

CACHE MEMORY

HOW CACHING WORKS

Cache memory acts as a buffer between the processor and the main memory so that the processor can read data from the fast cache rather than directly from the slow main memory in many cases. The proportion of memory accesses satisfied this way is referred to as the *cache hit ratio*. This is often 90% or higher.

Even within very large programs, only small portions of code generally get used at once. Most programs spend most of their time working in one small area of the code (called the current *working set*) at a time, often performing the same work many times over and over on different data, and then move to another area. Access to data is similarly repetitive.

If a program accesses a word of memory, chances are therefore high that in the near future it will access the same word again. Chances are also high that in the near future it will access a word close to the one just accessed. These two observations are known as the principles of *temporal* and *spatial* locality, respectively. Because of temporal locality, it pays to keep a word in the cache once it has been brought there. Because of spatial locality, it pays to load several adjacent words at once into the cache.

PARTS OF A CACHE

The cache, now part of the processor itself (*on-die L2 cache*), is made up of two main components :

- **the data store** : This is where the cached information is actually kept. When reference is made to « storing something in the cache » or « retrieving something from the cache », this is where the actual data goes to or comes from. The size of the cache refers to the size of the data store. A typical L2 cache size is 1MB. The larger the data store, the higher the probability that it will be able to satisfy most requests. A cache is generally arranged in 32- or 64-byte lines.

- **the tag RAM** : This is a small area of memory used by the cache controller to keep track of where in main memory the entries in the data store come from. There will be as many entries in the tag RAM as there are in the data store, although of course these entries are only a few bits wide, not 32 or 64 bytes wide like those in the data store.

For each cache entry, the tag contains the 4 (or 8, or 16...) most significant bits (MSB) of the address in main memory where that data is stored. When the cache controller wants to determine whether the data requested by the processor is already in the cache (i.e., when the controller checks for a *cache hit*), it compares these 4 (8, 16...) bits in one or several tag locations (depending on cache design, as explained below) with the corresponding MSB bits of the address it gets from the processor.

If they are identical, then the controller knows that an entry present in the cache is the one the processor wanted, i.e. we have a cache hit, and the controller reads the requested data from the appropriate line in the data store. (If no entry in the tag RAM matches, we have a *miss*, and a fetch from main memory is then required.) As the tag RAM is read first when checking for a hit, it must be still faster than the data store, because we want to be able to check the tag and still have enough time to read the data store within a single clock cycle if we have a hit.

In addition to these memory areas there is of course the **cache controller circuitry**. Most of the work of controlling the L2 cache is performed by the system chipset.

CACHE DESIGN (MAPPING TECHNIQUES)

As the cache is much smaller than the system memory, a very important factor in determining its effectiveness is how the cache is *mapped* to system memory - in other words, the first question to be decided in a cache design is : into which cache line should a given block loaded from memory be stored ? And, consequently, where in the cache should that block be looked for when needed ? There are three different ways this mapping can be done :

- **Direct mapped cache** : All main memory addresses with the same 4 (or 8, or 16...) least significant bits (LSB) share (i.e., are mapped to) the same single cache line; only one of these addresses can use that line at a given time. So there is only one location in cache memory where the contents of a given address in system memory can be stored - and subsequently found. This is the simplest and fastest technique because only one comparison needs to be made by the controller : only the MSB of the requested address need to be compared to the tag of the indicated line. (The middle bits of the address identify the line.) But the hit ratio is relatively poor because of possible data conflicts : if two frequently accessed blocks of main memory map into the same cache line, they will constantly be evicting each other from the cache.

- **Fully associative cache** : Any line in the cache can be associated with any main memory location. The cache controller has to check every entry in the tag RAM before signalling a hit or miss. This mapping technique offers the best theoretical hit ratio since it solves the problem of conflicting addresses, but it is very slow.

- **N-way set associative cache** : «N» here is a number, typically 2, 4, or 8. This is a compromise between the direct mapped and fully associative designs. The cache is divided into many sets of N lines each, e.g., 4. Each memory address is assigned a set (much the same way each memory address is assigned

a unique cache line in a direct mapped cache, based on its least significant bits), and can be cached in any one of those 4 lines within that set. In other words, within each set the cache is fully associative, hence the name. But 4 comparisons at most are needed, which makes it much faster than a fully associative cache. Moreover, the hit ratio is very high because two addresses that would be conflicting in a direct mapped cache can be stored in two different lines of the same set.

When a new entry must be added in a fully associative or N-way set associative cache and the cache is already full, an algorithm must be used to decide which cache line will be replaced, or *flushed*. Ideally, the LRU (least recently used) line is replaced : data that have not been accessed for a long time are less likely to be needed again in the near future.

WRITE POLICY

In addition to caching reads from memory, the system is also able to cache writes to memory. There are two different ways that the cache can handle writes, and this is referred to as the write policy of the cache.

- **Write-back cache** : When a write is to be made to system memory at a location that is currently cached, the new data is only written to the cache, not actually written to system memory. Later, if another memory address needs to use the cache line where this data is stored, it is saved (« written back ») to the system memory and then the line can be used by the new address.

- **Write-through cache** : With this method, every time the processor writes to a cached memory location, both the cache and the corresponding memory location are updated. The data just written is in the cache in case it is needed to be read by the processor soon, but we still have to initiate a memory write operation each time.

Comparing the two policies, in general terms write-back provides better performance, but at the slight risk of memory integrity. Write-back caching prevents the system from performing many unnecessary write cycles to the system RAM, which can lead to noticeably faster execution. However, as the RAM itself isn't actually updated until the cache line is flushed to make room for another address to use it, at any given time there can be a mismatch between the contents of several cache lines and the memory addresses that they correspond to. When this happens, the data in the memory is said to be stale, since it doesn't have the fresh information yet that was only written to the cache. Memory used with a write-through cache can never be stale because the system memory is written to at the same time that the cache is.

Normally, stale memory isn't a problem, because the cache controller keeps track of which locations in the cache have been changed and therefore which memory locations may be stale. This is done by using a single extra bit of tag RAM, one per cache line, called the *dirty bit*. Whenever a write is cached, this bit is set (made a 1) to tell the cache controller that the current

contents of the cache line must be written back to memory before that cache line is re-used for a different memory address. But the use of a write-back cache does involve the small possibility of data corruption should something happen before the « dirty » cache lines are saved to memory.

(Adapted)

Vocabulary

set : groupe, ensemble

(working set : groupe actif)

over and over = again and again

it pays = it brings good results

die = chip (µP)

tag : (littéralement) étiquette

to keep track of sthg : garder une trace de, surveiller

entry : entrée (= élément dans une série)

MSB → 10010110 ← **LSB**

to check for sthg = to check if sthg is present

to match sthg = to correspond to sthg, to be the same as

to map A to B = to make elements of A correspond to elements of B.

a memory block *maps* into a cache line =

must be stored in that cache line

direct mapped : à associativité directe

to share : partager, avoir en commun

n-way set associative : à associativité partielle

(à n voies)

at most >< at least

to evict sb from : chasser qqn de

to flush : vidanger

a policy : une politique

to update : mettre à jour

mismatch : non-correspondance, divergence

stale / fresh (air) : (air) vicié / frais, neuf

whenever = every time

Sans l'aide d'aucun document (sauf dictionnaires explicatifs), et après avoir lu l'ensemble du texte anglais ci-joint (Cache memory),

- A. (18 ⇒ 6) Sur une feuille distincte, traduire en français les phrases soulignées et numérotées de 1 à 11 dans le texte
- B. (22 ⇒ 10) Sur cette feuille même, répondre en français aux questions suivantes portant sur la compréhension de ce texte. (T/F ? = true or false ? Explain your answer.) Formulez votre réponse en y faisant figurer les termes de l'énoncé, pour montrer que vous l'avez bien compris. (NB : on peut me demander de corriger immédiatement la traduction d'un énoncé en cas de doute.)

1. What is referred to as the working set of a program ? Why is that concept important in relation to cache memory ? (2)

2. What two principles dictate what will be put in cache memory ? Explain. (2)

3. What does an entry in the tag RAM contain ? Why must the tag RAM be faster than the SRAM used by the data store ? (2)

4. When does a cache miss occur ? What happens then ? What proportion of memory accesses results in a cache miss ? (1.5)

Which component of the system determines that a cache miss has occurred ? How does it do so ? (2.5)

Nom :

5. What is called a mapping technique ? (1,5)

6. What two criteria are used to judge the effectiveness of different mapping techniques ? Explain in a few words. (2)

Which mapping technique is most efficient ? Why ? (Use the criteria defined above). (3)

7. An LRU algorithm (explain) is not used in a direct-mapped cache. T/F ? (2.5)

8. How can writes be handled by a cache system ? Explain. (1.5)

9. What is a *dirty* bit in this context ? What is it used for ? (1.5)

ATTENTION

Le texte sur les caches (examen de juin 2006) a été distribué lors de la dernière interro (Linux) et devait être préparé pour le cours suivant (10 mai). Il s'agissait d'une simulation d'examen, le questionnaire aurait dû être distribué, travaillé et corrigé collectivement ce jour-là. Cette deuxième partie de l'exercice n'a pas eu lieu, pour une raison dont je ne retrouve pas la trace écrite dans mon agenda de cours (correction de Linux trop longue ? imprévu ? c'était l'après-midi de l'exposé sur l'Europe...) Quoi qu'il en soit, comme la préparation devait être faite, j'ai considéré que le texte faisait partie de la matière.