# An Asynchronous Victim Cache

D. Hormdee, J.D. Garside, S.B. Furber
*Department of Computer Science,*
*The University of Manchester,*
*Oxford Road, Manchester M13 9PL, UK*
*{hormdeed,jdg,sbf}@cs.man.ac.uk*

**Abstract**

*Memory bandwidth is a limiting factor with many modern microprocessors and it is usual to include a cache to reduce the amount of memory traffic. Of the two commonly used cache write-policies, the copy-back approach is better than the write-through approach in this respect. The performance of both approaches can be further aided by the inclusion of a small buffer in the path of outgoing writes to the main memory, especially if this buffer is capable of forwarding its contents back into the main cache if they are needed again before they are emptied from the buffer. This is what is known as a victim cache.*

*For an asynchronous microprocessor it is logical that the cache system should be asynchronous as well; since a large degree of the flexibility of an asynchronous microprocessor would be lost if it were to use a standard synchronous memory interface. However, implementing a forwarding mechanism in an asynchronous system is more difficult because the data to be forwarded is flowing in a manner unsynchronised to the process which requires it.*

*This paper presents an architecture for a victim cache to resolve forwarding in a totally asynchronous environment. The resultant structure forms a key part of an asynchronous copy-back cache system for the Amulet3, a third generation asynchronous implementation of the ARM processor.*

Keywords: {victim cache, copy-back cache architecture, asynchronous design}

## 1. Introduction

The function of a cache is to alleviate the disparity between the processor's memory bandwidth requirement and the bandwidth provided by the main memory devices. It does this very effectively. However as the speed mismatch increases it becomes more important to reduce the traffic between the cache and the memory especially if the primary cache resides on-chip and the memory does not.

Many early caches used a *write-through* policy, where every write operation was conducted to the memory whether the addresses were cached or not. As many locations are written to several times in close succession this traffic can be reduced by suppressing all but the 'last' write operation. This is implemented by the *copy-back* policy which defers writing to memory until the cached copy is about to be overwritten. Multiple successive write operations are therefore combined, with a consequent reduction in memory bandwidth requirement. Previous studies have proved that the copy-back policy provides superior performance by reducing write traffic and power in a uniprocessor system [11] though it imposes an increased hardware cost.

Further advantage may be gained by using a *write buffer* [7] to defer write operations until the bus is not needed for a (more urgent) read operation without delaying the processor; this does not diminish the bus traffic but helps to reduce average read latency.

Another method of reducing memory traffic is to improve the cache hit rate. This can be done by:

- increasing the cache size
- increasing the associativity
- improving the cache utilisation

The first of these is easy, but has a significant cost. Studies have shown that the hit rate increases very gradually with increasing associativity above 8-ways [7]. However even if both of these parameters are fixed the hit rate can still be improved by using the cache more effectively; this is primarily controlled by the choice of which lines are cached, which is in turn determined by the choice of which lines are rejected from the cache when a new line is fetched.

Various selection algorithms have been tried but perhaps the most often cited 'best' policy is the Least Recently Used (LRU). Unfortunately LRU is difficult to implement efficiently for high-associativity caches and simpler algorithms are often adopted. One very easy-to-implement algorithm is to use pseudo-random rejection; this can work surprisingly well, but suffers as it can arbitrarily select a

much-used line for rejection, imposing both a bandwidth and latency penalty as the line is written out and then refetched.

A victim cache [10] can vastly alleviate this by storing a small selection of ‘rejected’ lines which can be brought back into the main cache with very little penalty if they are required again. Thus if the ‘wrong’ line is chosen (by any allocation algorithm) it is likely to be salvaged before it is lost altogether from the cache system. The net effect is an improvement in the overall choice made in cache line rejection. The victim cache can also double as a write buffer, holding rejected lines until the bus is available.

Victim caching and forwarding mechanisms have been implemented in many synchronous systems. In an asynchronous cache system other new implementational problems are introduced as the processor may be completely desynchronised with the bus traffic. This paper describes a justification for, and a possible implementation of, a victim cache and write buffer in an asynchronous environment.

## 2. Asynchronous cache systems

There is much literature from the past two decades describing synchronous cache organisations across the whole range of complexity and many strategies, mechanisms and architectures. However there are very few asynchronous caches and little is understood about this area.

Recent research [3] has shown that asynchronous microprocessors can offer lower power consumption and better electromagnetic emission profiles than their synchronous counterparts, but there have been 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. For instance, when a request is sent from the processor to the cache, the response time can be different (and unrelated to the system bus speed) depending on where in the cache system the data currently resides.

There are only a few related previous implementations in this area including the Amulet2e cache system [5], the TITAC-2 cache system [17] and the pipelined caches in asynchronous MIPS R3000 Microprocessor [13], all of which were single-ported and used a write-through strategy.

## 3. Cache architecture and its operation

To investigate a victim cache in an asynchronous environment a previously developed copy-back cache architecture [9] is used. This is briefly described below.

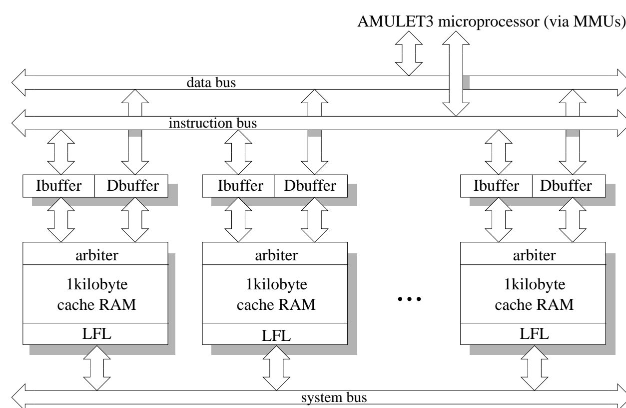


**Figure 1. Cache block organisation (after [4])**

Some constraints are placed on the cache by the processor architecture and its usage. The on-chip cache system (figure 1) employed here has to be unified but dual-ported to accommodate the Amulet3 processor [4], a third generation asynchronous implementation of the ARM architecture [1]. This has a Harvard-style memory interface but requires a unified memory map.

To save power and reduce the possibility of instruction and data conflicts the cache is subdivided into eight addressable 1-kilobyte blocks. Each block is identical comprising a number of cache lines and, in this model, is a fully associative CAM-RAM structure.

Within each block there are two *line-buffers* [8], one for each port. These effectively form another, split cache which can return data faster than the main cache, reduce the probability of conflicts further and reduce the energy dissipated by a cache access.

The other unusual feature of the cache – inherited from an earlier design – is the *Line Fetch Latch* (LFL) [14] which is used to allow the desynchronisation of the processor and line fetch process. The LFL acts as an extra cache line and has its own tag. When data is fetched it is streamed into this latch and picked up as required by the processor. The processor can then continue to use other parts of the cache unhindered. Therefore the cache is *non-blocking* [12][15] and implements *hit-under-miss*, meaning cache hits can be serviced even if a line fetch is still progress.

When a subsequent line fetch is needed the LFL must be copied into the main cache. In the write-through cache developed for Amulet2e [5] the line fetch simply overwrites an existing line in the RAM. In a copy-back cache the line to be overwritten must first be read so that – if necessary – it can be copied back to the main memory. It can then be determined if this line is ‘*dirty*’, i.e. if it has been modified whilst cached, and therefore needs to be copied back into memory. While waiting for the bus to be free, this line must be buffered.

## 4. Write buffering

By definition, 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 to create space for the newly fetched data. For obvious performance reasons any write should be performed *after* the read since the processor is waiting for the new data. This requires additional storage to hold the ejected line which is known as a *write buffer* [7].

The simplest write-buffer scheme has storage for a single cache line. Each time there is a cache miss this is updated whilst a new line is fetched; it is subsequently emptied into memory if it is 'dirty' or simply marked as 'done' if the write would be superfluous.

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*). Clearly this requires more than one slot in the write buffer. Whilst fairly straightforward to implement 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. *Arbitration* is required to make this decision which leads to non-deterministic (but correct) behaviour.

Allowing reads to overtake writes also risks incurring Read After Write (RAW) hazards [7]. 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. However 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 draining the write buffer before a read is performed or forwarding the required data to the processor directly from the write buffer. 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 (figure 2).

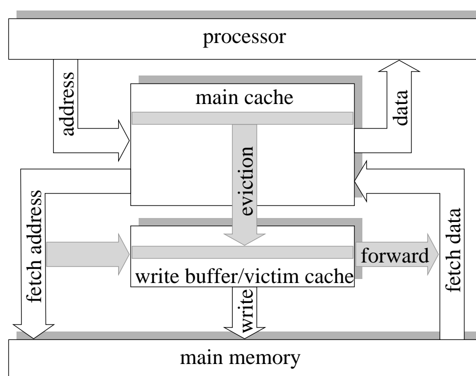


**Figure 2. Write buffer/victim cache position**

## 5. Victim cache

The victim cache was proposed by Jouppi [10] 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.

The victim cache can be considered as a memory with three different functions acting upon it:

- **Cache eviction** – a cache miss occurs and the main cache empties a line into the victim cache. 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, freeing space for re-use.
- **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.

However, there are only two independent, concurrent processes among these activities (filling and draining the victim cache) 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.

### 5.1. Victim cache implementation

Gilbert presented an asynchronous implementation of a reorder buffer [6] intended for use in a processor register bank, which 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 to the time it is overwritten. A similar approach can be used here, with the simplification that inputs and outputs are always in the same order.

The write buffer is a circular FIFO; write operations are made to the 'head' of the buffer and the write process strips entries from the 'tail' whenever the bus goes idle. The useful property of the circular buffer is that the data does not move and so can be read and forwarded despite the fact than another asynchronous process may be reading the data concurrently. This lifetime of the 'forwardable' data is fixed by the number 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 is maintained by the

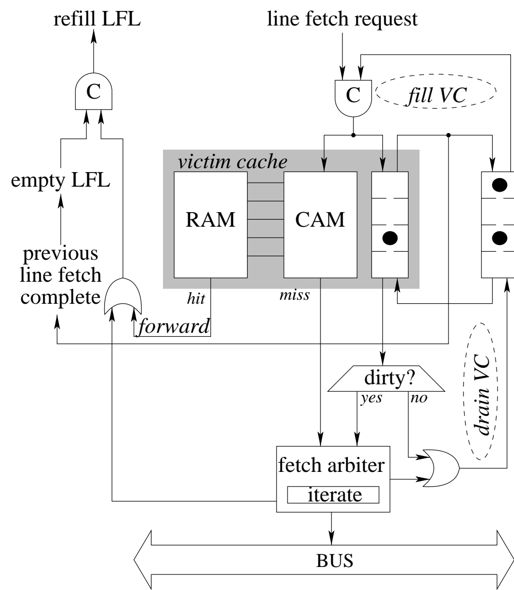


**Figure 3. Victim cache operation**

instruction decoder 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 forward 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 'head' of the queue must be excluded from the 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 line than it has storage locations.

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 (CAM) and their corresponding data are held in the data store (RAM). Before reading external memory a line fetch must be compared with these address tags and, if a match occurs, the data can be forwarded instead of fetching the line from the memory. This does not interfere with the (asynchronous) process of writing to the memory which may not have started, may be in progress or may have completed at this time. The cache line is therefore 'cleaned' and does not need writing again unless subsequently modified.

When forwarding, the line fetch process appears 'short circuited' and can occur in a single, on-chip cycle rather than four, slow bus cycles. This makes line fetch an asynchronous process with a highly variable delay!

Figure 4 shows an entire cache read control flow in this architecture including forwarding from the victim cache.

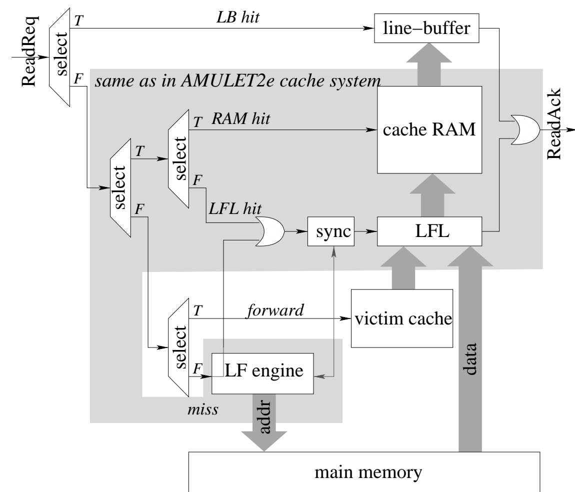


**Figure 4. Cache read control flow**

## 5.2. Victim cache storage

Three types of information are stored in each line of the victim cache: the address is held in a tag CAM allowing fast parallel look-up checks; the data are held in RAM, and a number of additional control markers, such as the 'dirty' bit must also be kept. There are also global 'head' and 'tail' pointers.

Two of the main independent processes, writing into and emptying out the victim cache, are steered by the head and tail respectively. Three extra bits for each data entry describe the data held:

- **Full** – the entry has been filled but not yet copied-out
- **Dirty** – the entry should be copied into the memory
- **Valid** – the entry may be considered for forwarding

When a line of data – along with its 'dirtiness' – arrives it is stored in an empty slot indicated by head. Head then moves on to the next slot. The valid and full bits are set.

The concurrent tail process waits for an entry to be full and then checks its 'dirtiness'. If it is dirty the process competes for the bus and updates the memory, otherwise this phase can be skipped. Lastly the full bit is cleared to indicate that the write phase is complete and the tail moves forward to the next entry. Note that this proceeds regardless of any, possibly concurrent, forwarding.

The function of the valid bit is to prevent the wrong data being forwarded. It is cleared at start-up when the victim cache is empty and the tag fields are undefined. However the valid bit 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 there can be, at most, one forwardable line. This removes the need for prioritisation logic to guard against the (unlikely, but possi-

ble) 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 flag 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 (depending on the dirty bit) regardless whether the data has been forwarded.

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.

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 5.4.

The eviction and copy back processes are independent and largely decoupled, although the head must not '*lap'* the tail. In practice the constraint is slightly more strict as is illustrated below.

### 5.3. Avoiding deadlock by using token queue

Once reads are allowed to overtake writes, there is a potential deadlock on cache line allocation in a copy-back cache because the victim cache can fill up. When the line fetch engine asks for data from the memory, the memory tries to send the data to the LFL. However the LFL must be emptied before it can store the newly fetched line. To empty the LFL requires allocation of a line in the cache RAM which must first be emptied into the victim cache before the LFL can be read. If the victim cache is full, a line must be written from it into the main memory, requiring the memory bus. This results in deadlock if the memory is busy performing the read.

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* [6] where tokens corresponding to the victim cache locations are circulated (figure 3). Initially the allowed number of tokens are placed in a buffer and 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 pending queue. As there is one fewer token than victim cache locations eviction will always stall before the last victim cache entry is filled.

### 5.4. Arbitration for the output bus

If a line fetch has evicted a dirty line there will be data in the write buffer waiting to be written. In a simple system this could be queued to be the next bus transaction, and this would be a wholly deterministic (i.e. arbitration free) mechanism. However, it is quite plausible that a second cache miss could occur before the first line fetch is finished. In this circumstance it is desirable for the fetch to overtake the 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. This therefore requires an arbiter [16] to decide whether it preempted the write starting. Because most standard asynchronous arbiters work on a 'first-come-first-served' basis – and the write will certainly have arrived first – this circuit needs to be specially biased to grant a read if at all possible. This arbiter is the only point of non-determinism needed in this scheme.

An extra optimisation could be made by detecting that forwarding has been performed *before* a write out (copy-back) has begun. In this case it would be possible cancel the write and instead return a dirty line to the cache. This would reduce the bus traffic a little more, but the cost in complexity is considerable. Although it has not been thoroughly investigated it seems probable that the extra cost is unlikely to be justifiable.

### 5.5. Distribution of the victim cache

As described in section 3 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 (figure 5a) or distributed amongst the blocks (figure 5b).

The choice for implementation is not an obvious one, both schemes have advantages (white) and disadvantages (shaded) as shown in table 1, some of which will only be quantifiable when layout is produced. In order to provide more data for comparison both styles have been simulated in the following section.

## 6. Modelling and simulation

Different cache designs (along with the Amulet3 core) were modelled and simulated using a functional model written in LARD [2]. LARD is a language with communication primitives, and it is less cumbersome to model asynchronous systems in such a language than it is to model the timing information of each communication path (provided by local request and acknowledge signals) as is required by

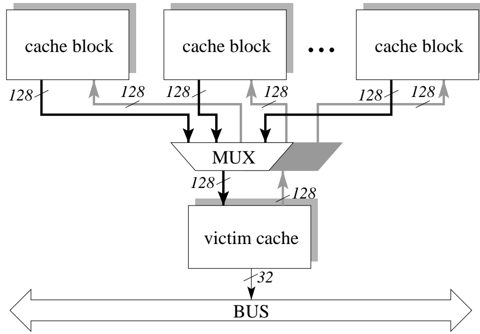


**Figure 5a. Central victim cache**

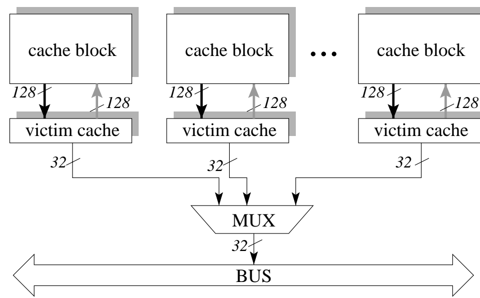


**Figure 5b. Local victim cache**

conventional hardware description languages such as VHDL and Verilog.

Absolute calibration of the models is difficult because the relationship between unit delay and the overall system performance is much more complicated in asynchronous systems. Dependencies between units mean that some delays can affect performance in other parts of the system so that their effects are magnified, whilst others are masked and have no overall effect. Moreover, delays can have one effect at one time and a different one at another time depending on the characteristics of the tasks.

When working with behavioural models early in the design process, measuring absolute performance is less significant than being able to understand the underlying influences on the system performance. It is most important to be able to identify those units which have the greatest effect on overall performance and focus on those regions.

| program name | description |
|---|---|
| Espresso | a two-level Boolean function minimisation program |
| STcompiler | a C compiler program |
| Sim | a program for local similarities with affine weights |
| Da | a heap implementation of Dijkstra's algorithm |
| DES | a fast and portable DES en/decryption program |
| Blackjack | a playing and betting strategies in blackjack program |
| Whetstone | a converted Whetstone double precision benchmark |
| MM | a matrix multiply program |

**Table 2: Benchmark descriptions**

## 6.1. Simulation environment

The cache is to be divided into eight 1-Kbyte cache blocks with lines of 4-words in length. Each block contains separate instruction and data line-buffers and a LFL. Random replacement algorithm is used to choose evicted lines. A copy-back scheme and write allocate policy are adopted.

## 6.2. Benchmark programs

Table 2 gives a brief description of C benchmark programs used in the simulations presented here. These programs were chosen for a number of reasons including the facts that they are fairly simple programs that, whilst easily compiled with existing ARM libraries, display a range of different memory access behaviour and reasonable miss rates. Also they are small enough to be representative of embedded applications and they are not too large for use with LARD simulations.

| | ***N*** **blocks with a central victim cache of** ***V*** **lines** | ***N*** **blocks each with a local victim cache of** ***V/N*** **lines** |
|---|---|---|
| tag comparison | bigger, hence slower tag array | faster |
| restriction on *V* | *V* can be any size, minimum of 1 line | *V* must be an integer multiple (minimum of 1) of *N* |
| wiring cost | 128-bit buses connecting blocks to victim cache are expensive | short local forwarding paths are much cheaper |
| forwarding ability | (*V* -1) lines can be considered for forwarding | (*V* - *N*) lines |
| stalls due 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 blocks (non-critical) victim cache drain |
| multiplexing | in critical path | everything is local |

**Table 1: Benefits of distributing the victim cache**

## 7. Results and evaluations

This section illustrates the advantages of a victim cache architecture with dual-ported asynchronous block-based copy-back caches, and investigates the distribution of such a victim cache. These are described below.

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 cache size, organisation and replacement strategy. Since the total cache size here is fixed to 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 6 contains five sets of results for a variety of total victim cache sizes. The 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 and a central victim cache
- 8-way cache and local victim caches
- 64-way cache[1] and a central victim cache
- 64-way cache[1] and 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.

The rightmost bar in each set shows 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. Each of these bars shows the miss rate of direct-mapped cache and of the fully-associative (in each block) cache.

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

- **Direct-mapped vs set-associative caches** - It is obvious that high associativity caches provide 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. This is simply because there are more conflict misses in direct-mapped caches causing rejection of lines that are likely to be required later. Nonetheless, as shown in figure 6, the advantage of the victim cache also reduces 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 ejected), makes the block-based associative cache the preferred choice.

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

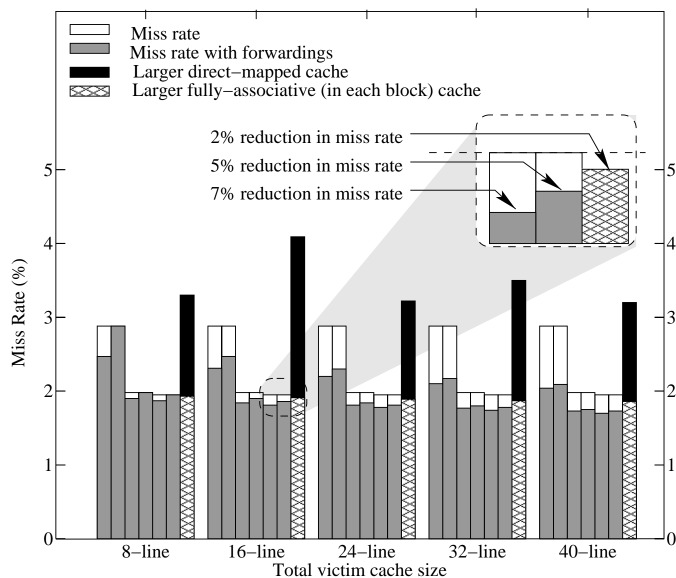


**Figure 6. The effect of the victim cache**

- **Victim cache distribution** - The results show what would have been expected from table 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 caches of this size is at its maximum. 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.
- **Efficient use of resource** - In fully-associative caches, using a few extra lines as a victim cache gives more benefit than extending the cache with them. Although the difference appears small the effect of the victim cache is magnified when considered as a proportion of the miss rate (inset, figure 6). Using a few extra lines of store as a victim cache rather than adding them to the main cache can give reduce the proportionate miss rate by an extra ~5%, the benefit increasing as the victim cache size increases.

  As shown by the dark bars in figure 6, increasing the size of a direct mapped cache only slightly has strange effects on the miss rate, as might be expected in a direct mapped cache which is not a power of two in size.

## 8. Conclusions

An architecture of a self-timed victim cache with a forwarding mechanism suitable for use with a dual-ported, block-based, copy-back cache within an asynchronous environment has been presented. The victim cache has been designed as a key part of the asynchronous copy-back cache system to work with the Amulet3 microprocessor. It is believed to be the first such design.

A forwarding mechanism is used to overcome the coherence problems caused by allowing memory reads to overtake writes, and gives a performance benefit too. In order to forward asynchronously the data in the victim cache cannot be removed by the write process without introducing hazards. This means the last few rejected lines are retained until they are overwritten. Adapting the queue technique, as used in the Amulet3 reorder buffer, makes data forwarding from the victim cache feasible without imposing undue synchronisation in an asynchronous environment.

Although studies are continuing, along with the asynchronous copy-back cache, it is clear that this design will be used in any future Amulet3 cache designs and we hope, in other asynchronous processors. We strongly believe that this is another step in bringing asynchronous processing into parity with the synchronous world.

## 9. Acknowledgements

The authors would like to thank the other member of the Amulet group for their contributions. D. Hormdee is supported by a scholarship from the Royal Thai Government. This support is gratefully appreciated.

## 10. References

[1] ARM Ltd., “ARM Architecture Reference Manual” ARM DDI 0100D 2000.

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

[3] S.B. Furber, J.D. Garside, P. Riocreux and S. Temple,“AMULET2e: An Asynchronous Embedded Controller”, *Proc. of the IEEE*, 87(2), pp. 243-256, February 1999.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] N. Jouppi, “Cache Write Policies and Performance”, In *Proc. Int. Symp. Computer Architecture (ISCA’93)*, 21(2), pp. 191-201, 1993.

[12] D. Kroft, “Lockup-free instruction fetch/prefetch cache organization”, In *Proc. Int. Symp. Computer Architecture (ISCA’81)*, pp. 81-85, 1981.

[13] 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.

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

[15] 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.

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

[17] 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.