8. Memory hierarchies and cache architecture

In this Section we complete the memory hierarchy treatment and study the main memory – cache hierarchy in-depth: cache architecture, caching levels, performance evaluation, and program optimizations.

Paged memory hierarchies

As introduced in Section 7, in a paged hierarchy objects are decomposed into fixed size blocks, which represent the *transfer unit* between the memory levels of the same hierarchy M2-M1, where M2 is the upper level.

In the following we'll denote by γ the capacity of a memory level (γ_1 or γ_2), by σ the page size (by definition, the same for M2 and M1), and by $\nu = \gamma/\sigma$ the number of pages of a memory level. Typical values of these parameters for VM-M and for M-Cache have been mentioned in previous Sections.

Locality and reuse

As introduced, locality and reuse are the basic properties to be exploited in order to optimize the memory hierarchy utilization. Let us define them formally. Consider the following simple program P:

```
int A[N];
∀ i = 0 .. N-1: A[i] = A[i] * A[i]
```

compiled as:

```
LOOP:   LOAD  RA, Ri, Ra
        MUL  Ra, Ra, Ra
        STORE  RA, Ri, Ra
        INCR  Ri
        IF<  Ri, RN, LOOP
        END
```

Let us observe a trace of logical addresses generated by the program execution, assuming that instructions are allocated in VM_P starting at address 0 and A at address 1024. The initial prefix of the trace is the following (in absence of interrupts and exceptions):

0, <u>1024</u>, *1, 2*, <u>1024</u>, *3, 4, 0*, <u>1025</u>, *1, 2*, <u>1025</u>, *3, 4, 0*, <u>1026</u>, *1, 2*, <u>1026</u>, *3, 4, 0*, <u>1027</u>, …

We observe that, inside a sufficiently wide temporal window, *some* (two in the example) *address sequences are intermixed*:

[0, 1, 2, 3, 4]

[1024, 1024, 1205, 1025, 1026, 1026, 1027, …]

In any sequence, addresses are relatively close one another compared to the size of the temporal window; in the example, they are consecutive or equal two-by-two. This concept corresponds to the ***locality*** property (also called *spatial locality*).

Moreover, some sequences are repeated several times in the temporal window (in the example, the first sequence), or some references are repeated in the temporal window (as in the second sequence of the example). This concept corresponds to the ***reuse*** property (also called *temporal locality*).

By organizing information in pages, the probability that referred information belongs to a limited number of pages, and that the referred pages vary slowly, is relatively high. In our example, the same code page is used for the whole duration of program execution, while the same page of data is used for 2σ consecutive references to A.

Thus, in the example:

- once the code page has been loaded into the lowest level of the hierarchy, the transfer time is no more paid provided that we are able to impose that this page is not replaced. This is the way to exploit the *reuse* property;
- once a page of A has been loaded into the lowest level of the hierarchy, the transfer time is no more paid for 2σ references to A, provided that such page is not replaced during the time interval of σ iterations. This is the way to exploit the *locality* property.

The total number of faults during the program execution is given by:

$$1 + N/\sigma \sim N/\sigma$$

The importance of reuse

The following example shows a computation in which potential reuse exists for the data too:

```
int A[N], B[N];
∀ i = 0 .. N-1
        ∀ j = 0 .. N-1
                A[i] = F (A[i], B[j])
```

The pages for the code (their number depends on the compilation of function *F*) are characterized by reuse.

The pages of A and B are characterized by locality. Moreover, the pages of B are characterized by reuse too: for each outermost iteration, the *N* innermost iterations refer all B's elements. Depending on the capacity of the lowest level M1, reuse is exploited provided that all B's words (*N* words) can be contained in M1 for the whole duration of the program execution.

For example, consider an M-Cache hierarchy where the cache capacity is γ_2 = 64K words and the block size is σ = 16 words, thus ν_2 = 4K blocks (the term *block* is used for the M-Cache hierarchy as synonymous of page). Assume that the D-RISC code consists in 128 instructions, and that *N* = 32K. The code allocation requires 8 blocks; B allocation requires 2K pages. *If proper mechanisms are provided at the assembler and firmware level*, then all the blocks of code *and* of B can be maintained permanently in cache once loaded for the first time. For array A, it is sufficient that only one block at the time is present in cache (the current block). As a results, 2K + 9 (~ 2K) blocks over 4K are used by this program execution. In this condition locality and reuse are exploited optimally. The total number of cache faults is given by

$$8 \text{ (for code)} + N/\sigma \text{ (for A)} + N/\sigma \text{ (for B)} = 2\,N/\sigma + 8 \sim 2\,N/\sigma$$

However, if (for example) N = 128 K, then the potential reuse on B *cannot* be exploited, since B cannot be contained entirely in cache. In this case, the number of cache faults for B is one order of magnitude greater:

$$\sim N^2/\,\sigma$$

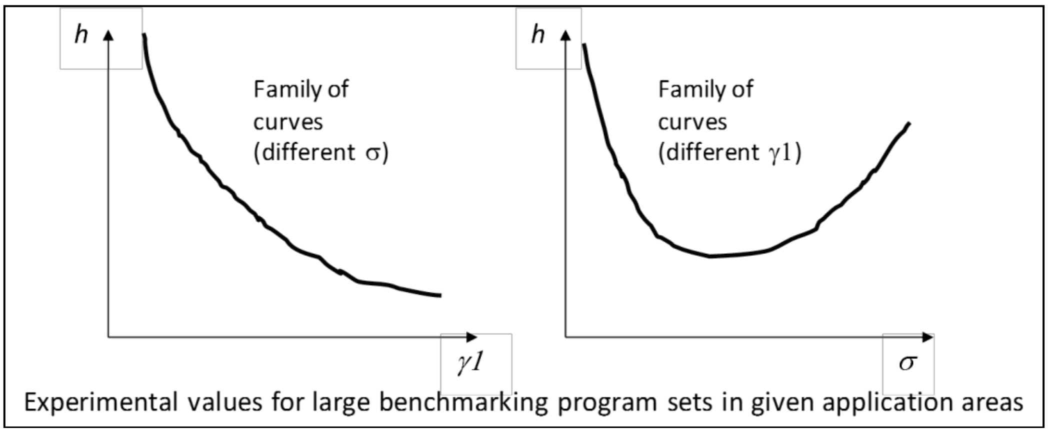
since only the locality property is exploited for B.

Notice that, in the latter case, a very marginal, or even no, advantage is achieved by maintaining in cache more than one block of B at the time.

The explained importance of reuse in program optimizations opens a very interesting area of research about *cache-aware* algorithms and computations. In our example, if B is too large, in principle it is possible to think about a different algorithm organized as a sequence of steps, during each of which only a smaller fraction of B is employed, so that reuse can be exploited during every step and for the whole computation (*block-structured algorithms*). This is a case of cache-awareness: the programmer is aware of the cache performance problems, and designs the application with proper algorithms. In alternative, the compiler could restructure the program by adopting a different algorithm. By now, the importance of these issues in real applications has been widely recognized [A5] and justifies intensive research efforts.

Optimal page size

As introduced, conceptually the fault probability *h,* as a function of γ_1 and σ, has the qualitative shape shown in the following figure:



While the function $h(\gamma_1)$ is quite self-explicative, the analysis of the function $h(\sigma)$ deserves more attention. The existence of a *minimum*, thus of an *optimal page size*, is consistent with the locality property: on one hand, by increasing σ the locality inside every single page is improved; however, by increasing σ the number of pages in M1 decreases and, under some values, this number is not sufficient to simultaneously maintain all the pages belonging to the intermixed sequences.

For example, in a computation like:

```
int A[N], B[N], C[N], D[N];
∀ i = 0 .. N-1: A[i] = F (A[i], B[i], C[i], D[i])
```

we have five intermixed address sequences (belonging to the code, A, B, C, and D), thus the block size must be such that at least five blocks are contained in the lowest level of the hierarchy.

Large benchmarks of programs for given application areas provide values of σ and γ_1, also according to the architectural characteristics of the memory supports. See the typical values for VM-M and M-Cache hierarchy, e.g.

- σ = 1K and γ_1 = 1G for VM-M (note: 1G per process),
- σ = 8 and γ_1 = 64K for M-cache.

Such values are very rough average evaluations that give just an order of magnitude. Each computation has its own optimal values of σ and γ_1, however these values cannot be exploited in practice, since σ and γ_1 must be necessarily fixed for each architecture at the firmware design time. We can say that the typical values obtained with the benchmarks are just used for designing a reasonable firmware architecture. It is a task of the compiler/programmer to exploit the firmware level at best.

Working set

Given a M2-M1 hierarchy, the working set of a program for such hierarchy is defined as *the set of pages (blocks) which, if simultaneously present in M1, minimize the fault probability*.

This concept is a quite fundamental one for the optimization of programs, since, in modern architectures, the page/block transfer time is one of the most important sources of performance degradation. Thus, the objective of an optimizing compiler is to recognize *how many* and *which* pages/blocks belong to the working set, and to verify whether the working set pages can be effectively maintained in M1.

In the examples, the working set WS for the M-Cache hierarchy is evaluated as follows:

- first example (single loop on A):

```
int A[N];
∀ i = 0 .. N-1:
        A[i] = A[i] * A[i]
```

WS is given by the code block and the current block of A. This very small working set can be effectively exploited in any machine;

- second example (double nested loop):

```
int A[N], B[N];
∀ i = 0 .. N-1
        ∀ j = 0 .. N-1
                A[i] = F (A[i], B[j])
```

WS is given by 8 blocks for the code, the current block of A, and *all* blocks of B (total: ~ N/σ blocks). This working set is effectively exploited if it can be contained in the cache capacity: the compiler knows this architectural characteristics (*the compiler knows all the most relevant architectural characteristics under the form of the abstract machine definition*), thus it is able to verify whether the program is implicitly optimized (at least from the cache exploitation point of view), or more sophisticated optimizations and program restructurings have to be implemented (cache-aware approaches).

The impact of the architecture and of the *specific* memory hierarchy is very important in studying the working set of a program.

Consider again the second example from the point of view of the *VM-M* hierarchy. This situation has strong differences compared to the M-Cache hierarchy. First of all, to be executed the program must have been already loaded into M, while the program is loaded into the cache only when it is running. For the VM-M hierarchy, the working set includes the whole program (code, A, and B), since this is the situation that minimizes the number of main memory faults. The system will try to load the whole program into M at creation time, thus implicitly satisfying also the reuse requirements for B. Of course, this is feasible because of the rather large capacity of the main memory. Possibly, some *annotations* are associated to applications (using the configuration file) in order to characterize them from the point of view of the main memory requirements. The process creation and loading will be affected by this feature: possibly, if the required conditions cannot be satisfied, the process creation is postponed or some main memory pages are subtracted to other lower-priority processes.

Paging strategies: on-demand vs *prefetching*

All previous examples have been done under the assumption that (for a M2-M1 hierarchy) a page/block is loaded into M1 only if and when it is affected by a fault. This is the basic, and most common, memory hierarchy strategy, called ***on-demand*** paging.

In principle, it is possible to think of a more efficient strategy by loading pages/blocks in advance with respect to their effective references. In the first example, a static analysis of the computation shows that, once the *i-th* block is loaded into M1, the next block to be used will be the *(i+1)-th* one. Thus, a proper architecture could provide to load the *(i+1)-th* block in parallel, while the program is working on *i-th* one. In this example, the number of fault reduces to zero (more precisely, to 2). Analogous considerations apply to the second example. This strategy is called ***prefetching***.

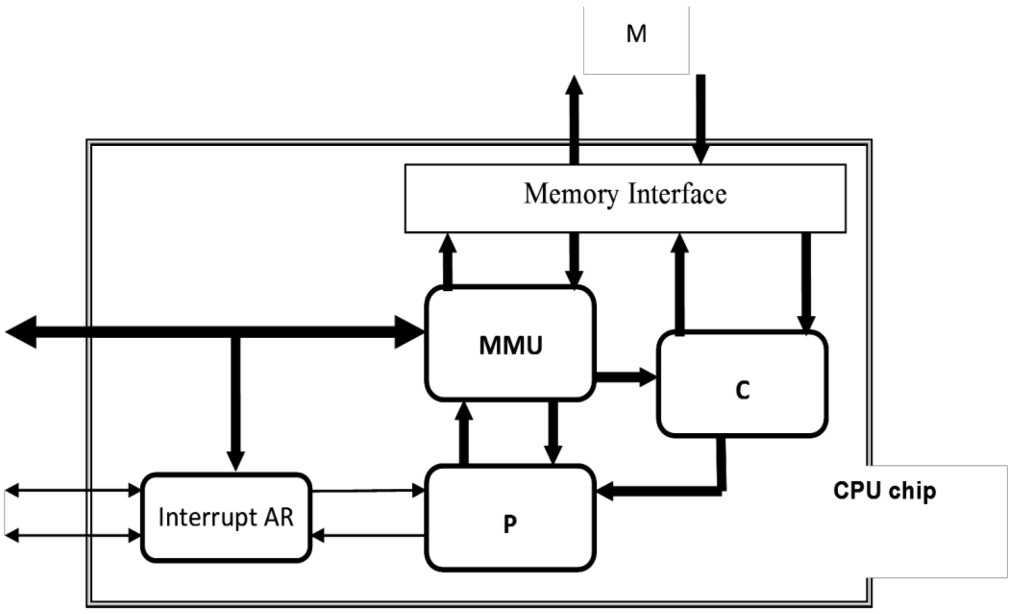
Provided that the firmware architecture is able to load new blocks in parallel with the processing to the currently loaded ones, prefetching technique has an impact on the assembler level and the compiler. That is, the compiler must detect that a certain form of prefetching can be applied and insert *special instructions or annotations* to the code.

Though interesting, the prefetching strategy is not so simple to be applied and implemented. Programs might have more complex access patterns: in general, after the *i-th* block the computation could require the *j-th* one, with $j \neq i+1$, or even *j* variable. Not only this situation must be detected and *j* possibly evaluated, but the *i vs j* relation must be represented at the assembler level. Many machines adopt a "by-default", i.e. automatic, prefetching strategy where the *(i+1)-th* block is *always* loaded in parallel to the *i-th* one utilization, without a compiler analysis. In fact, several benchmarks exist showing that some programs are under-optimized, because useless blocks are loaded and subtract space to other useful blocks. In the second example, applying prefetching to A can subtract useful (and much more important) space to B.

Automatic prefetching with $j = i+1$ can be effectively applied to *instructions*, and in fact this technique is adopted in many machines. In the same way, *reuse* to instructions is applied automatically, at least for small-medium size loops.

Cache memory architecture

The following figure shows a possible scheme of the CPU with the elementary architecture:



The cache processing unit C contains the *cache memory logical component* MC (e.g. 16K - 64K capacity) and all the needed Operation Part hardware resources. After the successful logical address translation, MMU sends the request, including the main memory physical address, to C.

The cache unit provides to translate the main memory physical address into the cache address, to serve the request and to send the access result to P. In case of *cache fault* (or *cache miss*) C requests the cache block to the main memory and stores it into a MC block, in general by replacing an existing block through a proper replacement strategy. The cache fault and the corresponding block transfer are *fully invisible to the processor*, which merely waits the access results from C.

Implementation of writing operations

Two main methods are used to implement writing operations in a memory hierarchy:

- ***Write Back***: a modified block is copied into the upper memory level only when it is replaced;
- ***Write Through***: every writing operation is executed simultaneously in cache and in the upper memory level.

Write Back is used also in the VM-M hierarchy, while Write Through cannot be applied in VM-M for obvious latency reasons. With Write Through, MMU can send the memory request both to the cache unit and to the upper memory level.

In cache-based architectures, because of the higher access latency of the main memory, Write Through requires that the writing operations are not implemented in a request-reply manner, i.e. the writing operation is launched but the final outcome is not waited by the processor/MMU (*asynchronous write*). Program completion time might be affected by problem of insufficient memory bandwidth.

Another issue related to writing operation concerns the ***cache fault handling*** when a writing operation is executed. Let us consider the following example:

int A[N], B[N];

$\forall\ i = 0 .. N\text{-}1: A[i] = F\ (B[i])$

Array A is used in a "write-only" mode: thus, when a cache fault occurs, it is not necessary that the corresponding block is transferred from the upper memory level into the cache; it is sufficient that the block is allocated in MC *without any block transfer*. The cache unit can be implemented exactly in this way, and the time overhead for handling such faults is negligible.

For this reason, though this program has $2N/\sigma$ faults for data (in an on-demand cache strategy), only N/σ block transfers occur. By convention, for the sake of performance evaluation, we will evaluate the number of faults as

$$N/\sigma$$

since we are interested in evaluating only *the cache faults causing block transfers*.

If the cache unit behaves in this way for the writing operations (i.e., no block transfer is requested), the compiler must take care to distinguish between write-only data structures and data that, at least with a non-null probability, can be read *and* written. If the first operation on such data is a reading one, there is no problem: the read fault causes the block to be loaded into the cache. For example:

int A[N], B[N];

$\forall\ i = 0 .. N\text{-}1: A[i] = F\ (A[i]\ , B[i])$

Each element of A is first read then written, thus the read operation causes the block transfer. In general, the correctness sufficient condition for read-write blocks is that *a block reading operation must the executed first*. In fact, if the first operation on such block is *not* a reading one, there are cases in which the referred data are inconsistent.

For example, let us consider the following execution sequence on a certain block_h, in which the *i-th* position is written before the *j-th* position is read, with $j \neq i$:

```
….
write $block_h[i]$
read $block_h[j]$
…
```

and the writing operation is *not* preceded by a reading operation on the same block. The fault occurring on the writing operation causes the block to be allocated in cache without transferring it, thus the reading operation returns an inconsistent value for the *j-th* position, which has not been read from the main memory.

In conclusion, if the cache unit is designed with the write-only optimization, then for read-write blocks the compiler must ensure that the first operation on the block is a reading one: if it is not, then the compiler must *force* such reading by restructuring the code in such a way that the writing operation of an element is preceded by a reading operation on an element of the same block (a LOAD instruction is sufficient). For example, the correct code for the previous example could be:

```
….
read $block_h[0]$
write $block_h[i]$
read $block_h[j]$
…
```

so that a fault is caused by the *read* $block_h[0]$ operation and the block is transferred from the main memory into the cache.

Address translation methods for cache memory

Three main methods exist for translating main memory addresses into cache addresses, characterized by precise *correspondence laws between main memory blocks and cache blocks*. Such method has to be efficiently implemented in the cache unit firmware design. Correspondence functions can be defined in an analytic way (the so called *direct* cache), or in a table-based way (*associative* cache), or a mix of these two solutions (*set associative* cache).

In the following, we denote by *BM* the identifier of a block in main memory, by *BC* the identifier of a block in cache, and by *D* the displacement of the information inside a block. For example, with a main memory of 4G maximum capacity (32-bit address), a cache of 64K capacity (16-bit address), and blocks of 8 words (3-bit displacement D), the main memory address is seen as the concatenation (BM, D), where BM is 29-bit wide (512M blocks in main memory), and the cache address as (BC, D), where BC is 13-bit wide (8K blocks in cache).

1. Direct method

In this method, a given main memory block corresponds to, i.e. it can be transferred into, a well-defined cache block. A simple and efficient correspondence function is the following one:

$$BC = BM \textbf{ mod } NC$$

where *NC* denotes the number of cache blocks. If, as usually, *NC* is a power of 2, then BM can be seen as (TAG, BC), where *TAG = MB* ***div*** *NC* and *BC = MB* ***mod*** *NC*. Thus, if the referred block is currently allocated in cache, the C address (BC, D) is just a part of the M address. For this reason, the address translation *per se* is very simple.

The address translation must be coupled with the verification that actually the referred block BM is the block currently allocated in the cache block *BC = BM mod NC*. For example, if the M address is (8K, D) the M block identified by BM = 8K corresponds to the BC = 0 cache block. If currently the 0-th C block contains just the 8K-th M block, then the (*BM.BC*, D) configuration is the correct cache address and the cache access is done in a single clock cycle. Otherwise, a cache fault condition occurs, and the 8K-th M block must be transferred into the 0-th cache block.

This verification is done by comparing the *BM.TAG* field of the M address with the TAG of the block currently allocated in the *BM.BC* cache block. Notice that the TAG information uniquely identifies the M block corresponding to the *BM mod NC* cache block. The implementation uses a register memory TAB, of *NC* capacity, in the cache unit Operation Part. Each location of TAB contains the TAG of the currently allocated block, if any, a presence bit, and other information for cache management (modified block, reuse, and so on). The location in the BC position is compared with *BM.TAG* in a single clock cycle.

Moreover, the verification and the cache access can be done in parallel in the same clock cycle, Thus, in this method the cache unit service time and latency is equal to τ. Taking into account the MMU latency, the cache access latency, as seen by the *elementary* processor, is given by:

$$t_C = 2\ \tau$$

The direct method is very efficient and simple to be implemented. However, the correspondence law is too “rigid”: in general, the consequence is an increased fault probability when more than one data structure is allocated in blocks corresponding to the same cache blocks.

Only some particular objects are not affected by the discussed, notably the *instructions* of a program. For data, we need a more flexible method.

2. *Associative method*

The maximum flexibility is achieved if the correspondence law is fully random, that is each M block can be dynamically allocated into *any* C block. The block replacement algorithm, e.g. LRU, assumes a special importance in this case.

As we know, this is exactly the method used in the VM-M hierarchy. Notice that any other method cannot be applied to the VM-M hierarchy, because M contains pages of more than one process.

The implementation of the associative method requires the utilization of an *associative memory* of *NC* locations (see Section 7) in the Operation Part. The key is BM and the returned information is BC, if the BM block is currently allocated in cache, or a cache fault condition.

The cache unit service time and latency is equal to 2τ, since the associative memory access and the cache memory access are sequential. Thus, taking into account of the MMU latency:

$$t_C = 3\ \tau$$

is the ideal cache access latency for the *elementary* processor.

Compared to the direct method, in the associative method the maximum flexibility is paid with increased access latency and with the utilization of a more complex hardware component (associative memory in place of a normal RAM). This issue of hardware complexity was relevant during the previous generations of computer architecture, while now it has a less meaningful impact.

3. Set associative method

If the hardware complexity issue is considered relevant, the trade-off between the direct and the associative method consists in a "mixed" correspondence law: in part analytic and in part associative. Let us logically organize the cache blocks into *sets*, for example 4 consecutive blocks per set. The analytical correspondence is established between M blocks and C sets, i.e. a given M block can be allocated in a given C set only. Inside the C set, the M block can be allocated in any cache block.

By denoting by *NS* the number of C block sets, and by *SET* the identifier of a C set, the direct correspondence can be efficiently defined as

$$SET = BM\ \mathbf{mod}\ NS$$

If *NS* is a power of 2, SET is simply given by the least significant log_2NS bits of BM. The method implies that, if the referred M block is currently allocated in the C set identified by SET, then we have to search for the block, inside the set, in which the referred block is allocated. Let $TAG = BM\ div\ NS$. The search is efficiently implemented with a RAM table, in the cache unit PO, each location of which contains *NC/NS* TAGS of the currently allocated M blocks, plus the usual C management information and presence bits. This table is acceded by SET and the obtained *NC/NS* TAGS are simultaneously compared with *BM.TAG*. If no comparison is successful, then a cache fault is detected. Otherwise, the position (say *J*) of the successful comparison uniquely identifies the block inside the set, thus the cache address is given by the concatenation (*SET, J, D*).

The cache access latency for the elementary processor is again:

$$t_C = 3\ \tau$$

The hardware complexity is comparable to the direct method, while the flexibility depends on the number *NC/NS* of blocks per set. In large benchmarks, it has been verified that 4-8 block per set are sufficient to achieve a fault probability close to the associative method. However, there are programs in which the *NC/NS* parameter is critical, i.e. when the program works simultaneously on more than *NC/NS* information corresponding to the same set.

As a conclusion, usually the cache memory is decomposed into two units: one for instructions only, and the other for data only. The ***Instruction Cache*** is a direct one, while the ***Data Cache*** is associative or set associative. This decomposition will be really exploited in parallel CPUs to increase the parallelism degree, while in the elementary processor it is just a technique to optimize the fault probability.