VICTIM CACHES – IN AN ASYNCHRONOUS ENVIRONMENT.

Saurabh Rawat[1], Dr Rakesh Kumar[2],

Anushree Sah[3], Sumit Pundir[4], Bhaskar Nautiyal[5]

Department of Electronics
Graphic Era University, Dehradhun (India)

ABSTRACT

When an asynchronous copy back cache architecture is designed to work with the AMULET processor, a third generation asynchronous ARM implementation, there is a problem of RAW hazard in basic write buffering 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. In this paper same technique is applied in asynchronous environment although implementing a forwarding mechanism in an asynchronous system is more difficult because data to be forwarded is flowing in an unsynchronized manner to the process which requires it.

I INTRODUCTION

1.1 Forwarding

A possible solution to forwarding in an asynchronous environment was introduced by Gilbert , an asynchronous implementation of a recorded buffer intended for use in a processor register bank. The recorder 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 is used in this paper. This allows memory write back 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 behavior 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.

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 above figure 1. Unlike the victim cache proposed by Jouppi ,where 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.

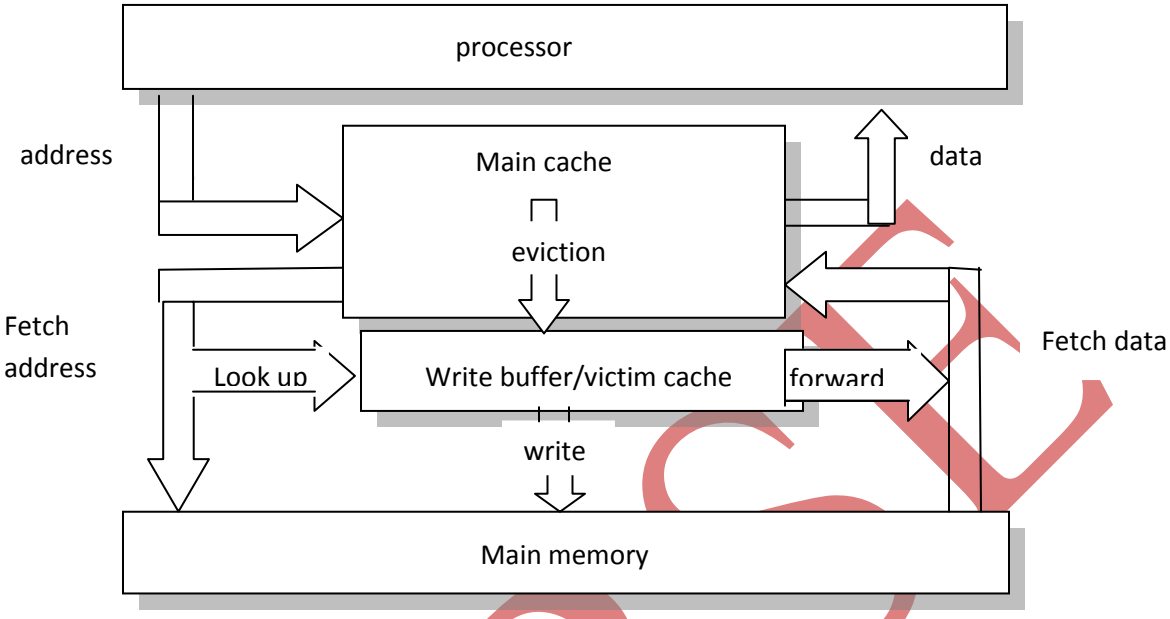


Figure 1: Write buffer/ victim cache position

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 lines. 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 2. Illustrates the different sizes of data transfer from/ to the cache system. 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.

II VICTIM CACHE PROCESSES

The victim cache was proposed by Jouppi as a method to reduce the impact of conflict misses in direct-mapped cache structures, but is easy to generalize 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 3 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 9CAM) 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:

2.1 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.

2.2 Cache eviction: A cache miss occurs and the main cache empties a line into the victim cache (shown in figure XX labeled 'fill VC'). The victim cache has to provide an empty storage location for the line at this time.

2.2 Buffered writes: The victim cache autonomously copies 'dirty' lines into the main system memory (shown in figure XX labeled 'drain VC'), freeing space for re-use.

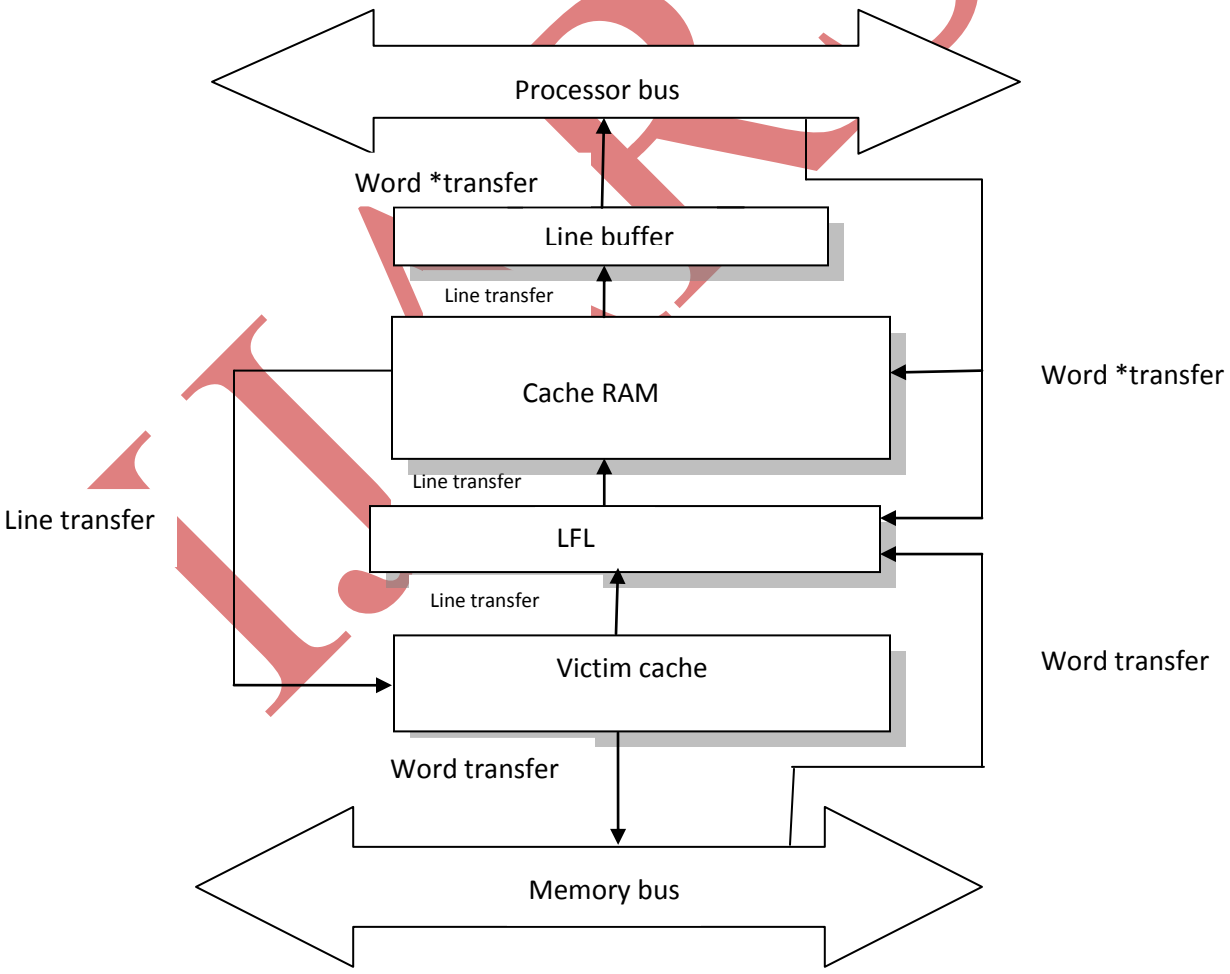


Figure 2 : Data transfer granularity

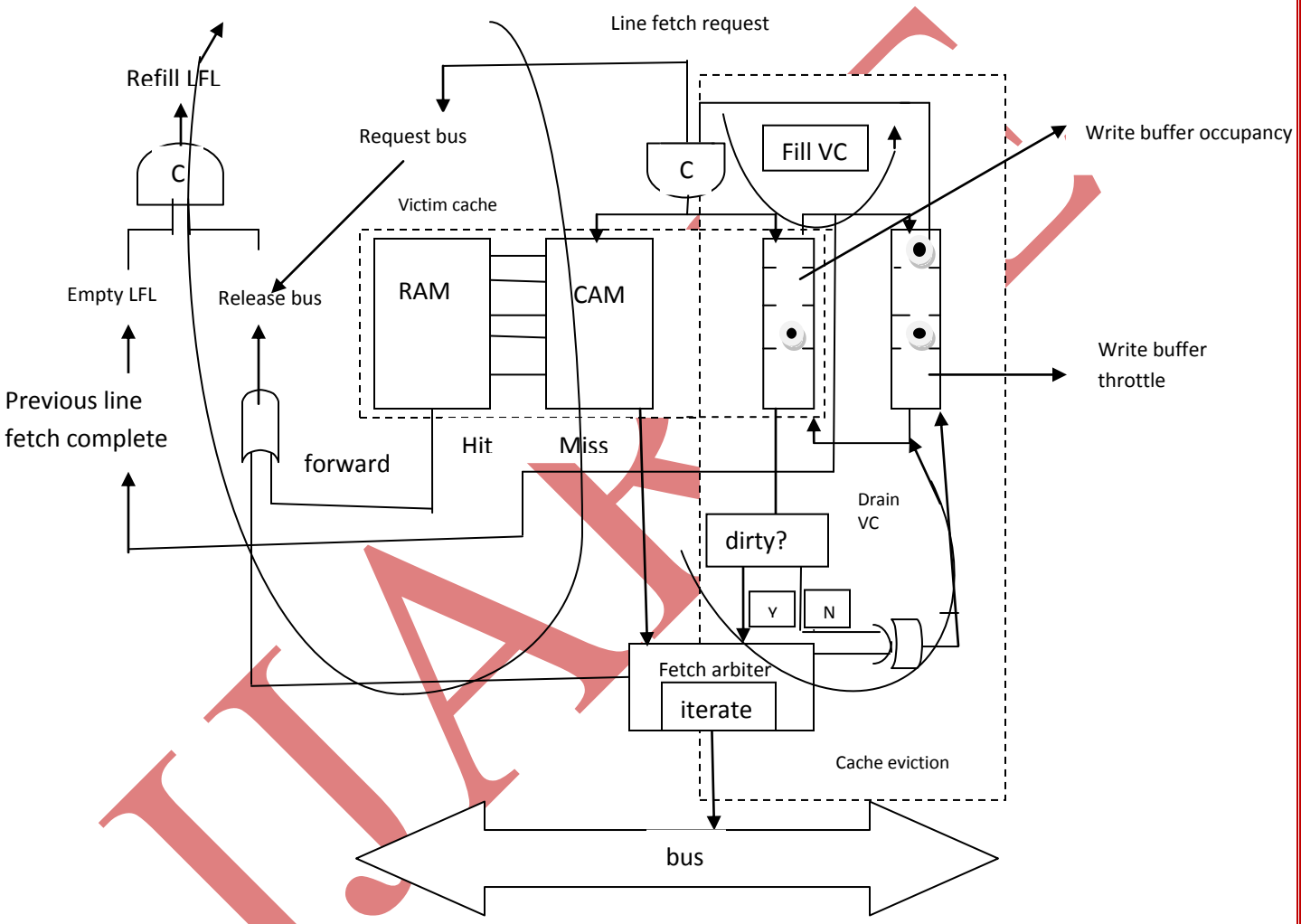


Figure 3 : Control flow in the victim cache

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 unsynchronized manner from the line-fetch/forwarding process that may require it.

III VICTIM CACHE IMPLEMENTATION

A similar approach to the one used in the reorder buffer in AMULET3 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 XX) 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 fordable data is fixed by the number of write buffer entries and is entirely independent of the copy-back process.

IV 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 RAM; and a number of additional control markers must also be kept. There are also global in and out pointers (as in figure X) steering the writing into the emptying out from the victim cache respectively. Three extra bits for each data entry describe the data held (also shown in figure 4):

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 the whilst in the main cache ;

Valid – the entry may be considered for forwarding.

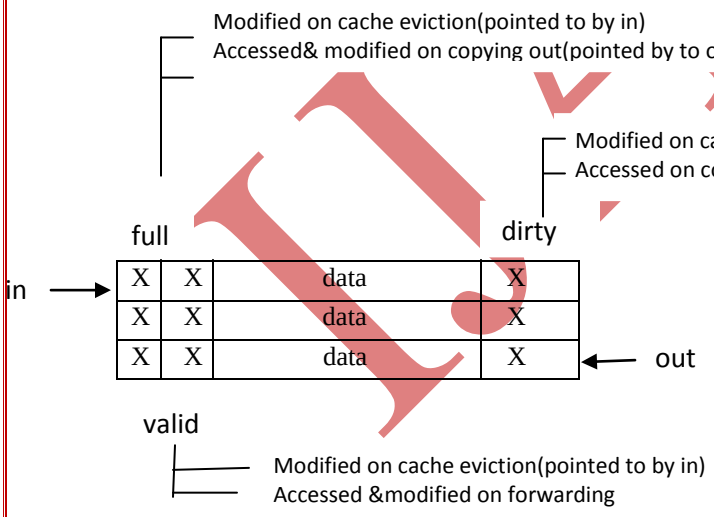


Figure 4 :Victim cache RAM structure

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 co0mpetes 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 the 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 forward able line matching any address. This removes the need for prioritization logic to guard against the (unlikely, but possible) chance that a line I 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 and 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 further sections.

V VICTIM CACHE OPERATIONS

The cache operations involved in forwarding are shown in figure 4 .Address (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 ,data can be forwarded directly from the victim cache (6*) instead of fetching the line from the 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 can be extended as illustrated in figure 5 . 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.

VI VICTIM CACHE BENEFITS

Figure 7 below 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. 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. In this approach 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.

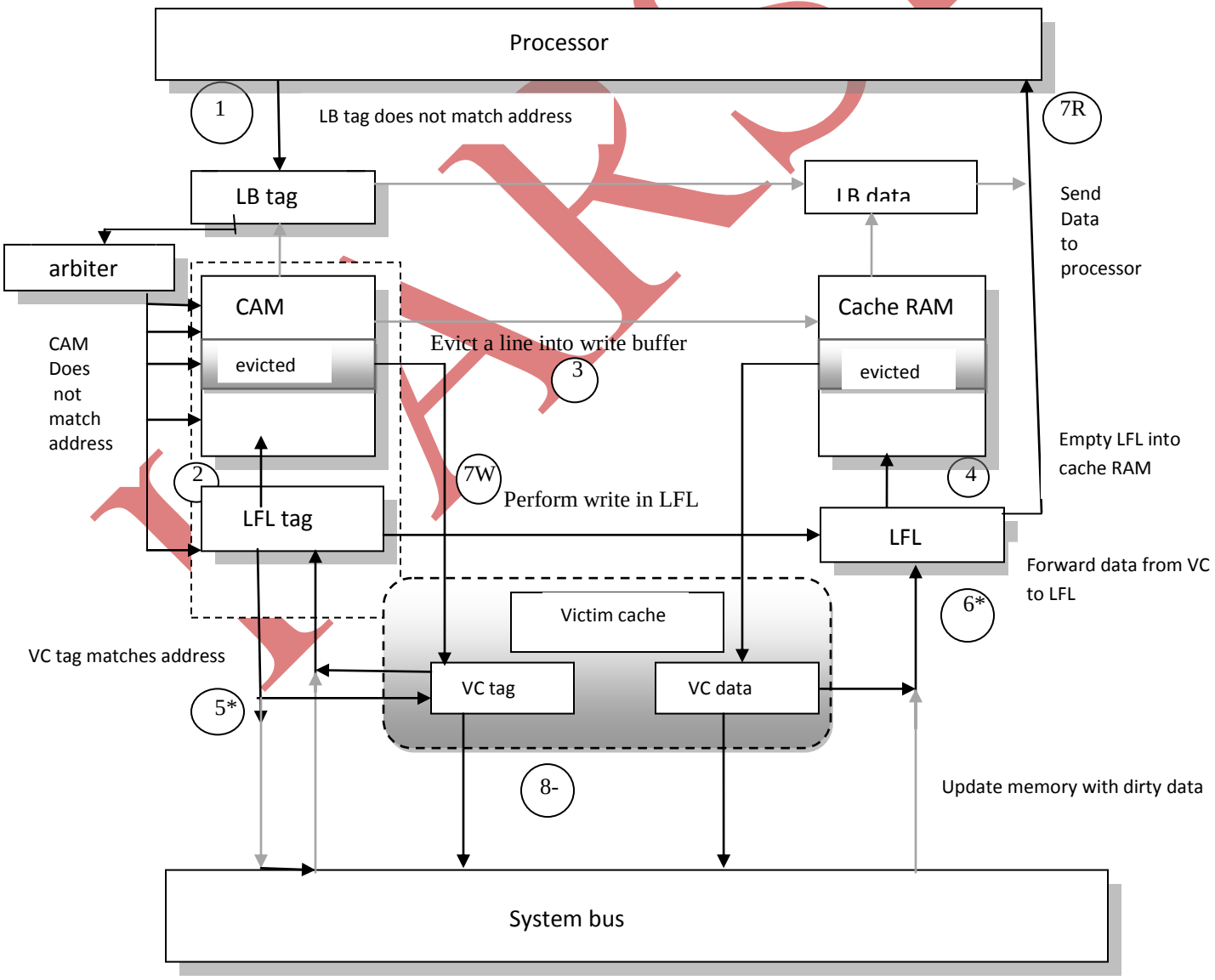
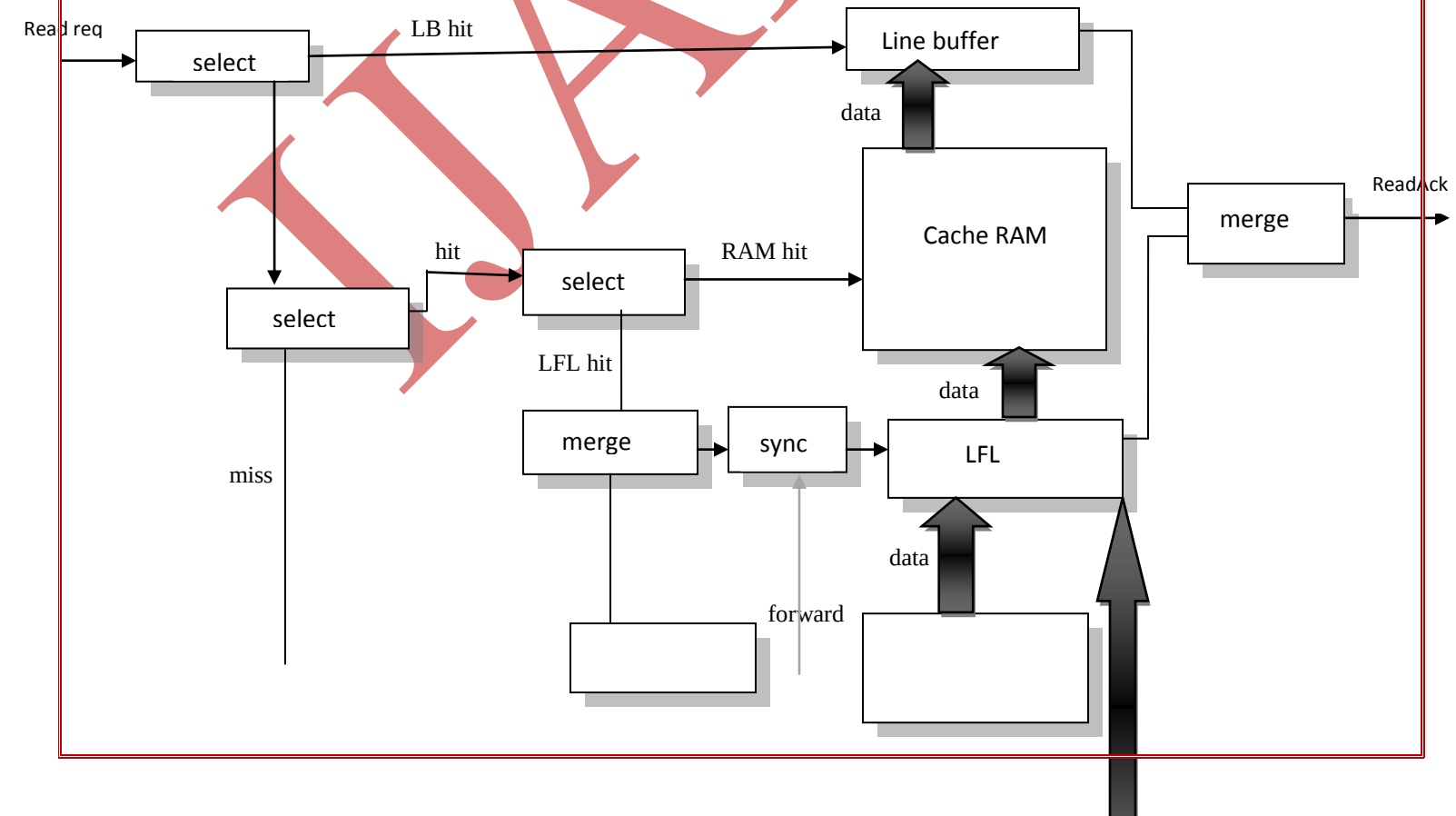


Figure 5 : Cache Forwarding Operations

VII DEADLOCK AVOIDANCE 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 victim cache become full. This is shown in figure 8 . 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 a deadlock if the memory is busy performing the read (and cannot service the memory writes). 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 where tokens corresponding to the victim cache locations are circulated. Initially, the allowed number of tokens is placed in a pool and then one is claimed before each eviction can begin. The tokens then reside in the victim cache until the copy out process returns them to the write buffer throttle. As there is one fewer token than the victim cache locations, eviction will always stall before the last victim cache entry is filled.



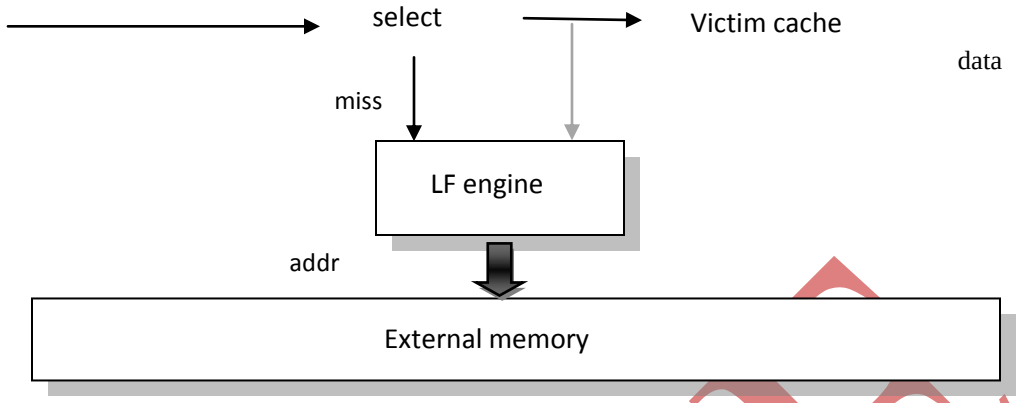
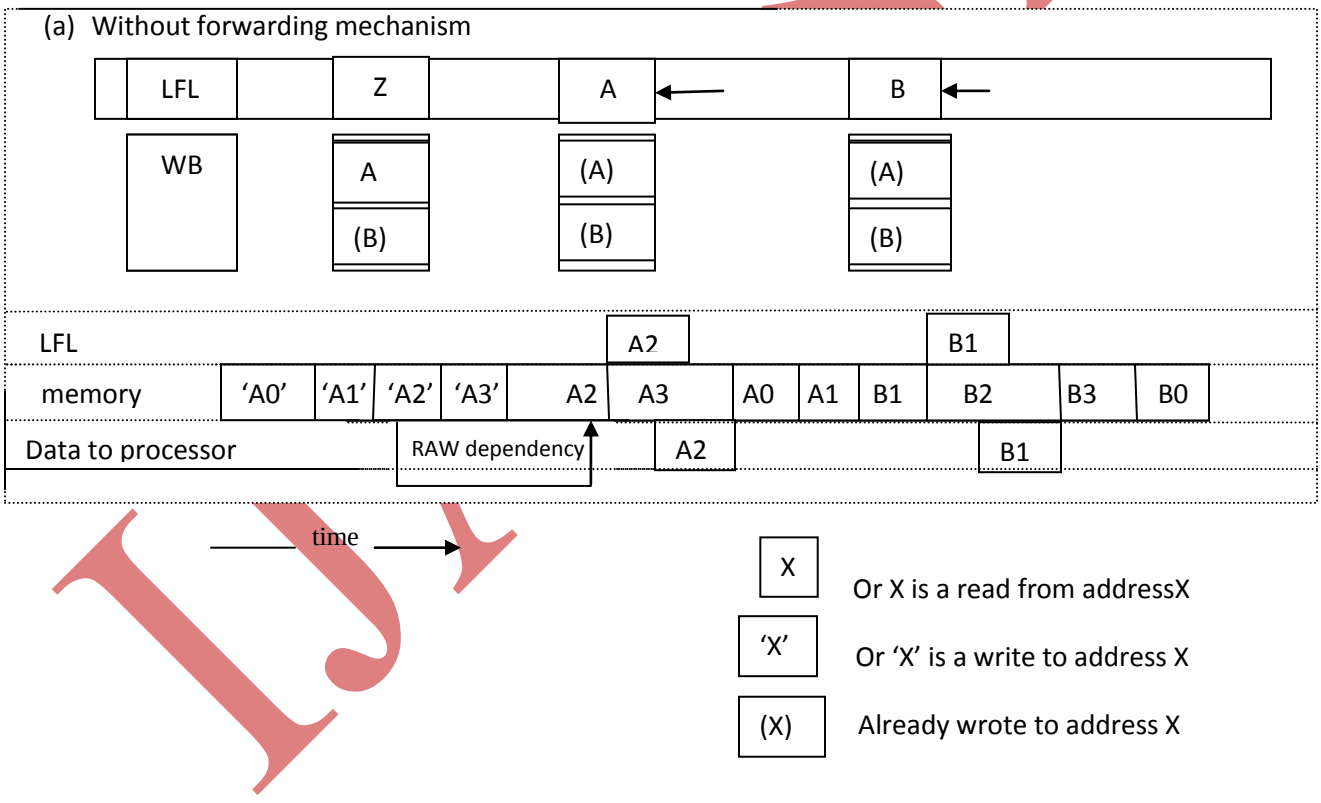
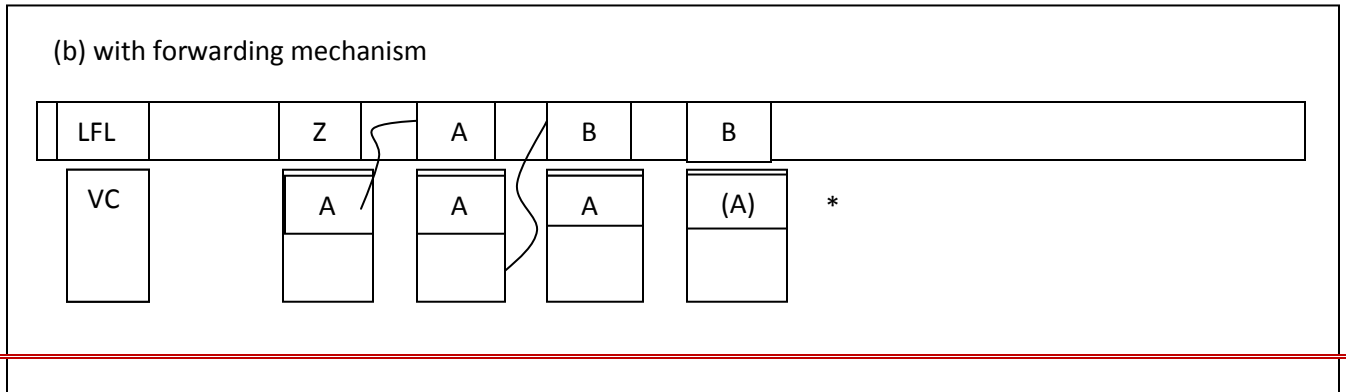


Figure 6 : Cache read request control flow with forwarding





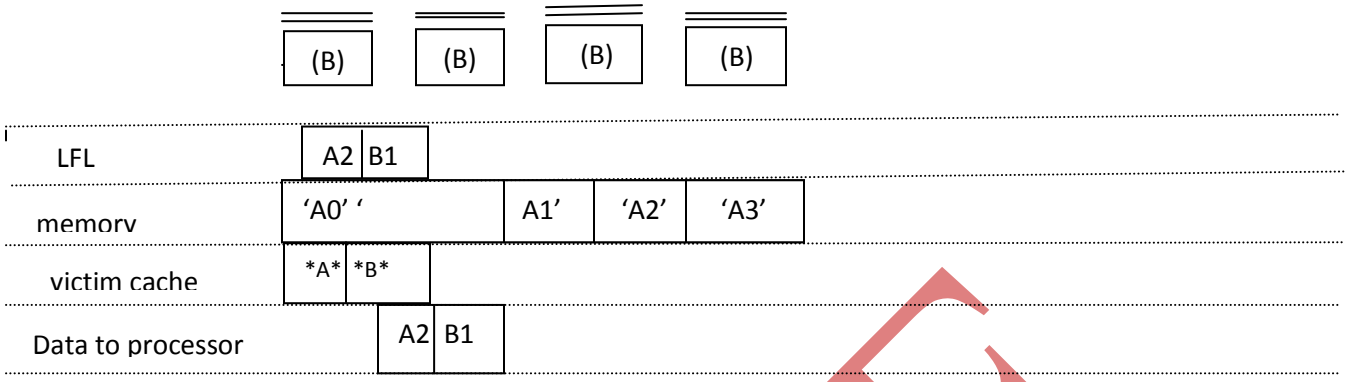


Figure 7 : Illustration of benefits of Forwarding.

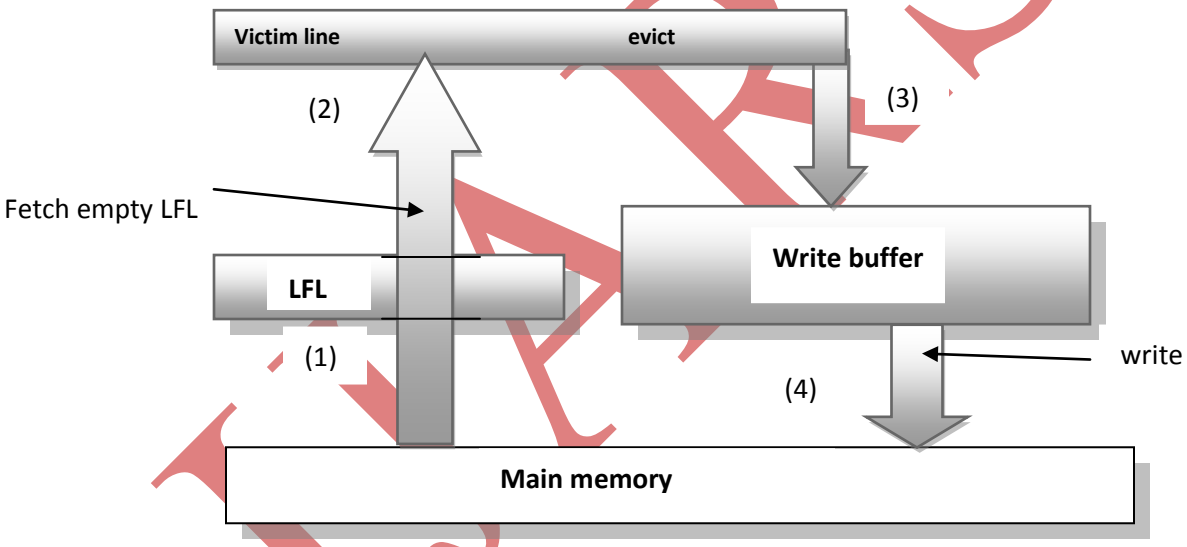


Figure 8 : Illustration of Deadlock Situation

VIII EXTENDING THE VICTIM CACHE TO REDUCE WRITE TRAFFIC

Figure 7 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 man 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 synchronization of the forwarding and copy-back processes before forwarding in performed for any data. Unfortunately, this synchronization 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.

IX VICTIM CACHE DISTRIBUTION

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: centralized 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.

9.1 Centralized victim cache

Having a centralized 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 entries.

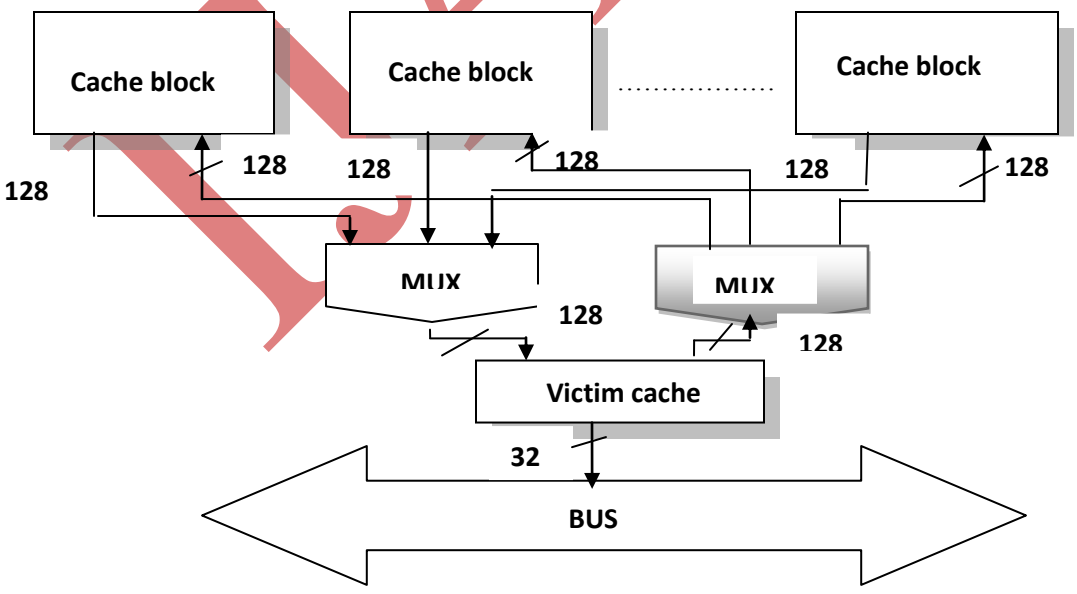


FIGURE 9; Centralized and shared victim cache

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

9.2 Distributed victim cache

For a cache system divided into N blocks, to provide the same total storage as the centralized 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 centralized scheme.

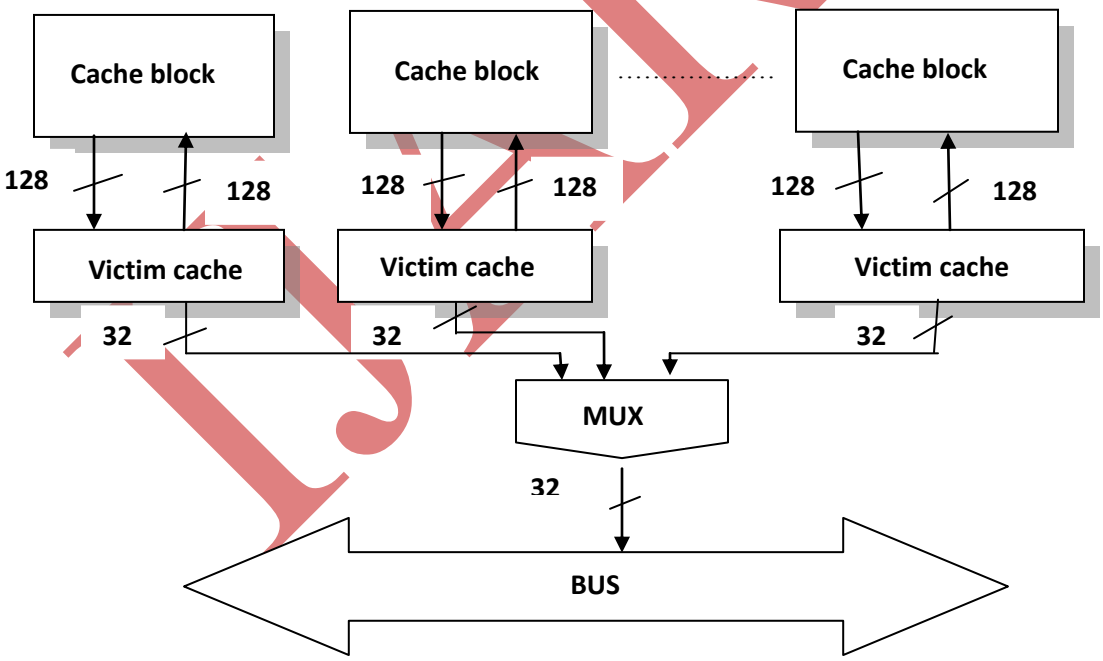


FIGURE 10: Distributed and localized victim cache

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, offers two further advantages over the centralized victim cache scheme. The first is cheap wiring using short, narrow (32 bit) local copy-back and forwarding paths. The second is that the multiplexing process becomes non-critical to performance. However, the 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 (bold) and disadvantages (unbold) summarized in table 11, some of which will only be quantifiable when layout is produced.

	Centralized 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

Table11: Benefits of distributing the victim cache

X CONCLUSION

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 paper 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 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 are discussed.

REFERENCES

[1] 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.

[2] 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.

[3] 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. 17 – 27.http://www.cs.wisc.edu/~markhill/spec92miss.html

[5] Johnson, Teresa L., and Hwu, Wen-mei W. *Runtime Adaptive Cache Hierarchy management via Reference Analysis.* ISCA '97, Denver, CO, USA, pp. 315 – 326.

[6] Jouppi, Norman P, and Wilton, Steven J. E. *Tradeoffs in Two-Level On-Chip Caching.* Research Report 93/3, October 1993, Compaq Western Research

[7] Steven Przybylski, Mark Horowitz, John L.Hennessy. *Performance Tradeoffs in Cache Design.* ISCA 1988: Honolulu, Hawaii, USA, pp. 290 – 298.

[8] Cheriton, D. R., Goosen, H. A. and Boyle, P. D.*Multi-level shared caching techniques for scalability in VMP-M/C.* Proceedings of the 16th annual International Symposium on Computer Architecture, 1989, pp. 16–24.

[9] Short, Robert L., Levy, Henry M. *A Simulation6Study of Two-Level Caches.* Proceedings of the 15th Annual Symposium on Computer Architecture. May 1988, pp. 81–88.

[10] Wan, Marlene, and George, Varghese. *Effect of Second Level Cache Parameterising Overall CachePerformance.*http://infopad.eecs.berkeley.edu/~varg/CS252_report /Final.html

[11] Hill. *A case for Direct Mapped Caches*. Computer,21:12. 1988, pp. 25-40.

[12] Smith, A. J. *Cache Memories*. Computing Surveys,14:3. September, 1982, pp. 473-530.