TSM_EmbHardw

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

03 – Cache / Cache Aware Programming / Scratchpad Memory / DMA

Hans Dermot Doran: Institute of Embedded Systems -

V 1.0	20.11.2011
V 1.1	15.11.2012: Minor changs – spelling mistakes etc
V1.2	19.11.2012: Post-lecture update slide 6.
V1.3	14.04.2014: Minor edits

Literature

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- J. Hardy “the Cache memory book”. Second Edition, Academic Press, San Diego 1998.

- For interested:
- Lam et al “The Cache Performance and Optimisation of Blocked Algorithms”
- Algorithms and Data Structures – Performance optimisation http://www.cie.bv.tum.de/~mundani/algdata/part06.pdf

Contents

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Cache – using Cache
- Scratchpad – using Scratchpad
- Memcpy – DMA – Hardware Acelleration

What you should be able to do after today

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- The student will have an introduction/refresh to caches and scratch pad memories.
- The student will be aware of various issues surrounding the use of caches and SPM and will understand the main issues surrounding cache-aware programming
- The student will be able to read and correctly implement cache-aware software on the NIOS II.

Thick Red Line

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Sobel is a cpu-bound function with multiple memory accesses -> reducing the mean access time increases execution speed
- Why cache is a statistical performance booster.
  - CAM -> set associative -> 2/4/x way associative
  - Cache architectures
  - Real life examples
- Using cache
- Case study

Zürcher Fachhochschule

5

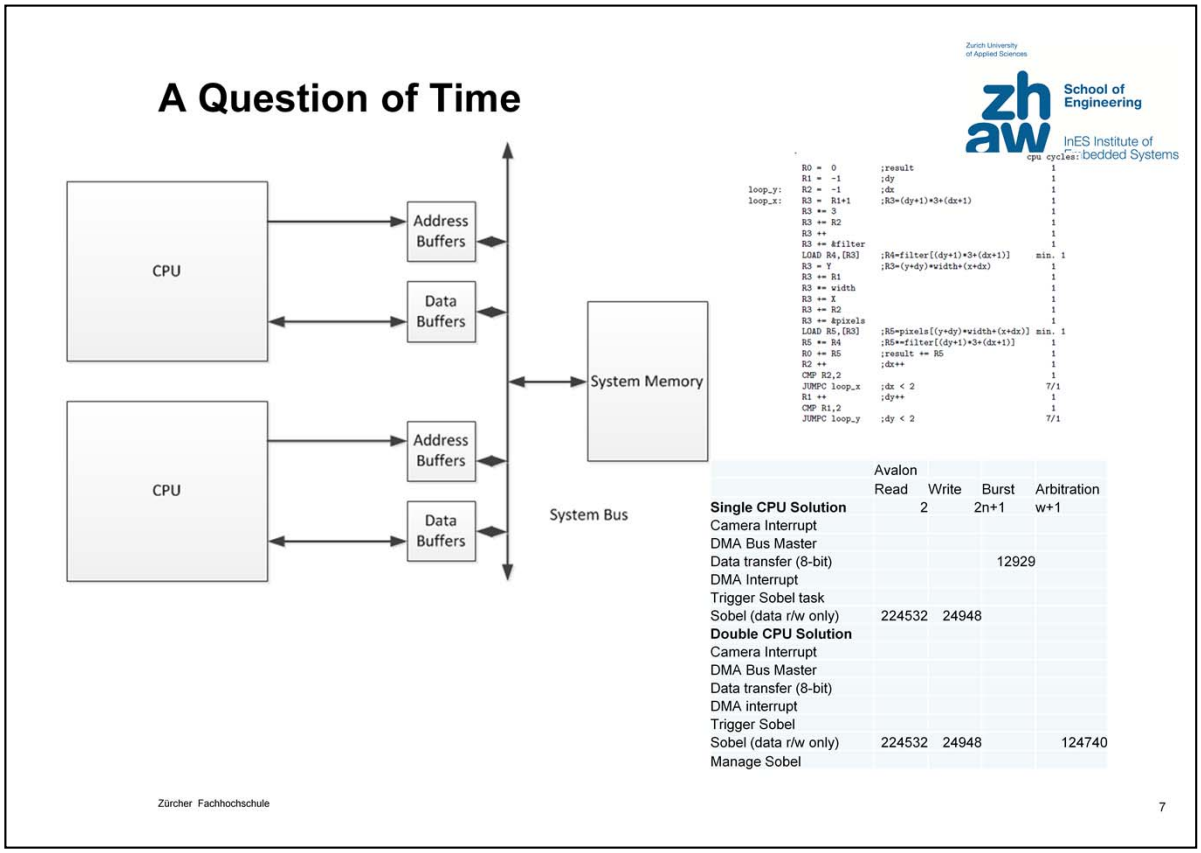
CPU bound algorithms are algorithms that use bursts of CPU time whereas **I/O bound** algorithms are algorithms that spend much time waiting for I/O

Programs tend to spend time, both code and data, in the same area. In code its in loops, and in data its in the same area of data. This is known as the **localisation principle**.

Contents

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Cache – using Cache
- Scratchpad – using Scratchpad
- Memcpy – DMA – Hardware Acceleration



In our Sobel example we have seen that we could feasibly increase our performance by adding another processor something like the above architecture. However we can also claim that despite increasing processing power we have introduce two inefficiencies – increased power and time wasted through additional bus arbitration. In our Sobel example so far we find ourselves in a **CPU-bound** situation, where the system characteristics are defined by the time it takes to complete CPU operations rather than an **I/O bound** system where we are limited by the speed of I/O accesses. These limitations are discrete. If a memory has a speed of say 20ns then CPU frequencies of up to about 50MHz can be well accommodated. Once a frequency of 60MHz is applied to the processor, wait states have to be applied to every read/write cycle. Designing in memory with a faster access time is a cost issue. In any system there are static delay times that do not improve with technology or the speed grade of a DRAM or SRAM, specifically, buffer delays, address set-up and hold times which serve to slow the access down regardless of the specification of the memory speed grade.

The net result of these considerations is that, for instance, the 5ns delay exhibited by the address and data buffers can be adequately compensated by moving memory off the system bus and closer (in some fashion) to the CPU. We do this by introducing an element called cache.

The point of the cache is clear – if the mean access time can be reduced then the CPU will suffer fewer wait states and the necessary system bus bandwidth will be lower. The general idea gets more pertinent if there are multiple potential bus masters in the system as the system bus accesses then suffer the issue of bus contention which needs to be resolved by an arbiter.

The idea of a cache works on the basis that a program spends time within a certain range of memory addresses – or in other words programs are loop driven, in affect the

behaviour we have seen with the Sobel algorithm where both the code and the data exhibit **temporal** and **spatial** locality. Spatial locality is the notion that a program executes code that is close to previous lines of code, and temporal locality is the notion that spatially close code executes in the same time frame.

If programs exhibit this kind of behaviour then we can optimise the execution time of the program **statistically** by introducing an extra memory layer (the cache) which temporarily holds the code or data being processed at that time.

Principle of Locality

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Programs tend to execute for longer periods of time on the same code on data in the same general location
  - **Localisation Principle**
    - Code -> loops
    - Data -> arrays
- Memory accesses are expensive in time
  - **Idea** – small but fast memory to store often used code and/or data -> decrease access times
- 2 types – Cache and Scratchpad memory

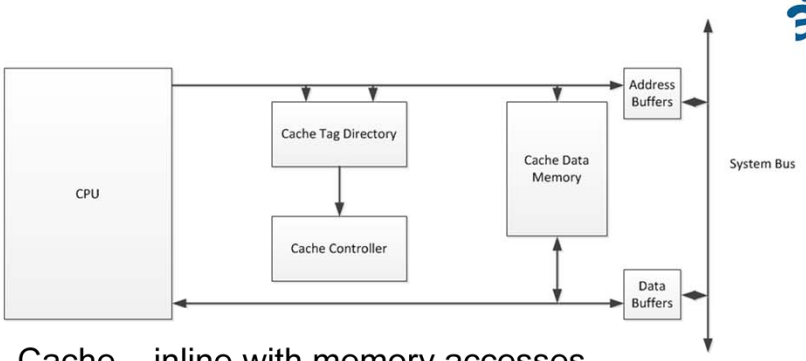
Zürcher Fachhochschule

8

CPU bound algorithms are algorithms that use bursts of CPU time whereas **I/O bound** algorithms are algorithms that spend much time waiting for I/O

Programs tend to spend time, both code and data, in the same area. In code its in loops, and in data its in the same area of data. This is known as the **localisation principle**.

Cache



- Cache – inline with memory accesses
  - Cache hit -> datum in cache
  - Cache miss -> data not in cache, processor must find datum in main memory
    - When loading data it will also be loaded into Cache
    - Next access to datum -> cache hit

Zürcher Fachhochschule

9

We know from the advertising of PC based systems that there are L1 and L2, and sometimes even L3 caches. The cache tied most directly onto the processor/controller – usually on-die memory, is called the **first level**, **primary** or **Level 1** (**L1**) cache and the **second level** (**third level**), **secondary** (**tertiary**) or **Level 2** (**L2/ Level 3/L3**) is **downstream** of this cache. We consider here L1 cache in the first instance as it is this we are usually confronted with in a large range of embedded situations.

From a hardware point of view the cache can work as follows: If the cache can supply the processor with a copy of what its looking for in the main memory then it will be serviced faster than if it had to go out onto the system bus to look for it. This particular case is called a **cache hit** – in which case the cache will place the data on the CPU bus and the main memory will not (be allowed to). If the design is intended to reduce bus bandwidth then no attempt will be made to access system memory unless there is no copy of it in the cache – the case of a **cache miss** or fault - the **cache directory** will decide whether there is a copy of the required datum in memory and the **cache controller** will manage the interaction of cache with system memory – formulated by **cache policies**.

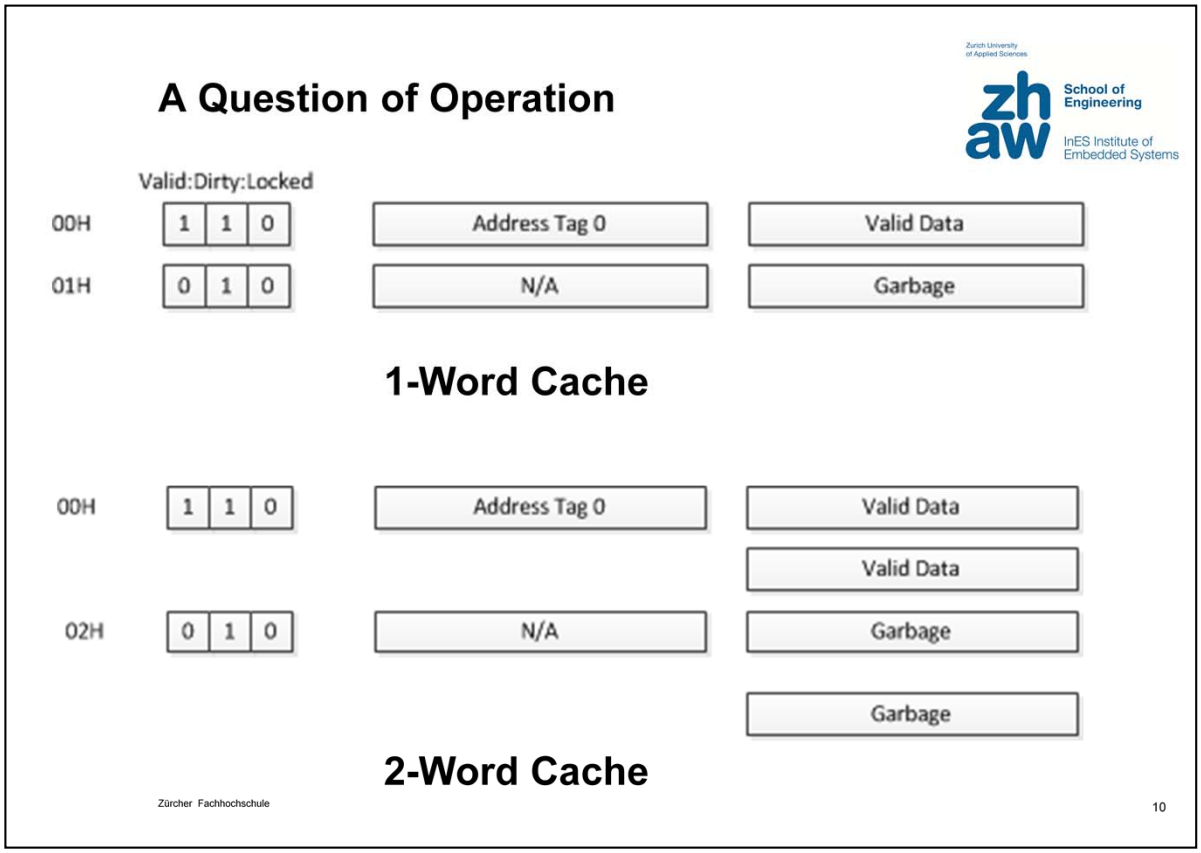
The cache size and policies affect the **hit rate**. The hit rate is the probability that the looked-for datum is resident in the cache. For example a cache with a hit rate of 80%, a very low number, will execute a read in 2 CPU cycles. For the remaining 20% the CPU must access the system bus at a cost of 3 additional cycles. Therefore for say 10 instructions the cost is

10*0.8*2 + 10*(1-0.8)*5 = 16 +10 CPU cycles.

Therefore for nearly 40% of the time the CPU would be unable to get to 20% of the instructions. If the hit rate was to be increased to a more reasonable number such as 90% then the arithmetic would be:

10*0.9*2 + 10*(1-0.9)*5 = 18 +5 CPU cycles

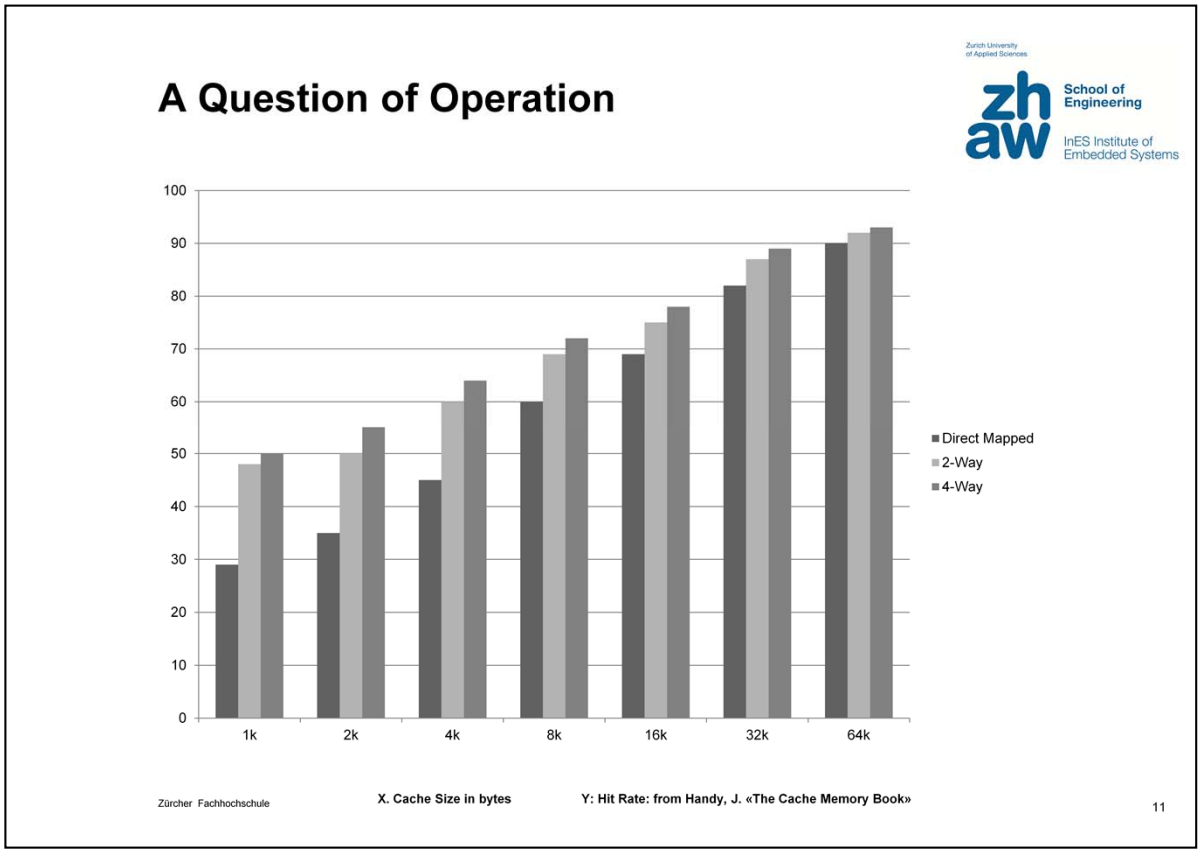
The miss rate is decreased by half.



The general operation pattern is as follows. When the system comes out of reset – and we presume that everything is initalised to zero - there is no data in the cache. Therefore the first memory read – lets say a code read – will cause a cache miss. The code will be loaded from system memory into cache data and the address into the cache tag directory. The CPU itself reads from system memory. This sequence is repetitive so if the CPU is executing a loop then sooner or later it will begin executing code that has been copied into the cache – and this is when the speed advantage of “on-board” faster memory kicks in.

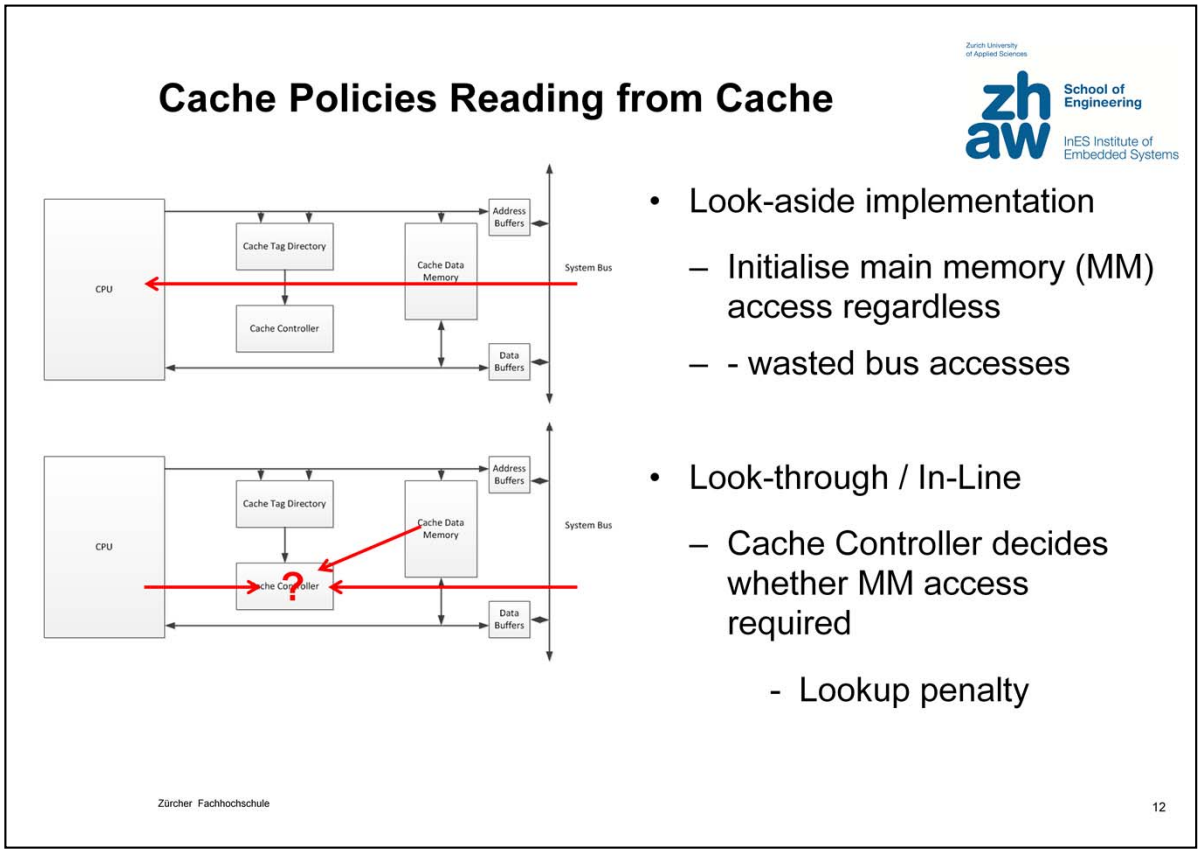
Datums are stored in cache **lines**. The depth of these lines (i.e. number of bytes) is design-specific. One of the important cache policies concerns the associativity of the cache. Obviously the cache is smaller than the main memory (but may be larger than the program and associated data) therefore some mapping system may be used. In most college courses **Content Accessible Memory** is used as the example to explain cache. CAM allows a data word to be presented at the inputs. A comparator array compares this input word with the entire contents of the CAM. If it finds a match then it generates the address where this word was found. The fact that a word can be found in this manner means that a word can be stored anywhere in the CAM – this is known as a **fully associative array**. Since CAM’s are relatively expensive (in terms of logic gates) there are some cheaper alternatives: One of the more popular is called the **set associative**. In this case the lower address bits are called the **set bits** – the number of bits being dependent on the size of the cache. The upper bits are stored in **the tag memory** and are hence known as the **tag bits**. What happens is that if there is a 4k cache? The datum at system memory address location 0x000000 will also reside at set address location 0x0000 in the cache – how much is dependent on the line size of the cache. Lets assume 4 words as in the NIOS. The next line of cache can be filled (if requested)

by the datum stored in system memory at the address 0x00004 and so on. The inherent advantage here is that then sequence of code as laid down in the system memory will be replicated – right upto the cache boundaries in cache memory.



Whereas the set associative is very fast it has one issue and that is illustrated by the following: If the code executing from 0x001000 to 0x001080 calls a function that resides at 0x002000 to 0x002040 then the 4k cache controller will continuously replace the lines given by the set address 0x0000 to 0x0040 because the tag addresses (0x010 and 0x020) associated with those set addresses do not match - this is known as **thrashing**. This can be (partially) solved by using a n-way cache. For a 2 way cache a 4k cache can hold each 2k of contiguous set addresses (i.e. set addresses to a boundary of 0x007FF) and two system memory locations with the set addresses 0x000-0x040 can be stored.

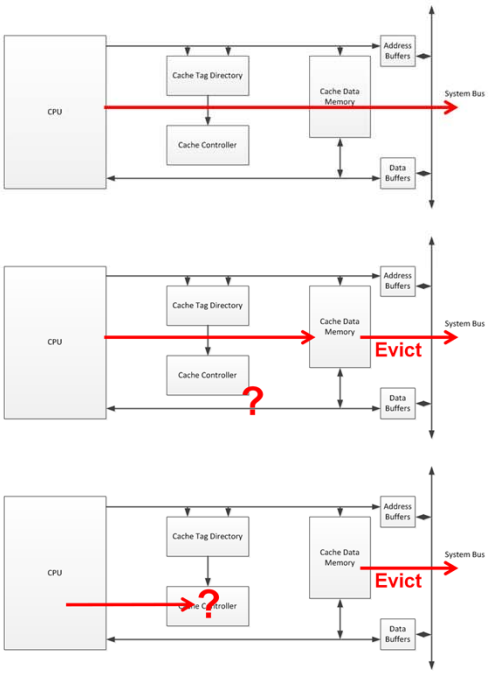
Some rules of thumb – doubling the cache size decreases the miss rate by 69%. Doubling the associativity decreases the miss rate by ca 20%. See the graphic for a better estimation.



There are four basic operations with respect to cache. These are **read hit** and **miss** and **write hit** and **miss**.

A design which allows all signals from the CPU to access the main memory regardless whether it's a hit or a miss is known as a **look aside** implementation. This makes for faster single-processor read miss accesses since by the time the cache controller realises it has a miss, the system memory is well on its way to getting the required data ready for the CPU. Obviously in a multi-processor design the wasted bandwidth is high and given the need for arbitration the notion of time saving cannot be so readily supported either. **Look through** or **in-line** caches are typical where the cache controller decides whether to initiate a system memory read/write or not. This kind of implementation massively reduces the CPU-required bandwidth at the cost of the so called **lookup penalty**, the time required to decide that a system memory access is required. In a read hit the system bus address and data buffers remain switched off and the Cache Data Memory buffers are switched on, allowing the datum to proceed to the CPU as if it were coming from system memory, only faster. During a read miss the opposite is true – after the lookup penalty the system bus address and data buffers are turned on and those of the Cache Data Memory to write. The data is sent to the CPU and "sniffed" by the data cache RAM. Equally the cache tag directory is updated with the new tag address. This is known as a **line fill/update** and any number of other buzzwords. A **compulsory line fill** is one which could not have been avoided using a better cache policy.

Cache Policies Writing to Cache



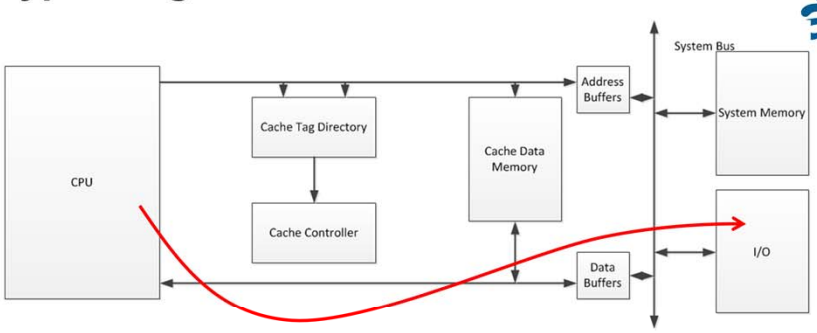
- 1 variable, two storage areas
- (Hit) Write-through policy
  - Update Cache and MM
- (Hit) Copy Back policy
  - Write into Cache
    - When dirty bit set evict to MM
- Write Misses
  - Possibilities too numerous for here

13

Write hits are generally handled in two ways: The first is that both the Cache Data Memory and the System memory are updated – this avoids the problem of cache coherence the fact that there are potentially two different copies of the datum at any one period of time in the system at the expense of additional CPU cycles being required for the system memory read. This is known as a **write-through** policy. A **copy back** policy is to only write the datum into the cache – speed, useful for instance when loop incrementers are used – at the expense of management effort to ensure **cache coherence**. The simplest method – and there are lots of others - is to implement a **dirty bit**. The dirty bit signifies that a cache line has been written to so that when the time comes for the cache line to be replaced it is not simply written over but **evicted** to system memory. There is the distinct possibility that the data cache implements a copy-back policy but the code cache doesn't, given that code is rarely written to. **Write misses** are also handled in a variety of ways. The cached line can be ignored and the result written directly to memory, or the cached line is written to both memory and cache or the line is simply written to cache and another policy implemented after that or for instance, and this discussion is out of the scope of this course since we are focused on living with caches rather than designing them, is to implement a **write buffer** wherein the write is cached in the cache and written to a buffer which is then used to update memory asynchronously from the processor.

The final policy is to update the cache line regardless of whether there is a cache hit or miss.

Bypassing the Cache



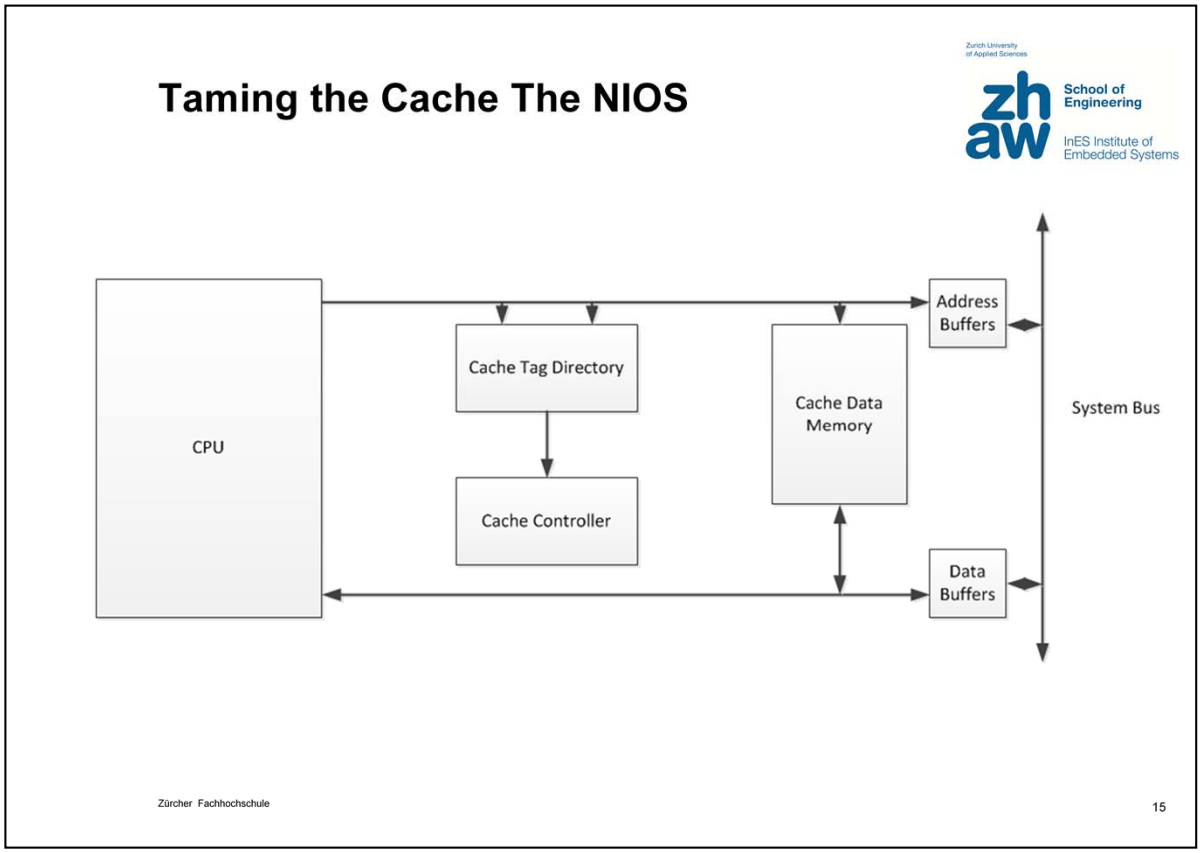
- I/O should not be cached
  - Avoid cache controller by locating I/O in reserved address range
  - C keyword `volatile` prevents optimisation of multiple reads but has no effect on cache operation
  -

Zürcher Fachhochschule

14

There is also a strong case to be made for ignoring cache entirely and implementing a uncached addresses. This is useful in systems where I/O is memory mapped, where processor status bits can be accessed or where communication with a co-processor occurs.

Last but not least there is the question of system restart behaviour: Like most memories both the cache memory itself and the tag memory comes out of reset with the presumption of being randomly filled with data. Some systems disable the cache until the software has time to fill the cache with enough data to begin normal operation. Other systems implement a thing called a valid bit – a bit that signifies that the data enclosed in the cache is valid or not. A cleared valid bit for a particular line would force a read or write miss whereas a set valid bit would allow a read/write hit.



This all leads us to the next question which is how are caches handled in real life.

In the NIOS the implementation is different for data and instruction cache: The instruction cache is a direct mapping (one way set associative) set at 32 bytes line and a read reads an entire line from memory in one burst cycle. Without a Memory Management Unit (MMU) the are 31 bit addresses available, the 32nd bit being used to bypass cache.

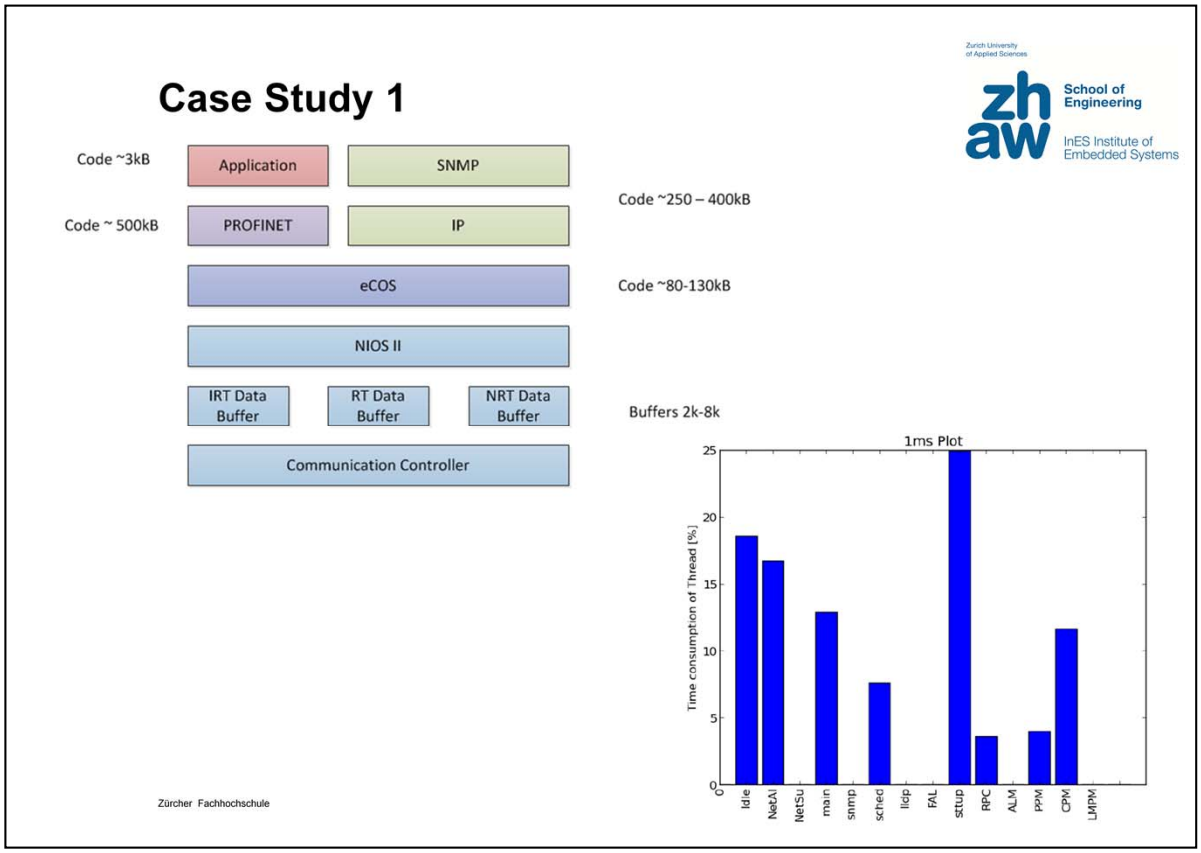
The data cache is also a direct mapped implementation but the line sizes can be configured to be 4, 6 or 32 bytes large. It is a write back implementation – means? It also implements **write allocation** which means that when a write miss occurs the entire line is read into cache and the write data merged as the line is being written into cache.

There are several instructions aimed specifically at the cache. The instruction cache can be invalidated by the instruction initi. After reset the device invalidates the line corresponding to the reset vector – this means that an instruction fetch from main memory is forced. The next 8 instructions must invalidate the entire instruction cache after which the program can proceed as normal. The data cache must also be initalised – with initd. Since the NIOS does not implement hardware cache coherence dirty data cache must be written back to memory with flushd. The NIOS implements two instructions for initd(a) and flushd(a), considering and not considering the tag line – why?

The NIOS also implements several instructions to read and write to memory or I/O bypassing cache – for instance ldbuio – these instructions are further used in the Hardware Abstraction Layer the NIOS provides. So instruction **Macros** the **IOW**_ and **IOR**_ ensure cache consistency. For instance if a read is done on an IO port using a standard read then the first read will read correctly as it is read from the system bus – but since it is also copied into cache all reads after this will be from cache and not via the system bus and hence the program will not function correctly. Normally a repeated read of a value that the compiler does not see to change will be optimised out by the complier

– this optimisation can be prevented by the use of the *volatile* keyword in C but it will ***not*** affect the operation of the cache.

What the NIOS doesn’t seem to offer is the ability to lock the cache.



In this particular case the software is so large and so little is known about the distribution and location of tasks within the OS and PROFINET stack, its not going to all fit in the NIOS cache AND the NIOS can't lock elements into its instruction cache.

The plot lower right shows the idle time (far left) on an ARM running at 666MHz at a PROFINET cycle time of 1ms. In the case of the NIOS II this was down to 5%. Any asynchronous traffic caused problems for the RT deadlines as the eCOS timeslice is 1ms as well and if the IP stack and SNMP stack needed the full timeslice then the system would collapse.

By increasing instruction cache to 16k (taking away data buffer size) and re-arranging task priorities we were able to reduce the processor utilisation to 80% and the node could easily handle a denial of service attack. On the other hand it was all pure blind luck – it might have not been possible to increase the cache any further and then we would have needed to get into the nitty-gritty of optimising individual threads in the entire system and hence producing something with non-standard "standard" components (OS, PROFINET stack …)

Contents

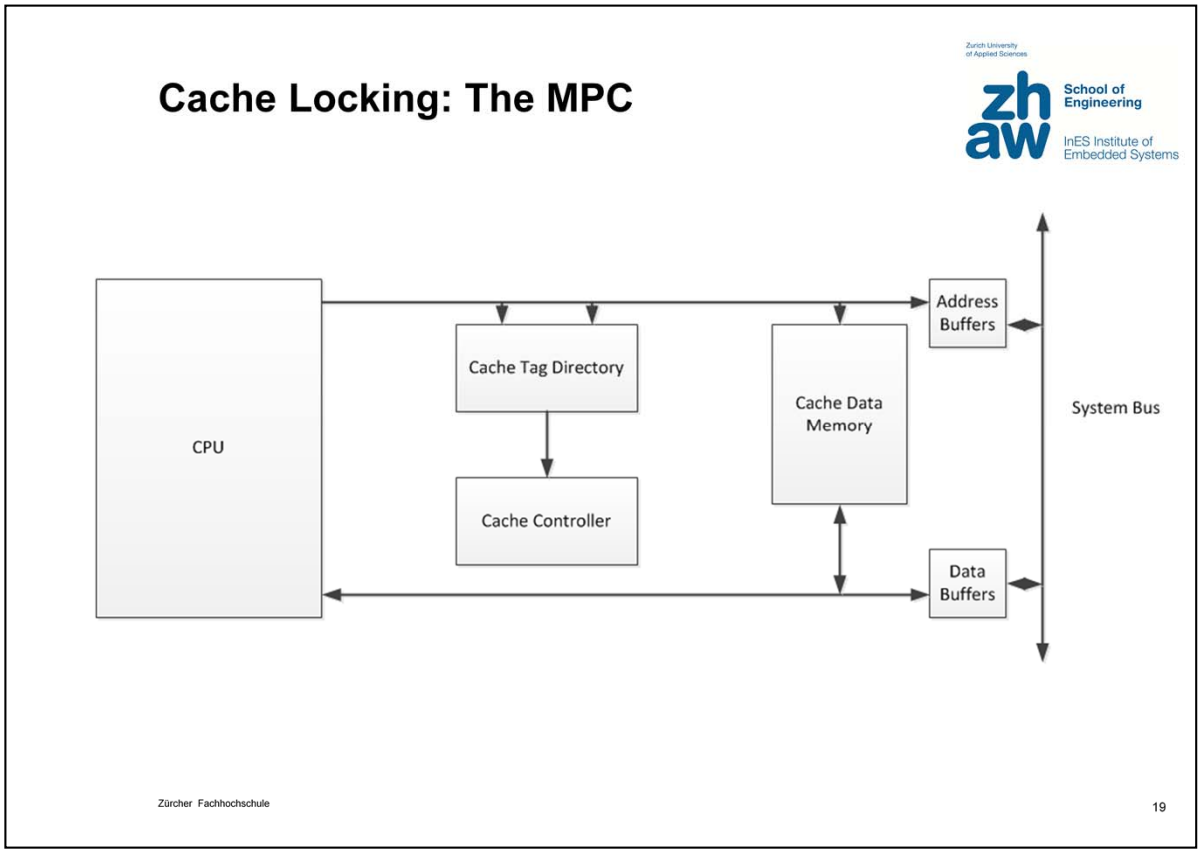
Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Cache – using Cache
- Scratchpad – using Scratchpad
- Memcpy – DMA – Hardware Acceleration

How to use Cache

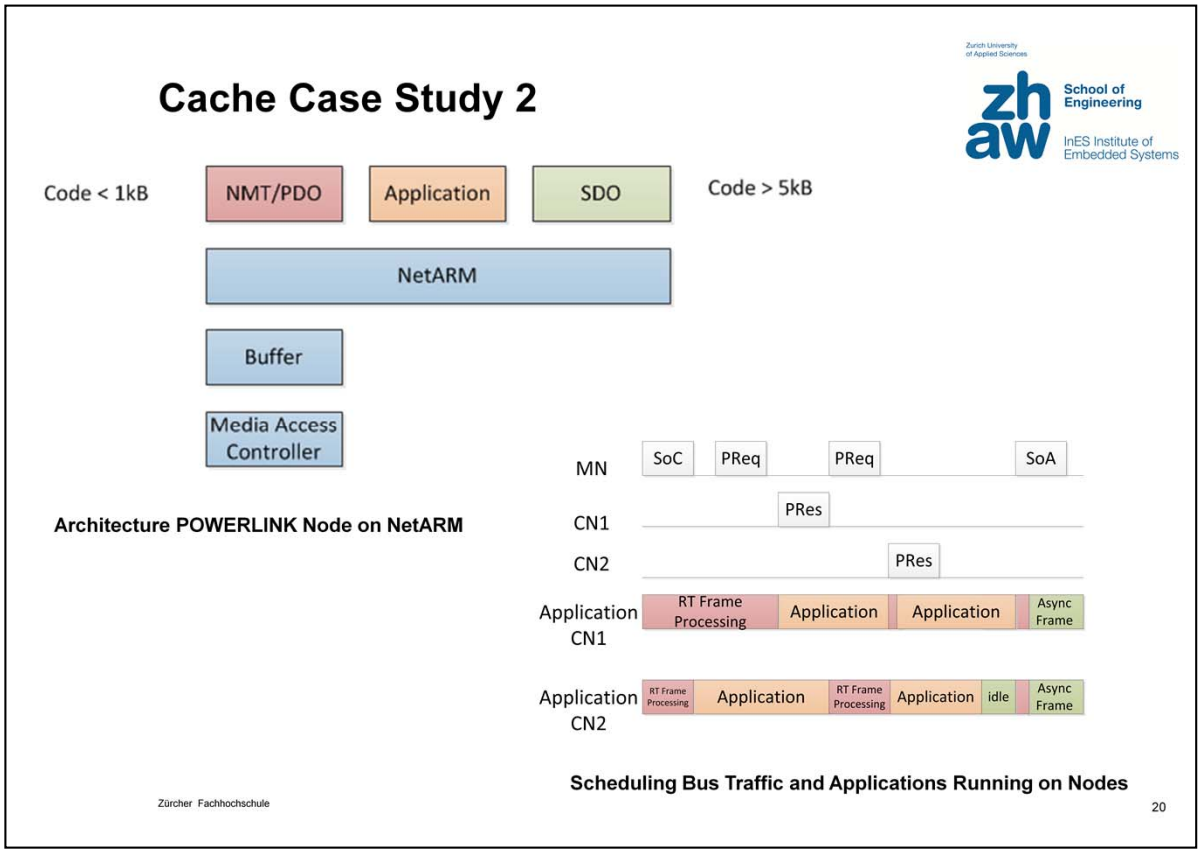
Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Cache Locking
- Cache Aware Programming
  - Alignment
  - Pre-Fetching
    - SW, HW, SW/HW
  - Access Optimisation
    - Loop Transformation
    - Optimised Storage



The cache for instance on the MPC8349e – a much more sophisticated beast by far – implements data and instruction caches of 32kBytes each with a line size of 8 words and are 8-Way associative. The operation of locking the cache is very useful in n-way associative caches since several routines can be locked in cache and yet space for “random” cache accesses is still available and the memory map of the compile-time binary does not need any particular mapping manipulation. The sequence for achieving this – typical for any locking sequence is: to 1.) Invalidate the caches 2.) load the caches – in the case of instructions these are done using a speculative fetch – in this particular case an instruction sequence (a divide and a branch) is carried out – the divide because it needs lots of cycles and the destination of the branch is loaded speculatively into cache. If the branch is not fulfilled then the processing of those instructions is cancelled but the instructions are still valid in cache. Note – what we should learn by this is that we cannot simply copy a region of instructions from memory to cache. It is then possible to lock either the entire instruction (data) cache or simply one Way. To things are to be observed – the first is that the code carrying out these speculative fetches must be in cache-bypassed memory (why?) and the interrupts must be disabled.

The use of a HAL is common in many Real Time Operating Systems as well. For instance eCOS offers a HAL which allows the programmer to set/define many cache parameters where allowed by the hardware and also offer macros for read and write instruction consistency. Given that multi threaded RTOS operate without memory protection and for the most part the caches work software transparent there is little that a RTOS requires from a caching system – on the other hand the caching system does have is important affects on the performance of the RTOS, or more specifically on the WCET of a piece of hard real time code.

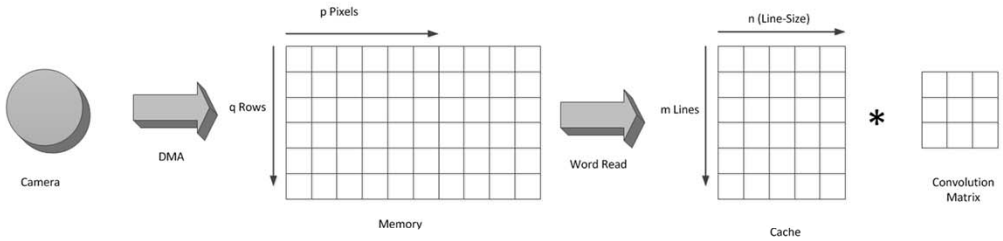


The general architetcure is shown above. The code sizes are given as approximations. From the scheduling – lower right – we can see that the function most often called within a cycle was the PDO/NMT. The Network management layer regulates the correctness of the cycle and the nodes response to it and the PDO handles the Real Time Data transfer. This task – function set was locked in cache and given the highest priority, a Fast Interrupt. The application, generally supposed to be a trivial one, was given the next highest priority of a standard interrupt and was allowed to take as long as it was going to take. The application programmer just had to make sure that enough time was left to process any incoming non real time frames.

The point of this case study is to show that since the code was small enough to fit into cache it could be locked in its entirety and therefore the functions called most often (PDO/NMT) would have the greatest benefit of the cache.

Case Sobel – Data Cache

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems



- Data transferred line for line per DMA to memory
- Read cache-line for cache-line into cache
- Multiplied by a second array (on stack)
- Stored in cache

Alignment 1

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Cache line of 16 bytes
- Assume tag address of 0x0000 end address of 0x000E: A read to 0x10008 produces either
  - Line read 0xYY0000 – 0xYY000E
  - Line read 0xYY0008 – 0yYY0016
  - Need to check what cache does
- Ought to align arrays on appropriate boundaries
- Need to consider: array storage in **row major order** (C/C++) or **column major order** (Fortran, Matlab)

http://www.cc.gatech.edu/~bader/COURSES/UNM/ece637-Fall2003/papers/KW03.pdf

Alignment 2

Zurich University of Applied Sciences

School of Engineering

InES Institute of

- Need to consider padding

Algorithm 3.4 Inter–array padding.

```
1: // Original code:                    1: // Code after applying inter–array padding:
2: double a[1024];                      2: double a[1024];
3: double b[1024];                      3: double pad[x];
4: for i = 1 to 1023 do                 4: double b[1024];
5:    sum+ = a[i] * b[i];               5: for i = 1 to 1023 do
6: end for                              6:    sum+ = a[i] * b[i];
                                        7: end for
```

- .. Or array merging

Algorithm 3.5 Array merging.

```
1: // Original data structure:
2: double a[1024];
3: double b[1024];

1: // array merging using multidimensional arrays:
2: double ab[1024][2];

1: // array merging using structures:
2: struct{
3:    double a;
4:    double b;
5: } ab[1024];
```

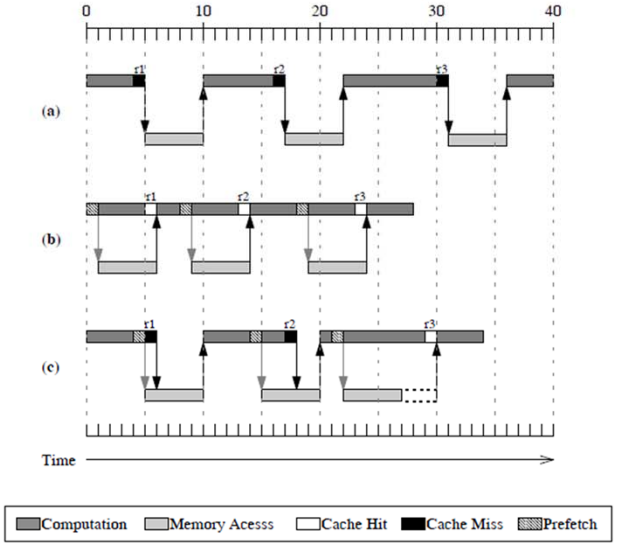
Zürcher Fachhochschule

23

http://www.cc.gatech.edu/~bader/COURSES/UNM/ece637-Fall2003/papers/KW03.pdf

The problem with arrays is that if they are both mapped to the same line in cache they interfere and thrashing occurs. Two ways round it are to put padding between the arrays (making sure the compiler puts them where you want them!!) i.e introducing an offset or to merge them.

Case Sobel – pre-fetches (1)



Standard accesses subject to bus arbitration

May result in a stall

Still only line-at-a-time

Some processors offer a prefetch instruction

Prefetch is non-blocking

Zürcher Fachhochschule

24

Several points to note: figure 2c – the pre-fetches are issued too late to avoid processor stalls for r1 and r2. the the case of r3 the data is in the cache a long time. This subjects the data to the replacement policy of the cache and hence it may be evicted before the data is actually needed. If this happens this is known as a **useless pre-fetch**. The data may also have replaced data required by the computation re4sulting in a cache miss on that data. This is known as **cache pollution**. Note also that using pre-fetches also increases the number of memory accesses per unit time and the memory must be able to handle this increased bandwidth.

Case Sobel – pre-fetches (2)

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

```
for (i = 0; i < N; i++)
```

```
for (i = 0; i < N; i++){
      fetch( &a[i+1]);
      fetch( &b[i+1]);
      ip = ip + a[i]*b[i];
}
```

```
for (i = 0; i < N; i+=4){
      fetch( &a[i+4]);
      fetch( &b[i+4]);
      ip = ip + a[i]*b[i];
```

```
fetch( &ip);
fetch( &a[0]);
fetch( &b[0]);

for (i = 0; i < N-4; i+=4){
      fetch( &a[i+4]);
      fetch( &b[i+4]);
      ip = ip + a[i]  *b[i];
      ip = ip + a[i+1]*b[i+1];
      ip = ip + a[i+2]*b[i+2];
      ip = ip + a[i+3]*b[i+3];
}
for ( ; i < N; i++)
      ip = ip + a[i]*b[i];
```

(d)

$$\delta = \left\lceil \frac{l}{s} \right\rceil$$

- Standard loop in (a)
- Pre-fetch the data for the next iteration in (b) – note a[0] and b[0] are compulsory cache misses
- Unrolled loop with cache line pre-fetch (c)
- Pipelining technique with prolog and epilog (d)
- prefetch distance given by ceiling of average latency divided by shortest path through the loop

25

The code of (d) **covers** all loop references as each reference is preceded by a matching prefetch.

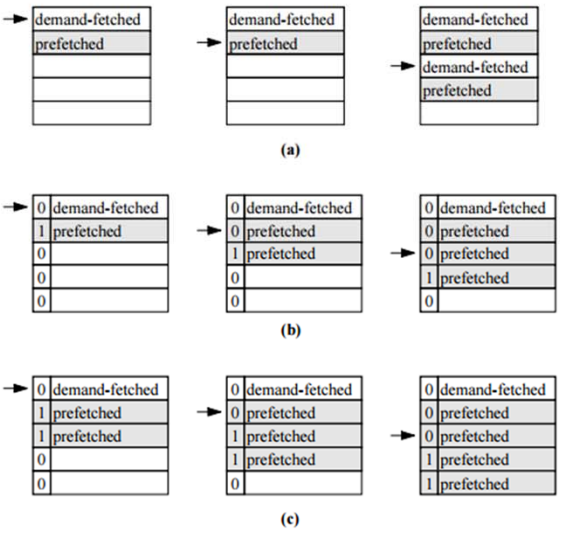
The ceiling function gives a pessimistic estimation and so the

Case Sobel – pre-fetch (3)

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Some compilers offer support for pre-fetching and pre-fetching instruction insertion
  - See GNU documentation
  - Manual insertion generally preferred
- Researchers report speed-up results vary from positive (+96%) to negative (-12%)
  - Largely depends on algorithm
- Increases instruction count
  - Researchers report up-to 100% for bwaves
- Static insertion
  - Not portable onto faster memories and different cache sizes

Hardware pre-fetch (1)



- Cache line fill on cache miss is a form of pre-fetching
- HW pre fetch operates on blocks (multiple lines)
- Three typical methods are:
- Pre-fetch on miss (a)
- Tagged pre-fetch (b)
- Sequential pre-fetching (c) k = 2

Hardware pre-fetch (2)

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

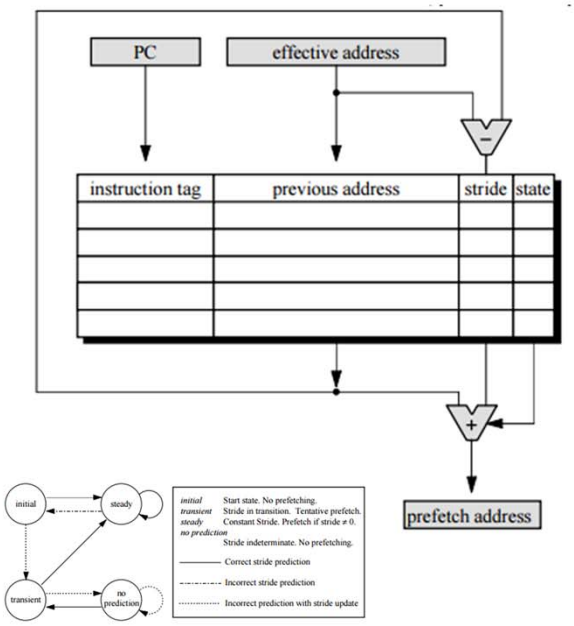
- Based on usage - fetch may not be initiated far enough in advance
  - **Degree of prefetching** K > 1: In practice not good for programs with poor spatial locality
  - Register based pre-fetching - If address register is inc or dec then pre-fetch in that direction
  - Buffer streams
    - Pre-fetched data into FIFO and then into cache when referenced
    - 8 streams and K = 2 produces good results
    - Memory bandwidth increases due to number of unnecessary pre-fetches.

Buffer streams open a whole new range of possibilities and are used to avoid cache pollution - To avoid constant re-filling of FIFO's a history buffer can be used to record most recent cache misses for block b and b+1 and then a stream allocated.

Then technology advances into the arena of non sequential fetch accesses including setting up a **reference prediction table** (RPT). A memory access is recorded and, on the second access the stride (distance from the first access) and when the hardware (state machine) thinks this is stable it then pre-fetches blocks with this stride.

Further information in: http://www.ece.lsu.edu/tca/papers/vanderwiel-00.pdf

Hardware pre-fetch (1)



- Reference Prediction Table

```
float a[100][100], b[100][100], c[100][100];

...

for ( i = 0; i < 100; i++)
    for ( j = 0; j < 100; j++)
        for ( k = 0; k < 100; k++)
            a[i][j] += b[i][k] * c[k][j];
```

(a)

Tag	Previous Address	Stride	State
ld b[i][k]	20,000	0	initial
ld c[k][j]	30,000	0	initial
ld a[i][j]	10,000	0	initial

(b)

Tag	Previous Address	Stride	State
ld b[i][k]	20,004	4	transient
ld c[k][j]	30,400	400	transient
ld a[i][j]	10,000	0	steady

(c)

Tag	Previous Address	Stride	State
ld b[i][k]	20,008	4	steady
ld c[k][j]	30,800	400	steady
ld a[i][j]	10,000	0	steady

(d)

Zürcher Fachhochschule

29

Then technology advances into the arena of non sequential fetch accesses including setting up a **reference prediction table** (RPT). A memory access is recorded and, on the second access the stride (distance from the first access) and when the hardware (state machine) thinks this is stable it then pre-fetches blocks with this stride.

Further information in: http://www.ece.lsu.edu/tca/papers/vanderwiel-00.pdf

HW-SW pre-fetch (1)

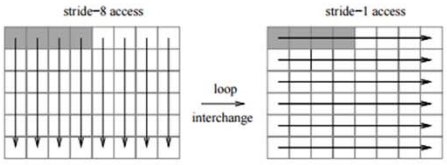
Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Automated pre-fetching works best with large regular memory access patterns
- HW-SW efforts revolve mainly around the idea of an semi-automated process
  - Compiler generates pre-fetch degree and the fetch instruction initalises the tagged-HW pre-fetcher
  - Programmable pre-fetch engine, tag, address and stride supplied by program
  - Pre-fetch engine external to processor running its own program via shared second-level cache (detection of blocks and stride) as well as direction from the processor

Data Access Optimisation (1)

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Have looked at getting the data into cache
  - Now need to look at optimising using the data in cache, i.e. reduce number of cache misses
- Effectively a problem of **Loop Transformations**
  - Difficult to determine which loop transformation to use
- Code generally has sufficient locality for instruction cache misses not to be a problem
- Three major loop transformations: **Loop Interchange**



Data Access Optimisation (2)

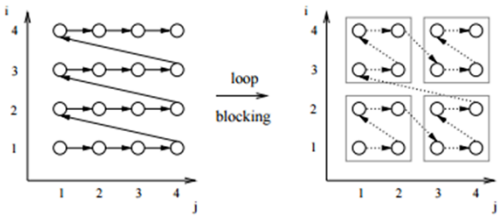
Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Loop Fusion

Algorithm 3.2 Loop fusion

```
1: // Original code:                 1: // After loop fusion:
2: for i = 1 to n do                2: for i = 1 to n do
3:    b[i] = a[i] + 1.0;            3:    b[i] = a[i] + 1.0;
4: end for                          4:    c[i] = b[i] * 4.0;
5: for i = 1 to n do                5: end for
6:    c[i] = b[i] * 4.0;
7: end for
```

- **Loop Blocking** or **Tiling**



Algorithm 3.3 Loop blocking for matrix transposition

```
1: // Original code:                 1: // Loop blocked code:
2: for i = 1 to n do                2: for ii = 1 to n by B do
3:    for j = 1 to n do             3:    for jj = 1 to n by B do
4:       a[i, j] = b[j, i];         4:       for i = ii to min(ii + B − 1, n) do
5:    end for                       5:          for j = jj to min(jj + B − 1, n) do
6: end for                          6:             a[i, j] = b[j, i];
                                    7:          end for
                                    8:       end for
                                    9:    end for
                                   10: end for
```

Data Access Optimisation (3)

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- **Example Picture storage row-major**
  - **Access row major and tiled (2*2)**

Storage = Row Major

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111

38	38		52	53
22	23	Per tile cache miss - tile access	36	37
6	7		20	21
36	37		4	5
20	21	Per 2 Sobel-x cache miss - row major access	50	51
4	5		48	49
34	35		34	35
18	19	Initial cache miss - row major access	18	19
2	3		2	3
32	33	Initial cache miss - tiled access	32	33
16	17		16	17
0	1		0	1

Sobel-x Access = Row Major

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111

Sobel-x Access = Blocked (Tiled) 2*2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111

Data Access Optimisation (4)

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- **Example: different picture storage options**

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(a)

0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15
16	17	20	21	24	25	28	29
18	19	22	23	26	27	30	31
32	33	36	37	40	41	44	45
34	35	38	39	42	43	46	47
48	49	52	53	56	57	60	61
50	51	54	55	58	59	62	63

(b)

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

I II III IV

(c)

Fig. 1. Various data layouts: block size 2×2 for (b) and (c). (a) Row-major layout, (b) block data layout, and (c) Morton data layout.

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.6881&rep=rep1&type=pdf

Zürcher Fachhochschule

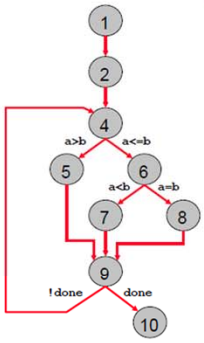
34

A classical issue in number crunching is matrix operations and matrix operations are efficient in row based operations for row major storage classes but very inefficient for column classes.

WCET revisited

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

```
   what_is_this {
1     read (a,b);
2     done = FALSE;
3     repeat {
4       if (a>b)
5          a = a-b;
6       elseif (b>a)
7          b = b-a;
8       else done = TRUE;
9     } until done;
10    write (a);
   }
```



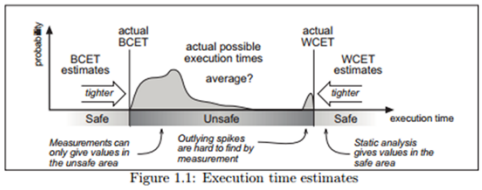


Figure 1.1: Execution time estimates

$$WCET = \sum_{i=1}^{N} c_i \cdot x_i$$

From: J. Engblom. Processor Pipelines and Static Worst-Case Execution Time Analysis. PhD thesis, Uppsala University, April 2002

Zürcher Fachhochschule

35

So formally:

WCET analysis is supposed to generate **safe** (no underestimation) and **tight** (small overestimation) estimations. The problem with using cache systems is that we are trading off the increase in speed gained by not going on the system bus with the unpredictability that a cache brings with it. There are two main issues. The first is that, especially in using 1-way associative sets, the problem of thrashing on a large scale can occur – especially in the case of interrupt servicing or task preemption when using an OS. This can be, in the case of ISR's and under certain conditions, be partially compensated by imaginative memory management during the linker stage so with a simple system as represented by the NIOS we should (?) be able to modify our linker scripts to ensure that particular pieces of code will be allocated particular lines of cache (see exercise).

For OS's which use relative code and dynamic loading this option is impossible and for all others only feasible with a heavily modified tool-set.

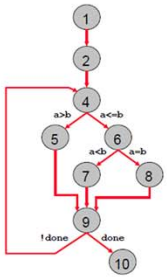
Cache locking can provide some relief if statically applied but this isn't always a practical solution. There have been some attempts to implement **dynamic locking** schemes, the central idea is to benefit from the speed increase due to caches but to avoid the non-determinism introduced by the cache. These solutions concentrate around the static and iterative code analysis. By dividing the code into **basic blocks** and assuming that these basic blocks are to be locked in cache the reloading is to occur on the entry and exit points of these basic blocks. If two basic blocks can be connected then they can be merged – thus a potential cache re-load can be avoided. Equally if a function is called and if it can be shown that the effort of reloading the cache is greater than the potential benefit of locking the code in cache then inlining can be applied to eliminate the

potential cache reload. Hence a convergence to a cache-aware best-case WCET can be achieved.

At the end of the day however – the use of scratch pad memories, or in NIOS terminology tightly coupled memories, are becoming the preferred tool.

Cache Analysis


```
   what_is_this {
1     read (a,b);
2     done = FALSE;
3     repeat {
4       if (a>b)
5         a = a-b;
6       elseif (b>a)
7         b = b-a;
8       else done = TRUE;
9     } until done;
10    write (a);
   }
```



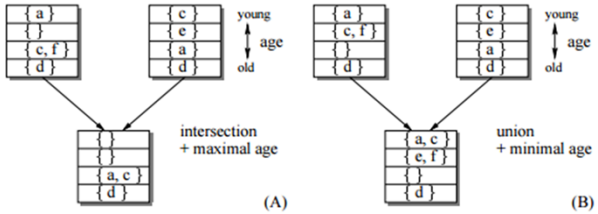


Figure 3. A) Join for the must analysis. B) Join for the may analysis.

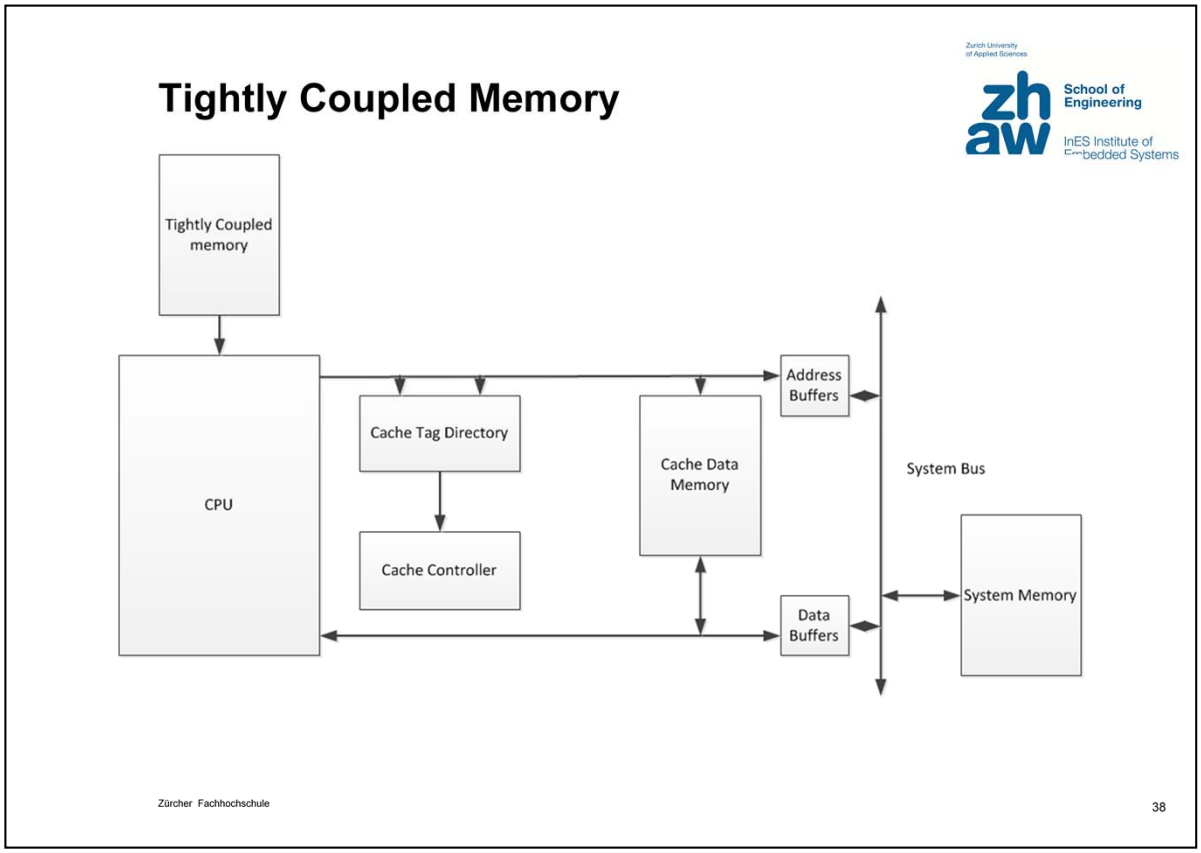
- Based on set theory and program analysis
- Definition of a concrete state and an abstract state
- Three analysis
  - Must: under all circumstances in cache
  - May: could be in cache
  - Persistence blocks never removed from cache

  Similar analysis possible for pipelines

Contents

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Cache – using Cache
- Scratchpad – using Scratchpad
- Memcpy – DMA – Hardware Acceleration



Tightly Coupled Memory or Scratchpad Memory is basically a RAM block in the same order of magnitude as the cache in both size and speed attached to a single processor. It is, different to cache memory mapped into the processors memory space. It enables the execution of both code and data. It allows the copying of code and/or data directly into the memory. This memory can be easily allocated at compile-time – on the other hand it, like cache, still requires in cases like our Sobel algorithm some form of management if the algorithms are to use the area efficiently.

TCM - fetching

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- TCM can function similar to joint cache
  - No discrimination between code and data
- Placement either at run-time or compile-time.
- Automated compile-time requires compiler/linker support
  - Which variables most likely to benefit – static code analysis
  - Stack in TCM
    - Issues with recursive functions -> bottom of stack needs to be transported to main memory
  - Heap in TCM
  - Code in TCM

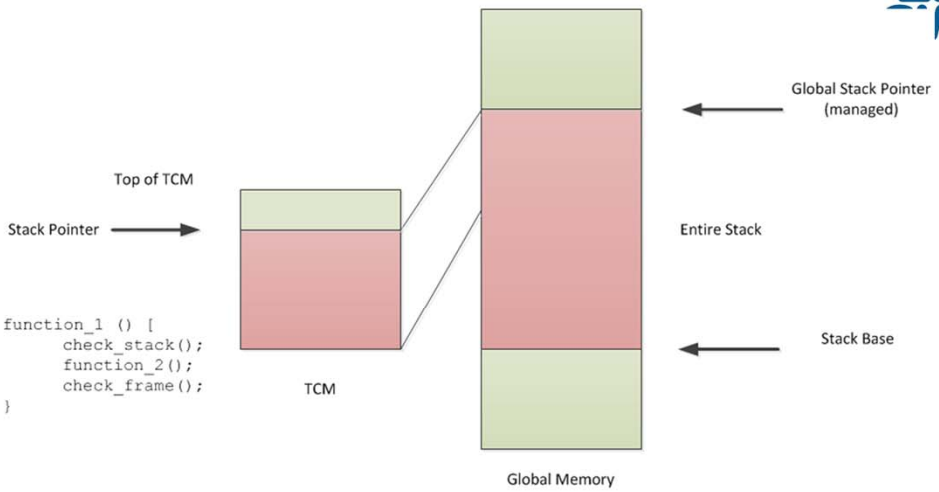
Contents

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Cache – using Cache
- Scratchpad – using Scratchpad
- Memcpy – DMA – Hardware Acceleration

TCM - stack

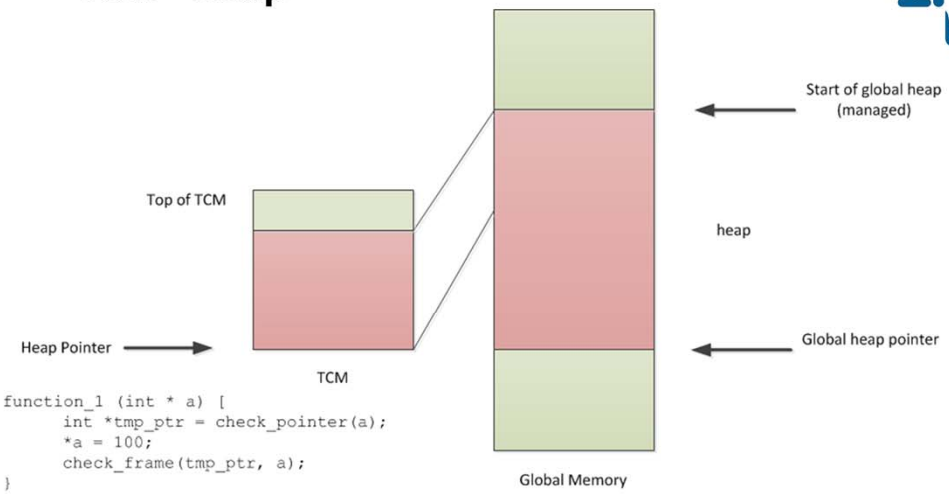
Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems



- Requires dynamic management
  - check_stack() – space in TCM? else flush stack to global memory
  - check_frame() – frame in TCM? Else fetch from global memory

TCM - heap

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems



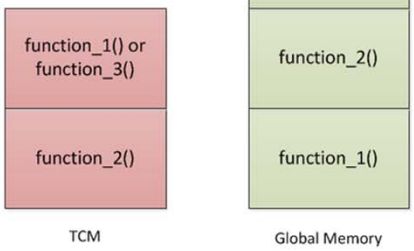

```
function_1 (int * a) [
      int *tmp_ptr = check_pointer(a);
      *a = 100;
      check_frame(tmp_ptr, a);
}
```

- Also requires dynamic management
  - check_pointer(a) – returns virtualised pointer
  - write_pointer(tmp_ptr, a) – writes value of virtual pointer into “real” pointer
  - __malloc() needs to manage transfer between live and run heaps

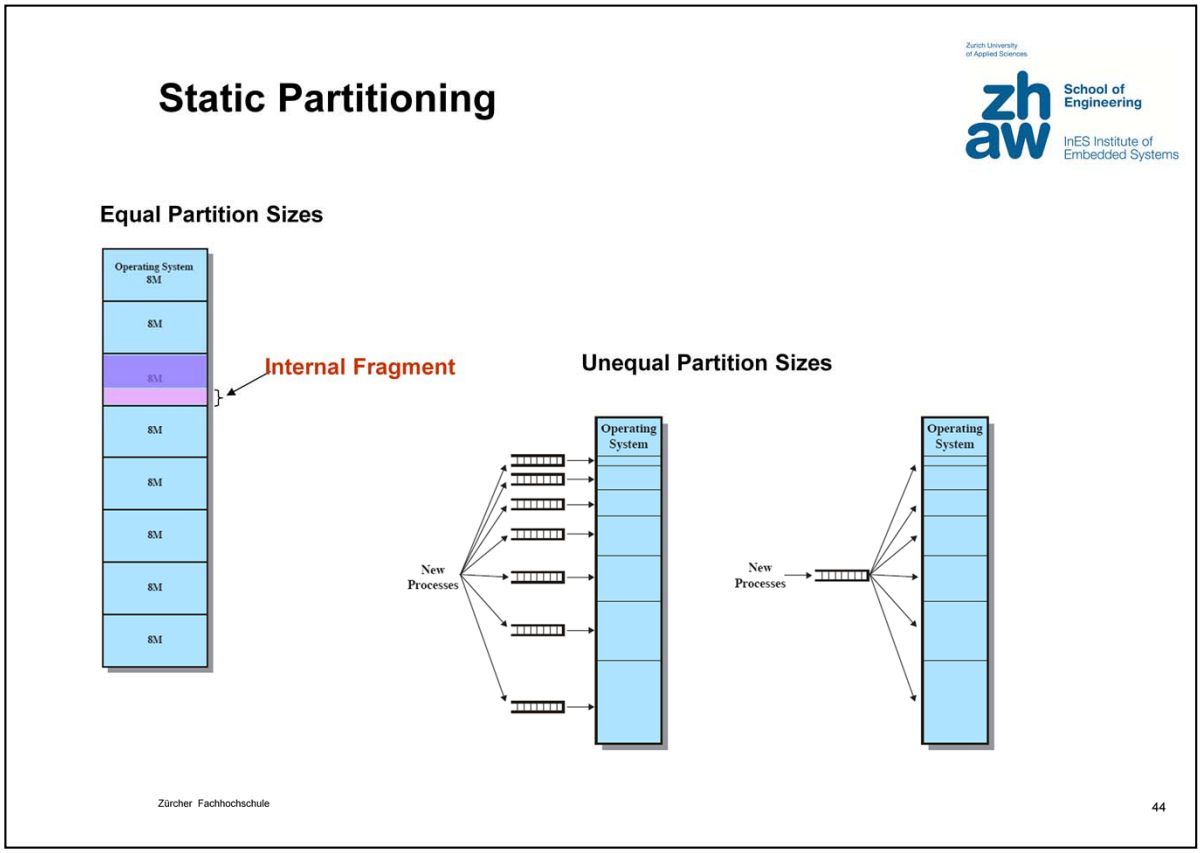
TCM - code

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

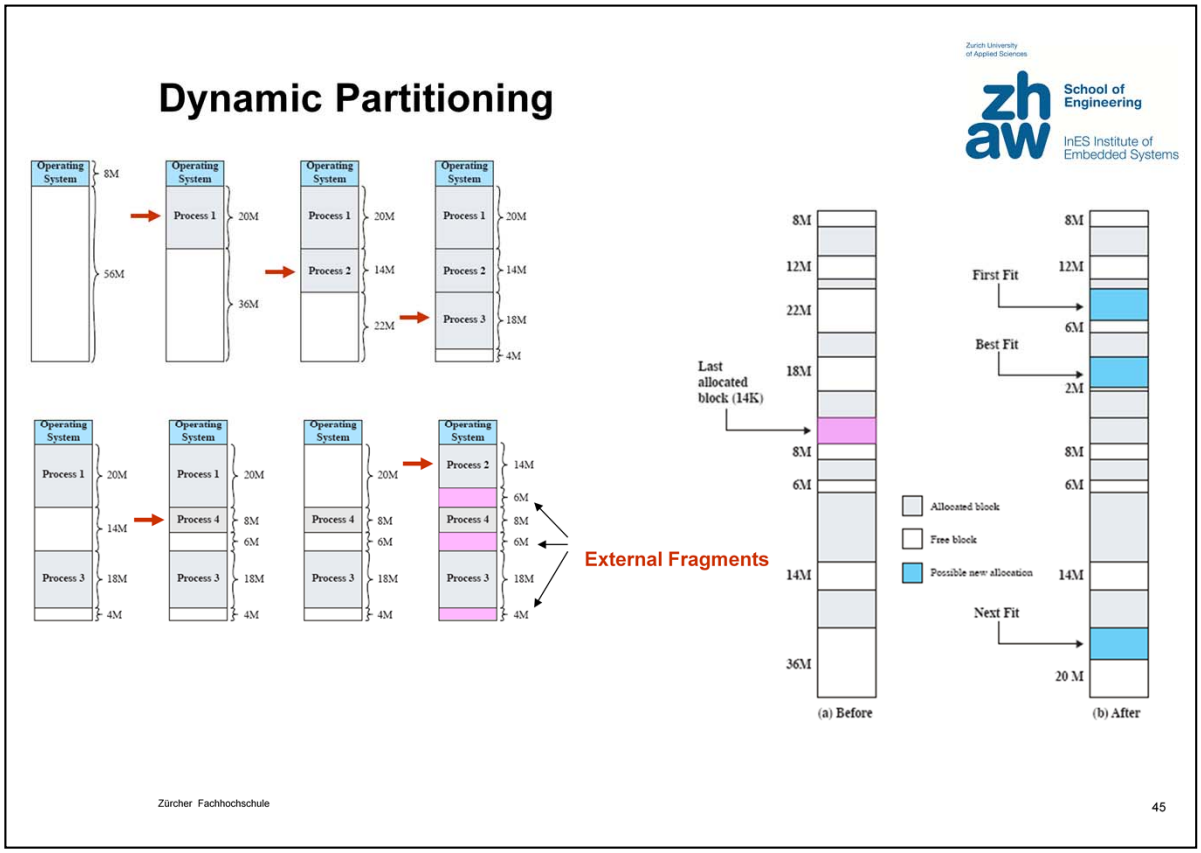
```
SECTIONS {
      OVERLAY {
            function_1()
            function_3()
      }
      OVERLAY {
            function_2()
      }
}
```



- Also requires dynamic management
  - Linker script locates the functions at specific memory locations
  - Overlay manager copies the functions into TCM
  - Requires management by programmer
  - Use of memory management unit also possible
  - Paging techniques (demand paging manager) also used i.e SW memory management unit



Static Partitioning is the simplest. By stipulating the size of memory and the size of partition one can fit a certain number of processes in the memory (**Question**: how many in our example above – how does one manage more?) since the size of the process and the size of the partition is never equal there is unused memory. This is called an **internal fragment**. One way to reduce the size of internal fragments is to make unequal partition sizes and queues for usage of these fragments: It runs the risk that if no smaller processes are needed then memory is not used which could be used for larger processes.



In this particular case internal fragmentation is resolved but as can be seen from the last slide **external fragmentation** appears. The question is: how is this external fragmentation managed.

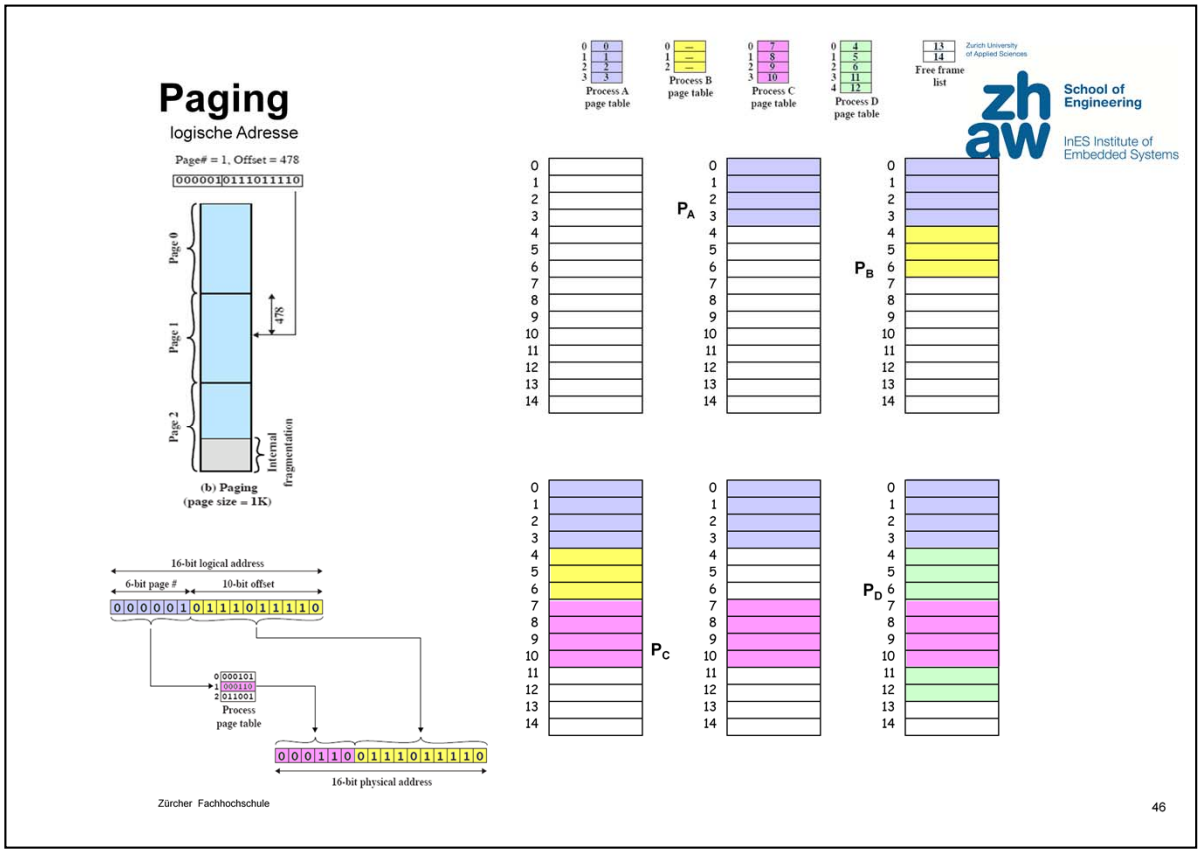
The simplest method is to compact the processes but this takes time (**Question**: how much time? **Question**: what computer system mechanisms can be used for compaction?).

Placement Algorithms:

The first method is to scan the memory from the top and take the **first fit**. Tests show that this *tends* to be the simplest and produces less fragmentation than the next algorithm, **next fit**. It also tends to concentrate the memory usage to the beginning of the memory.

next fit merely takes, from the position of the last allocation, the next best fit. The tendency is to increase fragmentation and compaction is more necessary than with other methods.

best fit: scans the entire memory and finds the best fit but it tends towards high fragmentation (lots of little fragments) and hence requires compaction more often than the other two algorithms.



Paging is beautifully elegant. You divide the process image up into pages of a specific size (determined by the OS) and you divide up the available memory into frames of equal size. Then you match the two. Management is by a thing called a page table where the entries are the frame numbers and the reference is the page number. Pages and frames are transparent for the programmer -> however page aware programming is a good idea.

Question: what does fragmentation look in such a system?

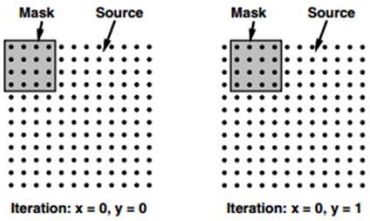
A logical address now has two components. A page number and an offset. The OS (**Question**: what else?) must then maintain the page table and convert a logical to a physical address by the method shown above.

Note: with a 16-bit logical address and 6-bit page number we have max 64 pages and a 10-bit offset means the pages are 1k large and so the maximum manageable memory is?

https://www.kernel.org/doc/gorman/html/understand/understand006.html for Linux

Cache + Scratchpad

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems



http://www.ics.uci.edu/~dutt/pubs/bc12-hipc02-panda.pdf

- Authors report placing mask in SPM and data is loaded into cache
- This avoids conflicts between mask and image array
- Achieved through defining a Total Conflict Factor
- Small and frequently used arrays confined to TCM whereas larger arrays accessed through cache mechanism

Zürcher Fachhochschule

47

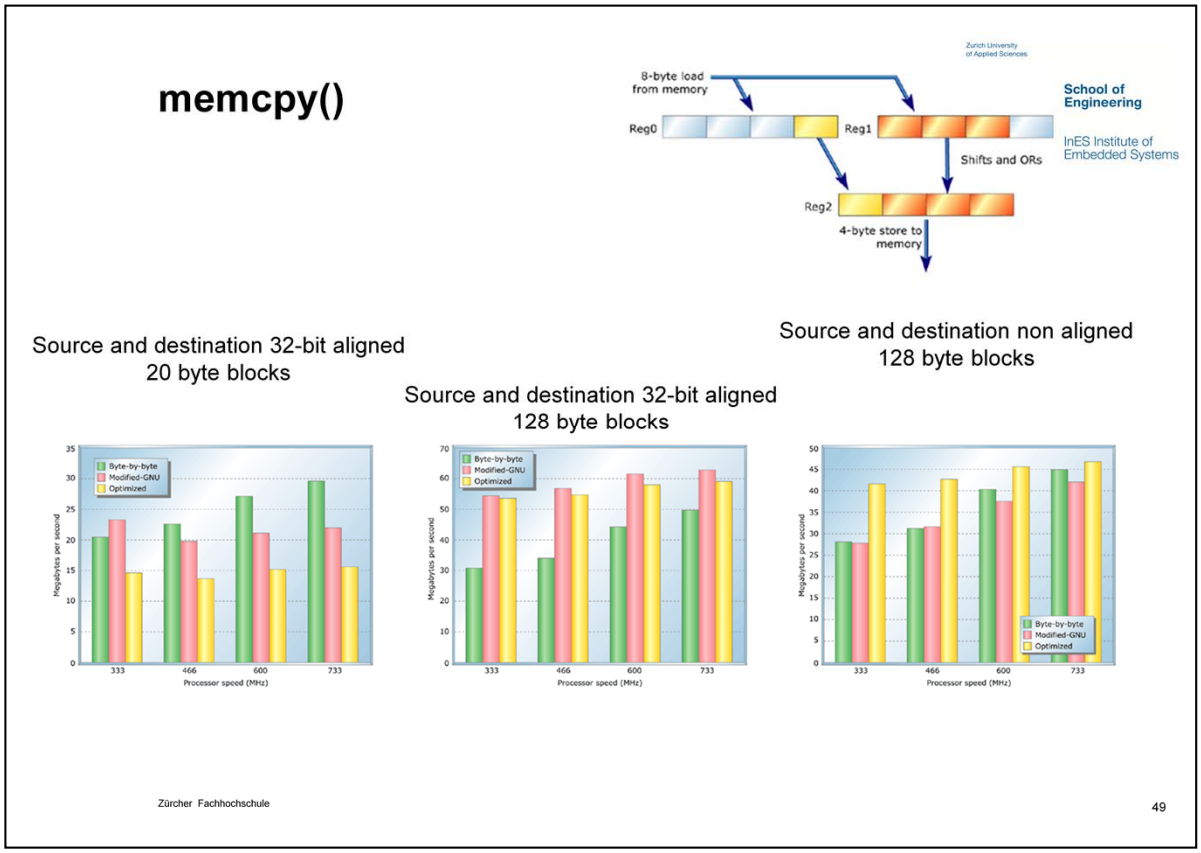
PANDA , DUTT, NICOLAU (2000) On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems. ACM Transactions on Design Automation of Electronic Systems, Vol. 5, No. 3, July 2000, Pages 682–704

PANDA , DUTT (2002) Memory Architectures for Embedded Systems-On-Chip. S. Sahni et al. (Eds.) HiPC 2002, LNCS 2552, pp. 647–662, 2002

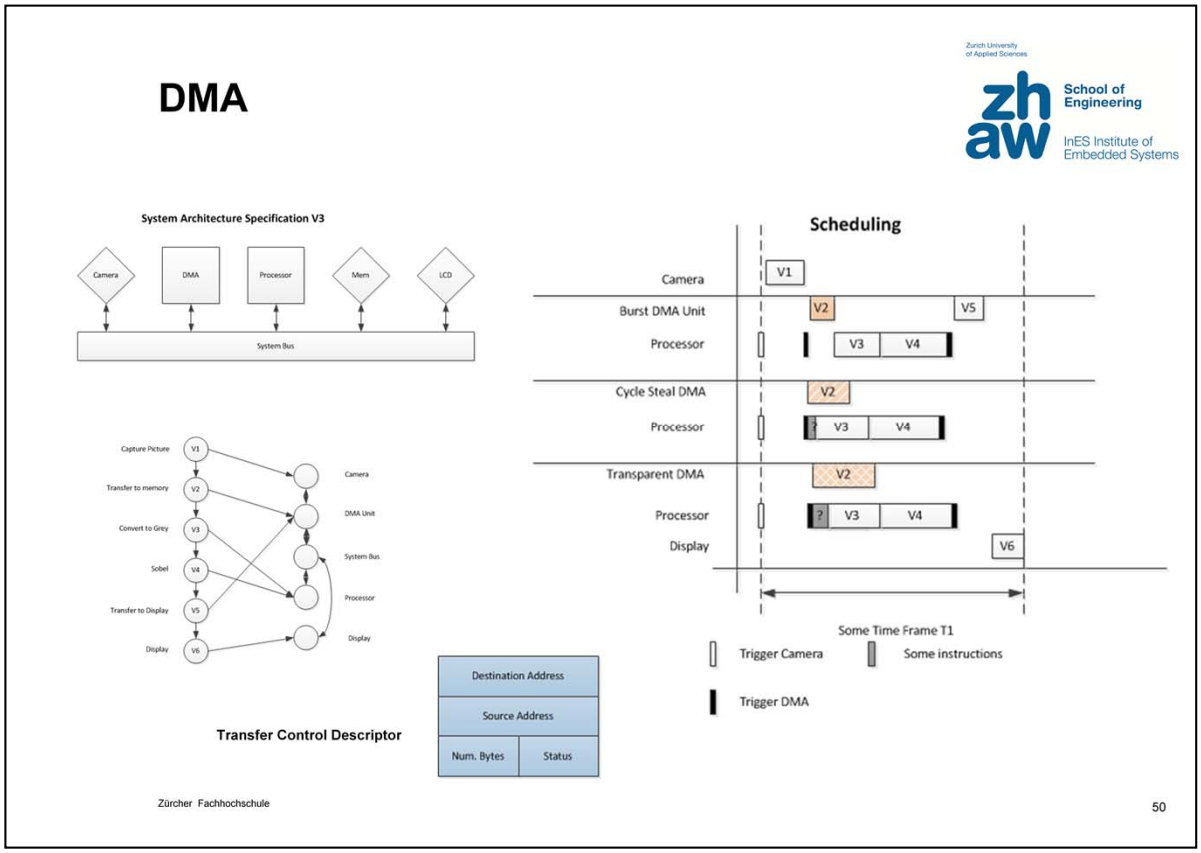
Contents

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Cache – using Cache
- Scratchpad – using Scratchpad
- Memcpy – DMA – Hardware Acceleration



memcpy() is a standard C library function used for copying blocks of memory around. It's the kind of thing that lots of people just use without considering what it does. Unfortunately its behaviour can cause lots of surprises. There is no standard implementation but there are several derivatives. The charts above were taken from an article from eetimes by Michael Morrow -> http://www.eetimes.com/design/embedded/4024961/Optimizing-Memcpy-improves-speed but the issue is well known. Morrow compares a simple byte-for-byte (green – first column), one that recognises byte alignment on 32-bit boundaries (pink – middle column) and one that manipulates the data as shown as shown in the diagram on the top. He then goes and compares the performance of the three algorithms on various data transfer sizes. Given that there is no cache enabled and the source and, in the first two cases, destination addresses are on 4-byte boundaries, the first chart shows the performance for 20 byte data blocks, the second for 128 bytes and the third for his optimised memcpy() for non-aligned source and destination addresses. Its clear that each algorithm has its pros and cons and the message here is that we should think about what we are trying to do before just using memcpy. In our Institute we have often re-written memcpy() for specific performance features.



We can allocate a DMA unit – here also we need to look at how DMA operates.

DMA is judged to be faster than a memcpy() because the instruction (and loop) overhead is expensive. On the other hand memcpy()'s are easy to schedule as they are merely part of the sequential program. DMA implements the loop in hardware which makes it faster than memcpy() but it is a different bus-master and hence it contests the bus (or resources such as memory) with a continuing executing resource, the CPU. It also requires scheduling. The whole point of the DMA is to work in parallel with a computing resource. This means that there are three steps

1.) Initalise DMA (CPU)

2.) Perform DMA (DMA -> non linear improvement)

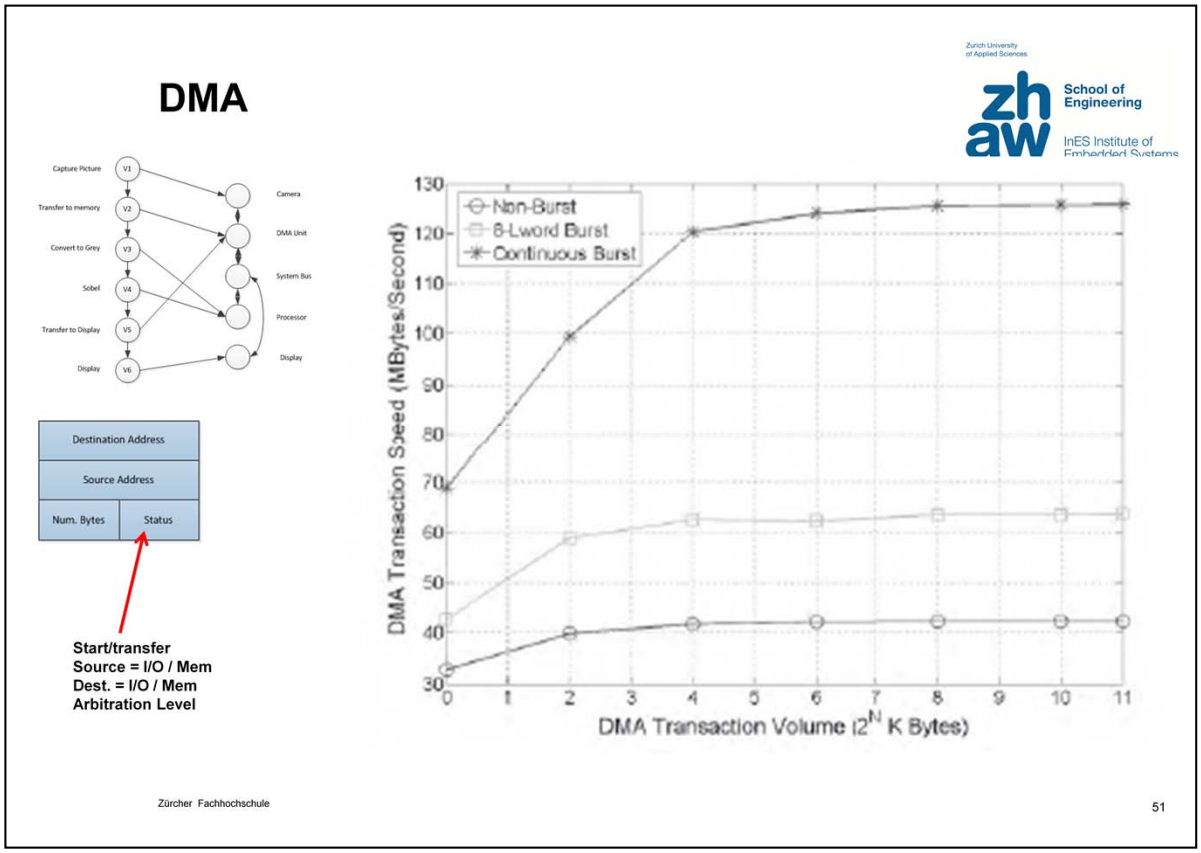
3.) Post-hoc processing (CPU)

There are several operation modes of DMA

Burst Mode: the contiguous block is transferred in one – this means that the resource (system bus and/or memory) is solely reserved by the DMA unit and the CPU has no access to the resource – i.e. either it essentially stops or, if it can run solely from cache, it does its bit internally.

Cycle Stealing Mode: by continually requesting and yielding the control of the bus/resource the DMA controller can force interleaving which, if the CPU is the priority master, should mean that the CPU is not much braked. This does however mean that the DMA transfer takes longer.

Transparent Mode: In this case the CPU never waits for the resource (f.i. system bus) and the DMA only transfers when the CPU doesn't need the bus. This requires that the DMA and the CPU must be intimately connected to ensure that the prediction (i.e. the fact that the CPU will not require the bus for a certain number of cycles) functions correctly. Obviously this affects the run-time of the program least but it does increase the length of the DMA transfer, and hence the achievable scheduling. …



… Flyby Mode: this isn't supported by many controllers but it transfers data in one cycle rather than the read/write cycle.

Burst transfers are – more or less – easily calculable. In Transparent and Cycle Steal Mode it becomes necessary to calculate the approximate WECT of the DMA transfer and its interaction with the program WCET. One example of how to do this is:

Tai-Yi Huang; Liu, J. W -S; Hull, D., "A method for bounding the effect of DMA I/O interference on program execution time," *Real-Time Systems Symposium, 1996., 17th IEEE* , vol., no., pp.275,285, 4-6 Dec 1996

In the diagram above Peng et. Al developed a DMA unit for PCI and made measurements using various settings as shown above and the type of bus cycle used and the bandwidth profiles above make interesting reading.

Yu Peng 2 Yanmeng Ba1 Bo Li3, **A Design of PCI-Bus High Speed Serial Communication Card.** Proc. Of the Ninth International Conference on Electronic Measurement & Instruments ICEMI'2009. (http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05273990 .

2-D DMA

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

```
for y = 1 to YCOUNT /* 2D with outer loop */
    for x = 1 to XCOUNT /* 1D inner loop */
    {
            /* Transfer loop body goes here */
    }
```

(a)

0x1	0x2	0x3	0x4
0x1	0x2	0x3	0x4
0x1	0x2	0x3	0x4
0x1	0x2	0x3	0x4
0x1	0x2	0x3	0x4

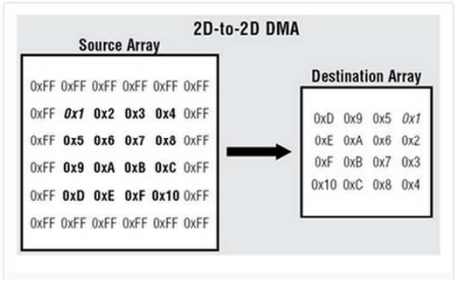
(b)

Address	Data
0	0x1
1	0x2
2	0x3
3	0x4
4	0x1
5	0x2
6	0x3
7	0x4
.	.
.	.
.	.
14	0x3
15	0x4

(c)

0x1	0x1	0x1	0x1	0x1
0x2	0x2	0x2	0x2	0x2
0x3	0x3	0x3	0x3	0x3
0x4	0x4	0x4	0x4	0x4

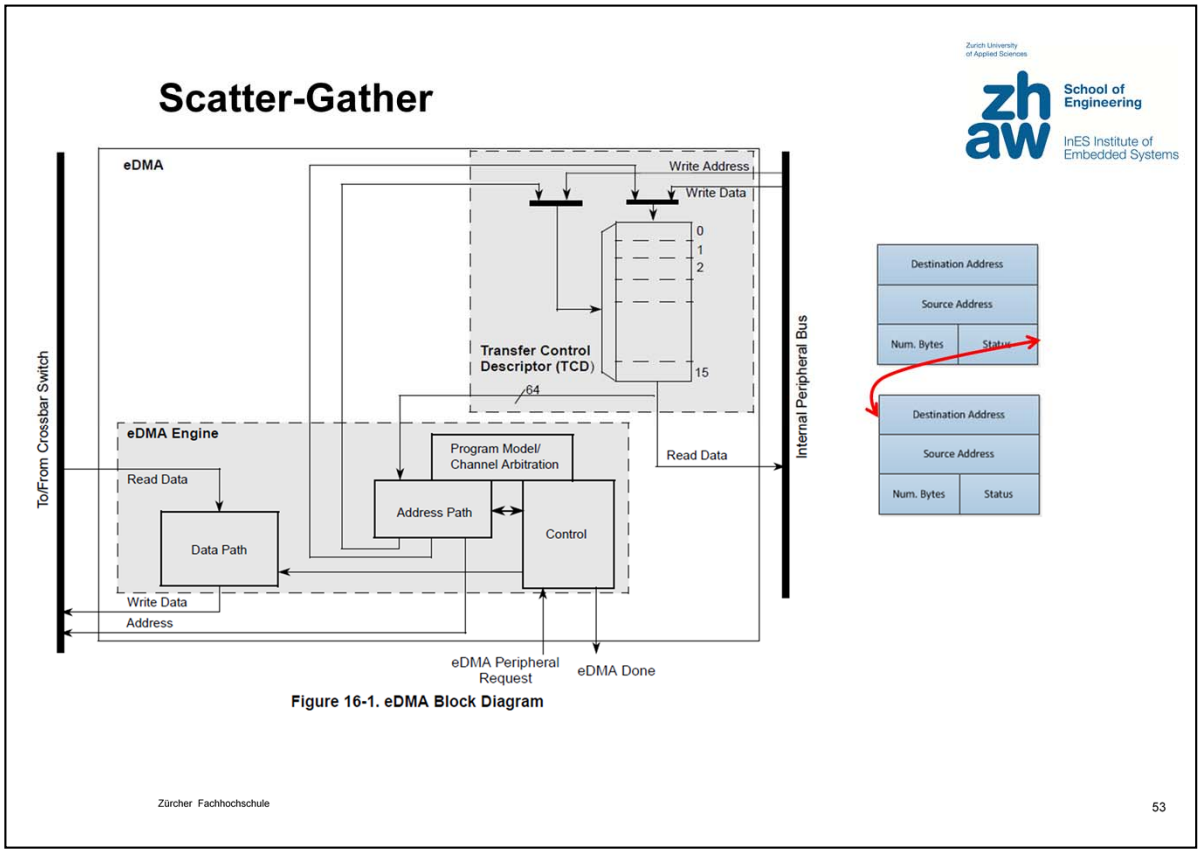
Source	Destination
XCOUNT =5	XCOUNT =20
XMODIFY = 4	XMODIFY = 1
YCOUNT = 4	YCOUNT = 0
YMODIFY = -15	YMODIFY = 0



Source	Destination
XCOUNT =4	XCOUNT =4
XMODIFY = 1	XMODIFY = 4
YCOUNT = 4	YCOUNT = 4
YMODIFY = 3	YMODIFY = -13

From: http://www.embedded.com/design/mcus-processors-and-socs/4006782/Using-Direct-Memory-Access-effectively-in-media-based-embedded-applications--Part-1

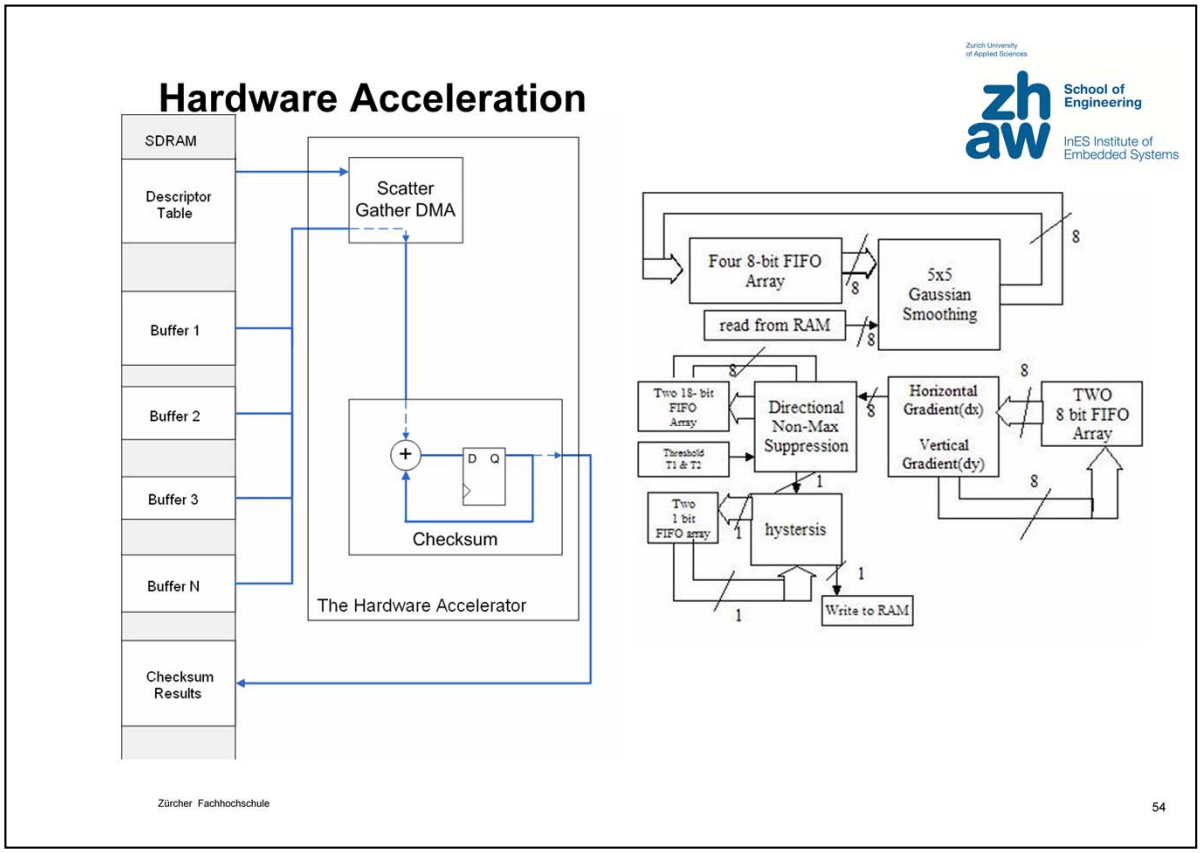
2-D DMA is useful for video processing



In microcontroller technology we are used to using DMA to collect data from some peripheral and move it into memory for us to process in software. Once it is in software however we generally tend to manipulate it from there in software, using for instance memcpy() or modified versions thereof.

We also know from our lectures that – specifically looking at our Sobel implementation – we may benefit from not only having data correctly placed in cache but data being transferred into system memory/cache in an order that considers softwares requirements. Such a beast exists in the form of a scatter-gather DMA and is widley used in fields such as high-performance networking. Scatter-gather is farily simple in that the DMA controller works off a list of transfer sizes – the DMA engine, which is SG capable, for the MFC5208R is shown above. Essentially it performs a series of DMA transfers and can be programmed to group these in scatter-gather mode – in essence leaving the host controller undisturbed for re-programming.

Scatter-gather is a feature of many operating systems – Ethernet drivers for one often requiring their use. http://www.skbuff.net/skbuff.html for linux networking and http://www.linuxjournal.com/article/7104 http://comp.ist.utl.pt/ec-sc/0304/docs/ecos-2.0b1/doc/html/ref/io-eth-drv-api-funcs.html for eCos. Linux requires SG for all memory manipulations over the standard page size (4k for many platforms) and given that uCLinux also organises itself into pages it will require it there too.

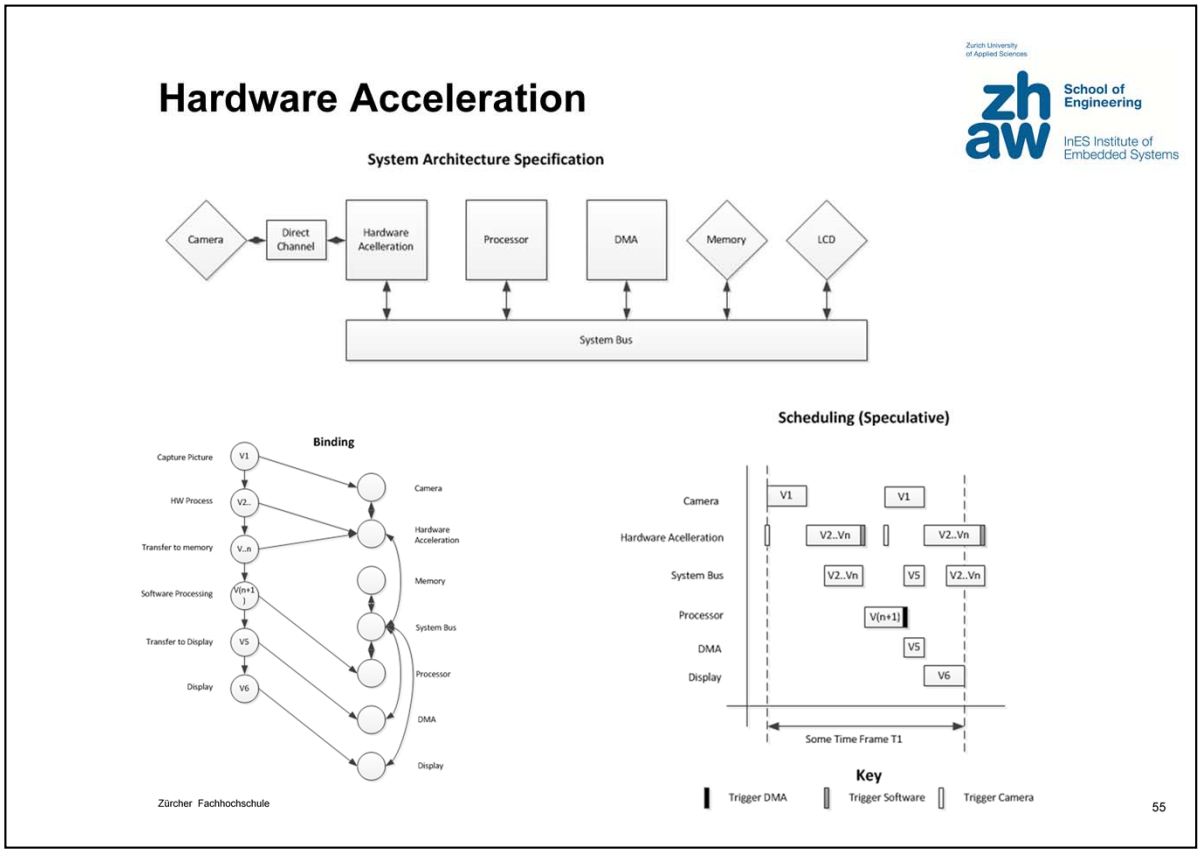


Hardware acceleration is a concept with many definitions – it is generally used when describing something that has been moved from software to hardware but also more specifically when connected with a data flow. Altera (left) uses it in this context when it describes an example implementation where a checksum generator is added to a scatter-gather DMA -> further details are available at: http://www.altera.com/literature/an/an417.pdf.

Altera uses a C2H compiler that generates code from a c file and replaces the c code with the hardware function. The

Take a look at Venkatesan et. Al (right) . http://www.uweb.ucsb.edu/~shahnam/HAoEDAoF.pdf . In this paper they generate a pixel every clock cycle and can process 30 frames/sec. Which, whilst they don't use a Sobel algorithm is interesting enough for us. Its questionable whether this is in the data path or not – but could be integrated into it.

The advantage of placing accelerators into the data stream is that is completely transparent for the programmer, the disadvanatge is that the origional data is lost and that the updatability for instance for specific operational environments is compromised.



Formally speaking we are beginning to exploit **Task Parallelism** and, to a certain extent **Data Parallelism**.

What the HA can easily do is to calculate the grey value from the RGB stream – the information required is local. But the Sobel is another matter as there are several steps before the final pixel value can be calculated. What could be exploited here is data parallelism which however does require (substantial) memory. This is what Graphics Processors do – exploit data parallelism.

Due to task parallelism we now have two processing elements which need to be synchronised – In the scheduling attempted above the system is synchronised by the hardware accelerator.

Conclusions

Zurich University of Applied Sciences
School of Engineering
InES Institute of Embedded Systems

- Cache is simple to organise in HW but requires thought to gain optimal benefit
- Tightly coupled memory is an embedded favorite but requires more run-time effort
- Moving data can be flexibly arranged and represents a first foray into parallel processing