# AMULET3i Cache Architecture

D. Hormdee, J. D. Garside
*AMULET Group, Department of Computer Science,*
*University of Manchester, Oxford Road*
*Manchester M13 9PL, UK*
*{hormdeed, jgarside}@cs.man.ac.uk*

## Abstract

*This paper presents an evaluation of a range of cache features applied to an asynchronous, dual-ported copy-back cache. The design has been optimised for the AMULET3 asynchronous microprocessor core, but the techniques developed are much more widely applicable. It is shown that using a copy-back cache with a victim cache would gives a noticeable performance improvement on the existing fabrication technology and that the benefits will increase with increasing cache/memory speed disparity. The design presented provides the processor with a unified, dual-ported view of its memory subsystem using multiple interleaved blocks each with separate line-buffers.*

## 1. Introduction

The performance of modern microprocessors is typically limited by their memory systems and it is usual to improve this by providing a memory hierarchy which includes some form of high-speed cache memory. There are many commercial and academic examples of synchronous cache architectures which range from low power caches, suitable for use in embedded systems (e.g. ARM940T and MIPS-X), to larger caches in high performance systems e.g. Intel x86.

Asynchronous microprocessors can offer lower power consumption and better electromagnetic emission profiles [1] than their synchronous equivalents. A number of asynchronous microprocessors have been constructed by various organisations around the world: these include the University of Manchester [2], the University of Tokyo [3] and the California Institute of Technology (Caltech) [4]; however there are relatively few. The primary interest in this work is in cache architecture for embedded processors, but many of the techniques developed should be applicable to larger, high-performance asynchronous caches.

This paper addresses the added complexity of supporting a Harvard-like processor architecture with a unified cache, requiring dual-ported memories capable of handling contention between the two independent asynchronous ports. In addition the implications of fetching and returning cache lines in an asynchronous environment are discussed.

## 2. Cache Fundamentals and Terminology

Cache principles are the same no matter whether the design is synchronous or asynchronous. The following decisions have to be made when beginning the construction:

**Write-Through vs. Copy-Back**

There are two basic cache variants determining how processor write operations are handled by a cache. In a *write-through* cache *all* write traffic is sent to the memory, to maintain coherency; since memory accesses are slow, writes are often buffered (see Section 4.1 – write buffer) to avoid slowing the processor down to the memory speed.

*Copy-back* caches take advantage of the temporal and spatial locality of writes (and reads) to reduce the write traffic leaving the cache; writes are suppressed if the data is cached. *'Dirty'* data is written back into main memory only when it is replaced; a clean line may be discarded on replacement [5].

**Write-Around vs. Write-Allocate**

There are two common options on a write miss: with a *write-allocate policy,* when a write miss occurs, the target line is fetched into the cache; with a *write-around policy* the line is modified only in the memory when a write miss occurs and there is no change to the cache [5].

**Degree of Associativity**

Since a cache entry represents a fragment of a larger store, each cache line has an address (or *tag*) associated with it, indicating which portion of main memory is stored into that cache line at that time [5]. If a block of main memory can appear only in one place (line) in the cache, the cache is known as *direct mapped*; if a block from main



1522-8681/01 $10.00 © 2001 IEEE

memory can appear anywhere in the cache, that cache is known as *fully associative*. When a block from main memory can appear in a limited ($n$) set of places in the cache, then it is known as *n-way set associative*; a *set* is a collection of cache-lines whose tags are checked in parallel [5].

**Line Replacement Strategy**

For associative caches, when a miss occurs, a decision must be taken as to which cache line to place the fetched data in. This can be an important choice because – except at start-up – this involves overwriting/replacing an existing cache line. The three most common strategies are *cyclic*, *Least Recently Used (LRU)* and *random* [5].

**Line Size and Fetch Size**

Although often these two sizes are the same they have different meanings. The *line size* is the amount of data memory in the cache associated with a single tag whilst the *fetch size* is the amount of data retrieved from main memory at one time. The ideal line size depends on the spatial locality of the application, whilst the fetch size depends on the efficiency of memory accesses. Typically, the fetch size is constrained by the number of external pins available. The relationship between line size and the performance is complex with no definitive optimum value [6,7].

**Cache Size**

The benefit of having larger caches is that more accesses can be satisfied from the cache. On the other hand larger caches tend to be (slightly) slower than small ones even when built with the same technology since there is a larger number of gates involved in addressing the cache – this is a reason for multiple levels of cache. Cache size is also limited by the available chip and board space. The performance of the cache is very sensitive to the nature of the workload, so it is impossible to arrive at the 'best' cache size [6].

**Unified vs. Split Cache**

In a *unified cache* data and instructions are located in a single cache. Accesses must be arbitrated in time unless a dual port cache is used. In a *split cache* separate caches allow data and instructions to be fetched simultaneously. This increases the potential memory bandwidth but risks introducing coherence problems between the caches. It also introduces a fixed limit on the apportioning of cache resources; a unified cache will adjust the proportion of cache used for code/data dynamically, yielding better occupancy and better hit rates.

An example of an embedded controller cache is the ARM940T cache system [8]. This comprises four 1-kilobyte instruction cache segments, four 1-kilobyte data cache segments and an 8-word write buffer. Each cache segment is fully-associative and consists of 64 CAM-RAM lines. It supports both write-through and copy-back modes. Although most copy-back caches allocate on write-misses, this particular example does not.

## 3. Previous Asynchronous Memory Systems

For an asynchronous microprocessor it is logical that the cache should be asynchronous as well; a large degree of the flexibility of an asynchronous microprocessor would be lost if it were to use a standard synchronous memory interface. Although synchronous caches are well understood, and comparison techniques to aid their development are well known, much less work has been done on asynchronous caches. Whilst asynchronous processor design has advanced rapidly in the past ten years, only limited attempts have been made to provide the necessary memory support for these processors. Below are some notable works in this area.

### 3.1. Asynchronous Caches

**The TITAC-2 cache system** [9]: The on-chip instruction cache is 8-kilobyte, direct-mapped with eight-words per line. It is *non-blocking* with *streaming*, (i.e. when the required word has been fetched the processor proceeds in parallel with any remainder of the fetch). No data cache is included.

**The Caltech asynchronous MIPS R3000 cache system** [10]: The cache system comprises a 4-kilobyte instruction cache with provision for branch prediction and prefetch and a 4-kilobyte write-through data cache with a write buffer. Both use 4-word cache lines.

**The AMULET2e cache system** [11]: Like many ARM caches this is built from (in this case four) independent 1-kilobyte CAM-RAM blocks with 64 cache lines of four words per line. Each block (set) is fully associative, the whole cache system being 64-way associative. Its write policy is write-through and write-around and it uses random replacement.

It incorporates arbitration free streaming, is *non-blocking*, and implements '*hit-under-miss*' meaning that cache hits can be serviced even if a line fetch is still in progress. To implement this without introducing synchronisation hazards a buffer called the *Line Fetch Latch (LFL)* (placed between the cache RAM and the main memory) was introduced [12].

Two common features of all these caches are that they are single-ported and use a write-through strategy.

### 3.2. AMULET3i Memory

Although not strictly speaking a cache, the RAM subsystem in AMULET3i [13, 14] is also worthy of note, particularly because it implements a unified memory model although the processor has separate instruction and data buses. As shown in Figure 1 (after [14]) it is an 8-kilobyte static RAM, divided into eight 1-kilobyte blocks; each block contains 64 lines of 4 words (one word is 4 bytes).

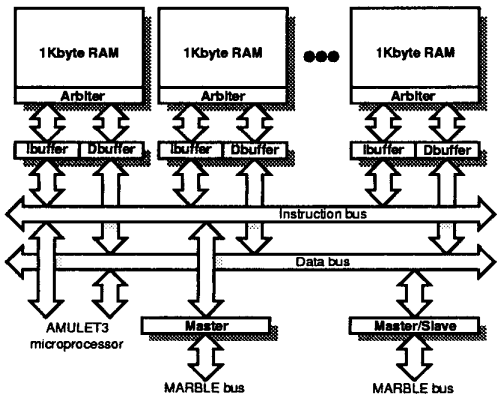


**Figure 1. AMULET3i RAM System**

Separate buses allow concurrent access to different RAM blocks. In a case of contention for the same RAM block each block has an internal arbiter. The block addresses are interleaved to spread loading and reduce contention.

To reduce contention further each RAM block retains the last line read on each port in a *Line-Buffer* [13]. Thus there are limited split caches each being 128 bytes, direct mapped and write through, although these only cache data already in the internal RAM. When using the internal memory many sequential reads can be served by the line-buffer without access to the RAM.

To maintain coherency the contents of the line-buffer are invalidated once there is a write hit in that line and they are replaced when a read hit in the RAM occurs. The line-buffer could be considered as limited 'level 0' cache.

Provision of separate line-buffers for instructions and data avoids interrupting the sequentiality of fetches, especially in the instruction stream. These latches lie after the sense amplifiers and, in fact, are necessary as part of the power-reduction strategy to allow the sense amplifiers to be switched off by a local, self-timed mechanism as soon as the read is resolved. Data is read from the RAM a whole line at a time and latched here. Future accesses may then be able to read data from these (faster) latches without cycling the RAM (and dissipating power). This decreases the average RAM cycle time, a fact exploited by the asynchronous nature of the processor.

## 4. An AMULET3 Cache Architecture

### 4.1. Environment

The first AMULET3 system [13] contained eight kilobytes of RAM (as discussed above) but this was memory mapped and not configurable as a cache. This system does, however, dictate the environment for the cache under development as shown in Figure 2. The major units in this figure perform the following functions:

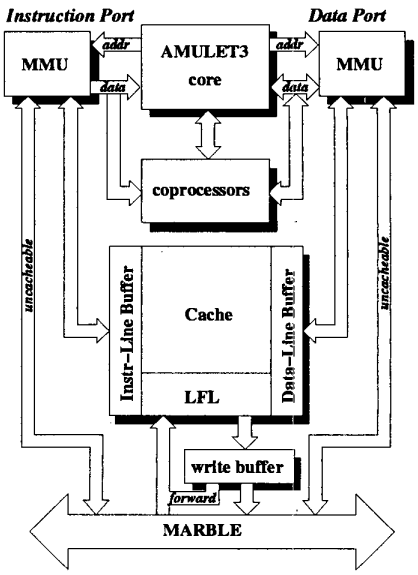


**Figure 2. AMULET3 Cache System**

*AMULET3 core*: the microprocessor in this system is code compatible with synchronous ARM implementations [15]. It has two 32-bit memory ports: the instruction port – which is read-only – and the full-function data port.

*MMUs* (Memory Management Units): located by each of the instruction and data ports, check whether a memory location is cacheable. If it is uncacheable, the memory access bypasses the cache. They also detect memory access permission violation and page faults, signalling these to the microprocessor. (MMUs were not included in the initial AMULET3i system).

*Coprocessors*: are used in the ARM architecture for system management tasks such as programming the MMUs, enabling cache features, locking down cache regions and flushing the cache and write buffer. Many of these operations are not supported here at present.

*MARBLE*: an on-chip asynchronous system bus [16] connecting the MMUs and cache to the other system components and the off-chip memory interface.

*Write Buffer* [5]: A significant write penalty is associated with the write-through and even copy-back strategies which slow the processor down to memory speed. A write buffer, which can accept the write information at a higher speed than the main memory speed, allows the processor to continue to the next task whilst the information in the write buffer is written into main memory.

## 4.2. Basic Architecture

Some constraints are placed on the cache model by the processor architecture and its usage. The cache is to be unified but dual-ported to accommodate the AMULET3 Harvard style memory interface. A number of features from earlier designs can be adopted.

The cache is to be divided into sub-blocks, like the AMULET3i RAM [13], to gain the advantages of modularity and dual-port access; the power consumption is also reduced. Arbitration will be performed only when both cache ports require access to the same block. This (typically) gives split cache performance but guarantees cache coherence. The line-buffer is also considered which yields a two-level cache structure.

Some AMULET2e [11] techniques are also adopted, notably the line fetch latch (LFL) mechanism [12]. To allow high degrees of associativity combined with adequate speed the AMULET2e pipelined CAM-RAM structure has been used (Figure 3) which allows tag look-up and data access processes to proceed in parallel.

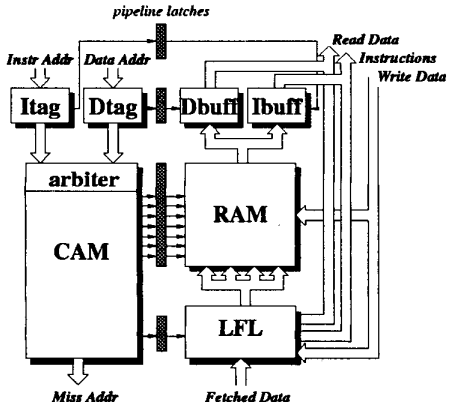


**Figure 3. Dual-Ported Asynchronous Cache Block**

In addition to combining these elements some new features have been developed. The most significant is the design of an asynchronous copy-back mechanism.

Applying the techniques discussed above to a dual-ported copy-back cache results in considerably more complexity. This arises because in a copy-back cache, data written to cached memory locations are retained locally by the cache which has to remember that the affected cache line is dirty. This dirty data then has to be written back to the memory when that line is about to be reallocated. The advantage that this provides is that memory bandwidth requirements are reduced. Extending the write buffer to support forwarding (i.e. it becomes a *victim cache*) further improves the performance and reduces memory bandwidth requirements.

Cache operation is discussed in the following sections.

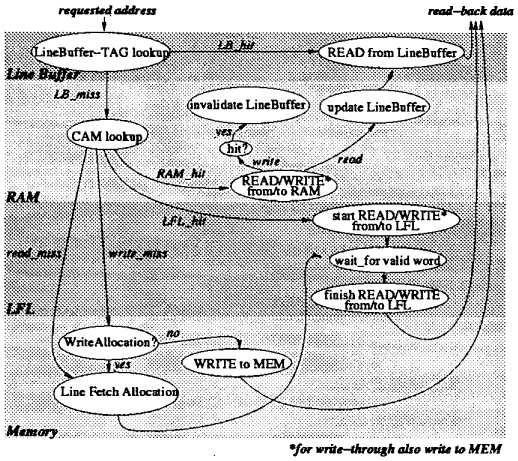


**Figure 4. Cache Operations**

## 4.3. Cache Operations

Figure 4 shows some of the possible activities that may occur for any cache access. The possibilities are:

- A read hit in the line-buffer. This can be satisfied quickly from the appropriate instruction or data line-buffer.
- A write hit on the instruction or data line buffer. This requires the contents of one or both line buffers to be invalidated and the written data stored into the cache RAM (for copy-back operation) or sent to the write buffer (for write-through operation).
- A read or write hit in the cache RAM or LFL. This can be satisfied quickly from the cache. The writes also have to proceed to the main memory in the case of a write-through cache. Reads also cause a line buffer replacement.
- A cache miss. This causes a line fetch and so stalls the processor whilst a memory access is performed by the line fetch process. Once the addressed word is fetched it is returned to the processor immediately and the remainder of the line is fetched concurrently with other cache accesses performed by the processor. Further activity may be necessary in a copyback cache if the line fetch engine has to return a 'dirty' line to memory.
- An uncacheable instruction or data access; passed on directly to the system bus. (Not shown in figure.)
- Cache flush on copy-back cache. The line-fetch engine scours the cache for dirty lines and copies them back to memory via the line fetch port onto the bus. (Not shown in figure.)

Each of these operations has a different characteristic speed. The asynchronous implementation allows each stage to operate to the best of its abilities at its 'natural' speed.

## 4.4. Line Fetch and Allocation

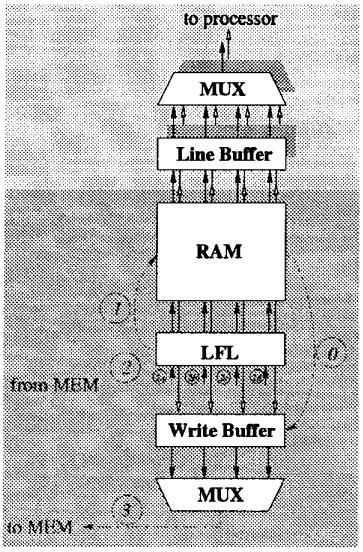


**Figure 5. Dataflow in Line Allocation**

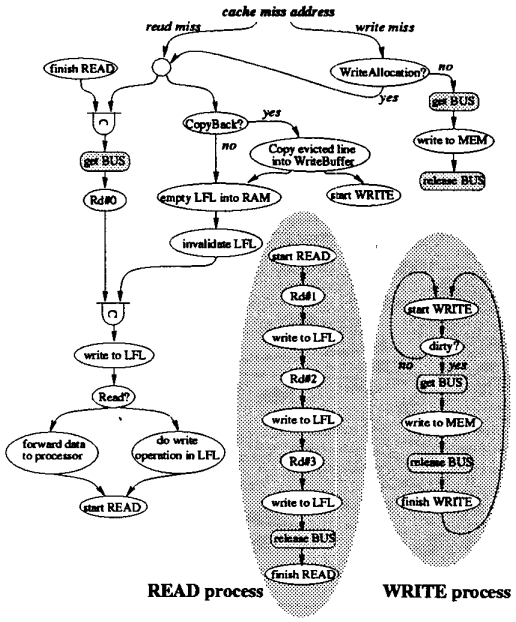


**Figure 6. Line Allocation Activities**

The line fetch mechanism is similar to that used in AMULET2e although the complexity is increased somewhat when a copy-back cache is employed. The key activities of the line fetch, which run partially in parallel with the memory access (as shown in Figure 5), are:

**Activity 0**: Select and reject a victim line from the RAM to the write buffer – regardless of whether that line is dirty or not.

**Activity 1**: Copy the old contents of the LFL into RAM.

**Activity 2**: Stream the fetched data into the LFL. Send the requested word (the first word fetched in a non-blocking scheme) to the processor.

**Activity 3**: Check if the victim line in the write buffer is dirty. If so, write it out to the memory when the bus becomes available.

The write-through cache (e.g. AMULET2e) performed only steps **1** and **2** as it is known that the RAM contents are *'clean'* and can be overwritten. In this case the write buffer (Figure 5) is unnecessary.

These activities have dependency and resource constraints and thus cannot be performed concurrently. However some overlapping is possible, subject to the restrictions that: both activities **0** and **1** must not occur at the same time as they both use the cache RAM. Similarly, both read process **2** and write process **3** cannot be performed at the same time as they both require memory access, although a clean line does not need to be written out and can purge itself from the write buffer during the line fetch. The operations required for line rejection and reallocation are summarised in Figure 6.

The two subprocesses for fetching a line from memory and emptying a line from the victim cache into the memory are shown on a grey background. The remainder of this figure is the main control thread of the cache. It also shows the contention between the line fetch and the writeout of dirty data for the use of the main memory bus.

## 4.5. Write Buffering

The mechanism described above introduced a write buffer. This is necessary as a place to swap out a potentially dirty line to leave space for newly fetched data. Although it would be possible to do without this it would require that the write operation, if needed, preceded the line fetch. Because the processor is waiting for data from the line fetch delaying this process would have a severe performance impact. The read therefore precedes the write to reduce the fetch latency. Both actions are performed every time a new line is fetched, so only one slot in the write buffer is required.

In general reordering state-changing operations is liable to cause hazards. In this case a read is sequenced before a write which potentially risks fetching data before it is modified by the write operation. However in this case this is not possible because the line being fetched caused a cache miss and so cannot be aliased to the rejected line.

With a single line write buffer, the evicted line can be side-lined allowing the read to be performed first. If a second line fetch is required then it must wait until the write

buffer is empty before it can begin, giving the ordering as **R1 W1 R2 W2**. This delays the performance critical **R2** operation. In order to reduce processor stalls when two or more line fetches are required in close succession, memory accesses could be reordered so that all outstanding reads are performed before the writes begin *(read-overtake-write)*. For the above example – two line fetches which both cause write operations – the memory accesses could be performed in the order **R1 R2 W1 W2** (assuming the **R2** request precedes the end of **R1**), resulting in a significant reduction in latency for **R2**. Clearly this requires more than one slot in the write buffer.

Whilst fairly straightforward in the synchronous domain, this can cause problems in an asynchronous implementation because of the lack of synchronisation between the input and output units of the write buffer. It is necessary to determine *if* a read operation is pending before a write burst begins. Because the write and a subsequently requested read are asynchronous, arbitration is required to make the decision between the line fetch process and the write buffer writeout.

Allowing a read to overtake any write other than the one for its corresponding evicted line introduces potential memory coherency hazards since the only write in the buffer that is *certain* not to conflict with the read is the line it evicted. Thus with a write buffer with more than one entry **R2** could clash with **W1**. Solutions to this problem include:

- Do not reorder. The write buffer must be drained before the read is performed. This would not take advantage of read-overtake-write.
- *Forward* the required data to the processor directly from the write buffer if it is fetched again.

Clearly the second option is preferable if some mechanism of forwarding can be provided without introducing hazards in the asynchronous environment.

### 4.6. Forwarding/Victim Cache

Forwarding in an asynchronous system is more difficult than in a synchronous one because the data that is to be forwarded is flowing in an unsynchronised manner to the process which requires it. A possible solution to this was introduced in the reorder buffer in AMULET3 [17] which forwards register values, and a similar technique can be used here. This allows memory writeback to proceed unimpeded but leaves valid data in the write buffer until it is overwritten. Addresses must also be held in the write buffer; before reading external memory a line fetch can 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' in the process.

This not only solves the coherency problem, it can reduce the number of memory cycles by intercepting line fetches of recently rejected addresses – for example due to bad luck with a random replacement algorithm – as lines which are still required will get back into the main cache before they are lost from the processor. The write buffer is now performing the function of a *victim cache*[5].

In this model the line fetch process is 'short circuited' and can occur in a single, on-chip cycle rather than four, slow bus cycles. This leads to an asynchronous process with a highly variable delay!

The line which is being updated in the victim cache need not be considered for the address comparison for forwarding purposes since it will never contain the required line. More seriously it *must* be excluded because the fetch (and, possibly, forward) and the write buffer insertion processes are asynchronous (Figure 6) so the contents of this location may be changing during the comparison process. Therefore the victim cache holds one fewer line than it has storage locations.

**Avoiding Deadlock**

When reads are allowed to overtake writes, there is a potential deadlock on cache line allocation in a copy-back cache because the write buffer (or victim cache) can start to 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 RAM which must then 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 because the memory is busy performing the read. The solution to this problem is to keep one slot in the victim cache empty at all times. One way to implement this solution is to use a token queue as presented in [17] where tokens corresponding to the write buffer locations are circulated, but – in this case – there is one fewer token than write buffer locations.

## 5. Simulation and Evaluation

In order to test different caching strategies and designs, simulation was performed using a functional model written in LARD (Language for Asynchronous Research and Development) [18] consisting of channel-communicating units each modelling separate cache blocks. The cache was initially built using a write-through strategy with an LFL and optional line-buffers. The model was used to allow the evaluation of the modifications to support a copy-back strategy with the victim cache and forwarding mechanism.

The benchmarks used are the same set that were used to

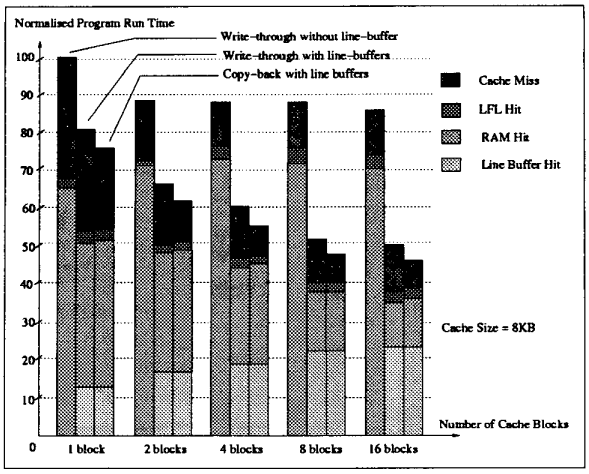


**Figure 7. Distribution of Run Time with/without Line Buffer**

evaluate the forwarding mechanism presented in [17]. Usually the number of cache misses due to writes varies dramatically depending on the benchmark used.

| program name | instr | data read | data writes | total |
|---|---|---|---|---|
| Dhrystone | 14741 | 1850 | 1736 | 18327 |
| ST_compiler | 236580 | 51083 | 51599 | 339262 |
| Espresso | 156257 | 42774 | 26550 | 225581 |

**Table 1:Benchmark detail**

Table 1 contains the details of benchmarks used in this simulation. Both ST_compiler and Espresso have some uncacheable accesses (12652 and 4138 respectively) due to file I/O which have been excluded from these results.

The cache parameters held constant during these simulations were:

- the cache line size (4 words/128 bits)
- memory access time (60ns as per SRAM data sheet[19])
- cache-RAM access time (8ns, same as AMULET3i SRAM)
- latch delay (1.5ns as per cycle time of single-rail long-hold data-path latch controller used in AMULET3i)
- random replacement strategy

To check the expected results a rough calculation was performed to estimate the expected difference between write-through and copy-back caches. The difference between the cache and external memory speeds here is a factor of 7.5; the proportions of different types of memory access is shown in Table 2.

| | Hit | Miss |
|---|---|---|
| Instruction fetch | 66.4% | 2.8% |
| Data read | 18.5% | 0.5% |
| Data write | 11.4% | 0.4% |

**Table 2: Memory Access Types**

(These particular figures are taken from Espresso.)

### 5.1. Copy-Back vs. Write-Through

For a copy-back cache, and assuming a large write buffer, it would be expected that only cache misses (3.7% of cycles) would be slowed to memory speeds. On the other hand a write-through cache will suffer this penalty on all write operations, a total of 15.1% of cycles. With the external memory being 7.5 times slower than the processor 'cycle' it would be expected that the copy-back cache would be limited by the processor alone (3.7% x 7.5 <100%) whereas the write-through cache should be limited by the memory bandwidth (15.1% x 7.5 = 113%).

This is a rough calculation but suggests a benefit of ~10% for the copy-back cache. This figure would be expected to increase rapidly as the disparity between the internal and external cycle increases, as will be the case in future microprocessor implementations.

Figure 7 shows the results of some LARD simulations to compare the (normalised) run times of the chosen benchmarks and contrast some of the cache parameters. The major effect which can be seen is the impact of the line-

buffer (which acts both as a fast 'level 0' cache and helps alleviate the problems of data and instruction fetch collisions. It also shows a small difference (about 10%!) in the performance of the write-through and copy-back caches.

Not having a line-buffer means there will be a full CAM look-up for every cycle (high power consumption) and also more arbitrations due to the two separate ports trying to access to the same cache block (low performance).

Not having *separate* line-buffers for the ports also means increasing arbitrations for instruction and data accesses to the same block for CAM look-up and the line-buffer contents may well get changed when a data access comes in between instruction accesses (or vice versa). This would then give fewer line-buffer hits and more line-buffer updating.

Simulations suggest that dual line-buffers should reduce the accesses to the cache RAM and LFL by ~40% with a resulting decrease in power -- since this also prevents full CAM look-up -- when compared to not having line-buffers.

Figure 7 also shows how varying the number of blocks affects the distribution of hits among the different units in the cache for a range of caches of the same size. The three columns for each group present run time with (from left to right): the write-through cache without line-buffer, the write-though with dual line-buffers and the copy-back with dual line buffers. The most significant impact comes from the introduction of *any* form of line-buffer, but as the number of blocks is increased the total level 0 cache size also increases with noticeable effect.

The effect of 'dual-porting' the cache can be seen in the step in performance in the leftmost columns.

## 5.2. Sequential/Parallel Line-Buffer Tag Look-Up

The line-buffer tag comparison and the CAM look-up could be performed either sequentially – as assumed previously (Figure 4)– or in parallel. Doing them in parallel would *appear* to provide higher performance; however the CAM look-up process follows the arbitration of the two cache buses. As the hit rates on the line-buffers are relatively high (e.g. ~40% of instruction fetches) the arbitration and CAM look-up is frequently unnecessary. Omitting these operations can therefore enhance performance – and save considerable power.

Therefore whilst parallelising these operations would make sense in a single-port cache, in this 'dual-ported' architecture checking the line-buffer tags *before* activating the CAM is noticeably beneficial.

This leads to a very variable association time for the cache. In an asynchronous implementation this does not impose much performance penalty; in a synchronous implementation it could impose a clock cycle overhead.

## 5.3. Cache Structure

A cache hit could occur in the line-buffer, the main cache RAM or the LFL. Since these all have different speeds, the unit in which the hit occurs affects the access time. This can be seen quite clearly in Figure 7.

Enlarging the cache – by adding more blocks of the same size – also has this effect although, interestingly, can be counterproductive in the current model. Figure 8 illustrates this with the (small) Dhrystone program which soon fits entirely in the cache; when line fetches effectively cease the last data fetched is left in the LFLs which require a full CAM access. Because this follows the arbiter it results in an increased access time.

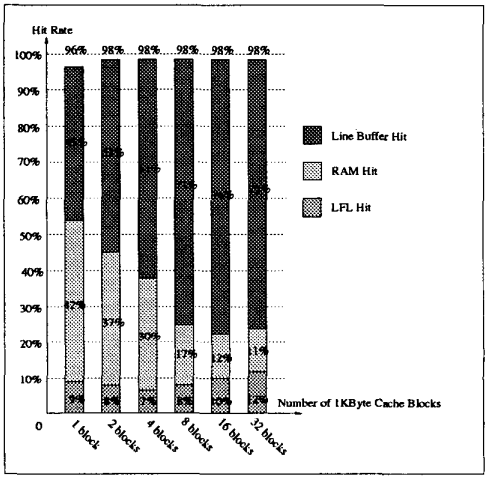


**Figure 8. Distribution of Cache Hits Locations**

It is known that Dhrystone, while quick to run, is a fairly poor benchmark for cache performance. Although this situation could arise for any program small enough to fit in the cache it is a fortunately small effect.

## 5.4. Allocation Strategies

A write allocation strategy assumes that in the near future the processor will access a line that has been recently written; normally a reasonable assumption. Applying write allocation in a copy-back cache also makes the forwarding mechanism from the victim cache easier since there is less control to check whether the write in the victim cache contains a whole valid line (clearly the write from the processor would not be a complete *line* of data).

From the simulation ~12% of (subsequent) hits occur in the lines allocated according to the write allocation. Looking at the data port alone, ~30% of all data hits are on lines allocated due to write allocation.

Figure 9 shows the decrease in write traffic with increasing cache size; if a write impacts an already dirty line this means that a write operation has been averted. Espresso experiences 86% or greater reduction in write traffic by the use of the copy-back cache.

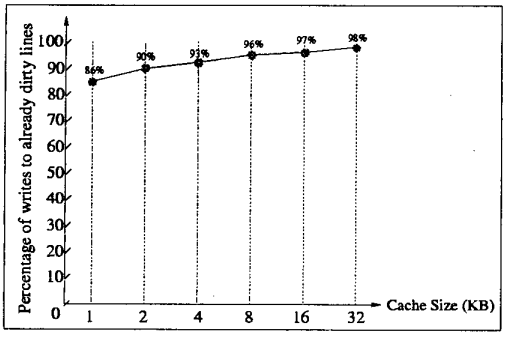


**Figure 9. Proportion of writes to 'dirty' lines**

## 5.5. Write Buffer and Victim Cache

Another issue in the comparison of copy-back and write-through caching is the size of the write buffers. Statistically a copy-back cache will produce less clustered write operations than a write-through cache, so it should not need as large a write buffer.

Both the write-through and copy-back caches require at least a single line write buffer for adequate performance and a copy-back cache *needs* one buffer entry to hold an evicted line.

The required depth of a write buffer is related directly to the number of pending writes which depends on the clustering of write operations as well as the ratio of processor speed to memory speed. It is likely to be fairly small. In a study of a (synchronous) write-through cache 2-4 entries in the write buffer were suggested [20].

The simulation shows that, to avoid pending writes delaying line fetches, Dhrystone requires two lines of write buffer with the copy-back cache, whereas Espresso and ST_Compiler require four lines.

Only where multiple misses with dirty victim lines occur in series would a write buffer with more than one entry be useful with a copy-back cache. However with forwarding the write buffer also acts as a victim cache and increasing the size of this cache is likely to be beneficial. There is a trade-off (hardware resource vs. performance) here since more lines in the victim cache allows more data to be forwarded back to the cache, the main cost being in silicon area.

In terms of victim cache, for Espresso 10% of the line fetches can be forwarded with a five entry cache. 30% of the subsequent cache hits are on this returned information. This obviously reduces memory usage with consequent speed and power benefits.

However, the benefits of forwarding depend on many parameters, including the size of the victim cache, number of cache blocks, block size, replacement strategy, program used to run etc. For example when running the ST_Compiler less than 5% of the misses can be forwarded from the victim cache and only a few percent of subsequent hits are on the returned data.

## 5.6. One or More Dirty Bits Per Line

A dirty bit is used to indicate whether the data is clean (it has not been modified) or dirty (it has been modified). The number of dirty bits per cache line determines the granularity of the interleaving of memory read accesses (to satisfy cache misses) with memory writes from the victim cache (assuming that a read is not allowed to interrupt the writing of a line). Using only one dirty bit per line means that the entire line must be written to memory (even if only one word is actually dirty) whereas the other extreme of using one dirty bit for each word means that only the dirty words will be written (i.e. lower bandwidth), and the latency incurred by stalled reads is lower. The latter approach is obviously more expensive to implement.

| 1 word dirty | 2 words dirty | 3 words dirty | 4 words dirty | none is dirty |
|---|---|---|---|---|
| 12% | 3% | 1% | 9% | 75% |

**Table 3:Dirtiness of Evicted Lines**

Table 3 shows the proportion of words dirtied in the cache. Since, in this simulation, the line size is fixed to 4 words, having a dirty bit per line would take nearly as twice the memory bandwidth as having a dirty bit per word. Although writing back the dirty data to the memory has to be done a word at a time due to the size of the bus, having to write a whole line (four words consecutively) might still be faster than writing four different words according to the sequential writing process in the memory. Further study is required to determine which scheme is more suitable for use with the AMULET3 core.

# 6. Conclusions

An asynchronous, dual-ported, copy-back cache architecture has been presented. It provides a unified view of memory for the Harvard-like AMULET3 core. The cache integrates a number of features used in earlier designs, such as a blocked memory structure, separate instruction and data line-buffers, a non-blocking line fetch mechanism with LFL, and hit-under-miss. In addition new features provide for a copy-back mechanism with write buffering without

imposing undue synchronisation in an asynchronous environment. The mechanism for ensuring memory coherency with the write buffer automatically introduces the facility for including a victim cache.

Although these features have been tailored for a particular processor they are all quite general in application. The case for copy-back caches has already been well demonstrated in the synchronous world; in this model the benefits have proved quite small (~10%) but this is because the speed differential between the cache and external memory is less than a factor of ten; the difference between copy-back and write-through caches is emphasised as this differential increases, as is currently happening. We believe this is the first solution to the problems of a copy-back cache in a totally asynchronous environment.

The need for a write-buffer to equalise bus loads – and allow writes to be deferred in favour of more urgent reads – is also demonstrated. The need to ensure memory coherency has forced the adoption of forwarding from this buffer. In order to forward asynchronously the data in the write buffer cannot be removed by the write process without introducing hazards. This means the last few rejected lines are retained indefinitely and therefore become a victim cache.

The line-buffers act as an extra level of cache and therefore deliver added performance; however in this implementation their most important role is to allow the separate instruction and data buses the maximum freedom of access to the memory. This has influenced their tags insofar as they are checked *before* beginning a main cache access. Line buffers would also give a benefit in a single bus cache, although in this case it could be better to run them in parallel with the tag look-up. As was demonstrated in the previous AMULET3 system the line-buffer yields slightly faster memory cycles than the cache, a feature which can be uniquely exploited in an asynchronous system.

Other features – notably as the LFL from AMULET2e and the blocked cache from ARM3 – have also been retained; these give, respectively, a fast forwarding and hit-under-miss capability and significant power reductions.

Although studies are continuing, it is already clear that these designs will be used in any future AMULET3 cache designs and, we hope, in other asynchronous processors. We believe that this is yet another step in bringing asynchronous processing into parity with the synchronous world.

## 7. References


[1] S.B. Furber, J.D. Garside, P. Riocreux and S. Temple,"AMULET2e: An Asynchronous Embedded Controller", Proceedings of the IEEE, February, 1999

[2] AMULET Group, URL http:// www.cs.man.ac.uk/amulet/

[3] Nanya-Nakamura Laboratory, URL http://www.hal.rcast.u-tokyo.ac.jp/titac2/

[4] Caltech Group, URL http://www.async.caltech.edu/

[5] J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, 1996

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

[7] A. Smith, "Cache Memories", ACM Computing Surveys, September, 1982

[8] ARM940T Technical Reference Manual, Technical Report No. ARMDD10144A, February 1999, Advanced RISC Machines Ltd. (ARM)

[9] A. Takamura et al., "TITAC-2: A 32-bit Asynchronous Microprocessor based on Scalable-Delay-Insensitive Model", Proceedings of ICCD'97, pp. 288-294, October, 1999

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

[11] J.D. Garside, S. Temple and R. Mehra, "The AMULET2e Cache System", Proceedings of Async'96 Aizu-Wakamatsu, Japan, March, 1996

[12] 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

[13] J.D. Garside, W.J. Bainbridge, A. Bardsley, D.M. Clark, D.A. Edwards, S.B. Furber, J. Liu, D.W. Lloyd, S. Mohammadi, J.S. Pepper, O. Petlin, S. Temple and J.V. Woods, "AMULET3i - an Asynchronous System-on-Chip" Proceedings Async 2000 pp. 162-175 IEEE Computer Society Press April, 2000 (ISSN 1522-8681 ISBN 0-7695-0586-4)

[14] "AMULET3H - 32-bit Integrated Asynchronous Microprocessor Subsystem", AMULET Group, University of Manchester, UK, version 1.0, 2000 URL www.cs.man.ac.uk/amulet/

[15] ARM Ltd., "ARM Architecture Reference Manual" ARM DDI 0100D 2000

[16] W.J.Bainbridge, "Asynchronous System-on-Chip Interconnect", PhD. thesis, Department of Computer Science, The University of Manchester, 2000

[17] D.A. Gilbert and J.D. Garside "A Result Forwarding Mechanism for Asynchronous Pipelined Systems", Proceedings of Async'97 Eindhoven, The Netherlands, April, 1997

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

[19] Hitachi HM514100 Series Datasheet

[20] A.J. Smith, "Characterising the Storage Process and Its Effects on the Update of Main Memory by Write-Through", Journal of the ACM 26(1) pp. 6-27, January,1979