INSTITUTO SUPERIOR TÉCNICO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

ORGANIZAÇÃO DE COMPUTADORES

LEIC

Primeiro Trabalho de Laboratório: Hierarquia de Memória

Version 1.0.3

IDENTIFICAÇÃO DO GRUPO:

Número:	**Nome:**

2022/2023

1 Introduction

The objective of this assignment is to design an optimized cache architecture targeted for a specific embedded application: a video encoder. This architecture shall be optimized in order to (i) minimize the monetary cost of the system, and (ii) improve performance by reducing the overall cache miss penalty time associated to the execution of the video encoding algorithm.

The rest of this section briefly describes the targeted application (Section 1.1), presents the requirements and restrictions for the memory hierarchy architecture (Section 1.2), and introduces DineroIV [1], the cache simulator that will be adopted to assess the performance of the memory hierarchy architecture (Section 1.3). In Section 2, guides the design of a cache structure that best matches the application requirements.

To facilitate the analysis, the kit supplied provides a program to generate a trace file which records the accesses to data memory generated by the execution of the encoding algorithm. A trace is a finite sequence of memory references usually obtained by the interpretive execution of a program or set of programs. This trace file will be the input to the necessary cache simulations by DineroIV.

Students MUST deliver the report of this work until the second lab of the third week

1.1 Targeted Application

The application whose performance we wish to improve is a video encoder, more specifically, a *motion estimation algorithm*. This algorithm is usually the most computationally intensive operation of a MPEG-x video encoding system. It aims to improve the efficiency of video compression by exploiting both the temporal and spacial redundancy within an image sequence. To achieve this, the motion estimation algorithm reduces the residual information between contiguous frames by tracking and compensating movement using previously encoded images.

For this assignment, we will work with a real implementation of a motion estimation algorithm. The C source code file of this implementation is available at the course webpage (file `motion_estimation.c`). Further details of motion estimation can be found on the Appendix A.

1.2 Memory Hierarchy Architecture

To improve the performance of the video encoder, it is necessary to design a dedicated memory hierarchy, to speed-up data accesses, taking into account the following set of requirements and restrictions.

The memory hierarchy consists of a dedicated SDRAM memory called *frame memory* and a set of *on-chip* and *off-chip* SRAM cache memories. The frame memory is responsible for accommodating the 8-bit pixel values of the video frames that will be considered for the motion estimation operation. To accommodate the video frames, this memory must have a capacity of 128 kBytes. Each pixel value is stored in its own memory address using a raster format (from left to right and from top to bottom). The cache memories interconnect the SDRAM memory and the motion estimation processor.

The characteristics of the memory devices that can be used in the design of this memory hierarchy are specified in the table below (1MB = $2^{20} Byte$).

Device	Technology	Access Time [ns]	Price [€ / MByte]
Cache L1	SRAM (on-chip)	$2 \times [0.7 + 0.35 \times \log_2(\#ways)]$	10
Cache L2	SRAM (off-chip)	$10 \times [0.7 + 0.55 \times \log_2(\#ways)]$	0.4
Frame Memory	SDRAM	140	0.01

Table 1: Characteristics of the available memory devices.

Due to current market restrictions, the overall price of the desired memory hierarchy (frame memory and caches) cannot exceed **€ 0.020**.

To design an optimal memory architecture, you should investigate a configuration that provides the best performance benefits for the targeted application while abiding by the price restrictions stated above.

Thus, a configuration is optimal when it produces the lowest cost value for the following normalized cost function. This function weighs both these factors by multiplying the *Mean Access Time* by the *Cost of the memory hierarchy* (Frame memory + Caches).

$$\textbf{Cost Function} = \textbf{Mean Access Time}\ [\mathrm{ns}] \times \textbf{Price of the memory hierarchy}\ [€] \qquad (1)$$

Students shall analyze multiple cache configurations, and choose the memory architecture that best satisfies the requirements. The procedure for conducting this analysis is described in Section 2. Some of the configuration options that will be explored are [2]:

- number of cache levels (only L1 or L1+L2);
- cache size;
- associativity;
- write policy (write-through vs write-back).

Next we describe the simulation environment that will be used in order to gauge the performance of the memory architecture so that the mean access time can be obtained.

1.3 Simulation Environment

To evaluate the performance of the memory architecture we will use DineroIV [1], a trace-driven cache simulator. DineroIV is capable of simulating a complete memory hierarchy consisting of various caches interconnected between the processor and the primary memory.

To perform a simulation, DineroIV takes a parameter set and a trace file as inputs. The parameter set specifies the characteristics of the caches to be simulated, for example the cache size, the block size, and the number of associativity ways. The trace file contains all memory references accessed by the program under evaluation. These traces specify whether a memory reference is an instruction fetch, a data read (LOAD), or a data write (STORE). For each reference, DineroIV simulates the behavior of the specified cache configuration and generates hits and misses as appropriate. At the end of the simulation, it produces a set of statistics which summarizes the performance of the simulation, including number of references, misses and memory traffic generated by the processor.

To show a concrete usage example, consider the following execution of DineroIV, which is launched from the command line:

```
dineroIV  -l1-usize 8k    -l1-ubsize 8
          -l2-usize 128k  -l2-ubsize 16   <   trace.xdin
```

In this example, DineroIV simulates a caching architecture comprising a cache L1 and a cache L2. The cache L1 has 8 kBytes and a block size of 8 bytes. The cache L2 has 128 kBytes and a block size of 16 bytes. The trace file that contains the read/write memory access patterns is provided in file *trace.xdin*.

DineroIV is extensively described in its user manual [3]. Nevertheless, we summarize the most relevant parameters for the purpose of this assignment. The basic command to run DineroIV is:

```
dineroIV [options] < tracefile
```

where the most relevant parameters are:

`-lN-Tsize P` - Sets the cache size of the specified level N cache to P bytes

`-lN-Tbsize P` - Sets the block size of the specified level N cache to P bytes

`-lN-Tassoc U` - Sets the associativity of the specified level N cache to U

`-lN-Tccc` - Computes Compulsory/Capacity/Conflict miss rates for the specified level N cache where:

T - is the cache type (u=unified, i=instruction, d=data);
N - is the cache level (1 = N), where level 1 is closest to the processor;
U - is an unsigned decimal integer;
P - is like U, but must be a power of 2, with an optional scaling suffix character (one of kKmMgG, for multiplication by 0x400, 0x100000, or 0x40000000).

Since DineroIV requires a trace file containing the sequence of memory accesses to be performed, for this assignment it is necessary to generate such a trace file for the motion estimation algorithm described in Section 1.1. To this end, the supplied implementation (program `motion_estimation.c`) includes two special functions (`frame_memory_read()` and `frame_memory_write()`) which register data memory accesses on the trace file (`trace.log`). This information will then enable the simulation of several cache configurations in order to obtain the one that offers the best performance levels within an acceptable cost.

2 Procedure

2.1 Understanding the Program

Without delving into the details of the signal processing application, analyze the flow of the C program. Observe the data access patterns and identify the critical sequence of accesses which may have a larger impact on the performance of the system.

2.2 Generation of the trace file

a) Open a Linux terminal and download the source file of the motion estimation algorithm (`motion_estimation.c`), as well as the input pixel data file (`table_tennis_qcif_3frames.yuv`), from the course webpage. Copy them into your working directory and compile the source file, by issuing the command:

```
gcc motion_estimation.c -o motion_estimation
```

b) Run the compiled executable file:

```
./motion_estimation
```

The executed motion estimation program will generate two output files:

`results.log` - with information about the result of the motion estimation procedure;
`trace.log` - with the trace of all frame memory addresses that were accessed during the program execution.

NOTE: The `trace.log` file occupies a significant amount of disk space (about 153 MBytes). In the event of any problem concerned with disk quota restrictions, run the above executable in the `/tmp` directory of your Linux terminal.

2.3 Cache L1

2.3.1 Theory of cache

1. Explain the different types of cache misses: *compulsory*, *capacity*, and *conflict*.

2. Explain the different types of cache writing-policies.

2.3.2 Cache L1: dimension and block size

a) Consider a memory hierarchy composed of a single cache memory (L1), which interconnects the SDRAM frame memory and the CPU. Considering the characteristics of the available memory devices (see Table 1), and the maximum total cost of the memory hierarchy, determine the maximum storage space of cache L1.

NOTES:

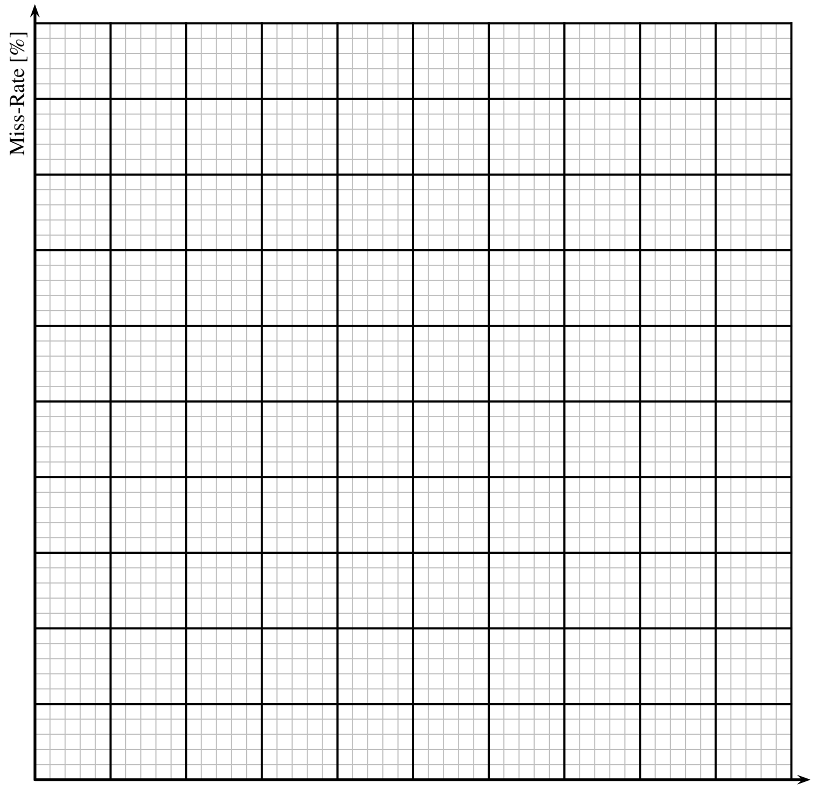
- the size of any of the memory modules (frame buffer, any cache) must be an integer power of 2: $\text{L1\_size} = 2^{\text{MAX}}$;
- do not forget to consider the cost of the 128 kByte frame memory.

b) Consider three different dimensions for the L1 data cache: $\text{L1\_size} \in \{2^{\text{MAX}}, 2^{\text{MAX}-1}, 2^{\text{MAX}-2}\}$. For each of these dimensions, and assuming a direct mapping configuration, use the dineroIV simulator to evaluate the resulting average data *miss-rate* considering the following block sizes: $\text{Block\_size} \in \{8, 16, 32, 64\}$.

Fill the following table with the obtained data:

	$\text{L1\_size} =$	$\text{L1\_size} =$	$\text{L1\_size} =$
Block size = 8 Bytes			
Block size = 16 Bytes			
Block size = 32 Bytes			
Block size = 64 Bytes			

c) For each L1 cache size, plot the variation of the *miss-rate* with the size of the block.



d) By considering the obtained results, select two L1 cache configurations (dimension and block size) that offer the best trade-off between the cost of the device and the resulting average *miss-rate*. Label in the previous plot the two configurations chosen.

L1_config_1	
Cache size:	
Block size:	
Miss-Rate:	
Cost = Price × Miss-Rate:	

L1_config_2	
Cache size:	
Block size:	
Miss-Rate:	
Price × Miss-Rate:	

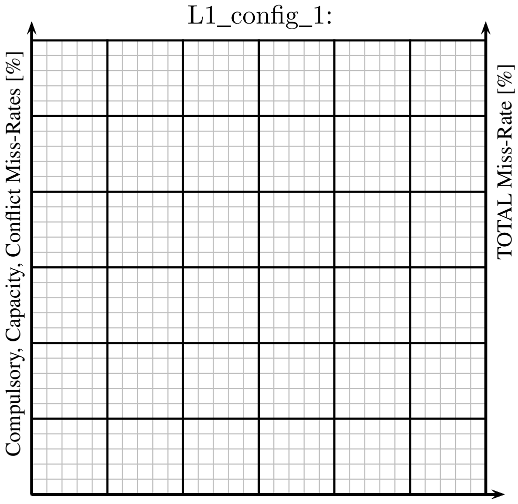
2.3.3 Cache L1: set associativity

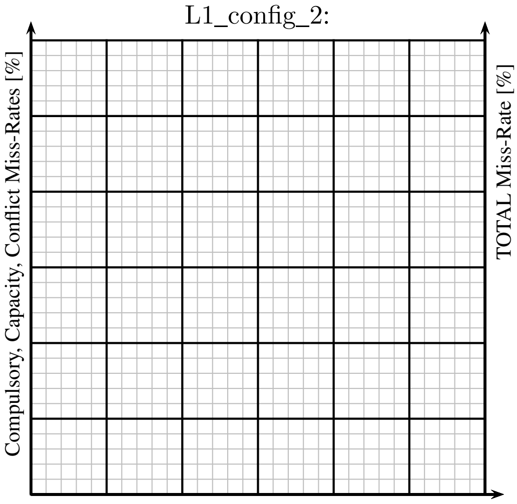
a) For each of the two L1 cache setups previously selected, evaluate the *compulsory*, *capacity*, *conflict* and *total miss-rates* when the following configurations are considered: set associativity of 1 (direct-mapped), 2, 4, 8.

Fill the following table with the obtained data:

	L1_config_1				L1_config_2			
Miss-Rate	1-way	2-ways	4-ways	8-ways	1-way	2-ways	4-ways	8-ways
Compulsory								
Capacity								
Conflict								
TOTAL								

b) For each L1 cache setup, draw a plot with the variation of the obtained *compulsory*, *capacity*, *conflict* and *total miss-rates* for the considered set associativity ways.


L1_config_1:


L1_config_2:

c) Comment the results above.

d) Write the expression that provides the mean *access time* as a function of the L1 cache hit $\left(p_{H}^{L1}\right)$ and miss $\left(p_{M}^{L1}\right)$ rates, the L1 cache hit $\left(t_{H}^{L1}\right)$ and miss $\left(t_{M}^{L1}\right)$ access times, and the time penalty associated to each associativity level, as expressed in Table 1. Consider a *non-blocking critical-word-first* load policy, where the bus occupancy rate has a lower impact in the performance of the cache.

e) Evaluate the mean *access time* of each configuration, considering the obtained *miss-rates* and the time penalty associated to each associativity level. Evaluate the resulting cost function, as defined in Eq. 1 (including the frame memory).

Fill the following table with the obtained data:

	L1_config_1				L1_config_2			
	1-way	2-ways	4-ways	8-ways	1-way	2-ways	4-ways	8-ways
Miss-Rate								
Access time								
Price								
Cost Function								

f) Draw conclusions:

2.3.4 Cache L1: write policy

a) By analyzing the sequence of memory accesses generated by the motion estimation algorithm (see Fig. 3), select the best setup for the cache writing-policy: *write-back* versus *write-through*, *write-allocate* versus *write-not-allocate*. Justify. (Note that the number of writes is much smaller than the number of reads.)

2.3.5 Cache L1: final selection

a) By considering the obtained results, select the L1 cache setup that offers the best compromise between the cost of the device and the resulting average access time.

L1_Config	
Cache dimension:	
Block size:	
Associativity:	
Write-policy:	
Miss-Rate:	
Access time:	
Price:	
Cost Function:	

2.4 Cache L2

Consider now that the obtained SRAM *on-chip* L1 cache is connected to a SRAM *off-chip* L2 cache in order to obtain a memory hierarchy composed of two caches (L1+L2), which interconnect the SDRAM frame memory and the CPU. After specifying the L1 cache we continue the design evaluating and specifying a L2 cache. (The real design process would be more complex since it would require the evaluation of several combinations of L1 and L2 caches.)

2.4.1 Cache L2: dimension

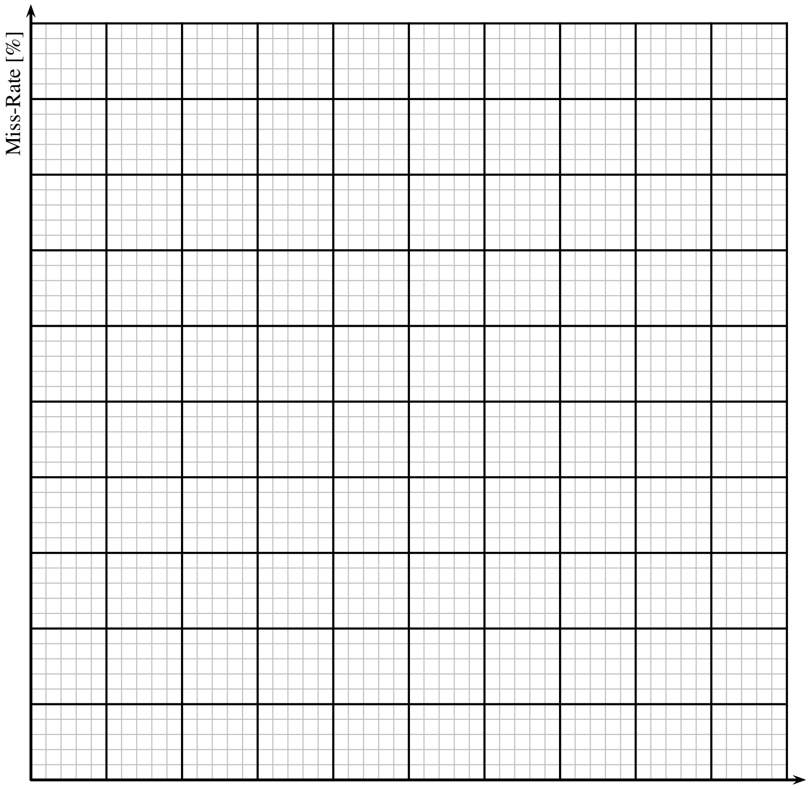
a) Considering the maximum cost of the whole memory hierarchy, as well as the price of L1 cache and the 128 kByte frame memory, determine the maximum storage space of L2 cache (an integer power of 2), considering the characteristics of the available memory devices (see Table 1).

b) For the obtained maximum storage space for L2 cache, adopting a **direct mapping configuration**, use dineroIV simulator to evaluate the resulting average data *miss-rate* considering the following block sizes: $(1 \times \mathrm{L1\_block})$, $(2 \times \mathrm{L1\_block})$, $(4 \times \mathrm{L1\_block})$ and $(8 \times \mathrm{L1\_block})$.

Fill the following table with the obtained data:

	Block Size	Miss-Rate
Block size = $(1 \times \mathrm{L1\_block})$		
Block size = $(2 \times \mathrm{L1\_block})$		
Block size = $(4 \times \mathrm{L1\_block})$		
Block size = $(8 \times \mathrm{L1\_block})$		

c) Plot the variation of the *miss-rate* with the size of the block.



d) From the obtained results, select the block size that offers the best trade-off between the resulting average *miss-rate* and the *time penalty* associated with each data fetch from the primary memory. Justify.

L2 Block Size =

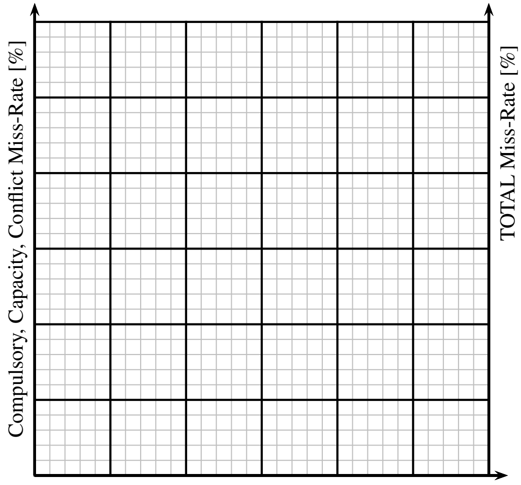
2.4.2 Cache L2:

a) Evaluate the *compulsory*, *capacity*, *conflict* and *total miss-rates* for the direct-mapped L2 data cache.

Fill the following table with the obtained data:

	Miss-Rate
Compulsory	
Capacity	
Conflict	
TOTAL	

b) Plot the variation of the obtained *compulsory*, *capacity*, *conflict* and *total miss-rate*.



c) Write the expression which provides the mean *access time* as a function of the L1 and L2 cache hit (p_H^{L1}, p_H^{L2}) and miss (p_M^{L1}, p_M^{L2}) rates, L1 and L2 cache hit (t_H^{L1}, t_H^{L2}) and miss (t_M^{L1}, t_M^{L2}) access times, and the time penalty, as expressed in table 1.

d) Evaluate the mean *access time* provided by the chosen configuration, considering the obtained *miss-rate* and the time penalty. Evaluate the resulting cost function, as defined in Eq. 1.

Fill the following table with the obtained data:

Miss-Rate	
Access time	
Price	
Cost Function	

2.5 Memory Hierarchy Configuration

a) By considering the obtained results, fill the following table with the selected characteristics for L1 and L2 cache memories, as well as the corresponding performance results of the overall memory hierarchy.

	Cache L1	Cache L2	Frame-Memory
Dimension [Bytes]:			128×1024
Block size [Bytes]:			–
Associativity:			–
Write-policy:			–
Local Miss-Rate [%]:			–
Price [€]:			
Global Miss-Rate [%]:			
Global Access time [ns]:			
Total Price [€]:			
COST FUNCTION [ns×€]:			

References

[1] dineroIV. Webpage. "`http://www.cs.wisc.edu/~markhill/DineroIV`", October 2017.

[2] David Patterson and John Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 4^{th} edition, 2011.

[3] Jan Edler and Mark D. Hill. *dineroIV User Manual*, 1997.

A Block-based Motion Estimation Algorithm

In this appendix, we provide more details on the motion estimation program used in this assignment. First, we clarify the purpose and context in which motion estimation is used, namely to improve the efficiency of video compression in popular video encoding standards. Then, we provide an overview of the key insight behind motion estimation which consists in reducing temporal redundancy of video frames. Lastly, we present the motion estimation algorithm implemented in `motion_estimation.c`.

A.1 MPEG-x and H.26x Video Encoding

Recently, there has been a general proliferation of advanced video services and multimedia applications, where video compression standards, such as MPEG-x or H.26x, have been developed to store and broadcast video information in digital form. The main goal of such video compression algorithms is to exploit both the spatial and the temporal redundancy within an image sequence in order to achieve the maximum compression as possible.

In video coding it is well established that video data tends to have a high degree of spatial and temporal redundancy. While spatial redundancies can be extensively exploited through the usage of the transform based algorithms (such as the Discrete Cosine Transform - DCT), temporal redundancies are usually exploited through the usage of prediction schemes. In fact, even in the presence of motion, we can easily expect a high degree of temporal redundancy between the pixels of a given region A in one image and the corresponding pixels, moved to region X, in the following image.

A.1.1 Reducing the Temporal Redundancy

Temporal redundancy is usually exploited by only encoding the difference between consecutive frames. Hence, if we denote the two contiguous frames in Fig. 1 by $\mathrm{I}(t_0 - 1)$ and $\mathrm{I}(t_0)$, a first approach to implement such prediction method would be to compress the *differences frame* $\Delta\mathrm{I} = \mathrm{I}(t_0) - \mathrm{I}(t_0 - 1)$.

However, as it can be seen in Fig. 1, such differences frame may still contain a large amount of information. The reason for this fact is mainly due to the presence of motion, which introduces a certain displacement to every region of frame $\mathrm{I}(t_0 - 1)$ into a new region at frame $\mathrm{I}(t_0)$.

One way to reduce the magnitude of this residual information in the differences frame $\Delta\mathrm{I}$ is to track and compensate such movement using the previously encoded images. The process of computing the changes among frames is usually referred to by *temporal prediction* with **Motion Compensation** (MC). Motion compensation is thus defined as the process of compensating for the displacement of moving objects from one frame to another. In practice, motion compensation is preceded by **Motion Estimation** (ME), the process of tracking and finding the corresponding pixels among consecutive frames.

Intuitively, one might expect that the ideal procedure for reducing the temporal redundancy level is the one that tracks every pixel from one frame to the following frame. However, this would be very computationally intensive, and such methods would not provide reliable tracking due to the presence of noise in the frames. Instead of tracking individual pixels from frame to frame, video coding standards only allow tracking of information for $M \times M$ pixel regions, commonly referred to as **Macroblocks** (MB). The macroblock dimension of 16×16 pixels is generally chosen, because it offers a good compromise between efficiently providing temporal redundancy reduction and requiring moderate computational requirements.

Hence, the result of such a prediction scheme will be a synthesized frame $\hat{\mathrm{I}}(t_0) = \mathrm{MC}(\,\mathrm{I}(t_0 - 1)\,)$, obtained with the macroblocks of the previous frame $\mathrm{I}(t_0 - 1)$, displaced by the corresponding **Motion Vectors** (MV). The differences frame, obtained by subtracting the current frame $\mathrm{I}(t_0)$ with the motion compensated frame $\hat{\mathrm{I}}(t_0)$, will be defined as:

$$\Delta\hat{\mathrm{I}}(t_0) = \mathrm{I}(t_0) - \hat{\mathrm{I}}(t_0) \tag{2}$$

$$\Leftrightarrow \quad \Delta\hat{\mathrm{I}}(x, y, t_0) = \mathrm{I}(x, y, t_0) - \mathrm{I}(x - u, y - v, t_0 - 1) \tag{3}$$

where $\mathrm{I}(x, y, t_0)$ are the pixel values at spatial location (x, y) in the current frame $\mathrm{I}(t_0)$, and $\mathrm{I}(x - u, y - v, t_0 - 1)$ are the *corresponding* pixel values at spatial location $(x - u, y - v)$ in the previous or reference

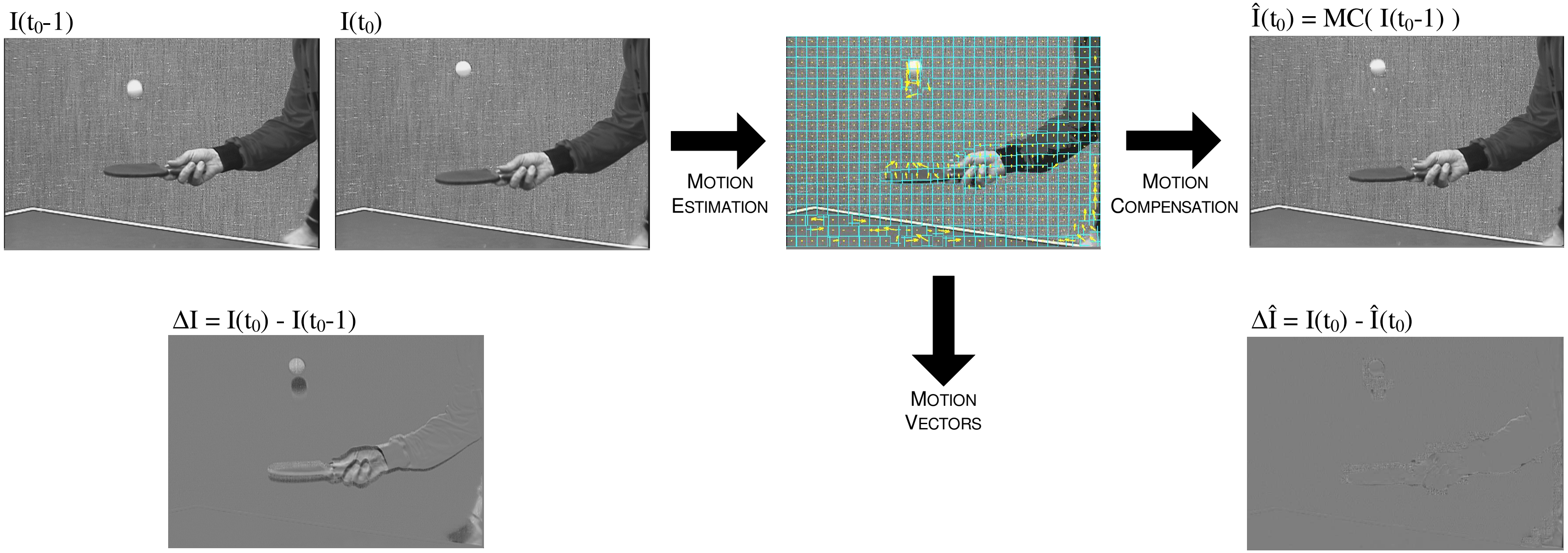

$I(t_0\text{-}1)$
$I(t_0)$
MOTION ESTIMATION
MOTION COMPENSATION
$\hat{I}(t_0) = MC(\ I(t_0\text{-}1)\ )$
$\Delta I = I(t_0) - I(t_0\text{-}1)$
MOTION VECTORS
$\Delta\hat{I} = I(t_0) - \hat{I}(t_0)$

Figure 1: Reduncing the temporal redundancy using the motion compensation algorithm.

frame $\mathrm{I}(t_0 - 1)$. The coordinates (u, v) are the output of the motion estimator and define the relative motion of a block from one frame to another. Hence, they are referred to as the motion vector for the macroblock at (x, y). $\mathrm{I}(x - u, y - v, t_0 - 1)$ is refereed to as the motion-compensated prediction of $\mathrm{I}(x, y, t_0)$ and $\Delta\hat{\mathrm{I}}(x, y, t_0)$ is the prediction residual for $\mathrm{I}(x, y, t_0)$.

As it can be seen in Fig. 1, such a residual frame will contain a significantly smaller amount of information. It is this differences or residual frame, as well as the computed motion vectors, that are encoded and transmitted to the video decoding system.

A.1.2 Motion Estimation

Motion estimation is usually the most computational intensive operation of any MPEG-x video encoding system. Fig. 2 illustrates this operation, as it is posed in most video coding standards. Given an $N_h \times N_v$ sized reference frame (previous image) and an $M \times M$ macroblock in the current frame, the objective of motion estimation is to determine the $M \times M$ block in the reference frame that better matches (according to a given criterion) the characteristics of the macroblock in the current frame. As it was referred before, the current picture is defined as the frame (or image) at time instant t_0. The reference picture is defined as the frame (or image) at time instant $t_0 - 1$.

The location of each macroblock is defined by the (x, y) coordinates of its top-left corner pixel. Ideally, the whole reference picture would be searched for the best match; however, this is usually computationally impractical. Instead, the search is restricted to a $[-p, +p]$ search region around the original location of the macroblock under processing in the current picture.

Matching Criterion

Let the pixels of the macroblock in the current frame be denoted as $C(x, y)$ and the pixels in the reference frame be denoted as $R(x, y)$. The Sum of Absolute Differences (SAD) cost function is defined as:

$$\mathrm{SAD}\,(i, j) = \sum_{k=0}^{M-1} \sum_{l=0}^{M-1} |C\,(x + k, y + l) - R\,(x + i + k, y + j + l)| \quad ; \quad -p \leq i, j \leq p \tag{4}$$

The best matching macroblock is defined as the macroblock $R\,(x + i, y + j)$ for which the $\mathrm{SAD}(i, j)$ measure is minimized. The coordinates (i, j) will then define the corresponding motion vector.

Since the matching procedure is carried out by considering rectangular regions, in video coding terminology such processing scheme is referred to as Block-Matching Algorithms (BMA).

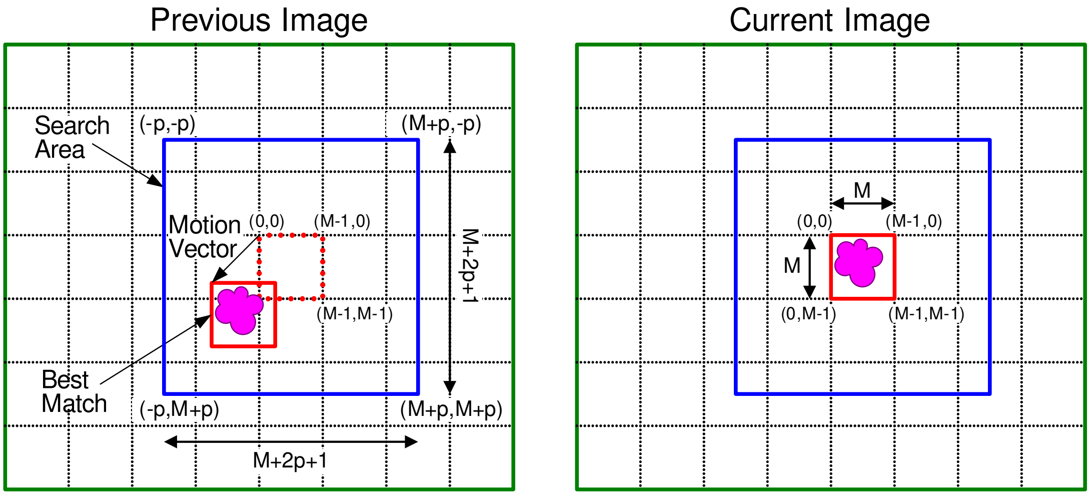


Figure 2: Motion estimation process.

Full-Search Block-Matching Motion Estimation Algorithm

```
/* ME algorithm for all macroblocks of the $N_h \times N_v$ image */
for $lin = 0$ to $\lfloor N_v/M \rfloor$ do
  for $col = 0$ to $\lfloor N_h/M \rfloor$ do
    /* Computation of the MV of the MB defined by its upper-left corner $(lin \times M, col \times M)$ */
    $Best\_SAD \Leftarrow \infty$                /* Minimum SAD initialization */
    /* Searches all position of the search window defined within the previous image */
    for $i = -p$ to $p$ do
      for $j = -p$ to $p$ do
        /* Processing of all the $M \times M$ pixels of the candidate MB under processing */
        $SAD \Leftarrow 0$                  /* SAD measure initialization */
        for $k = 0$ to $(M-1)$ do
          for $l = 0$ to $(M-1)$ do
            $SAD += \mid C\,(lin \times M + k,\ col \times M + l) - R\,(lin \times M + i + k,\ col \times M + j + l) \mid$
          end for
        end for
        /* Comparison between the obtained SAD and the minimum SAD computed so far */
        if $SAD\,(i,j) < Best\_SAD$ then
          $Best\_SAD = SAD$
          $MV(lin, col) = (i,j)$
        end if
      end for
    end for
  end for
end for
```

Figure 3: Full-search block-matching motion estimation algorithm.

Given Eq. 4, the most straightforward method to find the optimal motion vector for each macroblock is to compute $\mathrm{SAD(i,j)}$ at each location of the search space. This approach is referred to as the Full-Search Block-Matching (FSBM) algorithm.

For each motion vector there are $(2p+1)^2$ search locations. At each search location (i,j) the whole set of $M \times M$ pixels is compared. Each pixel comparison requires three operations, namely, a subtraction, an absolute-value calculation and one addition. Thus, the total number of operations per macroblock is $(2p+1)^2 \times M^2 \times 3$ arithmetic operations. For a picture resolution of $N_h \times N_v$ and a frame rate of F pictures per second, the overall complexity is $\frac{N_h N_v F}{M^2} \times (2p+1)^2 \times M^2 \times 3$ operations per second. For typical values usually adopted in broadcast digital TV (4CIF format: $N_h = 704$ pixels, $N_v = 576$ pixels, $F = 30$), $M = 16$ and $p = 16$, the full-search block-matching algorithm requires about 39.74 GOPS (Giga operations per second).

The pseudo-code implementation of the full-search block-matching algorithm to compute all the motion vectors $\mathrm{MV}(x,y)$ of an entire $N_h \times N_v$ pixels image is illustrated in Fig. 3. The C source code file of a real implementation of this algorithm can be obtained at the course webpage (file `motion_estimation.c`).