AN APPROACH ON COPY BACK CACHE ORGANIZATION FOR AN ASYNCHRONOUS MICROPROCESSOR

Saurabh Rawat[1], Dr Rakesh Kumar[2], Anushree Sah[3], Sumit Pundir[4], Bhaskar Nautiyal[5]

[1,2,3,4,5]*Asst.Professor, Department of Electronics, Graphic Era University, Dehradhun (India)*

ABSTRACT

The common feature of all the existing cache systems is their use of a write through policy. This paper describes of an asynchronous, copy back cache architecture for use with a Harvard like architecture processor core. Issues addressed here include the line allocation mechanism, non-blocking line fetches and out-of-order accesses. Detailed examples of a variety of cache operations are provided showing the flexible timing behavior that can b supported by the cache.

Keywords: AMULET Processor, Line allocation, Non blocking, Write Buffer.

I ARCHITECTURE

The cache architecture presented in this paper is intended to work with the AMULET3 microprocessor, a third generation asynchronous ARM implementation. Although the first AMULET3 system had no cache, it featured eight kilobytes of memory mapped RAM. The organization of this first system will affect the design of the cache in this paper. The top level organization of a possible and cache subsystem is shown in figure. 1. The major units in this figure are:

- An AMULET3 core: this is an implementation of the v4t ARM architecture .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 .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.
- Two MMUs (Memory Management Units): located next to each of the instruction and data ports, these check whether a memory location is cacheable. Uncatchable memory accesses bypass the cache. The MMUs also detect memory access permission violation and page faults, signaling 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 and write buffers. Many of these operations are not supported in the system proposed here.

- Marble: as on-chip asynchronous system bus connects the MMUs and cache to other system components and the off-chip memory interface.

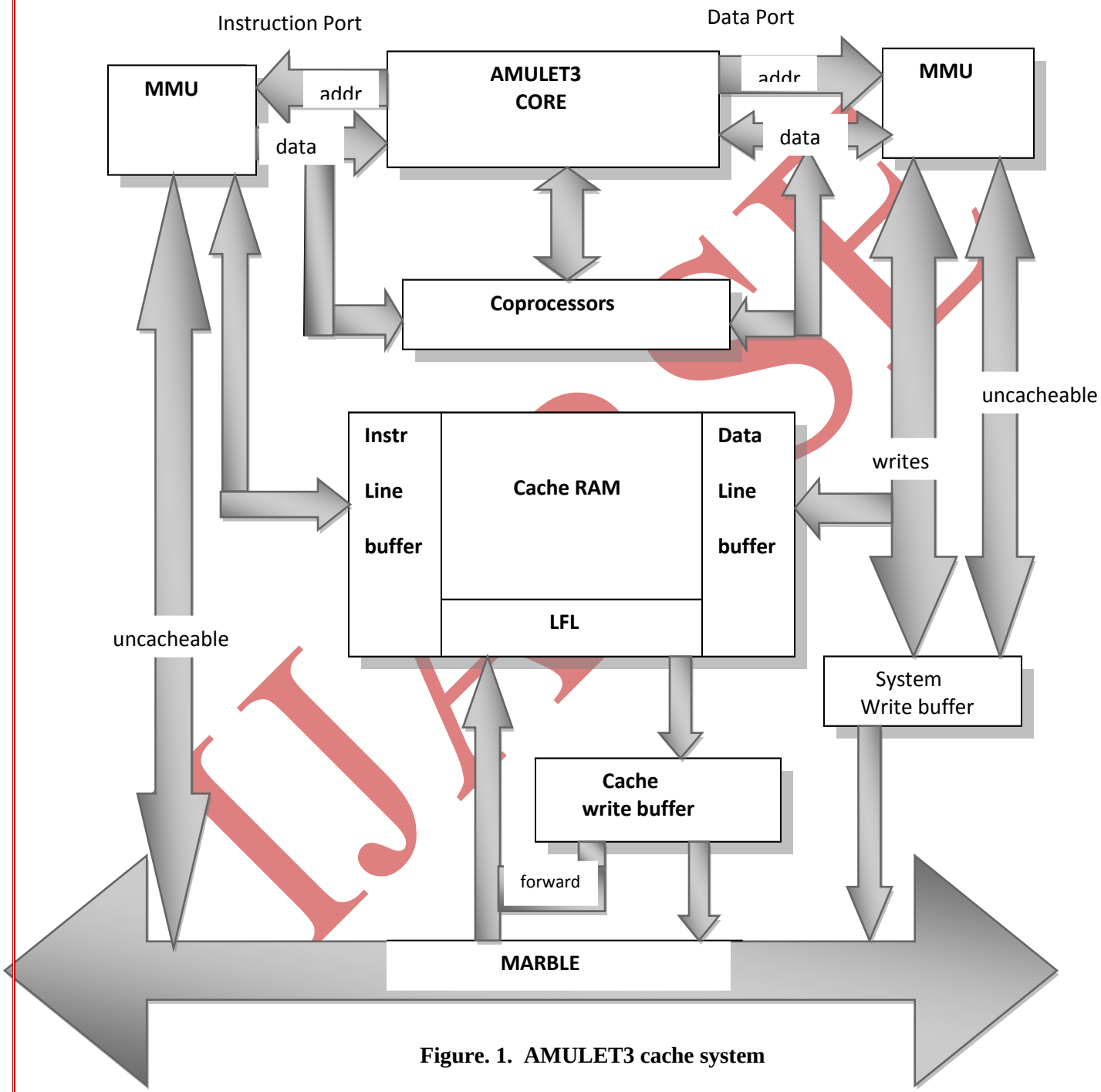

Figure. 1. AMULET3 cache system

- 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 in figure .A buffers uncatchable 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-buffer cache.

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 .this adds significant complexity because data written to cached memory location 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. In doing so memory bandwidth is better utilized, giving improved overall performance. The remainder of this paper describes in detail the cache operations and techniques that are adopted in the asynchrous copy-back cache for AMULET3. Throughout this cache description, it is assumed that the evicted cache lines are presented to the memory via the cache write buffer.

II. PSEUDO TWO-LEVEL CACHE STRUCTURE

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

2.1 '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 .2 resulting in a pseudo two-level 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. Its 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.

Level-1 cache

The true L1 cache is formed by parts shared between both instruction and data ports

- **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 cached in parallel because the cache access time is because it holds the newly fetched lines which are the only copy of the data in the cache system and behaves as an extension of the L1 Cache.

2.2 '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. The write buffer never generates a cache hit since, without forwarding, the only way to get data 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 rejection. A technique for overcoming this, the victim cache, is considered in the next chapter.

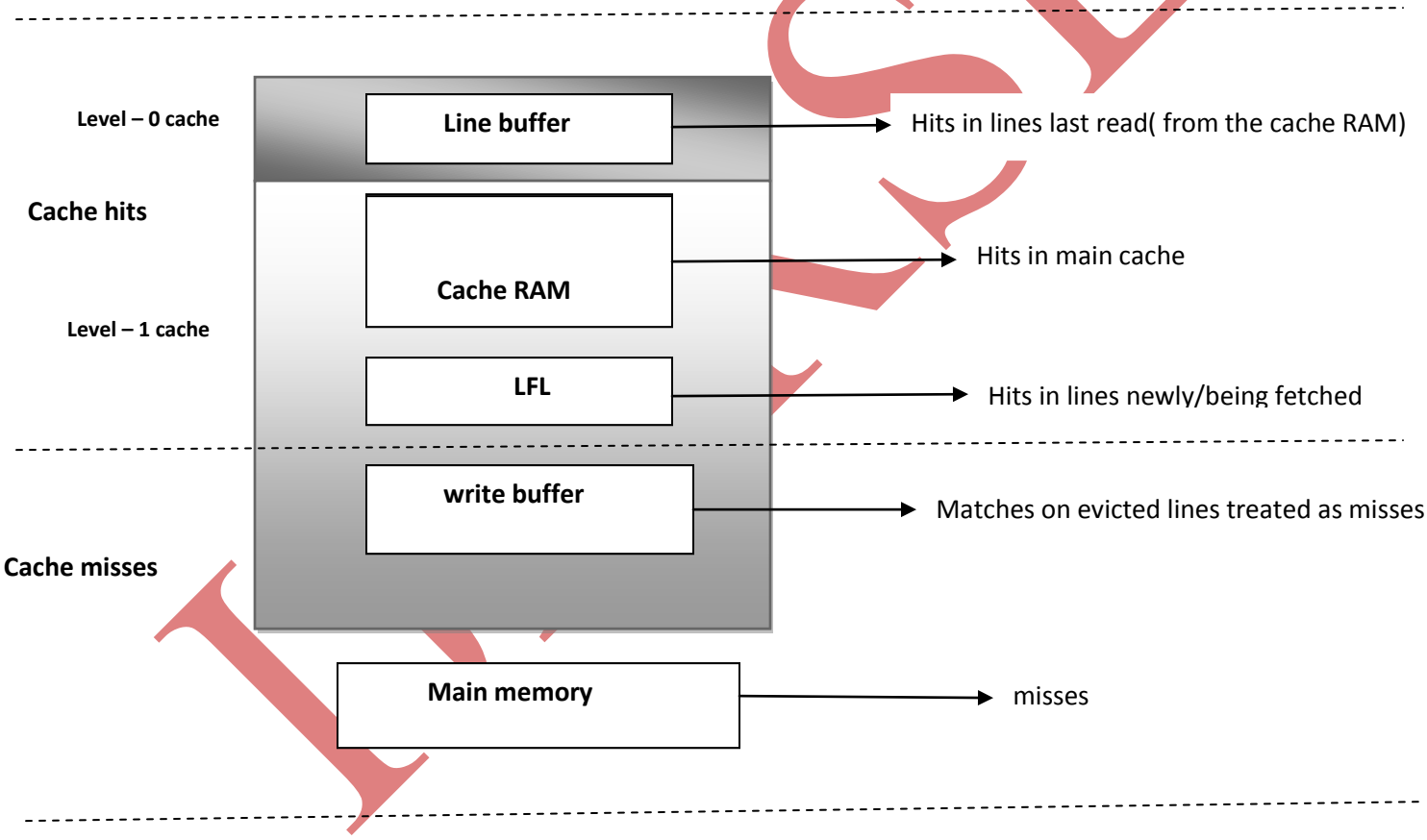

Figure: 2. Two level cache structure

III. LINE FETCH ENGINE

Figure .3 illustrates the control flow of a cache read request in this architecture. The top part shows the line-buffering adapted from the AMULET I memory system. The bottom part shows the line fetch technique adapted from the AMULET2e cache system .The line fetch engine in figure C is a separate unit from the cache. It takes a

line fetch address and interact 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.

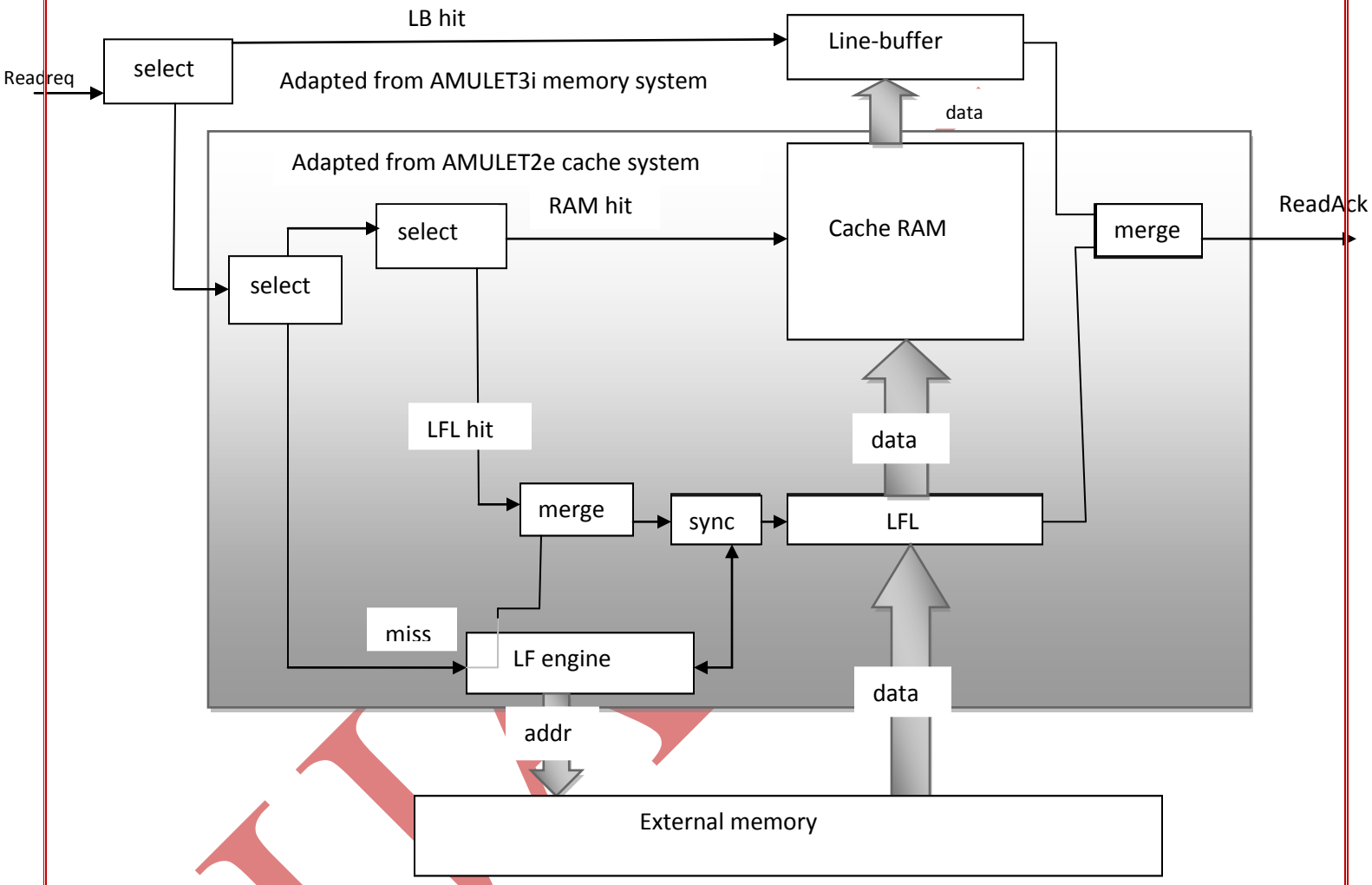

Figure: 3. Cache request steering control logic

IV. 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 4. The control of these operation is illustrated in a 2-phase control style 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 4 are:

- Activity 1: Select a victim line and eject it (from the RAM) to the writer 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 'emptying' the LFL. The request also reset the exclusive-OR gates' outputs, 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 synchronization logic is also used when an LFL hit wait until the requested word is present.
- Activity 4: After receiving the rejected line, the cache writes 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) 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 request to service.

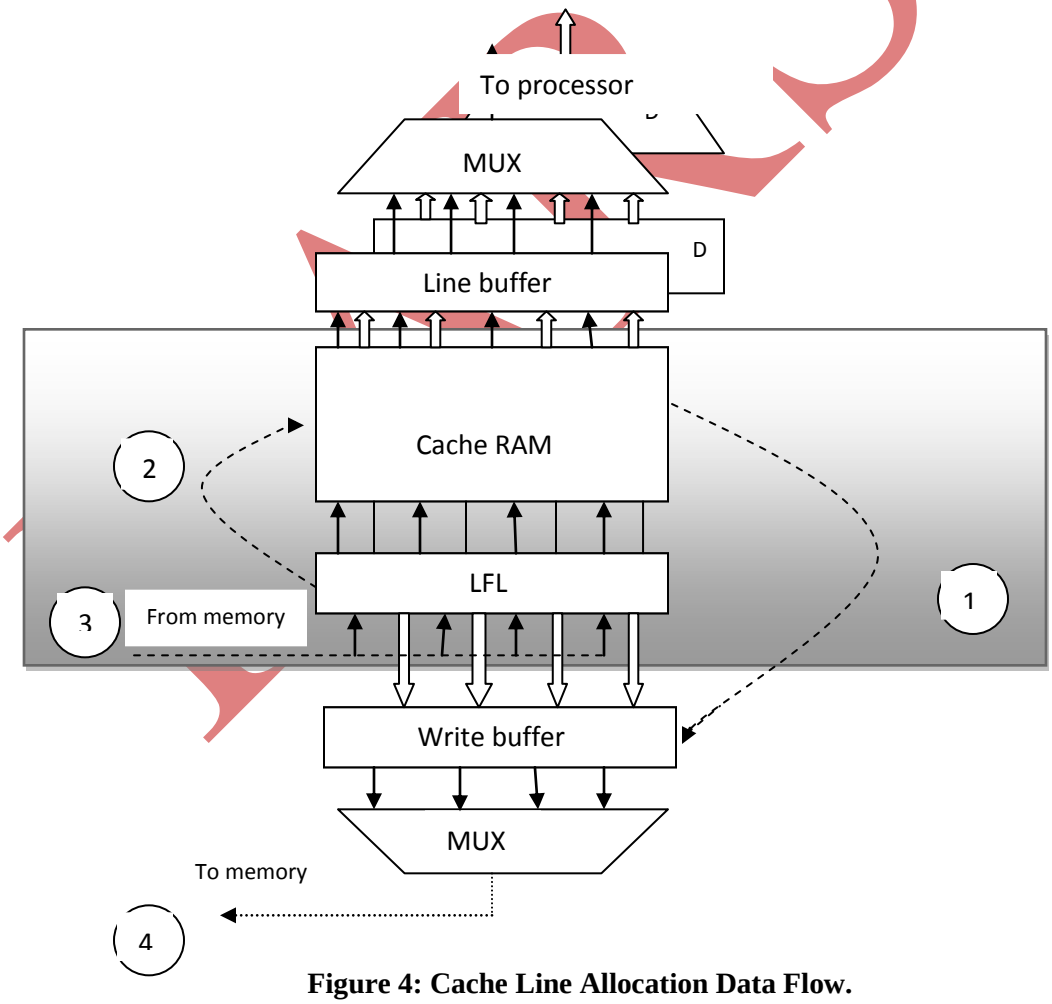

Figure 4: Cache Line Allocation Data Flow.

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

V. 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 requirement for a write-through policy is also included for comparison purposes. In this section, all possible activities are also illustrated in figure 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

5.1 Line-buffer read hit

A read hit in the line-buffer can be satisfied quickly from the appropriate fast asynchronous line-buffer. 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 request from the other port that may need to access the same cache block(including the other line-buffer) to proceed concurrently.

5.2 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,
- 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.

5.3 Cache RAM read hit

If an access does not hit in the line-buffer it is allowed, via an arbiter to serialize activity from the two ports, to access the 'main cache' system. Line-buffer write hits must also follow this path. The operations for a cache hit in the main cache RAM, 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. For a cache RAM read hit, the behavior 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 and the requested word is sent to the processor.

5.4 Cache RAM write hit

In the copy-back cache, whilst performing the write operation in the cache RAM for a write hit, 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 and then, when the bus is idle, the write request (in the queue) updates the main memory.

5.5 LFL read hit

The operation performed on an LFL hit) are much the same as those for a cache RAM hit. However, they are not identical since the LFL, buffers the fetched data which 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 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 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 since no cache updating occurs at any other cache level.

5.6 LFL write hit

There are a number of different ways a 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;

Option 4: 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 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 summarizes 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 miss	**write hit in LFL**	**write**
Option 1	up to 4 memory cycles + a cache write	4 memory cycles + a cache write
Option 2	up to 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 the 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.

5.7 Read miss

A cache miss is the most complicated and slower 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 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. The operations for a cache miss start with the usual (appropriate) line-buffer tag look-up followed by the CAM look up . 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 incurs the cost of a line fetch. However, prior to starting the line fetch, space must be created for the previously fetched line from line from the LFL to be inserted in the main cache; requiring an existing line to rejected. 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 . 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 and dirty data from the write buffer can be copied out to the main memory to maintain coherency.

5.8 Write miss

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 a memory, the write request is sent to the system write buffer and is emptied out when the memory bus is free. In a copy-back cache with write-allocation, a write miss requires a few additional operations. 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.Again, when the bus becomes idle the first dirty data in the cache write buffer queue is copied to the memory to maintain coherency.

VI. 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 paper illustrate the timing of 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.

6.1 Hits and misses in a non-blocking scheme

Figure E 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 line-buffer, line Y is in the cache and line A is in the LFL.

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 hope that they will be serviced by the 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.

The sequence of address requests….A0 B1 B2 A0 A1 A2 A3 B3 'A0' A1A2….

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 .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.

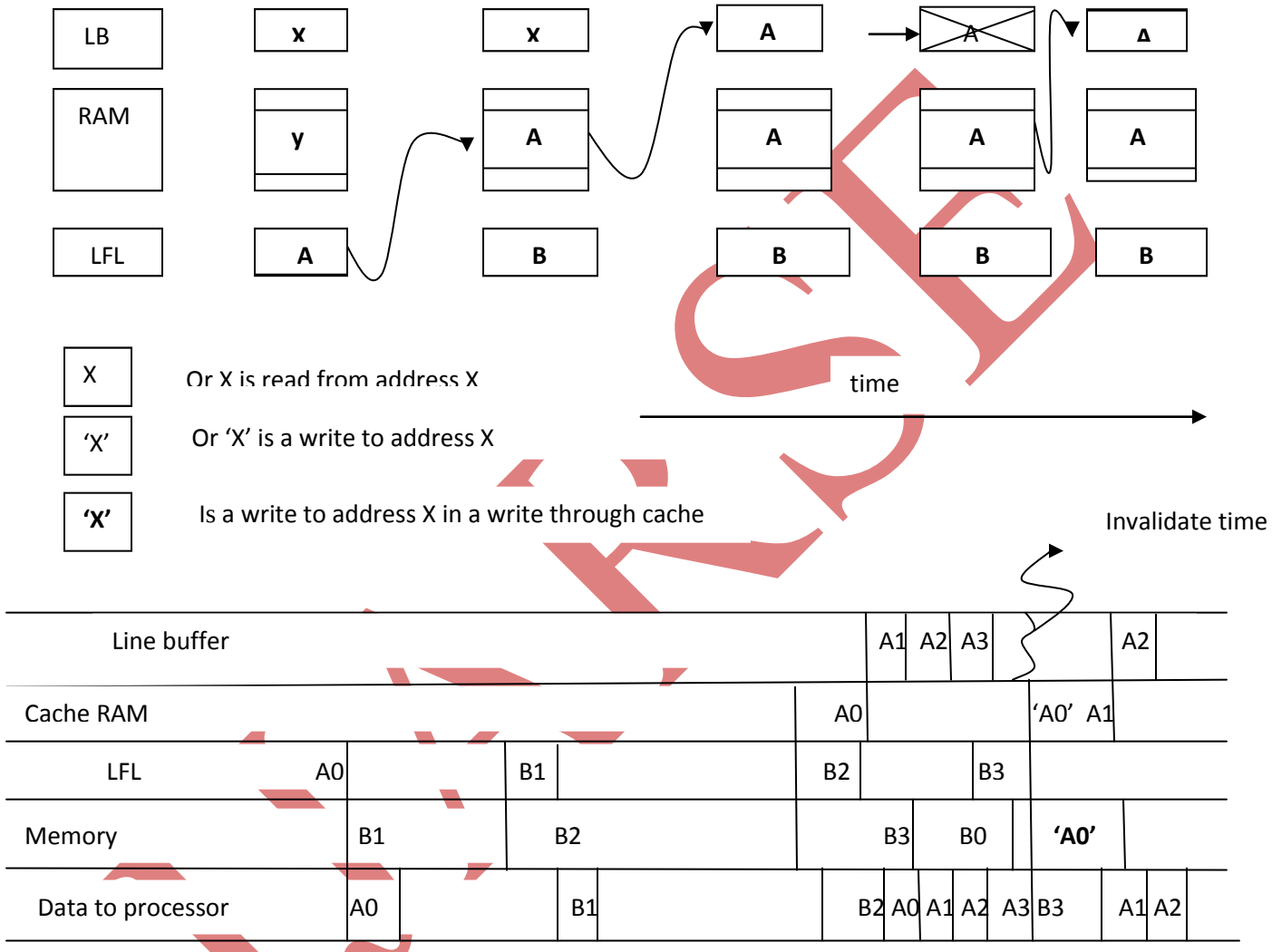

Figure 6.1: Hit Timing

6.2 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.

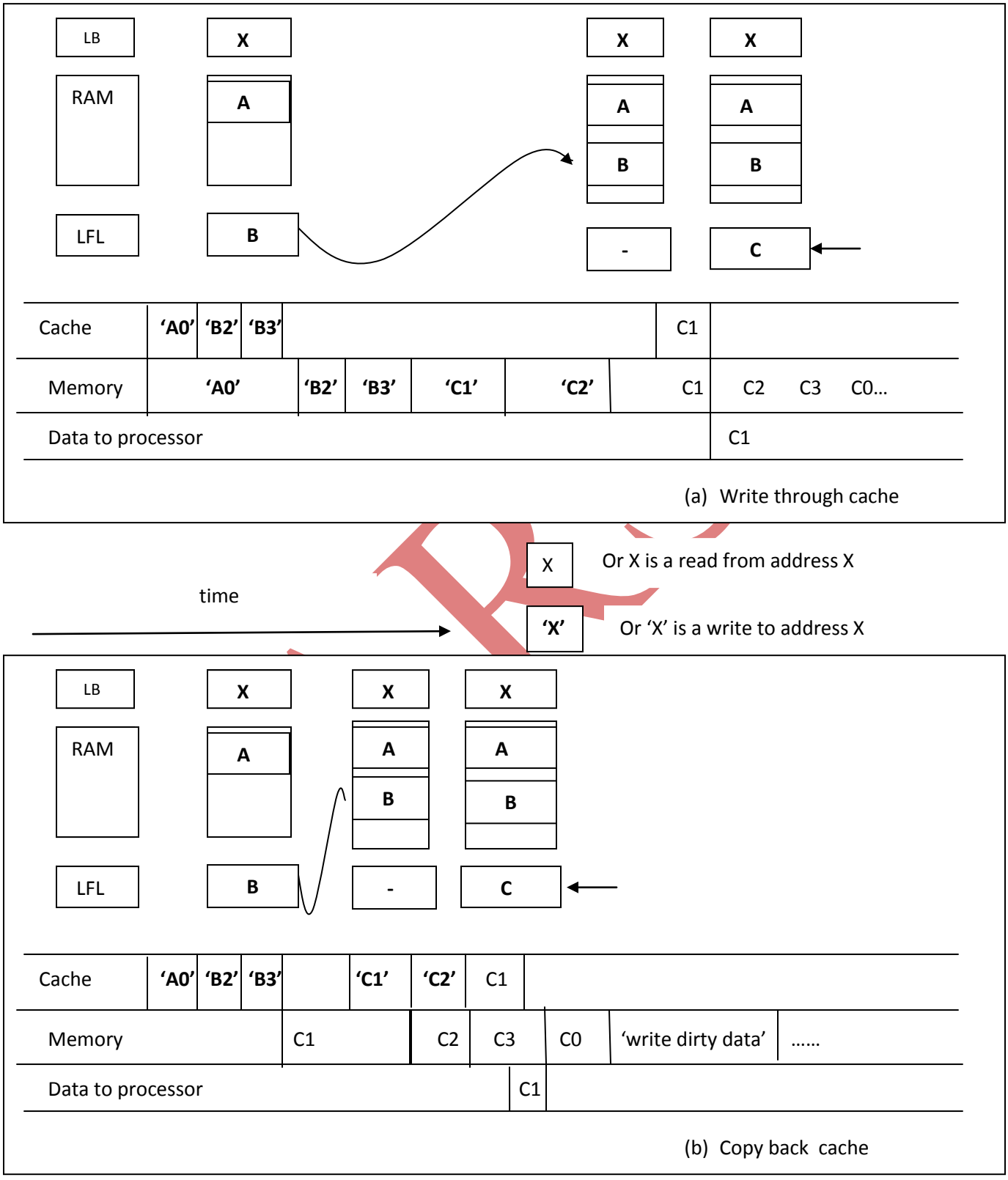

Figure 6.2: Timing for a sequence of writes

Figure F illustrates the benefits 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 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 these writes are handled entirely in the cache unlike in a write-through cache 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.

The writes C1 and C2 are cache write misses which are fairly simple write operations to the memory in the write-through cache but in a copy-back (which 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 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 ahead of subsequent reads from the same line.

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 write back 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.

VII. RESOLVING ORDERING PROBLEMS

To simplify the description of the cache's basic operation, it is 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.

Since the cache section is divided into (provisionally, eight) cache blocks, all of these could provide fast memory access 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 delay and a pipelined structure, there could potentially be more than 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 in 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 , ordered result collection and avoidance of the problem through single outstanding activities .

7.1 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 interleaving of the cache blocks. 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) .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.

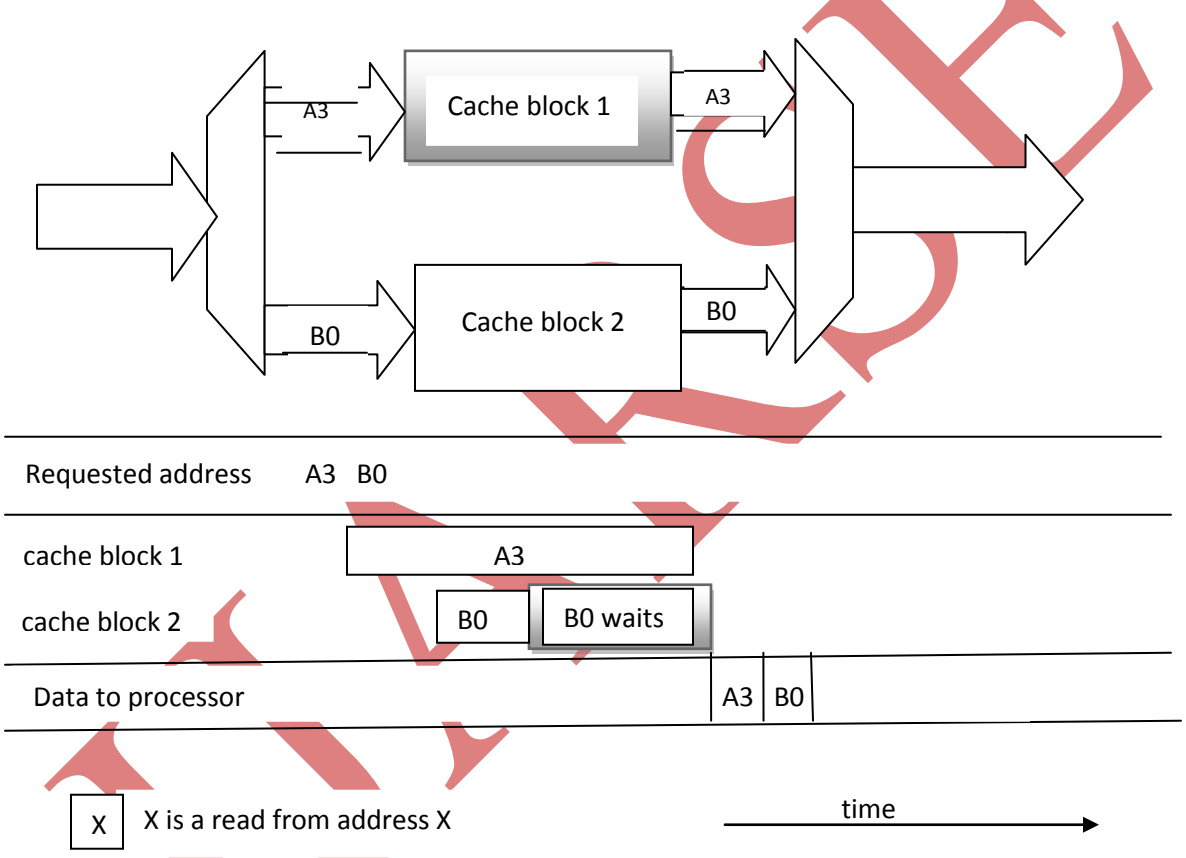

Figure 7: Order problem due to concurrent activities of different durations.

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, 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.

7.2 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 be 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 results 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) to collect accessed words from the right place. In fact, two separate FIFOs are used, one for each port. 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 spilt, along the zigzag, allowing the 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.

VIII. 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.

Solutions to this problem include:

- Do not reorder-The writer 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 .Clearly the second option is preferable if some mechanism of forwarding can be provided without introducing hazards in the asynchronous environment.

IX. CONCLUSION

This paper 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 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;
- copy-back write policy to reduce memory bandwidth;
- A write buffer with read-overtake-write support to reduce processor stall during for requested data.

REFERENCES

[1] A.J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T.K. Lee. \The Design of an Asynchronous MIPS R3000 Processor," Proceedings of the 17th Conference on Advanced Research in VLSI. Los Alamitos, Calif.: IEEE Computer Society Press, 1997.

[2] A.J. Martin. \Synthesis of Asynchronous VLSI Circuits," in Formal Methods for VLSI Design, J. Staunstrup, Ed. North-Holland, 1990.

[3] A.J. Martin. \The limitations to delay-insensitivity in asynchronous circuits," Sixth MIT Conference on Advanced Research in VLSI, W.J. Dally, Ed. Cambridge, Mass.: MIT Press, 1990.

[4] Mika Nystrom. \Pipelined asynchronous cache design." M.S. Thesis, Caltech CS-TR-97-21, 1997.

[5] Andrew Lines. \Pipelined asynchronous circuits." M.S. Thesis, Caltech CS-TR-95-21, 1995.

[6] H.P. Hofstee. Personal communication, September 1997.

[7] Gerry Kane and Joe Heinrich. MIPS RISC Architecture. Englewood Cli_s, N.J.: Prentice-Hall, 1992.Systems. Los Alamitos, Calif.: IEEE Computer Society Press, 1996.

[8] Jouppi, N. P. *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully- Associative Cache and Prefetch Buffers.* Proceedings of 17th Annual Int‟l Symposium on Computer Architecture, 1990, pp. 364-373.

[9] Przybylski, M. Horowitz, and J. Hennessy.*Characteristics of performance-optimal multi-level cache hierarchies*. Proceedings of the 16thInternational Symposium on Computer Architecture, 1989, pp. 114 –121.

[10] Baer, Jean-Loup, and Wang, Wen-Hann. *On the inclusion properties for multi-level cache hierarchies*. 25 years of the international Symposia on Computer Architecture (selected papers), 1998,pp. 345 – 352 [4] Gee, et al.. *Cache Performance of the SPEC92Benchmark Suite.* IEEE Micro, Vol. 13, Number 4, August, 1993, pp.