cache size increases.

8.4 Summary

The proposed cache organisation has been extensively simulated, and the distribution of cache access times shows considerable variability depending on whether the required data is located in the line-buffer, cache RAM or the LFL, with further variability resulting from non-deterministic arbitration delays when contention between the data and instruction ports arises. An asynchronous processor such as AMULET3 has no difficulty in operating in this environment, however, and can benefit from any small improvement in access times that the cache can yield.

Chapter 9: Conclusions

This thesis has presented the first asynchronous copy-back cache architecture. The design was developed to provide cache support for a third generation asynchronous microprocessor, AMULET3, which implements the ARM instruction set architecture. The processor is a Harvard-like architecture requiring a dual-ported but unified memory system that is capable of handling contention between two independent asynchronous ports. The original, existing AMULET3 system used only memory-mapped RAM.

This chapter summarises the work described in this thesis, draws conclusions and suggests possible directions for future work.

9.1 Architecture summary

The cache inherits techniques used in earlier AMULET systems, such as a line fetch mechanism, interleaved, fully associative sub-blocking and the dual-port memory based upon block-level arbitration and separate instruction and data line-buffers, a single line of each being sufficient. Major new features include the support of a copy-back write strategy with a write-allocate policy and a write buffer that implements a form of victim cache distributed between the blocks, with 2 lines per block, the minimum for forwarding being sufficient for satisfactory performance. To reduce the stall duration when a write-allocation occurs, and for good performance on a LFL write hit, the LFL used here allows individual words to be written by the processor.

As the majority of accesses are sequential, most of the necessary tag or CAM comparisons can be replaced by a simple circuit which detects if the new program counter or data access value maps to the same line as the previous one. Such comparisons will therefore be necessary only when accessing a different line or swapping the port used to access a block. Occurrences of this are simple to detect.

The resulting cache organisation has been extensively simulated, and the distribution of cache access times shows considerable variability depending on whether the data is located in the line-buffer, cache RAM, LFL or victim cache. Further variability arises from non-deterministic arbitration delays when contention between the data and instruction ports arises. An asynchronous processor such as AMULET3 has no difficulty in operating in this environment, however, and can benefit from any small improvement in access times that the cache can yield. The contents of each block are combined such that any cache access may pass through 1, 2 or all 3 stages (the line-buffers, cache RAM or LFL and the victim cache) depending on where (if at all) a cache hit occurs.

This cache design shows that the benefits of copy-back operation are available to asynchronous as well as clocked processors and, in addition, the asynchronous environment allows the straightforward inclusion of features such as line-buffers that offer significant performance and power benefits which, in a clocked system, would create difficulties due to their non-deterministic temporal properties.

The summary of the proposed cache architecture is depicted in figure 9.1.

9.2 Future work

The goal of the work presented in this thesis was to propose an entirely asynchronous cache system to work with a dual-ported microprocessor aiming for embedded systems. However, although the cache design has been optimised for the AMULET3 Harvard-like core, the techniques presented in this thesis are applicable to any other processor design (whether ARM compatible or not).

There are several areas in which the proposed cache architecture could be used in the future, as well as many possible expansions. This section describes suggested further work to improve the cache, categorised into 4 parts: architectural design alternatives, additional features, formal validation and VLSI implementation.

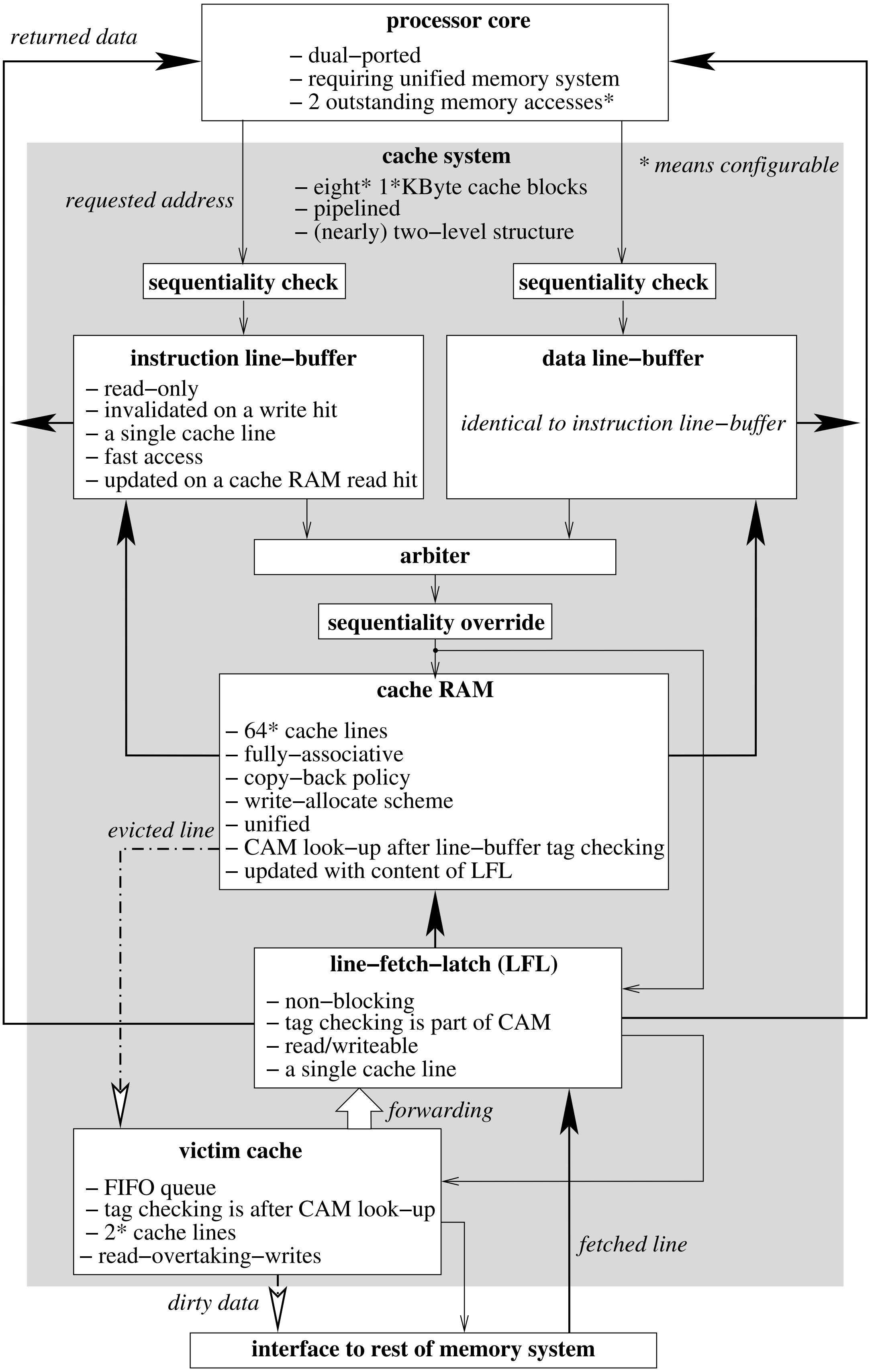


Figure 9.1: Cache architecture summary

Architectural design alternatives

There are 3 architectural issues described in this thesis that may warrant further investigation.

First, because the LFL is only emptied on a cache miss, a line may be accessed many times whilst stored in the LFL. Every such access incurs the additional performance cost of a line-buffer tag look-up and an arbitration, and burns power in the CAM which is checked concurrently with the LFL tag. Ideally, the content (complete line) in the LFL should be promoted to the cache RAM and line-buffer where any further accesses to it would occur, in the same manner as a hit in the RAM updates the line-buffer. This promotion should occur when it would not interfere with other cache accesses but this requires additional timing information that is not easily available in an asynchronous design. Furthermore, this promotion also requires an additional arbitration for updating the line-buffer.

Similarly, the forwarded data from the victim cache should be promoted directly to the line-buffer (as well as copying it into the main cache RAM via the LFL). This operation is simpler than the LFL promotion since the ‘complete’ line for forwarding is guaranteed in the victim cache hence the only requirement for this promotion is an arbitration for updating the line-buffer.

Lastly, updating the processor’s write-data into the LFL requires synchronisation for merging the fetched word and the write-data. An alternative approach to improve this includes option 6 described in section 5.6.6 which uses a shadow LFL to avoid processor stall in an arbitration-free manner.

Additional features

The cache presented includes the majority of features required by embedded systems. One notable exception, however, is support for locking down regions of the cache. With the fully associative CAM-RAM structure, this would be fairly straightforward to include, requiring modification of the method used to choose which line to evict.

Although this work has focused on uniprocessor support, there is no obvious reason why the cache could not be extended to support a cache coherency protocol such as the MESI protocol (Modified, Exclusive, Shared, and Invalid) [39], allowing use in multiprocessor systems.

Formal validation

Although a large number of simulations have been executed on a number of ARM programs, leading to high confidence that the design is correct and deadlock-free, formal proof of its correct behaviour and accurate performance analysis would be highly desirable. Unfortunately, current asynchronous formal validation tools, e.g. Rainbow [83], cannot handle designs of this complexity.

VLSI implementation

The ultimate aim would be to develop a VLSI implementation of the proposed cache architecture and analyse its cost and performance. The finished cache system can be used in either a single-ported or dual-ported general-purpose embedded system, or could easily be included in a multiprocessor system acting as a shared cache memory. Again arbitration would be performed only when more than one request attempts to access the same cache block at the same time.

Figure 9.2 illustrates a suggested layout organisation for a single block of the proposed cache system showing approximate dimensions for the major components. The cost estimates are all relative to a 6-transistor single-ported SRAM cell as used in the main cache RAM.

Using the same approach allows the cost of the proposed cache to be compared with a range of other caches (of the same size) as in table 9.1. The simplest of these is a direct-mapped cache (shown in figure 9.3) which has only a data RAM, a tag RAM, sense-amplifiers and a data latch for power reasons [94].

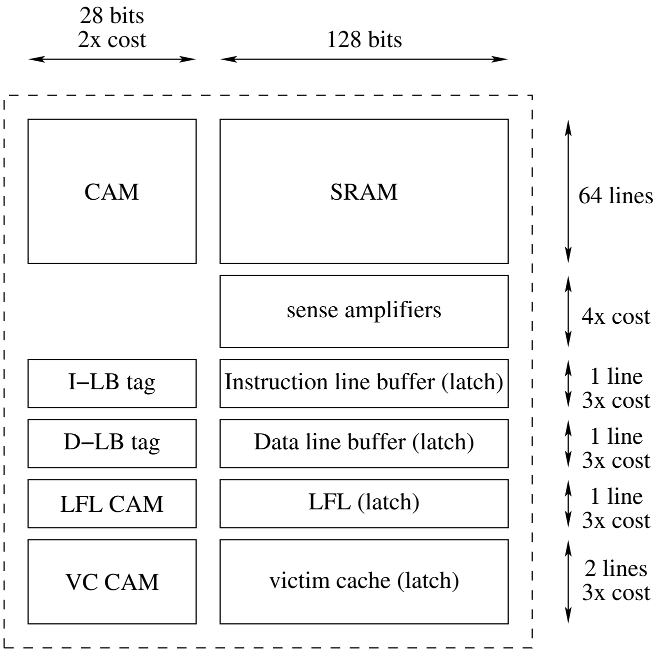


Figure 9.2: Suggested layout organisation of the proposed cache

The AMULET2e cache and the AMULET3i RAM structures are described earlier in section 4.3. The final comparison is against a dual-ported cache built with the same components as in the direct-mapped case accept for the use of the dual-ported SRAM and the replications of tag store, sense amplifiers and data latch required to support the second port.

memory systems	**cost relative to a simple direct-mapped cache**
direct-mapped single-ported cache system	100%
AMULET2e cache system	123%
AMULET3i RAM system	88%
dual-ported cache system	160%
proposed cache system	*130%*

Table 9.1: Cache cost comparison

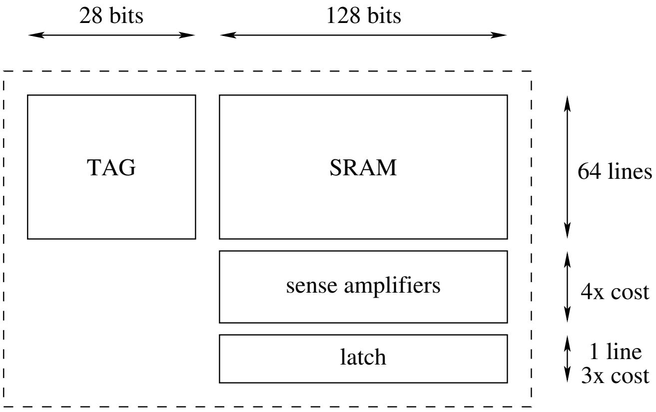


Figure 9.3: Layout organisation of a direct-mapped cache

9.3 Summary

The work described in this thesis shows that it is possible to adapt synchronous memory hierarchy techniques for use in asynchronous framework. The performance gap between the current range of asynchronous processors and main memory is therefore now no longer a hurdle to the widespread use of asynchronous processors. This is yet another step in bringing asynchronous processing into parity with the synchronous world.

9.4 Future prospects

Although a paradigm shift from synchronous to asynchronous is unlikely to happen in the digital electronics business in the near future, there are niche areas where asynchronous solutions are used today, and demonstrators such as the DRACO chip, show how asynchronous techniques can be applied to whole systems. These systems can now be more general through the use of asynchronous caches allowing asynchronous benefits to be applied to a much wider range of applications.

However, given the current dominance of synchronous design, hybrid systems comprising both design styles are the most likely route to market for asynchronous techniques. One philosophy based upon this method is the GALS (Globally Asynchronous Locally Synchronous) approach which uses asynchronous design for the interconnection of synchronous modules from different timing domains. Research on this is ongoing at a number of sites [73,76].

Within a GALS framework it is easy then to replace individual clocked modules on a case-by-case basis with asynchronous equivalents, and an asynchronous processor coupled with the cache described in this thesis would be a promising early option for substitution.

References

[1] A. Agarwal, M. Horowitz and J. Hennessy, “An Analytical Cache Model”, *ACM Transactions on Computer Systems*, 7(2), pp. 184-215, May 1989.

[2] “AMD-K6-III processor datasheet”, Advance Micro Devices Inc., 1999.

[3] S.S. Appleton, S.V. Morton and M.J. Liebelt, “Cache Design for Asynchronous VLSI RISC processor”, in *Proc. Australian Microelectronics Conference*, pp. 91-95, July 1995.

[4] “ARM Architecture Reference Manual” ARM Limited, ARM DDI 0100D 2000.

[5] “ARM940T Technical Reference Manual”, ARM Limited, 1999.

[6] “The Asynchronous Logic Home Page”, URL *http:// www.cs.man.ac.uk/async/index.html.*

[7] “The Asynchronous Bibliography”, URL *http://www.win.tue.nl/~wsinap/doc/async.html*.

[8] J.-L. Baer and W.-H. Wang, “Multi-level Cache Hierarchies: Organisations, Protocols and Performance”, in *Journal of Parallel and Distributed Computing*, 6(3), pp. 451-476, 1989.

[9] W.J. Bainbridge, “Asynchronous System-on-Chip Interconnect”, *Ph.D. Thesis*, Department of Computer Science, The University of Manchester, UK, 2000.

[10] A. Bardsley, “Implementing Balsa Handshaking Circuits”, *Ph.D. Thesis*, Department of Computer Science, The University of Manchester, UK, 2000.

[11] L.A. Belady, “A Study of Replacement Algorithms for a Virtual-storage Computer”, in *IBM Systems Journal*, 5(2) pp. 78-101, 1966.

[12] K. van Berkel et al., “The VLSI-Programming Language Tangram and Its Translation into Handshake Circuits”, in *Proc. European Conf. on Design Automation*, pp. 384-389, 1991.

[13] “Blackjack version 1.0”, URL *http://gd.tuwien.ac.at/perf/benchmark/aburto/bj/.*

[14] E. Brunvand, “The NSR Processor”, in *Proc. Annual Hawaii Int. Conf. System Sciences*, pp. 428-435, 1993.

[15] J. Bunda, D. Fussel and W.C. Athas, “Increasing Instruction Fetch Energy-Efficiency of a VLSI Microprocessor”, *Technical Report CS-TR-92-40*, The University of Texas at Austin, 1992.

[16] D. Callahan, K. Kennedy and A. Porterfield, “Software Prefetching”, in *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 40-52, April 1991.

[17] “C Converted Whetstone Double Precision Benchmark version 1.2”, URL *http://www.netlib.org/benchmark/whetstone.c*.

[18] T.-F. Chen, “Techniques for the Efficient Analysis of Cache Performance”, *Journal of Information Science and Engineering*, 12(4), pp.483-509, December 1996.

[19] T.-F. Chen and J.-L. Baer, “Effective Hardware-Based Data Prefetching for High Performance Processors”, in *IEEE Transactions on Computers*, 44(5), pp. 609-623, May 1995.

[20] T.-F. Chen and J.-L. Baer, “Reducing Memory Latency via Non-blocking and Prefetching Caches”, in *SIGPLAN Notices*, 27(9), pp. 51-61, 1992.

[21] K.-R. Cho, K. Okura and K. Asada, “Design of a 32-bit Fully Asynchronous Microprocessor (FAM)”, in *Proc. Midwest Symp. Circuits and Systems*, pp. 1500-1503, August 1992.

[22] J. Cortadella et al., “Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers”, in *Proc. IEICE Transactions on Information and Systems*, E80-D(3), pp. 315-325, March 1997.

[23] I. David, R. Ginosar and M. Yoeli, “Self-Timed Architecture of a Reduced Instruction Set Computer”, in *Proc. Asynchronous Design Methodologies*, pp. 29-43, 1993.

[24] M. E. Dean, “STRiP: A Self-Timed RISC Processor”, *Ph.D. Thesis,* Department of Electrical Engineering, Stanford University, USA, 1992.

[25] DSP56000/DSP56001 Digital Signal Processor User’s Manual.

[26] C.J. Elston et al., “Hades: Towards the Design of an Asynchronous Superscalar Processor”, in *Proc. Asynchronous Design Methodologies*, pp. 200-209, May 1995.

[27] R. M. Fuhrer et al., “Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines”, *Technical Report TR CUCS-020-99*, Columbia University, New York, 1999.

[28] S.B. Furber, “ARM System-on-Chip Architecture”, Addison Wesley Longman, Second Edition, 2000.

[29] S.B. Furber et al., “AMULET1: A micropipelined ARM”, in *Proc. IEEE Computer Conference*, pp. 476-485, March 1994.

[30] S.B. Furber et al., “AMULET2e: An Asynchronous Embedded Controller”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’97)*, pp. 290-299, IEEE Computer Society Press, April 1997.

[31] S.B. Furber, J.D. Garside and D.A. Gilbert, “AMULET3: A High-Performance Self-Timed ARM Microprocessor”, in *Proc. Int. Conf. Computer Design (ICCD’98),* October 1998.

[32] S.B. Furber et al., “AMULET2e: An Asynchronous Embedded Controller”, in *Proc. of the IEEE*, 87(2), pp. 243-256, February 1999.

[33] H. van Gageldonk, “An Asynchronous Low-Power 80C51 Microcontroller”, *Ph.D. Thesis*, Eindhoven University of Technology, The Netherlands, 1998.

[34] J.D. Garside et al., “AMULET3i - an Asynchronous System-on-Chip”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’2000)*, pp. 162-175, IEEE Computer Society Press, April 2000.

[35] J.D. Garside, S. Temple and R. Mehra, “The AMULET2e Cache System”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’96)*, pp. 208-217, IEEE Computer Society Press, March 1996.

[36] D.A. Gilbert, “Dependency and Exception Hanlding in an Asynchronous Microprocessor”, *Ph.D. Thesis*, Department of Computer Science, The University of Manchester, UK, 1997.

[37] D.A. Gilbert and J.D. Garside “A Result Forwarding Mechanism for Asynchronous Pipelined Systems”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’97)*, pp. 2-11, IEEE Computer Society Press, April 1997.

[38] J.R. Goodman, “Using Cache Memory to Reduce Processor-Memory Traffic”, in *Proc. Int. Symp. Computer Architecture (ISCA’83)*, 1983.

[39] J. Handy, “The Cache Memory Book”, Academic Press, Second Edition, 1998.

[40] “Heaps”, URL *http://www.cosc.canterbury.ac.nz/~tad/alg/heaps/heaps.html.*

[41] J.L. Hennessy and D.A. Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann, Second Edition, 1996.

[42] M.D. Hill, “A Case for Direct-Mapped Caches”, in *IEEE Computer*, 21(12), pp. 25-40, December 1988.

[43] M.D. Hill et al., “Design Decisions for SPUR”, in *IEEE Computer*, 19(11), pp. 8-22, November 1986.

[44] C.A.R. Hoare, “Communicating Sequential Processes”, in *Communications of the ACM*, 21(8), pp. 666-677, 1978.

[45] D. Hormdee, “A Proposed Asynchronous Dual-Ported Cache Architecture”, in Proc. *7th UK Asynchronous Forum*, Newcastle upon Tyne, Decemeber 1999.

[46] D. Hormdee, “An Asynchronous Dual-Ported Copy-Back Cache Architecture”, in Proc. *8th UK Asynchronous Forum*, London, June 2000.

[47] D. Hormdee and J.D. Garside, “AMULET3i Cache Architecture”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’2001)*, pp. 152-161 IEEE Computer Society Press, March 2001.

[48] D. Hormdee and J.D. Garside, “An Asynchronous Copy-Back Cache Architecture”, in *Proc. Postgraduate Research in Electronics, Photonics, Communications and Software (PREP 2001)*, April 2001.

[49] D. Hormdee, J.D. Garside and S.B. Furber, “An Asynchronous Victim Cache”, in *Proc. Int. Euromicro Symp. Digital System Design (DSD’2002),* September 2002.

[50] D. Hormdee, J.D. Garside and S.B. Furber, “An Asynchronous Copy-Back Cache Architecture”, submitted to *Microprocessors and Microsystems Journal.*

[51] “IBM PowerPC 405 Processor Core User’s Manual”, IBM Coperation, 2001.

[52] “Intel Pentium 4 Processor in the 423-pin Package at 1.30, 1.40, 1.50, 1.60, 1.70, 1.80, 1.90 and 2GHz Datasheet”, Intel Corporation, 2001.

[53] “Intel StrongARM SA-1110 Microprocessor Brief Datasheet”, Intel Corporation, 2000.

[54] L. Janin and D.A. Edwards, “Debugging Tools for Asynchronous Design”, in *Proc. 10th UK Asynchronous Forum*, July 2001.

[55] “LARD Documentation Home Page”, URL *http:// www.cs.man.ac.uk/amulet/ projects/lard/index.html.*

[56] J.S. Liptay, “Structural Asspects of the System/360 Model 85 Part II: The Cache”, in *IBM System Journal*, 7(1), pp. 15-21, 1968.

[57] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", in *Proc. Int. Symp. on Computer Architecture (ISCA'90)*, pp. 364-373, 1990.

[58] N.P. Jouppi and S.J.E. Wilton, "Trade-offs in Two-Level On-chip Caching", in *Proc. Int. Symp. Computer Architecture (ISCA'94)*, pp. 34-45, April 1994.

[59] N. Jouppi, "Cache Write Policies and Performance", in *Proc. Int. Symp. Computer Architecture (ISCA'93)*, pp. 191-201, 1993.

[60] M.B. Kamble and K. Ghose, "Analytical Energy Disspation Models For Low Power Caches", in *Proc. Int. Symp. Low Power Design (ISLPD'97)*, August 1997.

[61] T. Kilburn et al., "One-Level Storage System", in *IRE Transactions on Electronic Computers*, Vol.EC-11, No. 2, pp. 223-236, April 1962.

[62] J. Kin, M. Gupta and W.H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure", in *Int. Symp. Microarchitecture (MICRO'30)*, pp. 184-193, 1997.

[63] A. C. Klaiber and H. M. Levy, "Architecture for software controlled data prefetching", in *Proc. Int. Symp. Computer Architecture (ISCA'91)*, pp. 43-63, May 1991.

[64] U. Ko, P.T. Balsara and A. K. Nanda, "Energy Optimisation of Multi-Level Processor Cache Architectures", in *Proc. Int. Symp. Low Power Design (ISLPD'95)*, 1995.

[65] P.M. Kogge, "The Architecture of Pipelined Computers", Hemisphere Publishing Corporation, 1981.

[66] R. Kol and R. Ginosar, "Kin: A High Performance Asynchronous Processor Architecture", in *Proc. Int. Conf. Supercomputing (ICS'98)*, July 1998.

[67] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization", in *Proc. Int. Symp. Computer Architecture (ISCA'81)*, pp. 81-85, 1981.

[68] A.J. Martin et al., "The Design of an Asynchronous Microprocessor", in *Proc. Advanced Research in VLSI*, MIT Press, pp. 351-373, 1989.

[69] A.J. Martin et al., "The Design of an Asynchronous MIPS R3000 Microprocessor", in *Proc. Advanced Research in VLSI*, MIT Press, pp. 164-181, September 1997.

[70] "Matrix multiply tests – C language, version 1.0", URL *http://gd.tuwien.ac.at/perf/benchmark/aburto/mm/*.

[71] R. Mehra, “Micropipelined Cache Design Strategies for an Asynchronous Microprocessor”, *M.Sc. Thesis*, Department of Computer Science, The University of Manchester, UK, 1992.

[72] R. Mehra and J.D. Garside, “A Cache Line Fill Circuit for a Micropipelined Asynchronous Microprocessor”, in *IEEE Technical Committee on Computer Architecture Newsletter*, October 1995.

[73] S. Moore et al., “Point to point GALS interconnect”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’2002)*, pp. 69-75, IEEE Computer Society Press, April 2002.

[74] D. Morris and R.N. Ibbett, “The MU5 Computer System”, The Macmillan Press, 1979.

[75] S.V. Morton, S.S. Appleton and M.J. Liebelt, “ECSTAC: A Fast Asynchronous Microprocessor”, in *Proc. Asynchronous Design Methodologies*, pp. 180-189, May 1995.

[76] J. Muttersbach, T. Villiger and W. Fichtner, “Practical Design of Globally-Asynchronous Locally-Synchronous Systems”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’2000)*, pp. 52-59, IEEE Computer Society Press, April 2000.

[77] T. Nanya et al., “TITAC: Design of a Quasi-Delay-Insensitive Microprocessor”, 11(2), in *Proc. IEEE Design and Test of Computers*, pp. 50-63, 1994.

[78] M. Nyström, “Pipelined Asynchronous Cache Design”, *M.Sc. Thesis*, California Institute of Technology, USA, 1997.

[79] K. Öner and M. Dubois, “Effects of Main Memory Latencies on the Performance of Non-blocking Caches”, *Technical Report #CENG-92-34*, University of Southern California, 1992.

[80] L.A. Plana et al., “SPA - A Synthesisable Amulet Core for Smartcard Applications”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’2002)*, pp. 201-210, IEEE Computer Society Press, April 2002.

[81] B. Prince, “High Performance Memories”, John Wiley & Sons Ltd., Revised Edition, 1999.

[82] S. Przybylski, M. Horowitz and J. Hennessy, “Characteristics of Performance-Optimal Multi-level Cache Hierarchies”, in *Proc. Int. Symp. Computer Architecture (ISCA’89)*, pp. 114-121, May 1989.

[83] “The Rainbow Project”, URL *http://www.cs.man.ac.uk/fmethods/projects/AHV-PROJECT/ahv-project.html.*

[84] M. Renaudin, P. Vivet and F. Robin, “ASPRO-216: a Standard-Cell QDI 16-bit RISC Asynchronous Microprocessor”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’98)*, pp. 22-31, IEEE Computer Society Press, March 1998.

[85] W.F. Richardson and E. Brunvand, “Fred: An Architecture for a Self-Timed Decoupled Computer”, in *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems (Async’96)*, IEEE Computer Society Press, March 1996.

[86] J. Ruegg, Sozobon Limited, 1991. Public domain software available from the author: hans@wldrdg.uucp.

[87] S. Segars, “The ARM9 Family: High Performance Microprocessors for Embedded Applications”, in *Proc. Int. Conf. Computer Design (ICCD’98),* pp. 230-235, October 1998.

[88] C. Seitz, “System Timing”, Chapter 7 of *Introduction to VLSI Systems* by C. Mead and L. Comway, Addison Wesley, Second Edition, 1980.

[89] “Sim”, URL *http://gd.tuwien.ac.at/perf/benchmark/aburto/sim/.*

[90] A.J. Smith, “Cache Memories”, *ACM Computing Surveys*, September, 1982.

[91] A.J. Smith, “Line (Block) Size Choice for CPU Caches”, in *IEEE Transactions on Computers,* 36(9), pp. 1063-1075, 1987.

[92] A.J. Smith, “Second Bibliography on Cache Memories”, in *Computer Architecture News*, 19(4), pp. 154-182, June 1991.

[93] J.E. Smith and J.R. Goodman, “Instruction Cache Replacement Policies and Organisations”, in *IEEE Transactions on Computer*s, 34(3), pp. 234-241, March 1985.

[94] J Sparsø and S.B. Furber, “Principles of Asynchronous Circuit Design - A System Perspective”, Kluwer Academic Publishers, 2001.

[95] “SPEC Benchmark Suite Release 1.0”, October 1989.

[96] “SPEC CPU2000 V1.2”, URL *http://www.spec.org/osg/cpu2000/.*

[97] R.F. Sproull, I.E. Sutherland and C.E. Molnar, “Counterflow Pipeline Processor Architecture”, in *IEEE Design and Test of Computer*s, 11(3), pp. 48-59, 1994.

[98] W. Stallings, “Computer Organization and Architecture: Design for Performance”, Prentice-Hall International, Fourth Edition, 1996.

[99] C. Su and A. Despain, “Cache Design Tradeoffs for Power and Performance Optimisation: A Case Study”, in *Proc. Int. Symp. Low Power Design (ISLPD’95)*, 1995.

[100] I. Sutherland, “Micropipelines”, *Communication of the ACM*, 22(6), pp.720-734, June 1989.

[101] A. Takamura et al., “TITAC-2: A 32-bit Asynchronous Microprocessor based on Scalable-Delay-Insensitive Model”, in *Proc. Int. Conf. Computer Design (ICCD’97)*, pp. 288-294, October 1997.

[102] “VHDL++”, URL *http://*www.it.dtu.dk/~asytools/.

[103] R.P. Weicker, “Dhrystone: A Synthetic Systems Programming Benchmark”, *Communications of the ACM*, 27(10), pp. 1013-1030, October 1984.

[104] T. Werner and V. Akella, “Asynchronous Processor Survey”, in *IEEE Computer*, 30(11), pp.67-76, December 1997.

[105] N.H.E. Weste and K. Eshraghian,, “Principles of CMOS VLSI Design: A Systems Perspective”, Addison Wesley, Second Edition, 1993.

[106] “UltraSPARC III Cu User’s Manual”, Sun Microsystems, May 2002.

[107] Wm.A. Wulf and S.A. McKee, “Hitting the Memory wall: Implications of the Obvious”, in *Computer Architecture News*, 23(1), pp. 20-24, March 1995.

[108] M. Zhang and K. Asanivic, “Highly-Associative Caches for Low-Power Processors”, in *Kool Chips Workshop, Int. Symp. Microarchitecture (MICRO’33)*, December 2000.