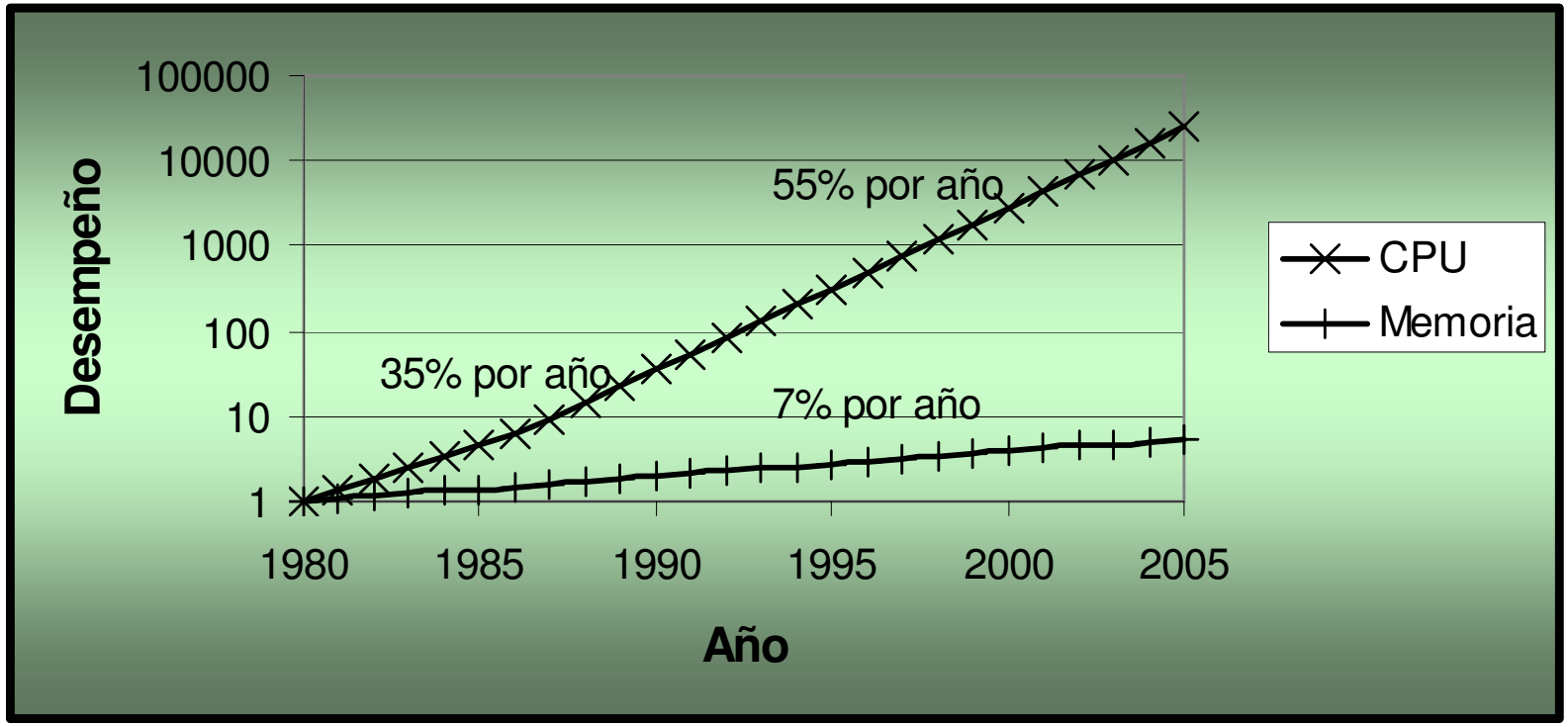
JERARQUÍAS DE MEMORIA

Organización de Computadoras

Facultad de Ingeniería
Universidad de Buenos Aires
2011

Introducción

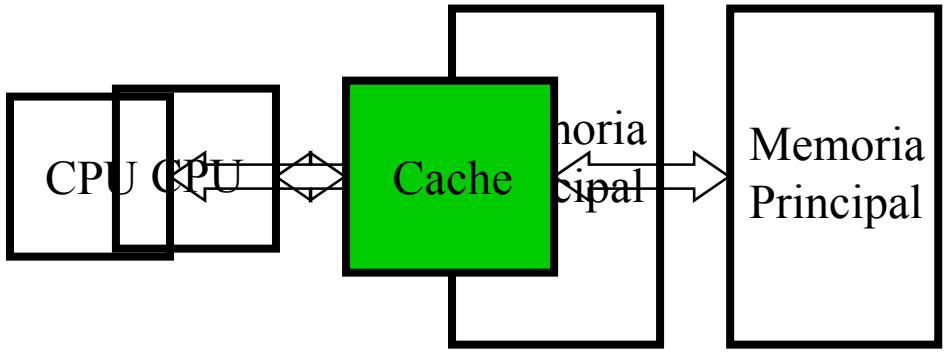
La Brecha CPU-Memoria



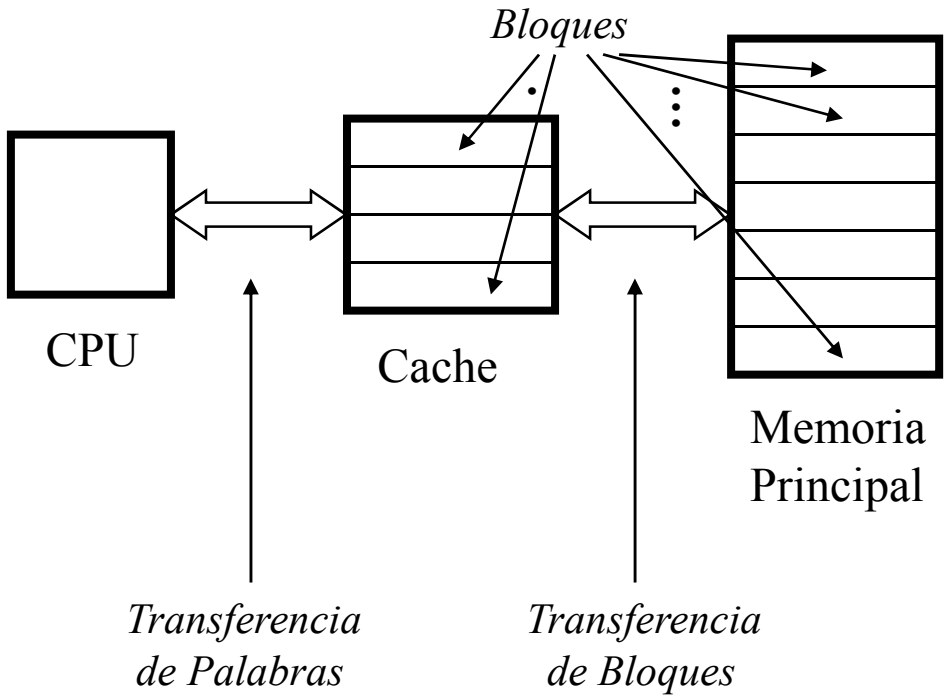
El Principio de Localidad para las Referencias a Memoria

- Los programas referencian la memoria localmente
  - Localidad espacial
  - Localidad temporal

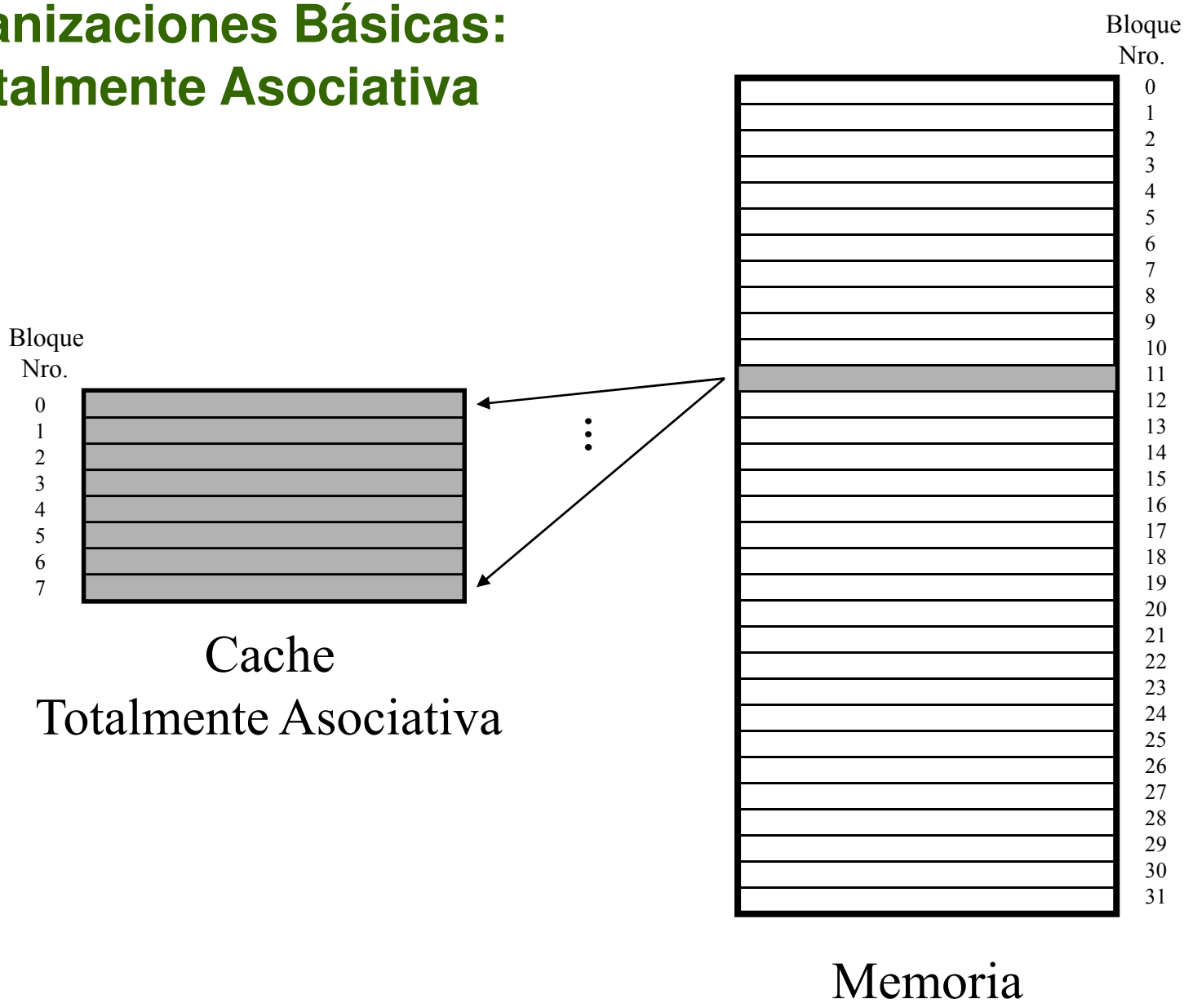
La Memoria Local o Memoria Cache



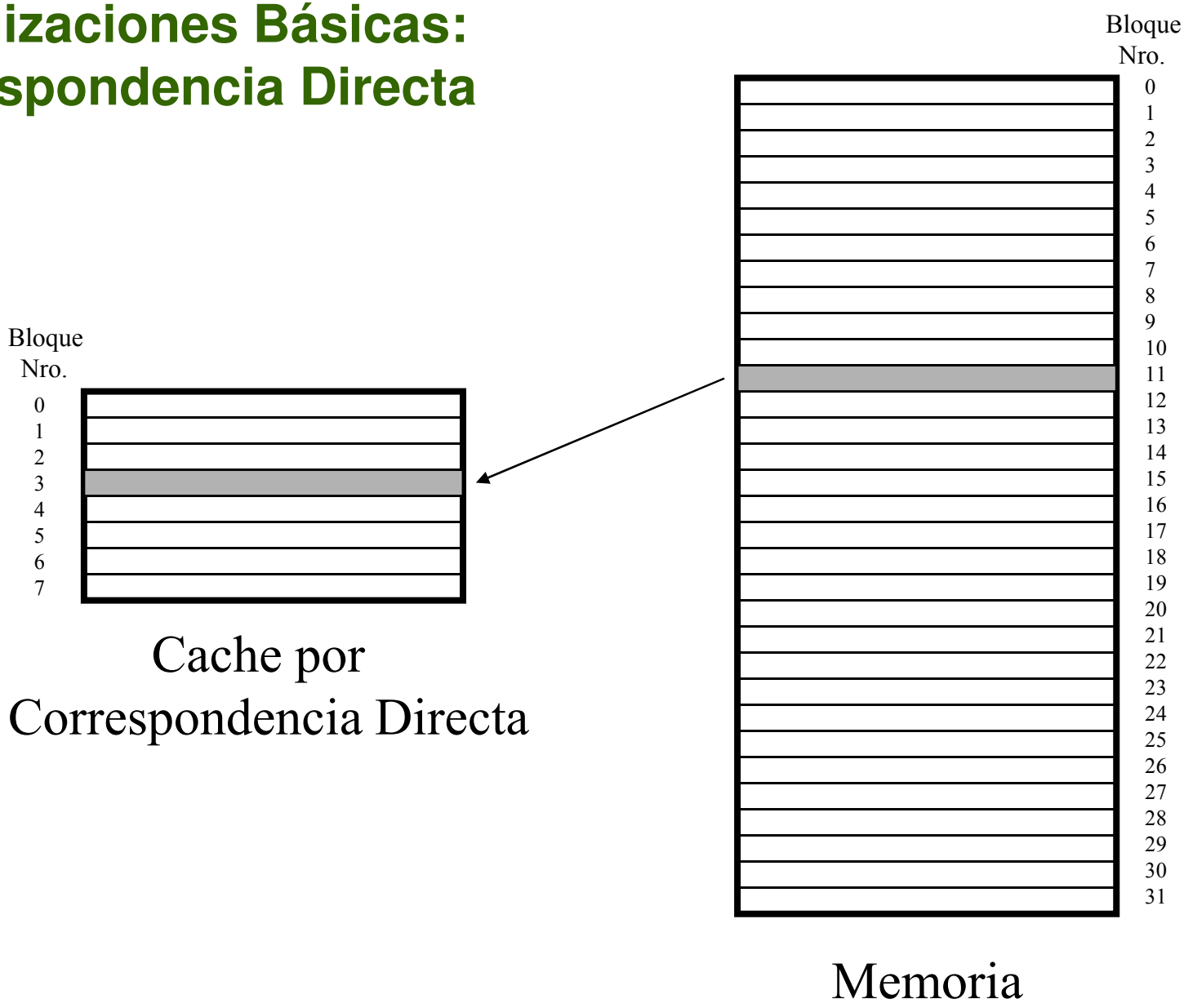
Funcionamiento Básico



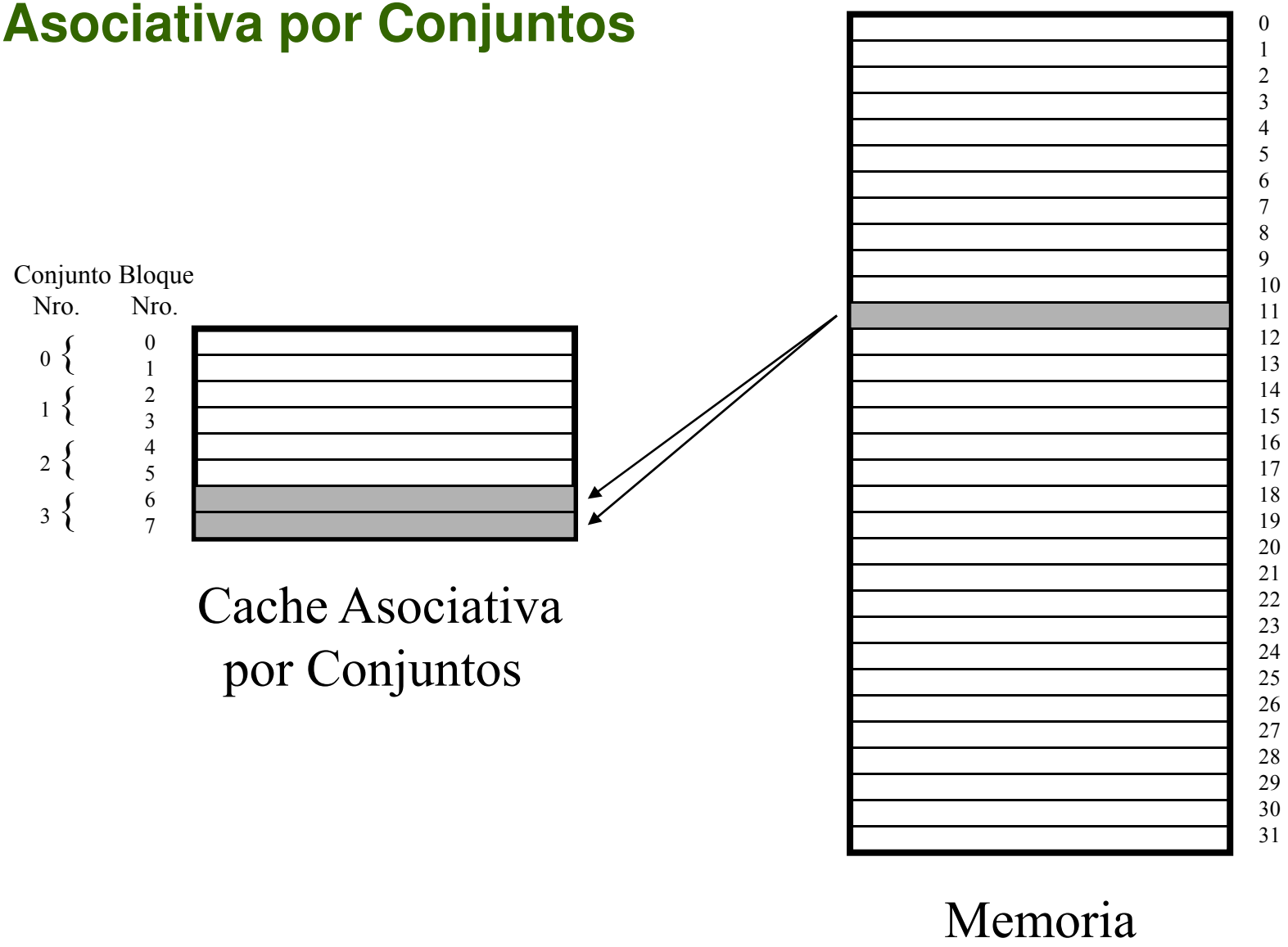
Organizaciones Básicas: Totalmente Asociativa



Organizaciones Básicas: Correspondencia Directa



Organizaciones Básicas: Asociativa por Conjuntos



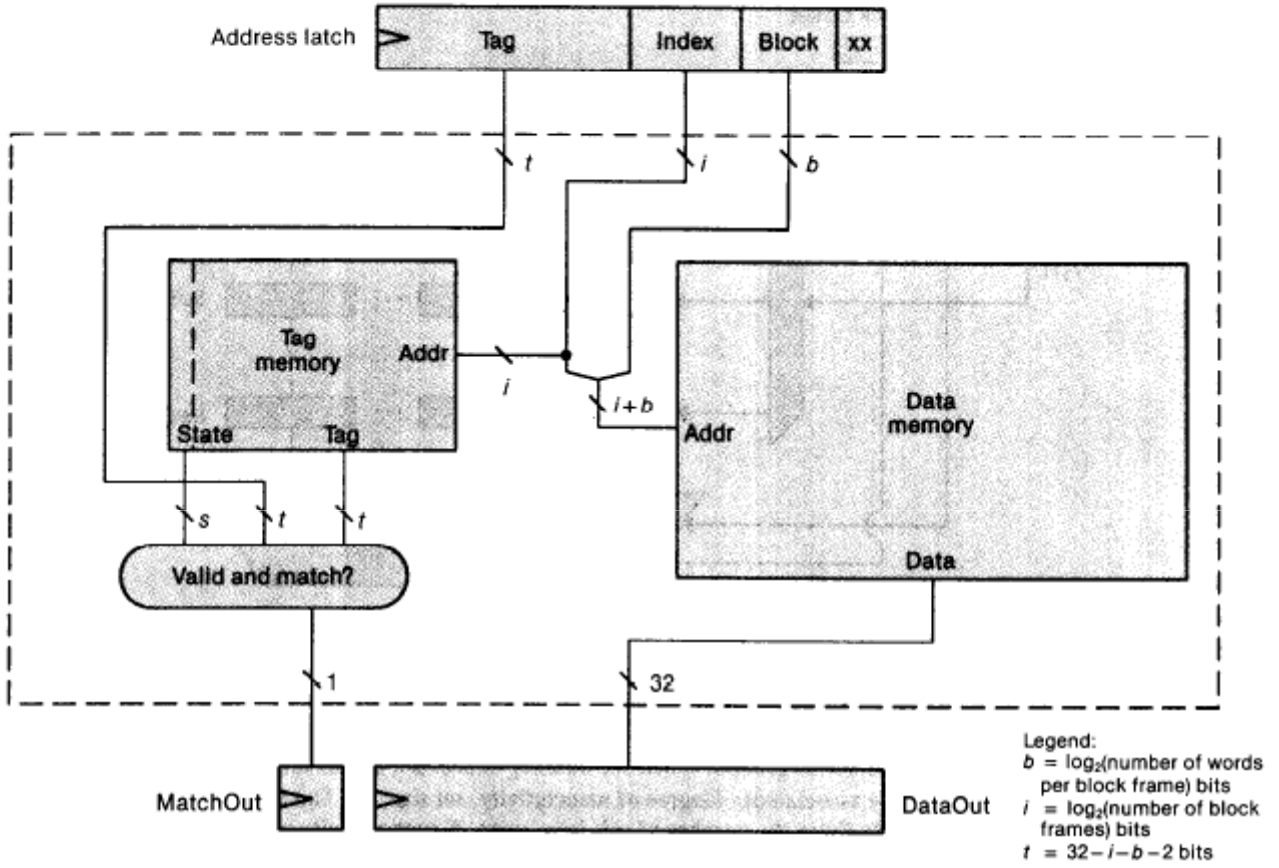


Figure 2. A direct-mapped cache. The access logic (hit, not miss logic) for a direct-mapped cache using bit selection to select the set (block frame) of the reference has three components. The first component, the data memory, holds all cached data and instructions. The second component, tag memory, holds the state bits and address tag associated with a cached block. The last component, the match logic, produces a single bit indicating whether the referenced block is present.

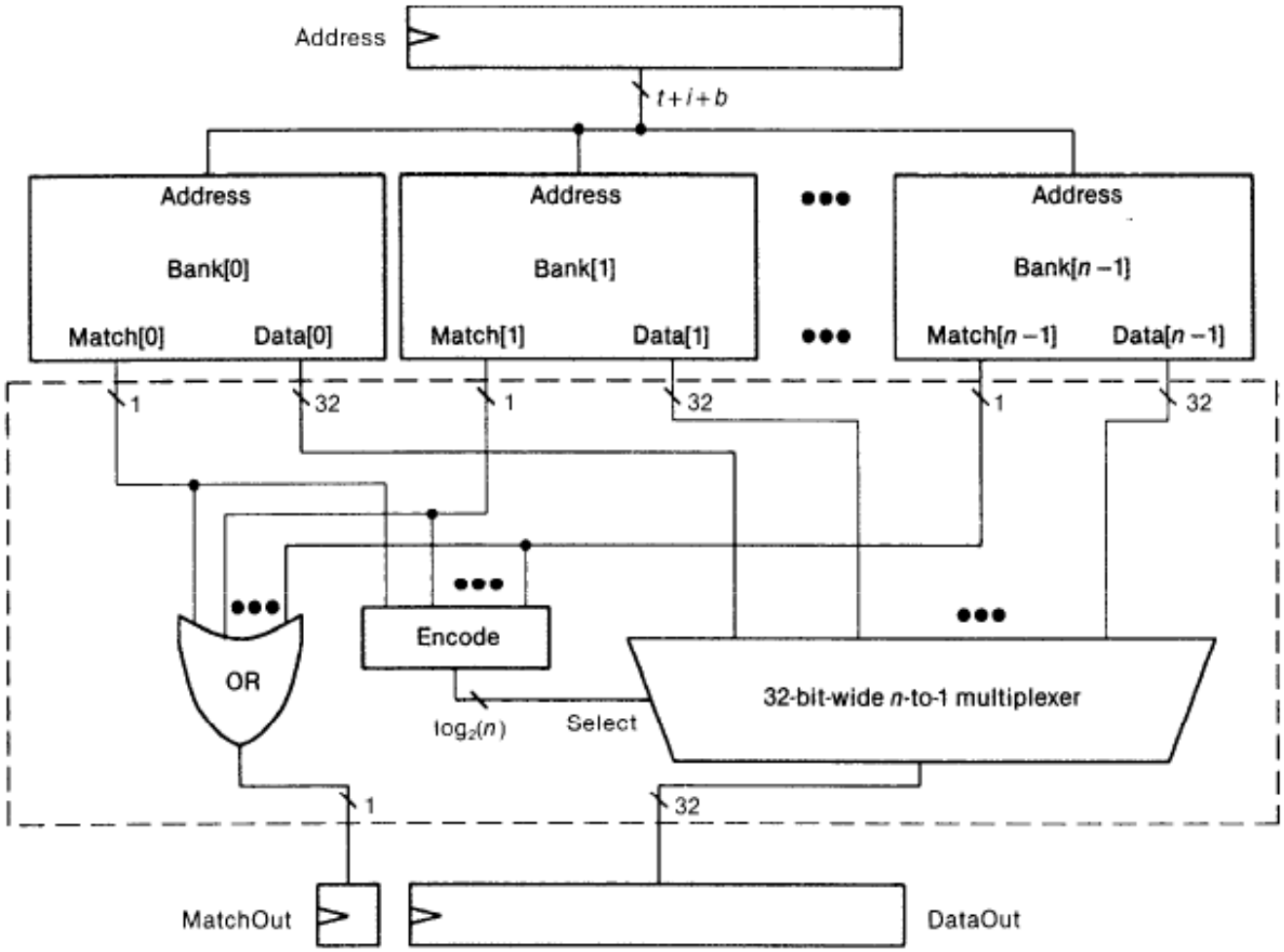


Figure 3. A set-associative cache. Cache access (hit) logic for an n-way set-associative cache of c blocks consists of n banks and the logic to combine bank results. Each bank can be thought of as a direct-mapped cache of c/n blocks and can be implemented using the logic in the dashed box of Figure 2.

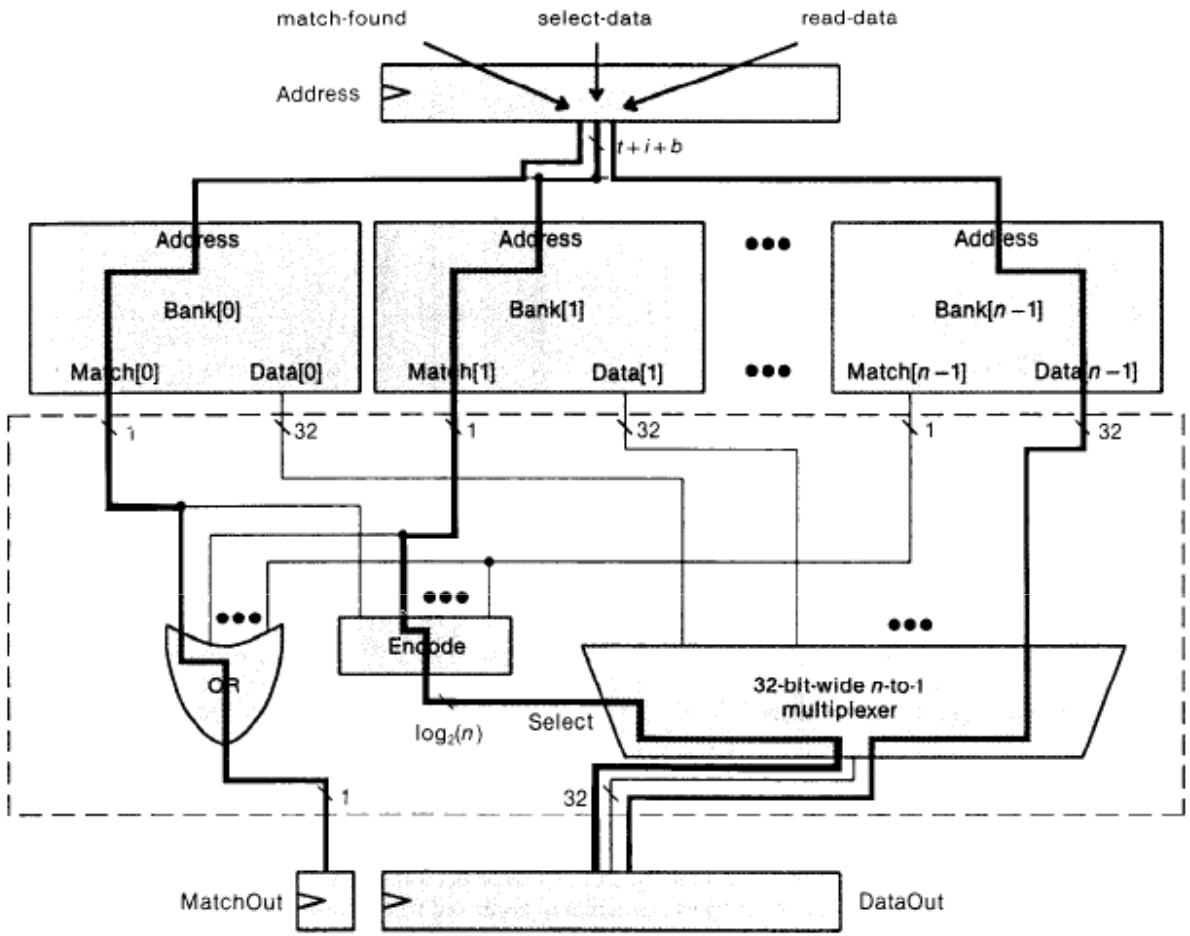


Figure 5. Timing paths in a set-associative cache. The three timing paths in the cache hit logic for an *n*-way set-associative cache are (1) match-found, which signals a cache hit or miss (Address to Match[*i*] to MatchOut); (2) select-data, which selects the data word that corresponds to the tag that matched (Address to Match[*i*] to Select to DataOut); and (3) read-data, which provides the data on a cache hit (Address to Data[*i*] to DataOut). Path select-data is not needed in a direct-mapped cache.

Políticas de reemplazo

- LRU
- FIFO
- RANDOM
- Basadas en Distancias de Pila LRU
- Óptima (Belady 1963)

Que pasa en una escritura?

- Write Through
- Write Back

Alocación: -Write Allocate

-Write No-Allocate

Cache Performance

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty} \\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \\ &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

Ecuación de desempeño de CPU con Memoria cache

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

Optimizaciones para mejorar el desempeño en sistemas con memoria cache

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

- Reducir la Penalidad de miss
- Reducir el Miss Rate
- Reducir el tiempo de Hit

Reduccion de Penalidad de miss Caches multinivel

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

and

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

so

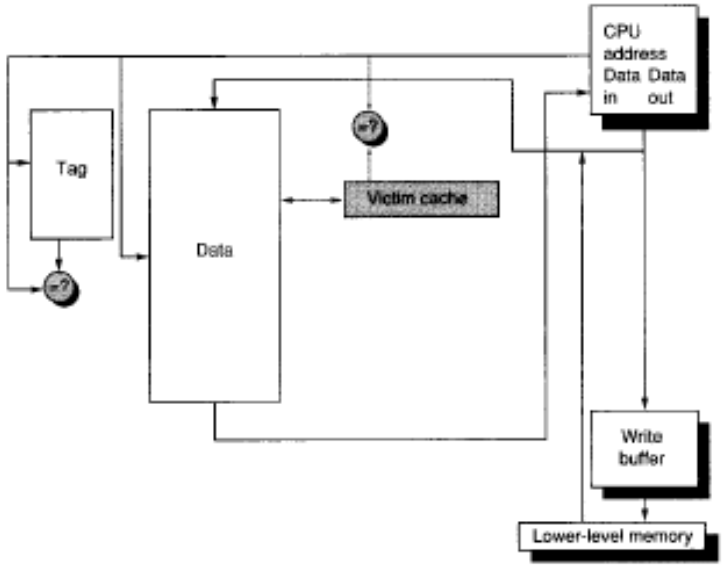
$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

Reduccion de Penalidad de miss Critical Word First, Early Restart

- *Critical word first*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Critical-word-first fetch is also called *wrapped* fetch and *requested word first*.
- *Early restart*—Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

Reduccion de Penalidad de miss

- Prioridad a Lecturas sobre las escrituras
- Merging Write buffers
- Caches de Víctimas



Reducción de la Tasa Miss

Causas de los misses en memoria cache?

Herramientas de Modelado y Análisis

Una Clasificación para los Desaciertos en Memoria Cache

- Modelo de las “Tres C”
  - A favor:
    - Tipos conceptuales
      - Obligatorios
      - Capacidad
      - Conflicto
    - Muy difundidos
  - En contra
    - Da resultados incorrectos
    - No clasifica individualmente

Otro Modelo: de las Tres C Determinístico "D3C"

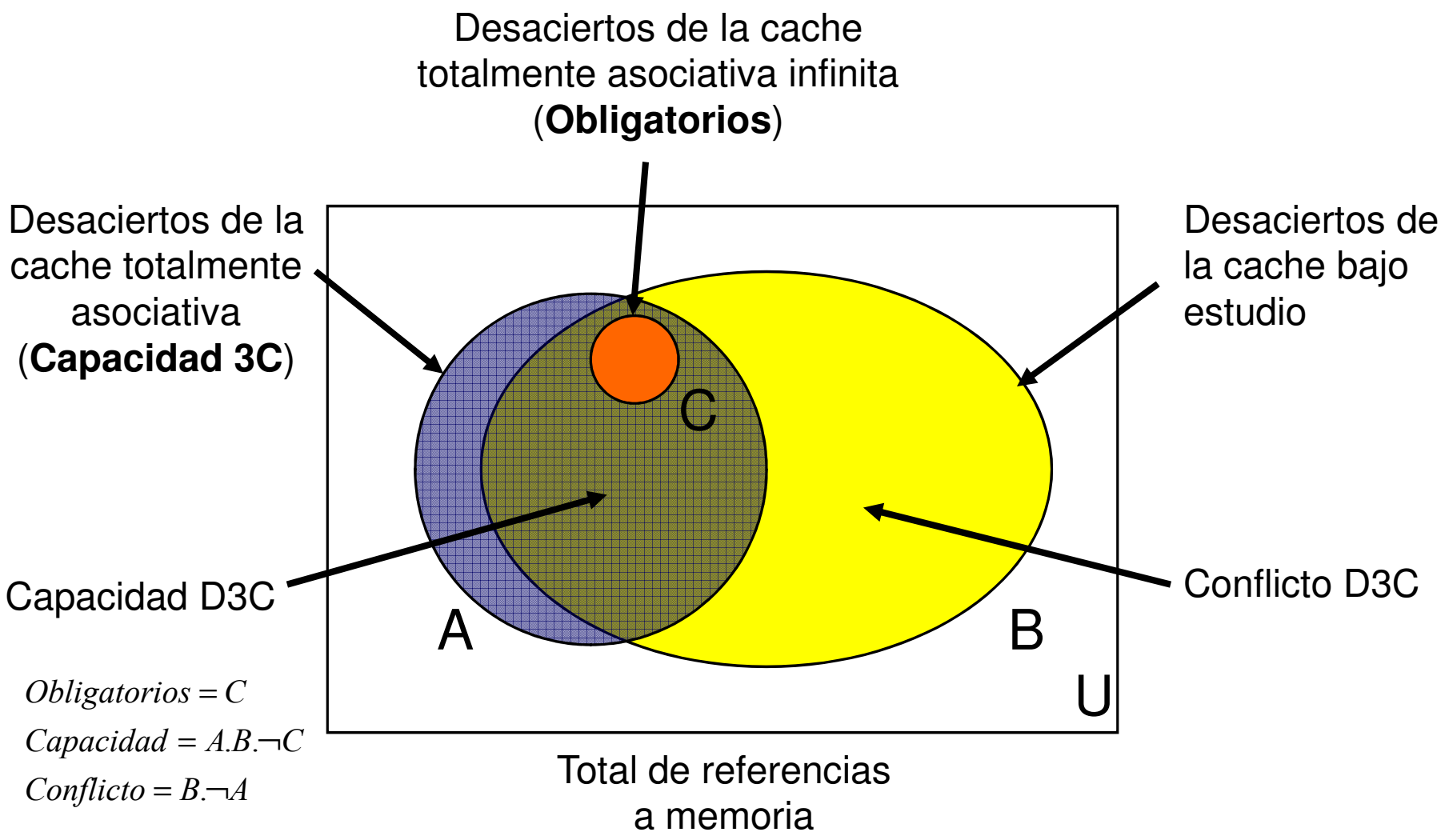
- Obligatorios
- Capacidad
- Conflicto

Definición operacional:

$$Obligatorios : D \rightarrow no\ computable$$

$$Capacidad : D > B$$

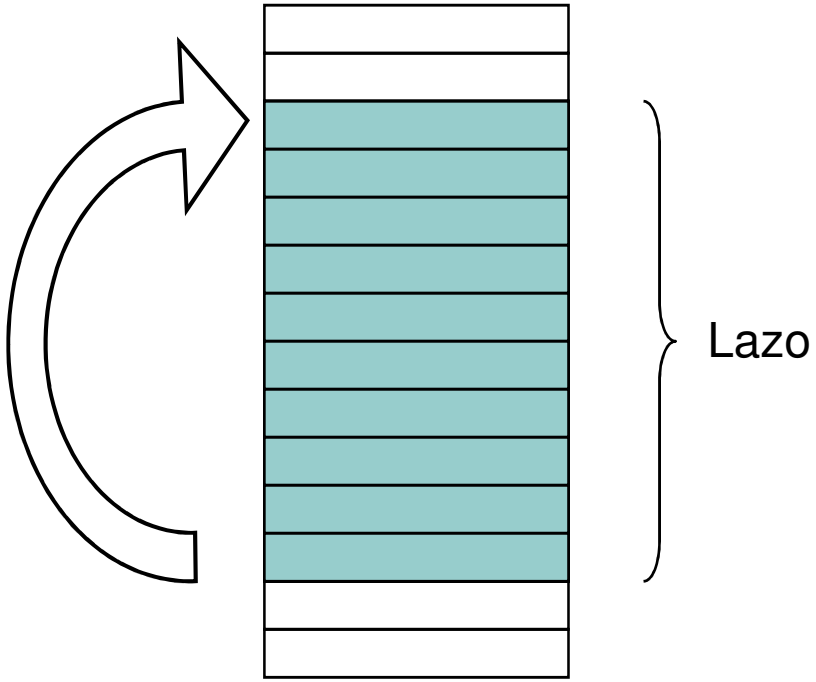
$$Conflicto : D \leq B$$


Desaciertos de la cache totalmente asociativa infinita (**Obligatorios**)
Desaciertos de la cache totalmente asociativa (**Capacidad 3C**)
Desaciertos de la cache bajo estudio
C
Capacidad D3C
Conflicto D3C
A
B
U
$Obligatorios = C$
$Capacidad = A.B.\neg C$
$Conflicto = B.\neg A$
Total de referencias a memoria

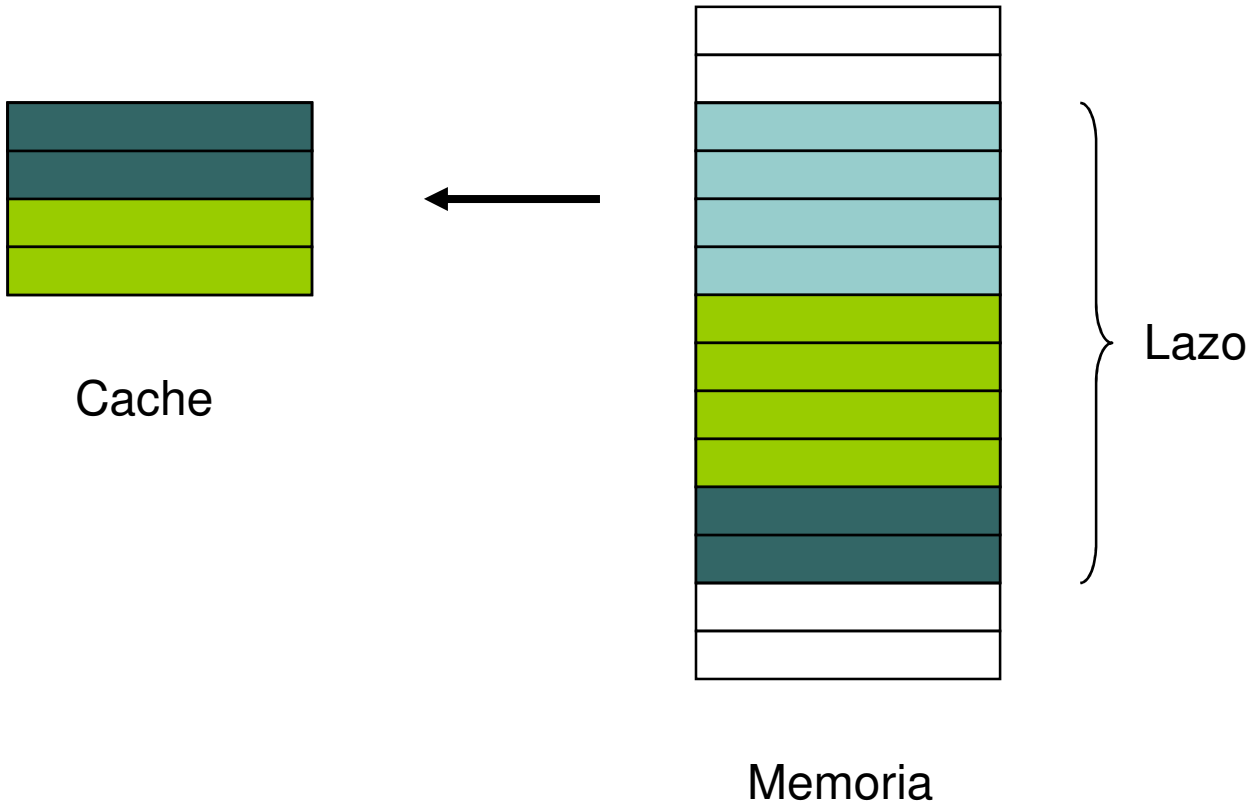
El Modelo del Lazo Simple

- Aplicaciones con desempeño pobre bajo LRU
  - Memoria Virtual
  - Memoria Cache
    - Lazos: componente importante en la mayoría de las aplicaciones.

El Modelo del Lazo Simple



El Modelo del Lazo Simple



Uso de la Clasificación D3C para el Modelo del Lazo Simple

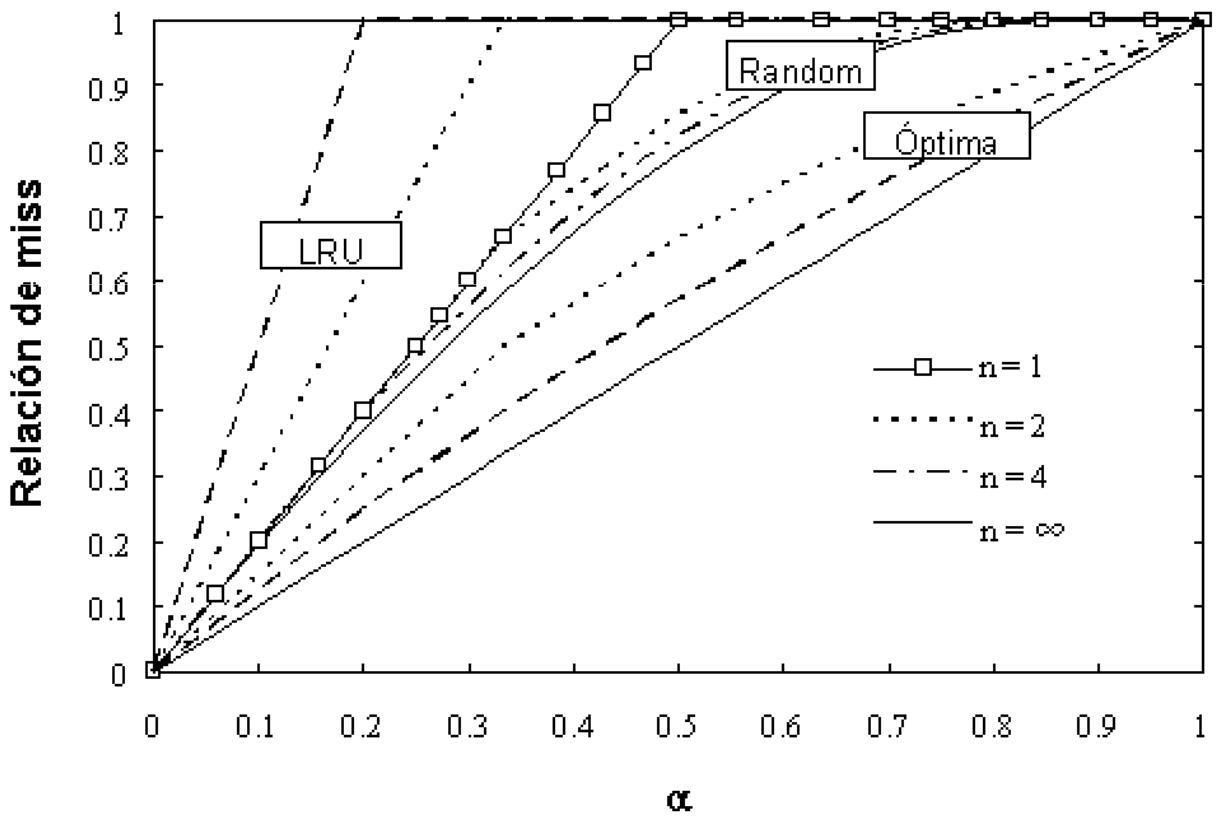
	Obligatorios	Capacidad	Conflicto	Total
3C	0	1	-0,4	0,6
D3C	0	0,6	0	0,6

Cache LRU asociativa por conjuntos de grado 2, lazo 20% más grande que la memoria cache.

El Modelo del Lazo Simple

- Definiciones
  - L tamaño del lazo
  - C tamaño de la memoria cache
  - $\alpha=(L-C)/L$
- Organizaciones analizadas
  - Todas las asociatividades
  - Políticas de reemplazo LRU/FIFO, Random, Óptima y No Reemplazo

Relación de Desaciertos para el Modelo del Lazo Simple



Uso de la Clasificación D3C para el Modelo de las Referencias a Memoria al Azar

	Obligatorios	Capacidad	Conflicto	Total
3C	0	$1-\frac{B}{M}$	0	$1-\frac{B}{M}$
D3C	0	$\frac{\binom{M-1}{B}-\frac{\binom{M-N}{B}}{N}}{\binom{M}{B}}$	≠0	$1-\frac{B}{M}$

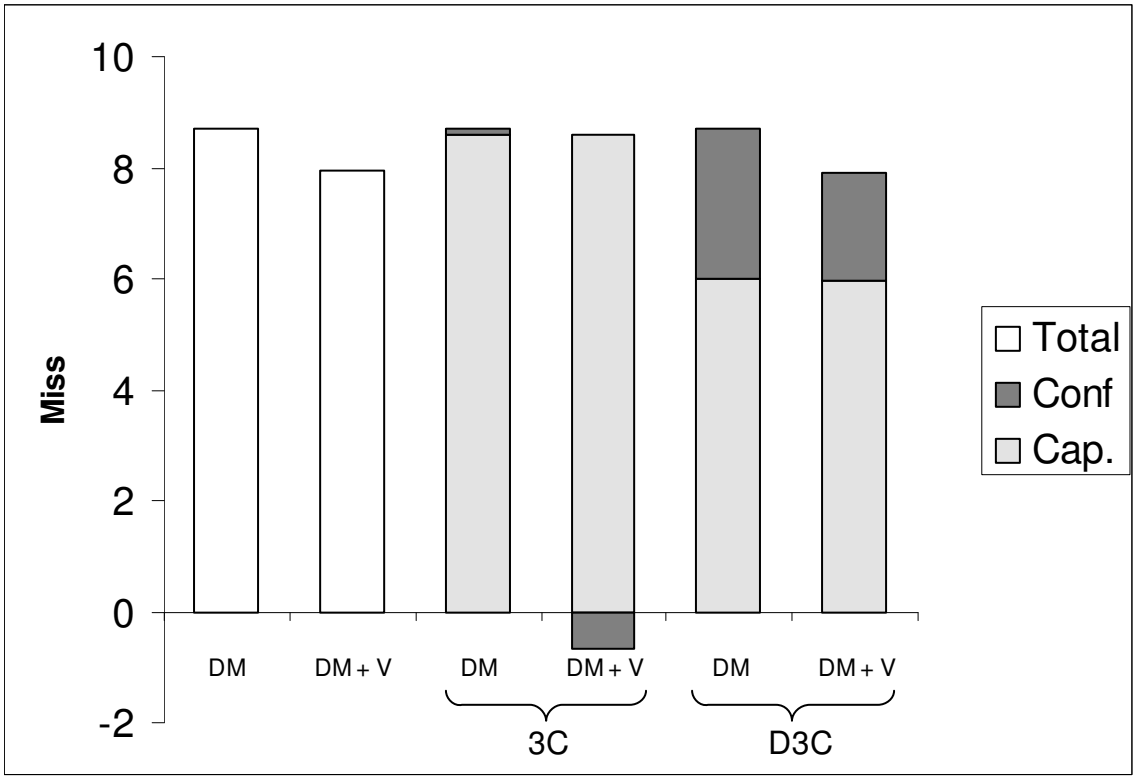
Uso de la Clasificación D3C para el Caso de la Anomalía de Belady

B = 3	Obligatorios	Capacidad	Conflicto	Total
3C	5	5	-1	9
D3C	5	4	0	9

B = 4	Obligatorios	Capacidad	Conflicto	Total
3C	5	3	3	10
D3C	5	3	2	10

Secuencia: 0,1,2,3,0,1,4,0,1,2,3,4, cache totalmente asociativa FIFO

Uso de la Clasificación D3C en la Cache de Víctimas

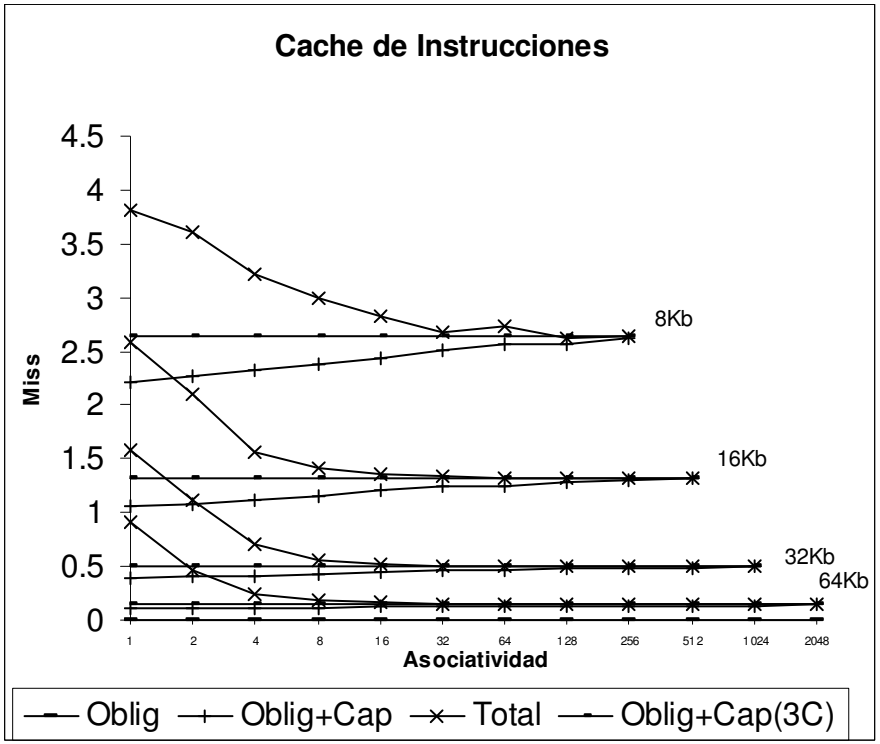


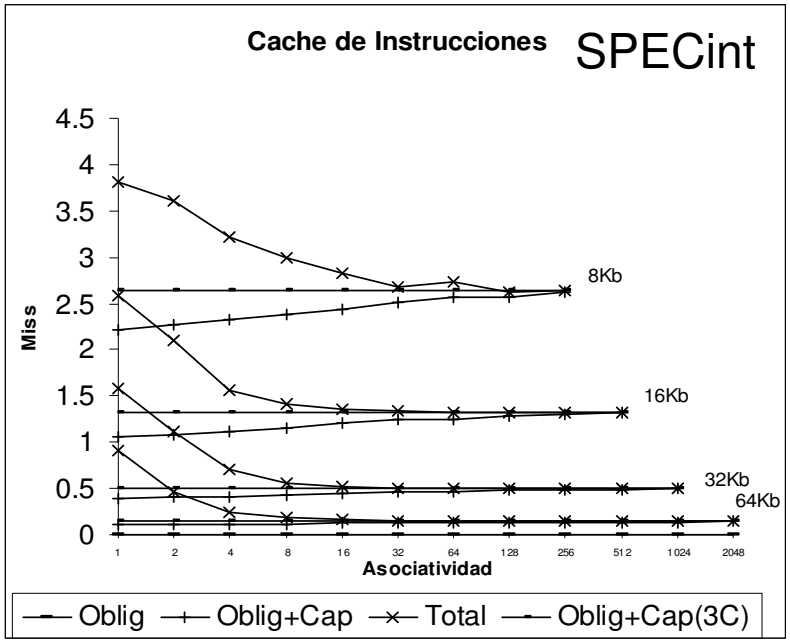
Benchmark Su2cor, cache DM de 8 Kbytes y cache de víctimas de 128 bytes (4 bloques de 32 bytes)

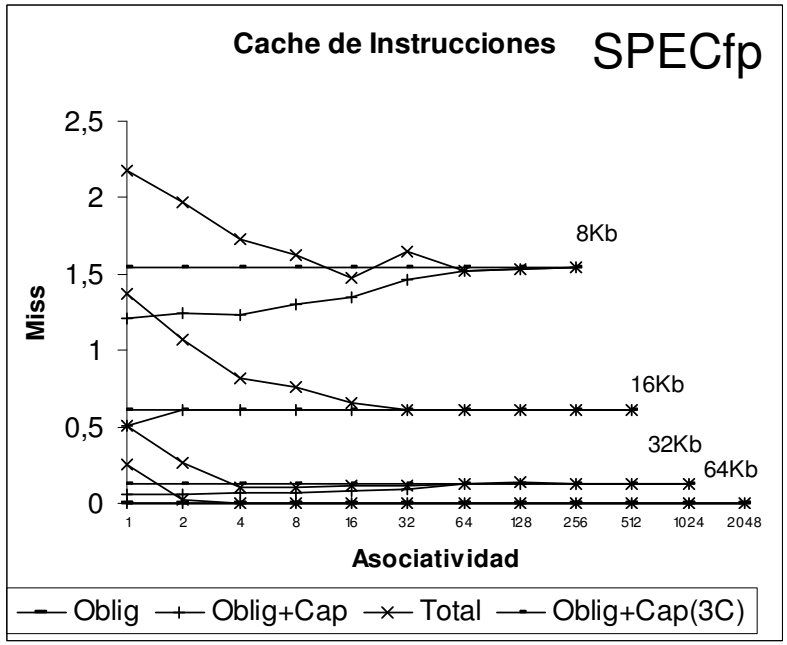
Metodología Experimental

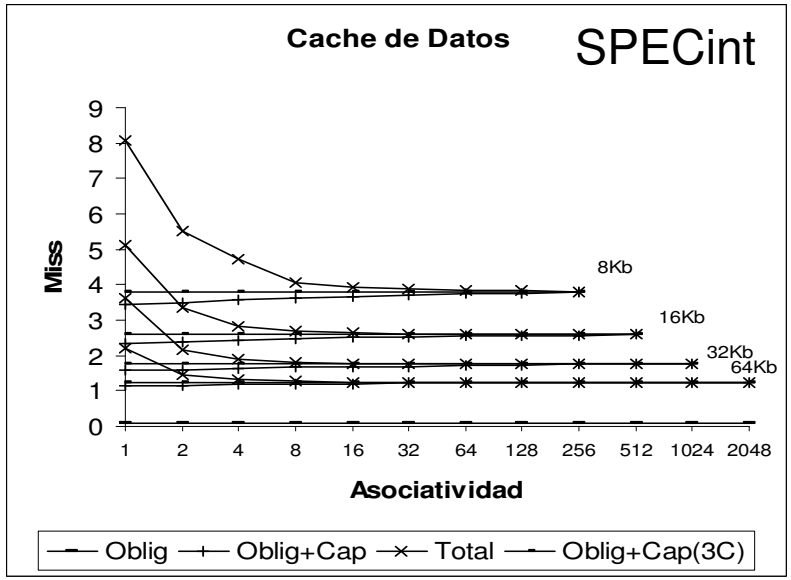
- Simulaciones manejadas por trazas
- Benchmarks SPEC95
  - SPECint95:
    - compress, go, gcc, m88ksim, li, perl, ijpeg, vortex
  - SPECfp95
    - applu, apsi, turb3d, su2cor, hydro2d, swim, tomcatv, wave5, mgrid, fppp.
- Trazas
  - Herramienta: ATOM
  - Formato: ASF
  - Logitud: 700 a 1200 millones de instrucciones

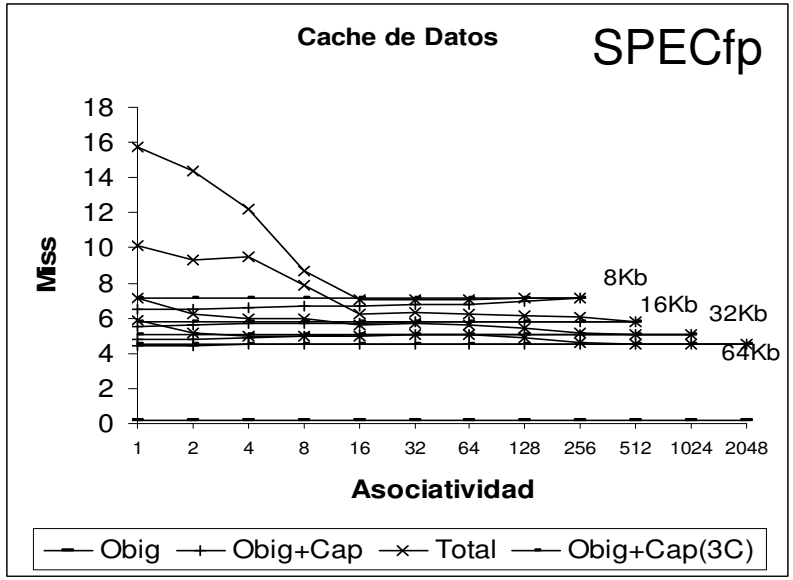
Benchmarks SPECint95




Cache de Instrucciones
SPECint
Miss
Asociatividad
8Kb
16Kb
32Kb
64Kb
Oblig
Oblig+Cap
Total
Oblig+Cap(3C)

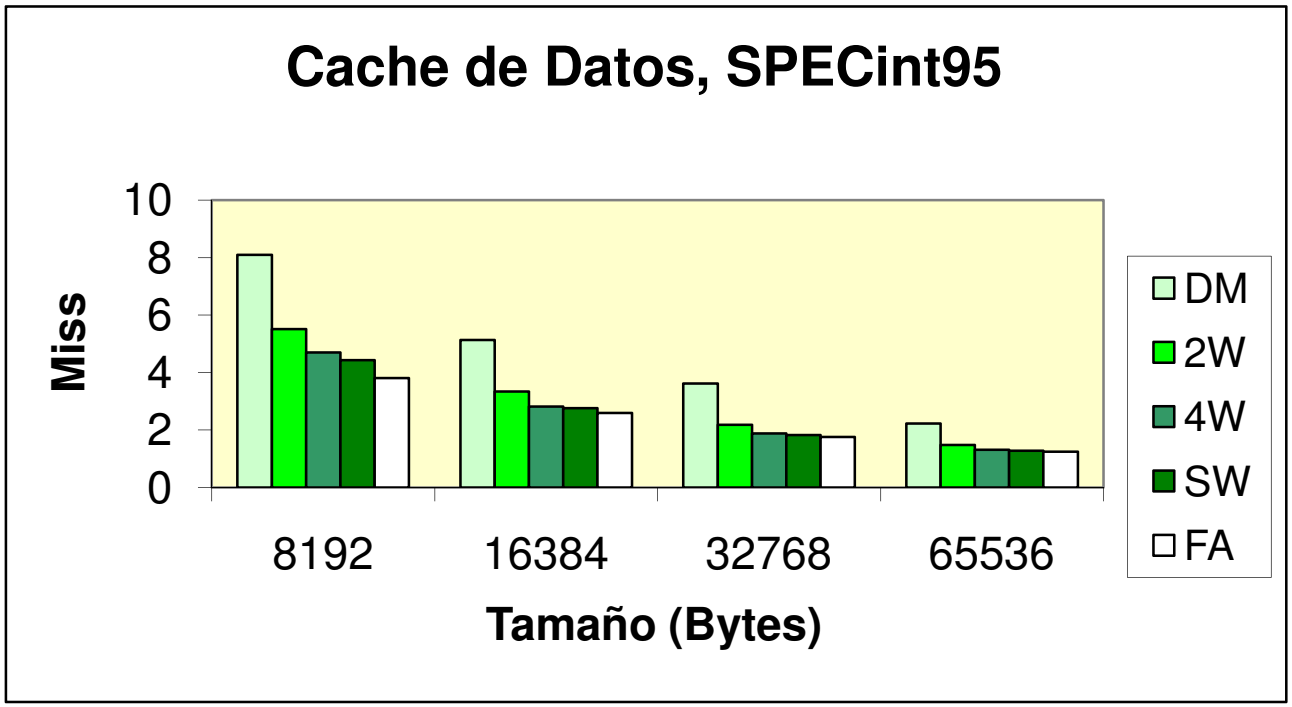

Cache de Instrucciones
SPECfp
Miss
Asociatividad
8Kb
16Kb
32Kb
64Kb
Oblig
Oblig+Cap
Total
Oblig+Cap(3C)


Cache de Datos
SPECint
Miss
Asociatividad
8Kb
16Kb
32Kb
64Kb
Oblig
Oblig+Cap
Total
Oblig+Cap(3C)


Cache de Datos
SPECfp
Miss
Asociatividad
8Kb
16Kb
32Kb
64Kb
Obig
Obig+Cap
Total
Obig+Cap(3C)

Reducción de la Tasa Miss

- Caches más grandes y más asociativas

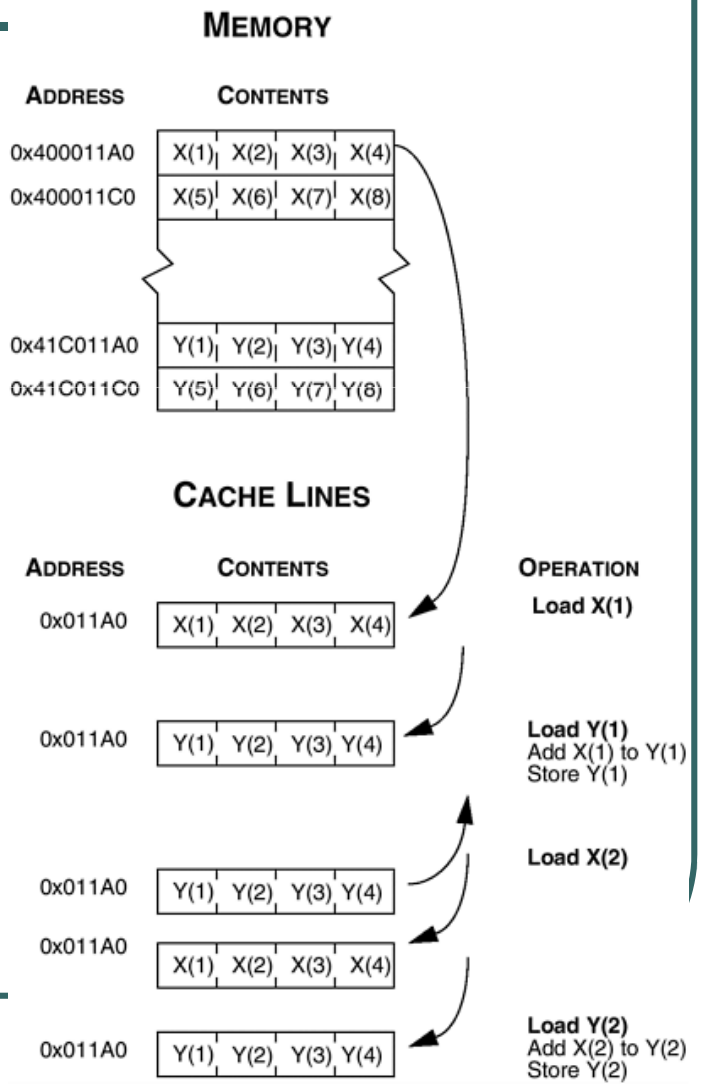


Reducción de la tasa de miss. Asociatividad mas alta.

- La organización de correspondencia directa sufre de *thrashing*.

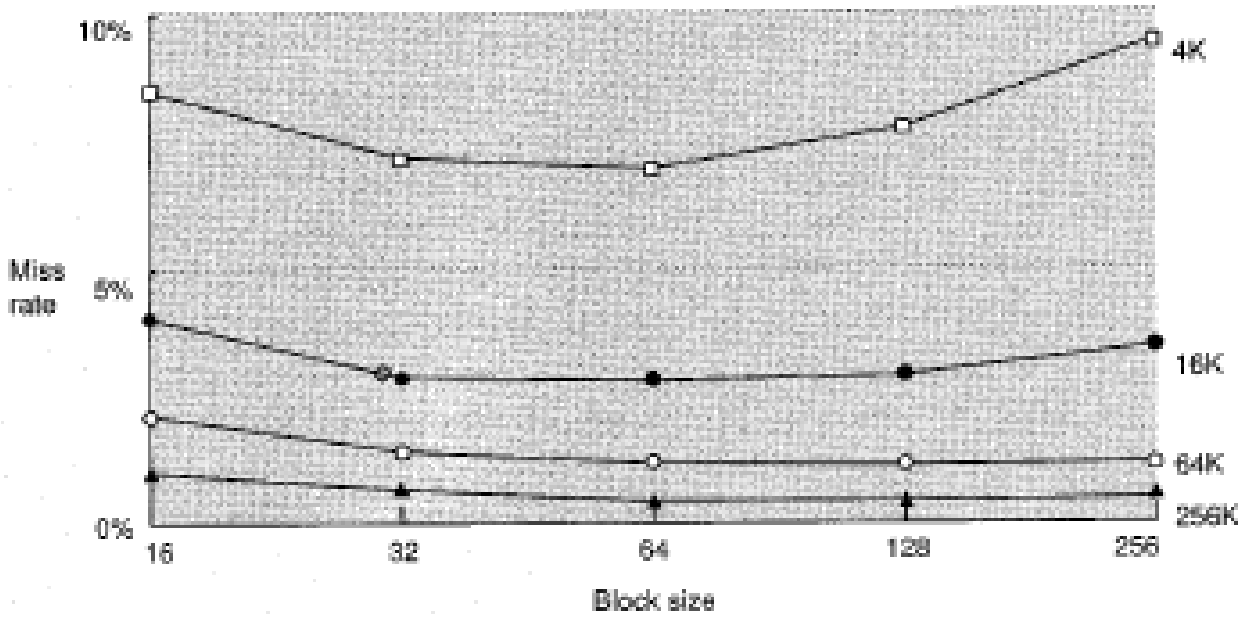
```
double x[MAX], y[MAX];

for (i=0; i<MAX; i++)
  y[i] = x[i] + y[i];
```

- El esquema asociativo por conjunto lo reduce o elimina.



Reducción de la Tasa Miss

- Bloques más grandes



Reducción de Ciclos de stall en cache via paralelismo

- Caches no bloqueantes
- Hardware prefetching
- Software prefetching
  - Programa
  - Compilador

  Opciones

  Buffer prefetching

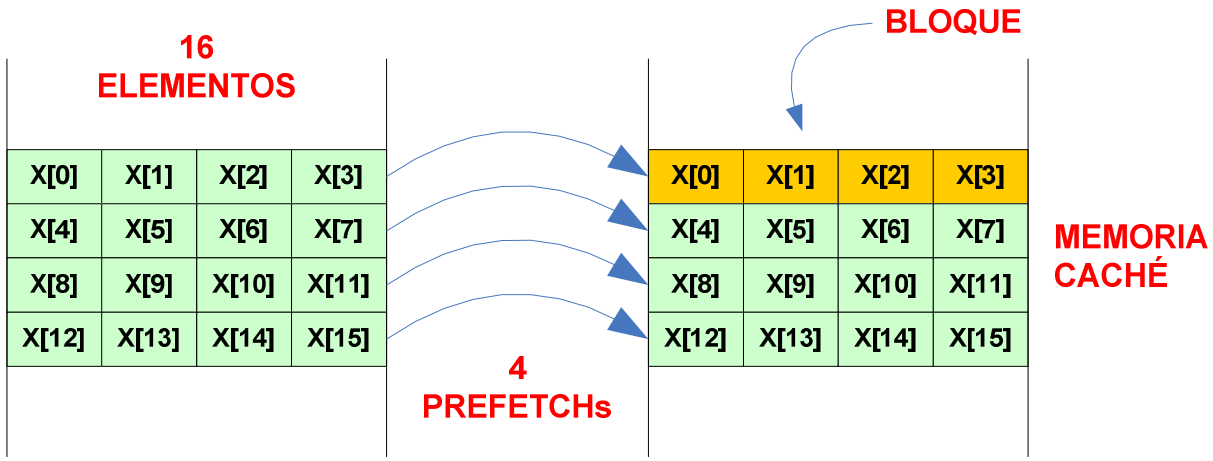
  Cache prefetching

Redución de la tasa de miss. Optimizaciones del Software

- Lectura adelantada (*prefetching*) de datos e instrucciones.
- Reemplazo de elementos de arreglos por escalares.
- Intercambio de bucles.
- Operación en bloques (blocking algorithms).
- Rellenado de arreglos (*array padding*).
- Reducción de solapamiento (*aliasing*).

Optimización de Memoria #1: Lectura Adelantada (*prefetching*)

- Especula con los accesos futuros a datos e instrucciones.
- Se implementa a nivel de *hardware* o *software*.
- Suficiente un *prefetch* por bloque de la caché.



Lectura Adelantada (cont.)

- Útil en bucles con patrones de acceso simples (aplicaciones científicas).
- En lenguaje C, es posible realizar una lectura adelantada mediante `__builtin_prefetch()`, si la arquitectura lo soporta.

```
int x[MAX];

for (i=0; i<MAX; i++)
{
   /* Prefetch */
   __builtin_prefetch( &x[i] );

   acum += x[i];
}
```

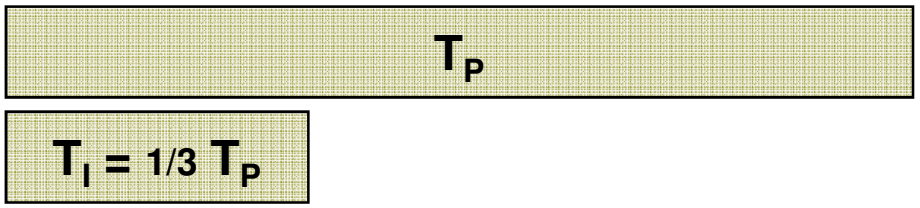
Desde MIPS IV, se soporta *prefetching* mediante la instrucción `pref`

Lectura Adelantada (cont.)

- ¿Con cuánta anticipación debe leerse un bloque desde memoria principal a caché?
- Si se hace cerca del instante de uso, podría ser tarde.
- Conviene aplicarlo con mayor anticipación.
- Deberá considerarse:
  - Cuántos tiempo insume el *prefetch*.
  - Cuántos tiempo insume una iteración del bucle.
- Por ejemplo:

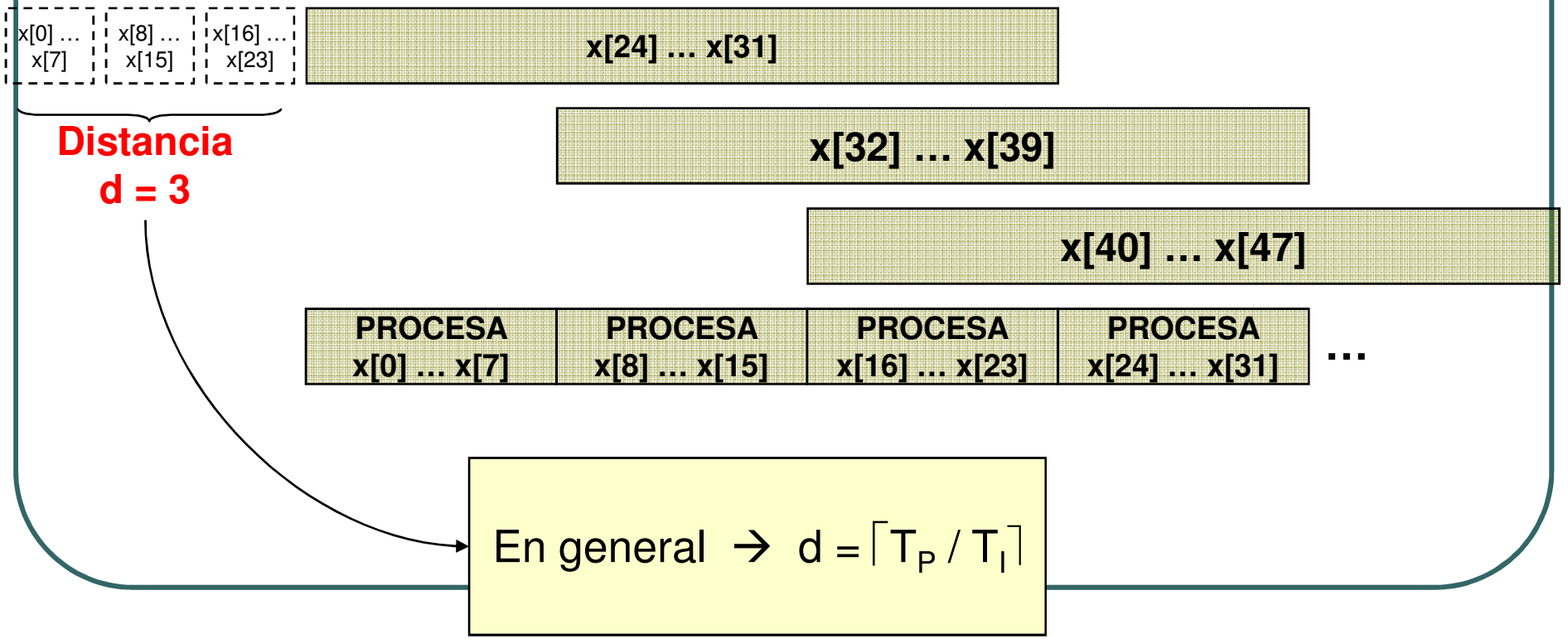
T_P = Tiempo *prefetch*

T_I = Tiempo iteración



Lectura Adelantada (cont.)

- Se debe generar un *pipeline* para que el bucle no deba esperar.



Lectura Adelantada (cont.)

```
for (i=0; i<MAX; i++)
{
  acum += x[i];
}
```

```
#define DELTA_PREFETCH 24
...

for (i=0; i<DELTA_PREFETCH; i+=8)
{
  /* Prefetching */
  __builtin_prefetch( &x[i] );
}

for (i=0; i<MAX; i+=8)
{
  /* Prefetching */
  __builtin_prefetch( &x[i+DELTA_PREFETCH] );

  acum += x[i];
  acum += x[i+1];
  acum += x[i+2];
  acum += x[i+3];
  acum += x[i+4];
  acum += x[i+5];
  acum += x[i+6];
  acum += x[i+7];
}
```

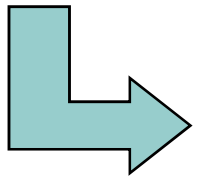
Agregar una estructura condicional (`if..else`) o desenrollar el bucle, para no ejecutar el *prefetch* en todas las iteraciones.

FACTOR DE DESENROLLADO IGUAL A LA CANT. DE ELEMENTOS POR BLOQUE DE CACHÉ

Optimización de Memoria #2: Reemplazo de Elem. de Arreglos

- Pocos compiladores intentan mantener los elementos de un arreglo en registros, entre iteraciones sucesivas.
- Reemplazar el elemento de una arreglo por una variable temporal, para que pueda mantenerse en un registro.

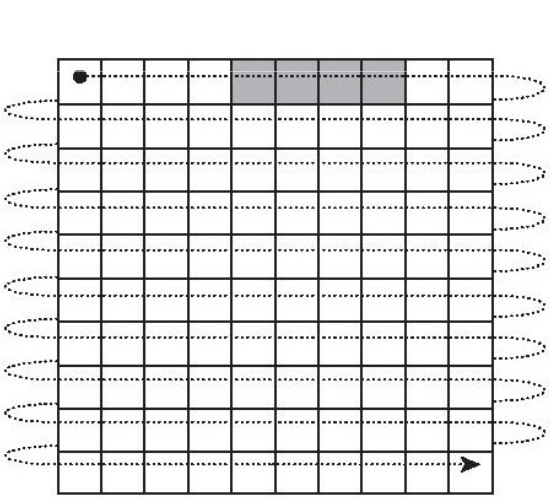
```
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
   {
      c[i][j] = 0;
      for (k=0; k<N; k++)
         c[i][j] += a[i][k] * b[k][j];
   }
```



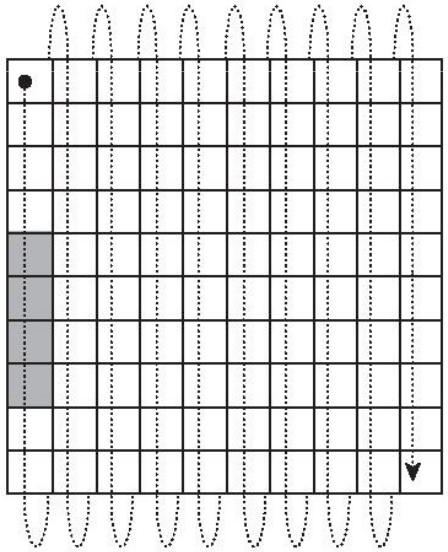
```
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
   {
      aux = 0;
      for (k=0; k<N; k++)
         aux += a[i][k] * b[k][j];
      c[i][j] = aux;
   }
```

Optimización de Memoria #3: Intercambio de Bucles

- Según el lenguaje, un arreglo multidimensional podría almacenarse en memoria, ordenado por filas o por columnas.



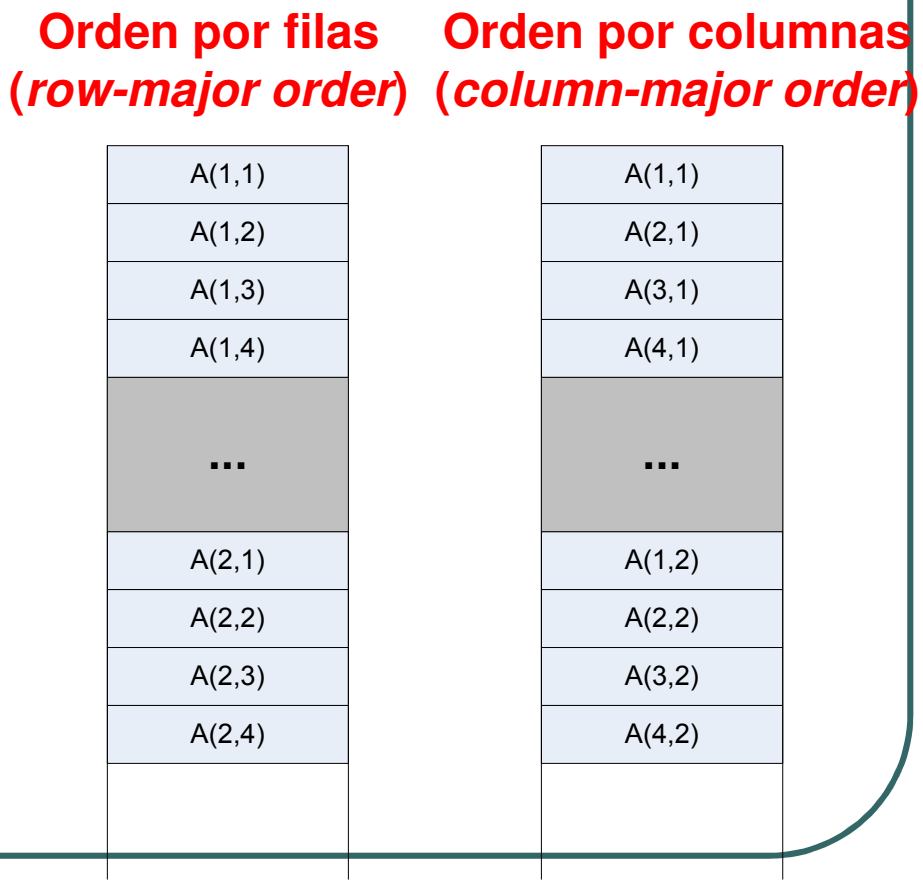
Orden por filas
(*row-major order*)



Orden por columnas
(*column-major order*)

Intercambio de Bucles (cont.)

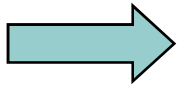
- Se busca acceder al arreglo en pasos unitarios (*unit stride*).
- Esto permite aprovechar la localidad espacial en la memoria caché de datos.



Intercambio de Bucles (cont.)

- C/C++, Java, Pascal utilizan *row-major order*.
- Fortran y versiones viejas de BASIC, utilizan *column-major order*.
- Por lo tanto, deben *intercambiarse* los bucles, de ser necesario:

```
/* No óptimo */
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    x[i][j] = 0;
```



```
/* Óptimo */
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    x[i][j] = 0;
```

Intercambio de Bucles (cont.)

Usando row-major order:

```
D1 cache:         2048 B, 32 B, 2-way associative

      Ir I1mr I2mr       Dr D1mr D2mr      Dw   D1mw   D2mw

       .    .    .        .    .    .       .      .      .  int main(void)
      10    1    0        0    0    0       1      0      0  {
       .    .    .        .    .    .       .      .      .      int matriz[256][256];
       .    .    .        .    .    .       .      .      .      register int i,j;
       .    .    .        .    .    .       .      .      .
   1,028    2    2      513    0    0       1      1      1      for (i=0; i<256; i++)
 263,168    0    0  131,328    0    0     256      0      0         for (j=0; j<256; j++)
 262,144    1    0  131,072    0    0  65,536  8,192  4,051            matriz[i][j] = 0;
       .    .    .        .    .    .       .      .      .
       1    0    0        0    0    0       0      0      0      return 0;
       2    0    0        2    0    0       0      0      0  }
```

Desaciertos de escritura en la matriz (1 cada 8 escrituras)

Intercambio de Bucles (cont.)

Usando column-major order:

```
D1 cache:         2048 B, 32 B, 2-way associative

      Ir I1mr I2mr       Dr D1mr D2mr      Dw    D1mw   D2mw

       .    .    .        .    .    .       .       .      .  int main(void)
      10    1    0        0    0    0       1       0      0  {
       .    .    .        .    .    .       .       .      .     int matriz[256][256];
       .    .    .        .    .    .       .       .      .     register int i,j;
       .    .    .        .    .    .       .       .      .
   1,028    2    2      513    0    0       1       1      1     for (j=0; j<256; j++)
 263,168    0    0  131,328    0    0     256       0      0        for (i=0; i<256; i++)
 262,144    1    0  131,072    0    0  65,536  65,532  4,051           matriz[i][j] = 0;
       .    .    .        .    .    .       .       .      .
       1    0    0        0    0    0       0       0      0     return 0;
       2    0    0        2    0    0       0       0      0  }
```

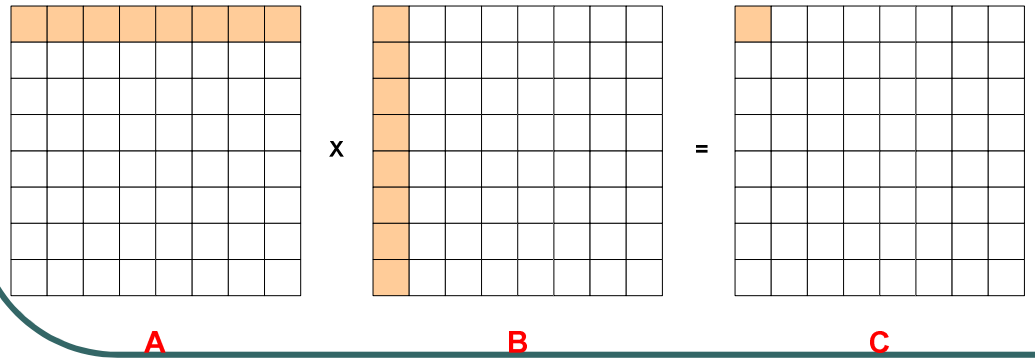
Desaciertos de escritura en la matriz (8 cada 8 escrituras)

Optimización de Memoria #4: Operación en Bloques

- Permite decrementar la cantidad de desaciertos *de capacidad* cuando se opera con arreglos grandes.
- Mejora la localidad temporal (mantiene en la caché, datos que se usarán en el corto plazo).

Operación en Bloques (cont.)

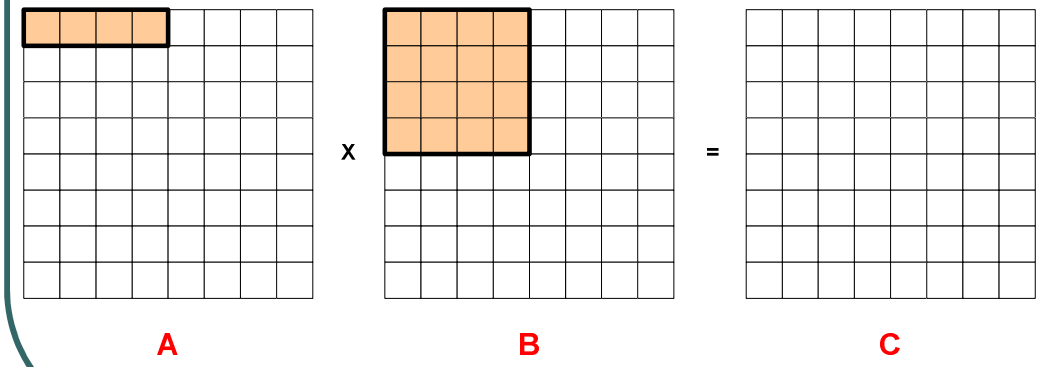
- Ejemplo: multiplicación de matrices.
- Elevada cantidad de desaciertos.
- Para una caché 2WSA 2-KB con bloques de 32 bytes, multiplicar dos matrices de 64x64 enteros, produce una tasa de desaciertos del 11,3%




```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
  {
    aux = 0.0;
    for (k=0; k<N; k++)
      aux = aux + a[i][k] * b[k][j];
    c[i][j] = aux;
  }
```

Operación en Bloques (cont.)

- Solución: operar en bloques pequeños.
- Se reducen los *accesos conflictivos*, ya que los bloques pequeños pueden ser mantenidos en la caché.

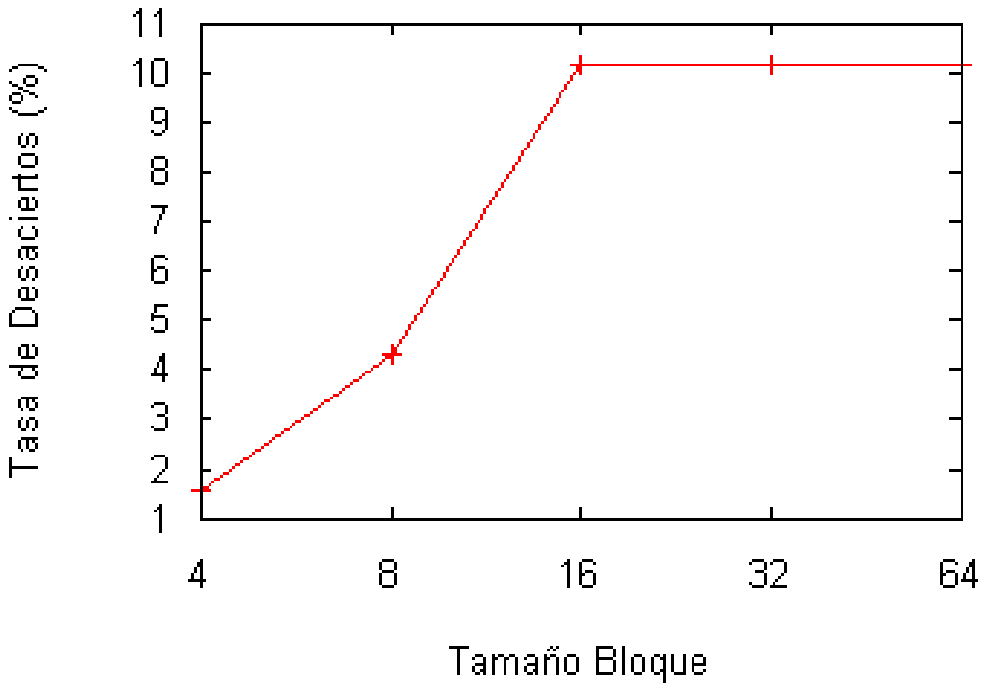



```
for (jj=0; jj<N; jj+=B)
  for (kk=0; kk<N; kk+=B)
    for (i=0; i<N; i++)
      for (j=jj; j<jj+B; j++)
      {
        aux = 0.0;
        for (k=kk; k<kk+B; k++)
          aux = aux + a[i][k] * b[k][j];
        c[i][j] += aux;
      }
```

Operación en Bloques (cont.)

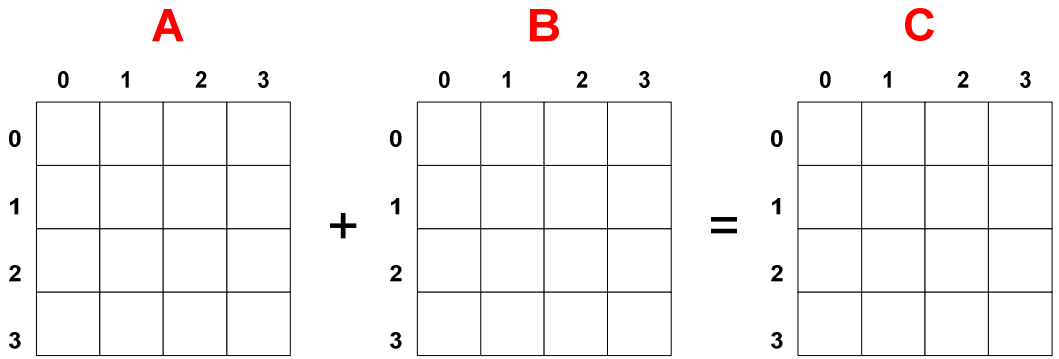
```
#define N 64

for (jj=0; jj<N; jj+=B)
  for (kk=0; kk<N; kk+=B)
    for (i=0; i<N; i++)
      for (j=jj; j<jj+B; j++)
      {
        aux = 0;
        for (k=kk; k<kk+B; k++)
          aux += a[i][k] * b[k][j];
        c[i][j] += aux;
      }
```

2WSA 2-KB (bloque de 32 bytes)

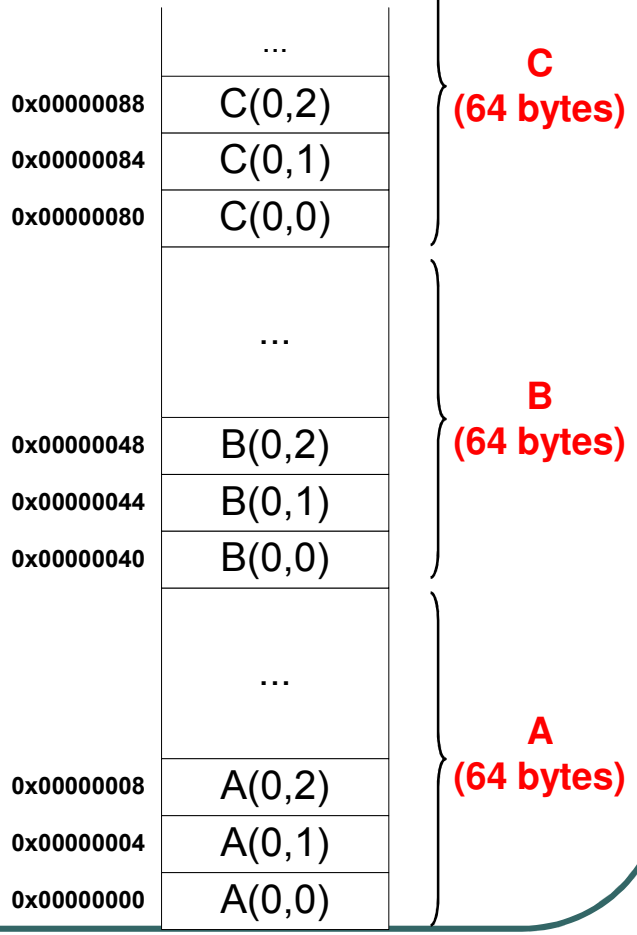


Optimización de Memoria #5: Rellenado de Arreglos (*padding*)

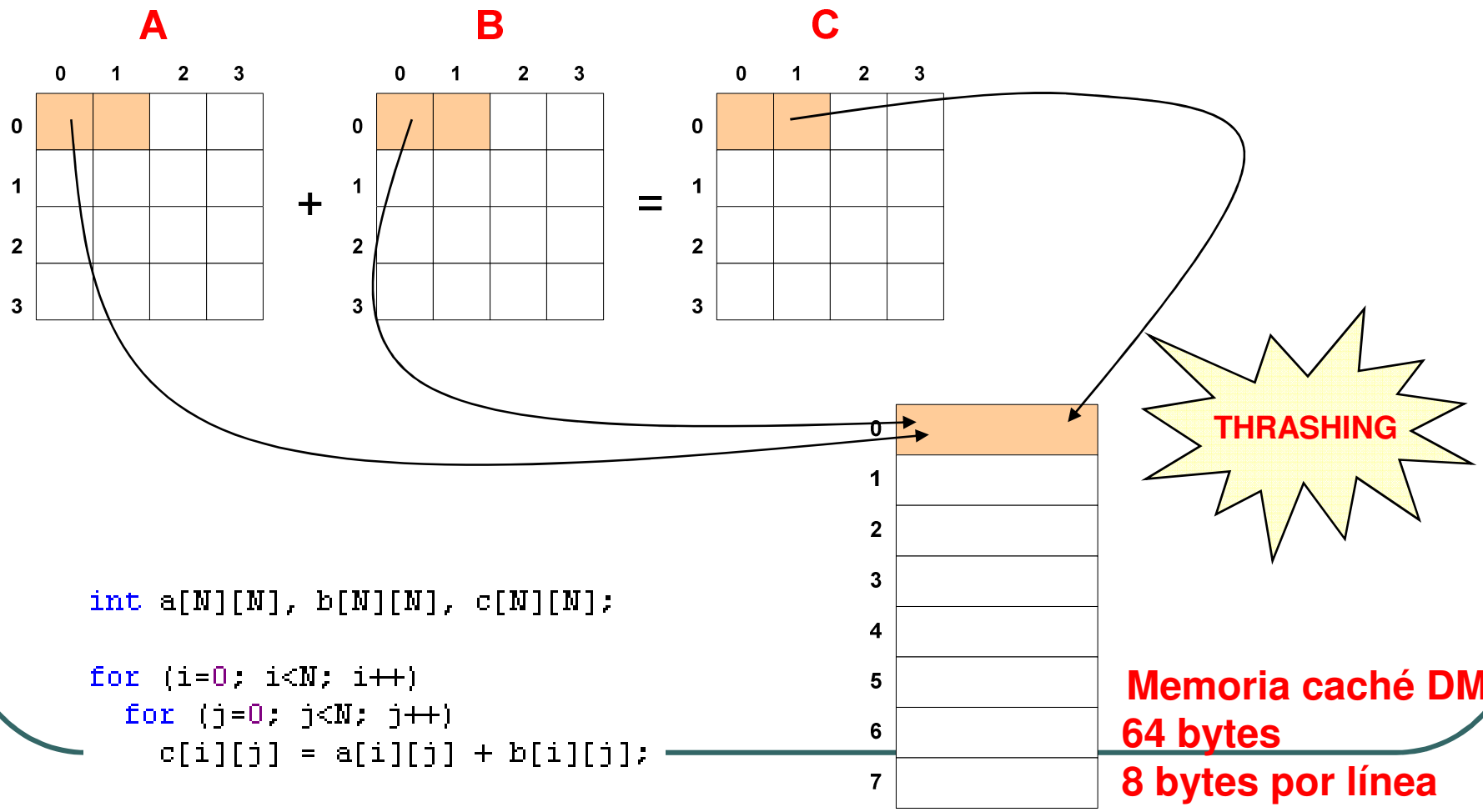
- Suma de matrices.




```
int a[N][N], b[N][N], c[N][N];

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

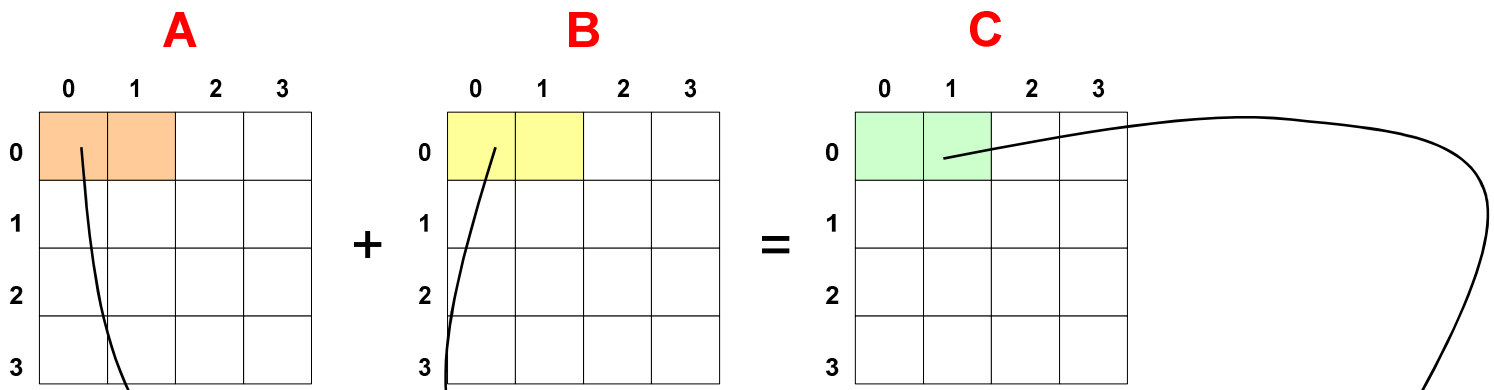


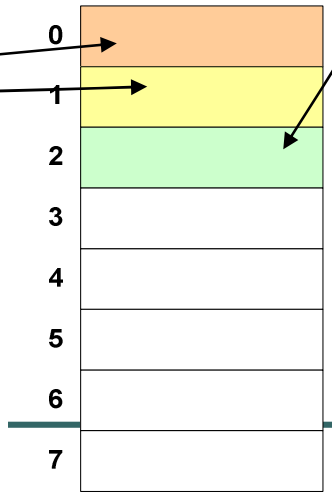
Rellenado de Arreglos (cont.)




```
int a[N][N], b[N][N], c[N][N];

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

Rellenado de Arreglos (cont.)






```
int a[N][N], pad1[2], b[N][N], pad2[2], c[N][N];

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

Memoria caché DM
64 bytes
8 bytes por línea

Rellenado de Arreglos (cont.)

- Se cambia la forma en que los arreglos son almacenados en memoria principal.
- Reduce los accesos *conflictivos*.
- Disminuye el *thrashing* y, por consiguiente, la cantidad de desaciertos.
- Como desventaja, utiliza mayor cantidad de memoria.

Rellenado de Arreglos (cont.)

Sin rellenado:

```
D1 cache:          4096 B, 32 B, direct-mapped

     Ir I1mr I2mr      Dr   D1mr D2mr     Dw   D1mw D2mw
      .    .    .       .      .    .      .      .    .   #define N 64
      .    .    .       .      .    .      .      .    .
      .    .    .       .      .    .      .      .    .   int main(void)
     11    1    0       0      0    0      2      0    0   {
      .    .    .       .      .    .      .      .    .       register int i,j;
      .    .    .       .      .    .      .      .    .       int a[N][N], b[N][N], c[N][N];
      .    .    .       .      .    .      .      .    .
    260    0    0     129      0    0      1      1    1       for (i=0; i<N; i++)
 16,640    2    1   8,256     28    0     64      0    0          for (j=0; j<N; j++)
 57,344    2    1  32,768  8,192  468  4,096  4,096  255             c[i][j] = a[i][j] + b[i][j];
      .    .    .       .      .    .      .      .    .
      1    0    0       0      0    0      0      0    0       return 0;
      3    0    0       3      1    0      0      0    0   }
```

Desaciertos de lectura (8 cada 8 lecturas)

Desaciertos de escritura en la matriz (8 cada 8 escrituras)

Rellenado de Arreglos (cont.)

Con rellenado:

```
D1 cache:          4096 B, 32 B, direct-mapped

    Ir I1mr I2mr      Dr   D1mr D2mr     Dw D1mw D2mw
     .    .    .       .      .    .      .    .    .   #define N 64
     .    .    .       .      .    .      .    .    .
     .    .    .       .      .    .      .    .    .   int main(void)
    11    1    0       0      0    0      2    0    0   {
     .    .    .       .      .    .      .    .    .       register int i,j;
     .    .    .       .      .    .      .    .    .       int a[N][N], pad1[8],
     .    .    .       .      .    .      .    .    .           b[N][N], pad2[8], c[N][N];
     .    .    .       .      .    .      .    .    .
   260    0    0     129      0    0      1    1    1       for (i=0; i<N; i++)
16,640    2    1   8,256     92    0     64    0    0          for (j=0; j<N; j++)
57,344    2    1  32,768  1,082  469 4,096  536  255             c[i][j] = a[i][j] + b[i][j];
     .    .    .       .      .    .      .    .    .
     1    0    0       0      0    0      0    0    0       return 0;
     3    0    0       3      0    0      0    0    0   }
```

Rellenos

Desaciertos de lectura de matrices A y B (1 cada 8 lecturas)

Desaciertos de escritura en la matriz C (1 cada 8 escrituras)

Algo sobre Valgrind

- Herramienta para profiling y debugging.
- Site: http://valgrind.org/
- Soporta cualquier lenguaje compilado.
- Pública y de código abierto.
- Aplica la técnica de *instrumentación dinámica de código*.
- Ejemplo de ejecución:

```
valgrind --tool=<tool> my_prog
```

Algo sobre Valgrind (cont.)

- Cachegrind es el módulo para simulación de memorias caché.
- Junto con la herramineta `cg_annotate`, es posible detectar el origen de los desaciertos en caché.

```
$ valgrind --tool=cachegrind --D1=2048,2,32 prueba
```

(genera un archivo `cachegrind.out.<pid>`)

```
$ cg_annotate --<pid> prueba.c
```

IMPORTANTE: no olvidar compilar `prueba.c` utilizando la opción `-g` para incluir información de *debugging* útil para Valgrind).

Redución del tiempo de hit

- Caches más pequeñas y más simples

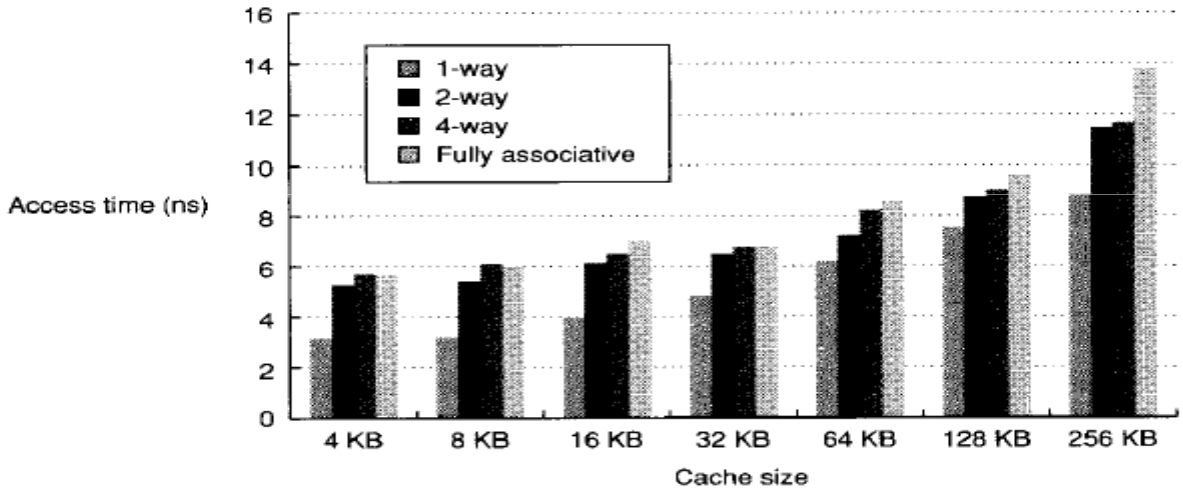
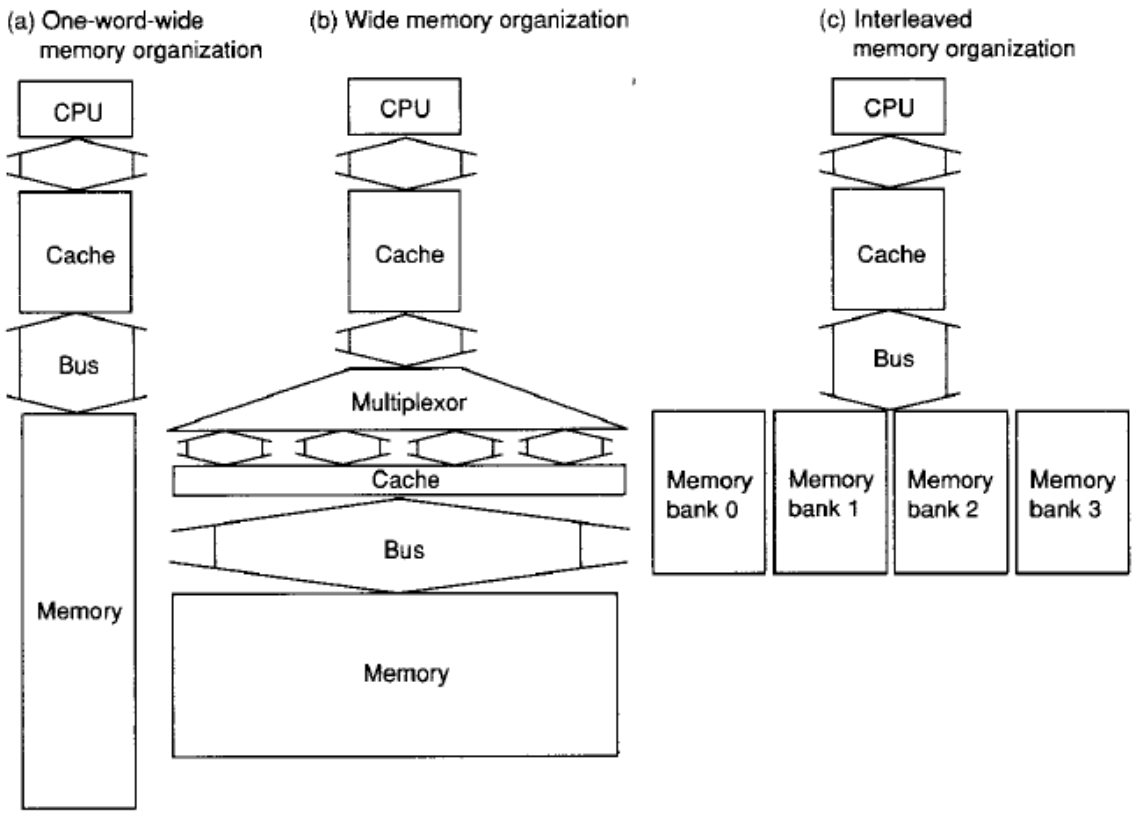


Figure 5.24 Access times as size and associativity vary in a CMOS cache. These data are based on Spice runs used to validate the CACTI model 2.0 by Reinman and Jouppi [1999]. They assumed 0.80-micron feature size, a single read/write port, 32 address bits, 64 output bits, and 32-byte blocks. The median ratios of access time relative to the direct-mapped caches are 1.36, 1.44, and 1.52 for 2-way, 4-way, and 8-way associative caches, respectively.

Redución del tiempo de hit

- Evitar traducción de páginas
- Cache pipeline
- Trace Caches (intel p4)

Mejorar el ancho de banda de la memoria principal.



Intercalado

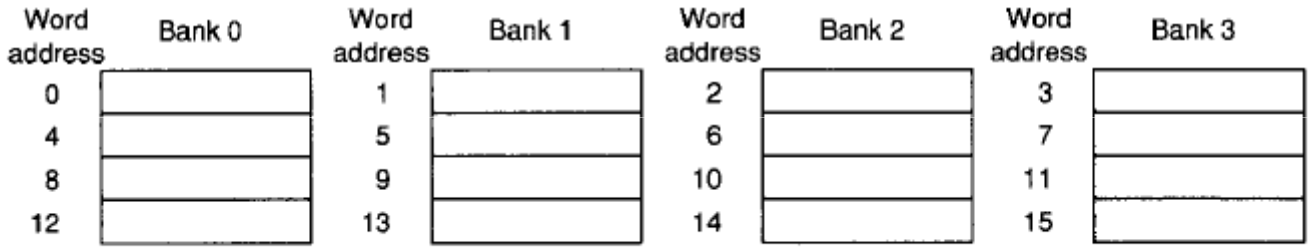


Figure 5.28 Four-way interleaved memory. This example assumes word addressing: With byte addressing and 8 bytes per word, each of these addresses would be multiplied by 8.

Tecnología DRAM

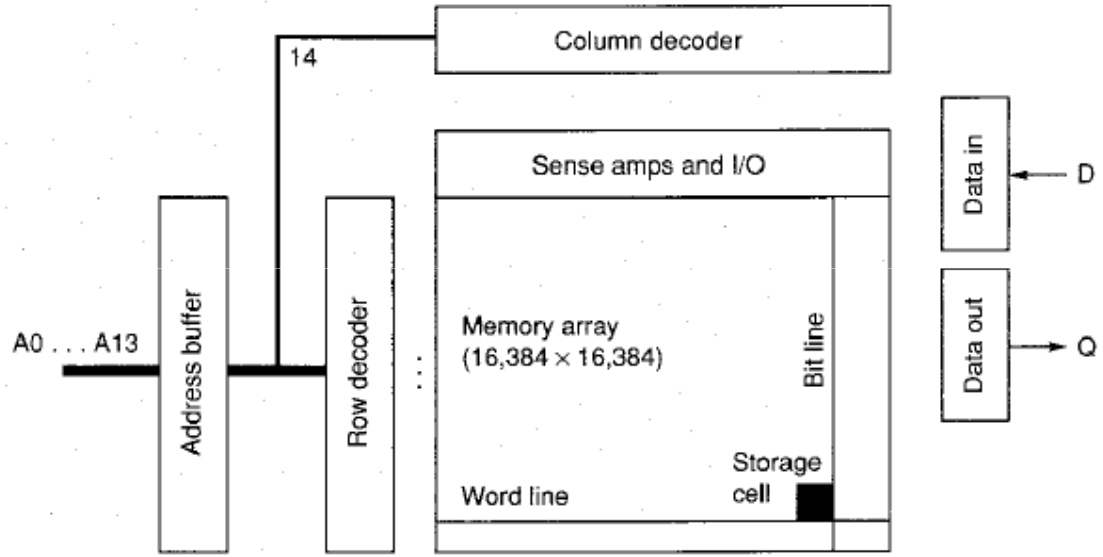
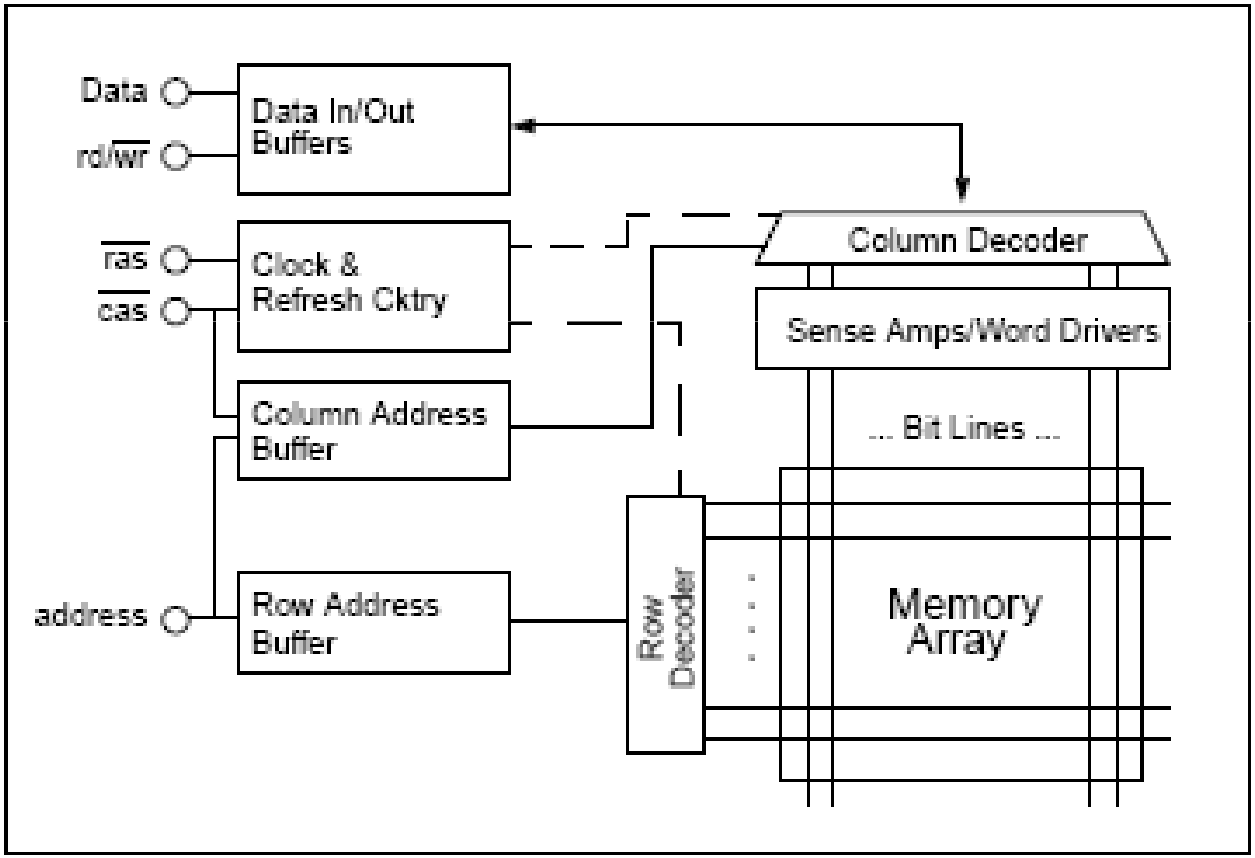


Figure 5.29 Internal organization of a 64M bit DRAM. DRAMs often use banks of memory arrays internally and select between them. For example, instead of one 16,384 × 16,384 memory, a DRAM might use 256 1024 × 1024 arrays or 16 2048 × 2048 arrays.

Year of introduction	Chip size	Row access strobe (RAS)		Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
		Slowest DRAM (ns)	Fastest DRAM (ns)		
1980	64K bit	180	150	75	250
1983	256K bit	150	120	50	220
1986	1M bit	120	100	25	190
1989	4M bit	100	80	20	165
1992	16M bit	80	60	15	120
1996	64M bit	70	50	12	110
1998	128M bit	70	50	10	100
2000	256M bit	65	45	7	90
2002	512M bit	60	40	5	80

Figure 5.30 Times of fast and slow DRAMs with each generation. Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access accompanied the switch from NMOS DRAMs to CMOS DRAMs.

DRAM Convencional



DRAM de Modo Página Rápido

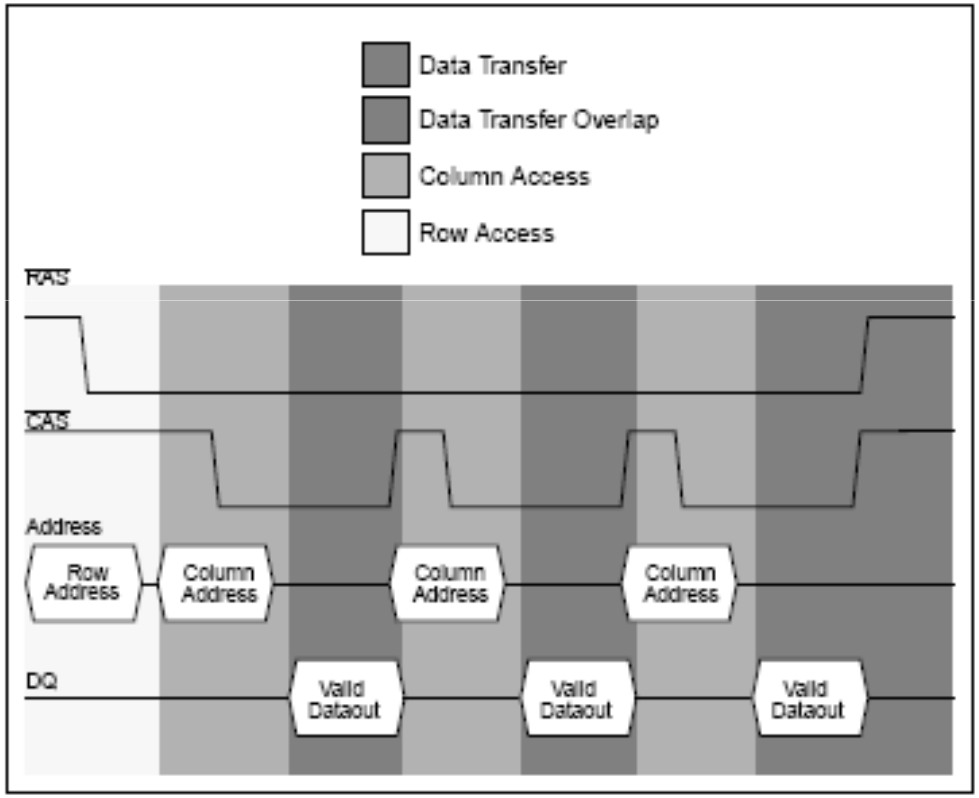


Figure 2: FPM Read Timing. Fast page mode allows the DRAM controller to hold a row constant and receive multiple columns in rapid succession.

DRAM de Modo Página Rápido (EDO)

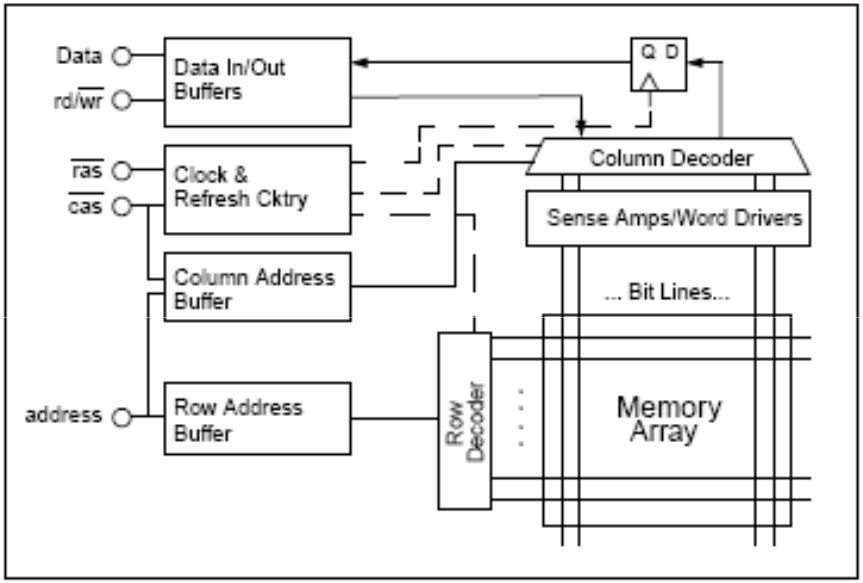


Figure 3: Extended Data Out (EDO) DRAM block diagram. EDO adds a latch on the output that allows CAS to cycle more quickly than in FPM.

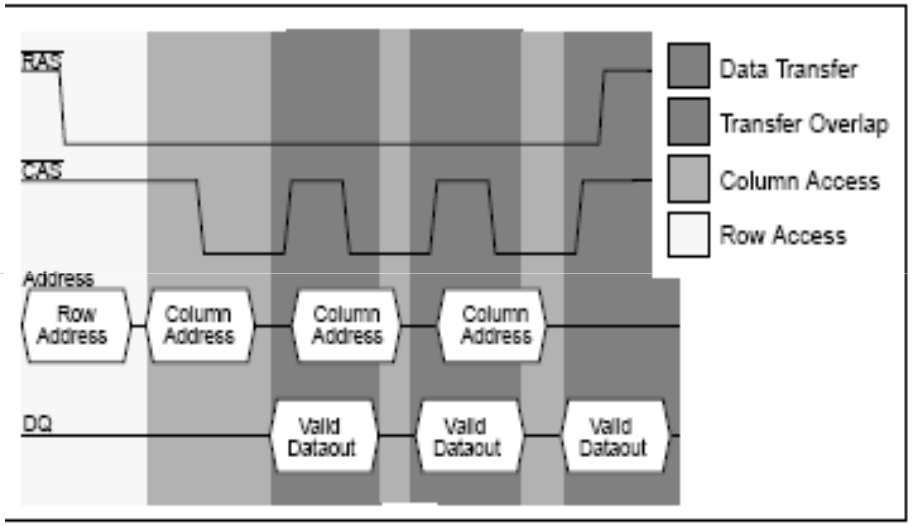


Figure 4: EDO Read Timing. The output latch in EDO DRAM allows more overlap between column access and data transfer than in FPM.

RAM Sincrónica (SDRAM)

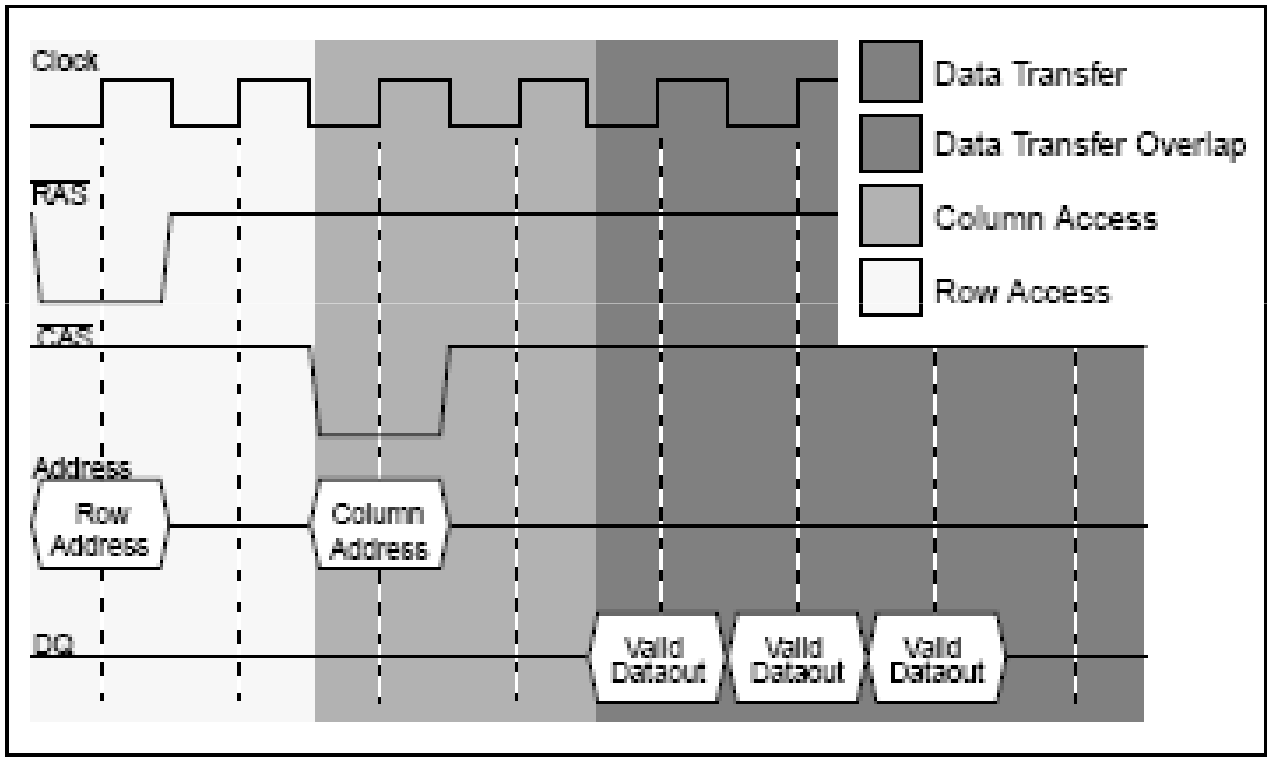


Figure 5: SDRAM Read Operation Clock Diagram. SDRAM contains a writable register for the request length, allowing high-speed column access.

Evolución DRAM sincrónica

- SDRAM
- DDR
- DDR2
- DDR3
- DDR4, 5, etc (video)

Memoria Virtual

Memoria Virtual

	Programa puede superar la memoria fisica	Protección	Compartir	Direcciones Virtuales
Overlays	x			
Registro Cerco		x		x
Memoria V. Páginada	x	x	x	x
Memoria V. Segmentada	x	x	x	x

Memoria Virtual Páginada

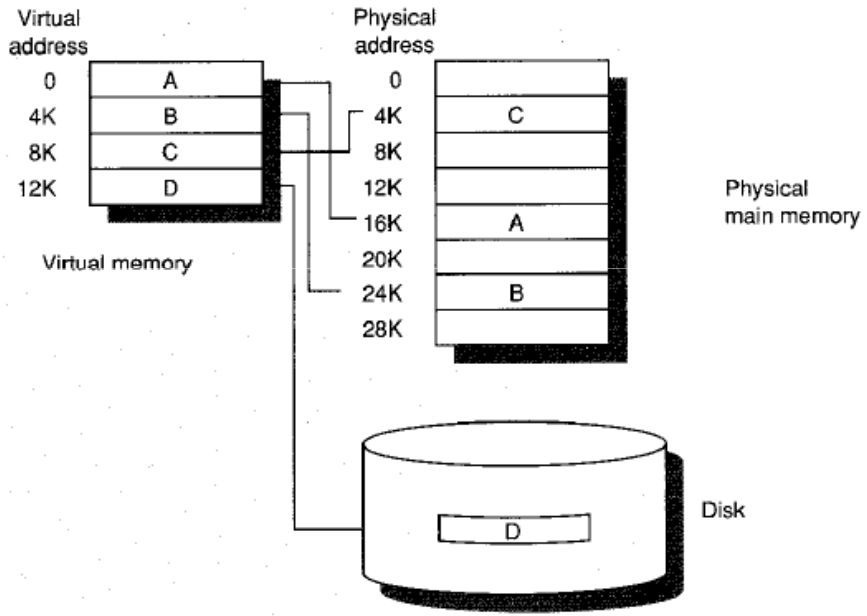
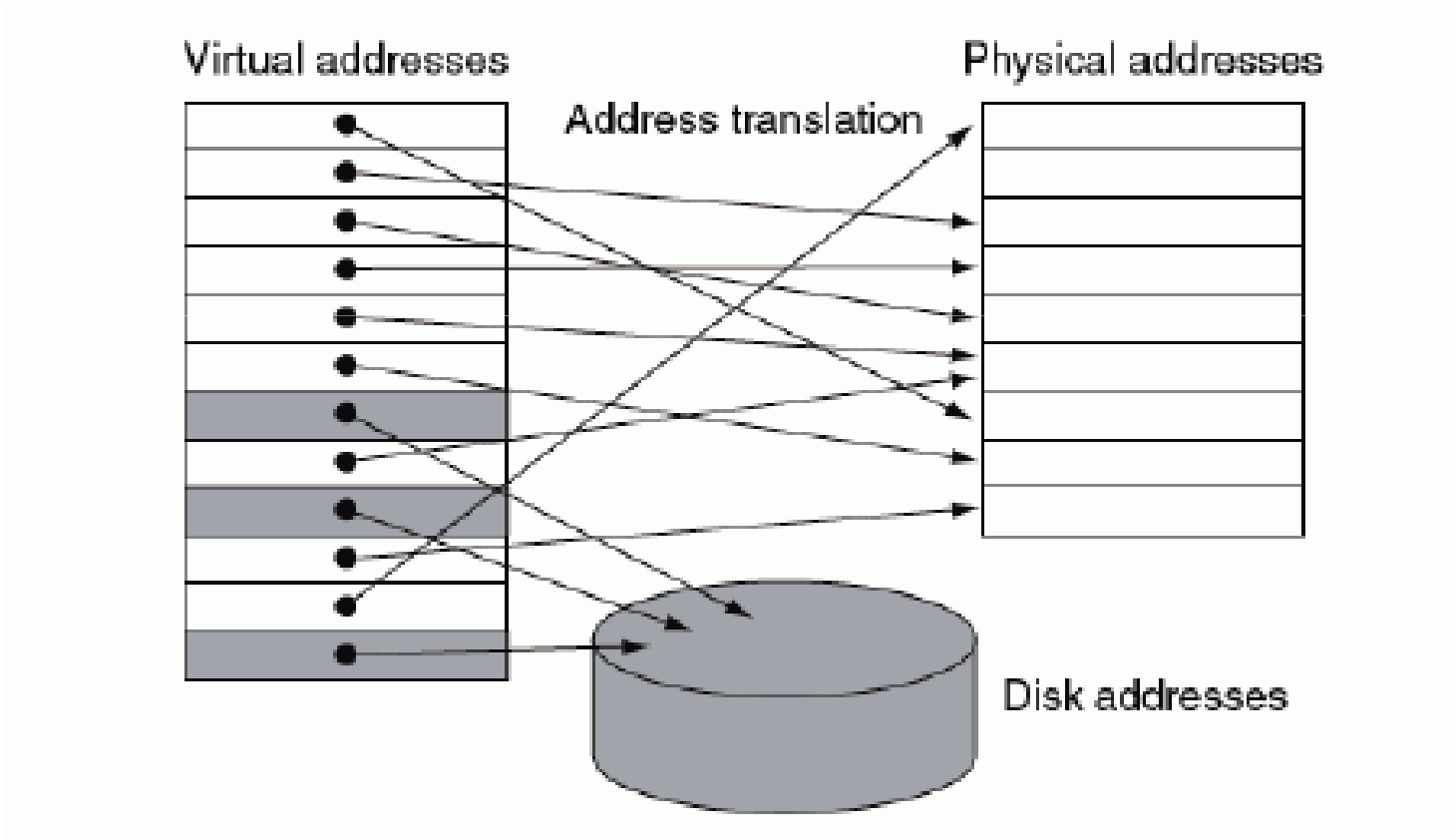


Figure 5.31 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

Memoria Virtual Páginada: tamaños de los espacios virtual y físico



Paginado vs Segmentado

Segmentado: bloques de tamaño variable → Dos palabras por Dirección: nro segmento y offset

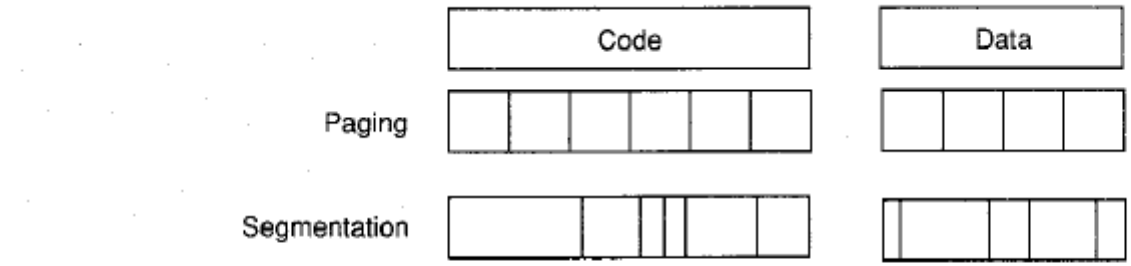


Figure 5.33 Example of how paging and segmentation divide a program.

Comparación parámetros Memoria Virual–Memoria Cache

Parameter	First-level cache	Virtual memory
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–3 clock cycles	50–150 clock cycles
Miss penalty	8–150 clock cycles	1,000,000–10,000,000 clock cycles
(access time)	(6–130 clock cycles)	(800,000–8,000,000 clock cycles)
(transfer time)	(2–20 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.1–10%	0.00001–0.001%
Address mapping	25–45 bit physical address to 14–20 bit cache address	32–64 bit virtual address to 25–45 bit physical address

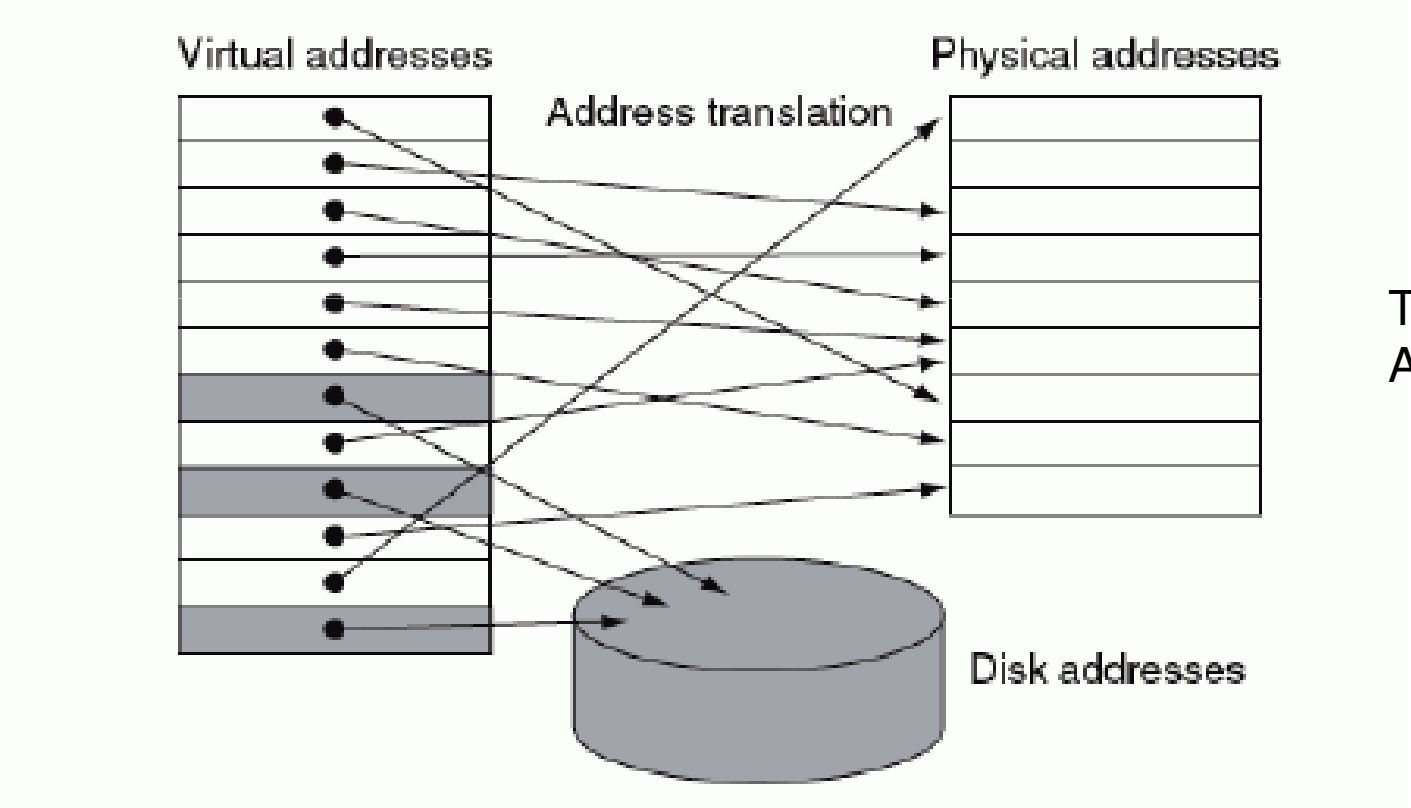
Figure 5.32 Typical ranges of parameters for caches and virtual memory. Virtual memory parameters represent increases of 10–1,000,000 times over cache parameters. Normally first-level caches contain at most 1 MB of data, while physical memory contains 32 MB to 1 TB.

Paginado vs Segmentado

	Page	Segment
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Hard (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

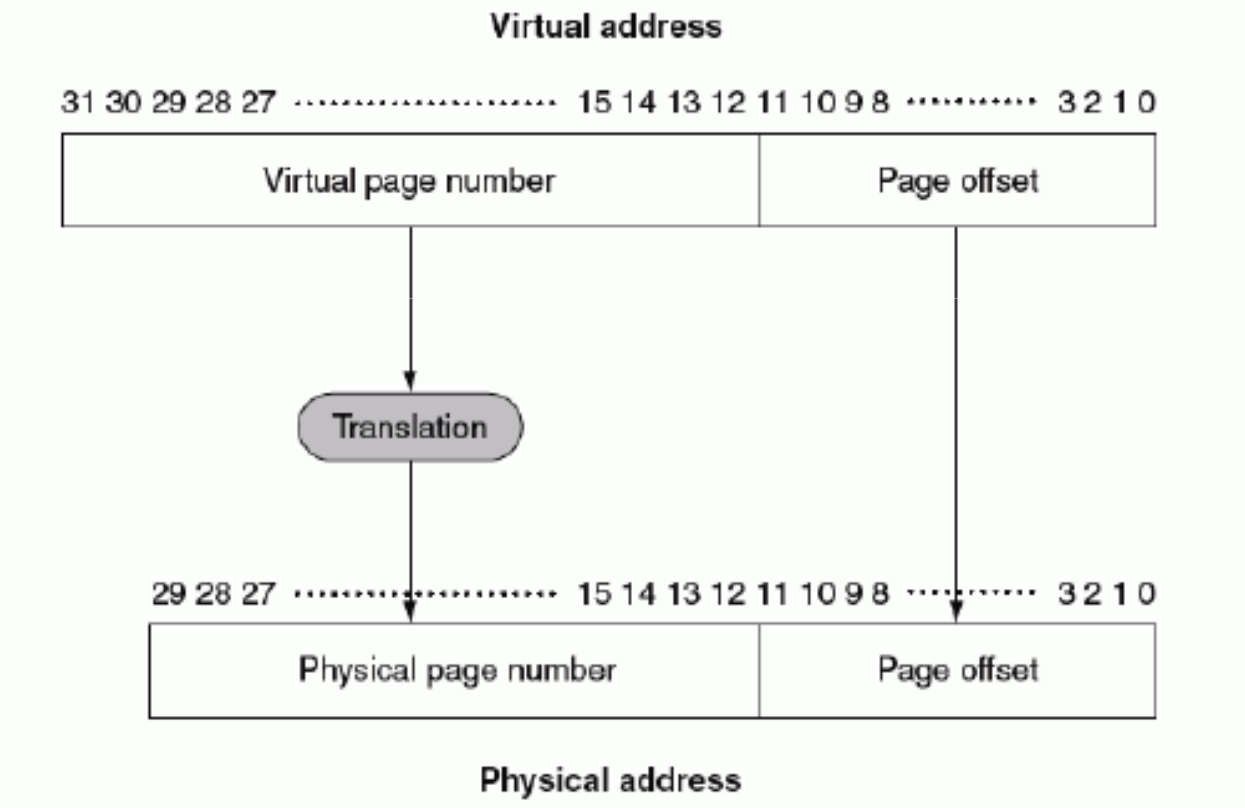
Figure 5.34 Paging versus segmentation. Both can waste memory, depending on the block size and how well the segments fit together in main memory. Programming languages with unrestricted pointers require both the segment and the address to be passed. A hybrid approach, called *paged segments,* shoots for the best of both worlds: Segments are composed of pages, so replacing a block is easy, yet a segment may be treated as a logical unit.

Mapeo de las Páginas Virtuales



Totalmente Asociativo

Proceso de traducción



Traducción: tabla de Traducción de páginas.

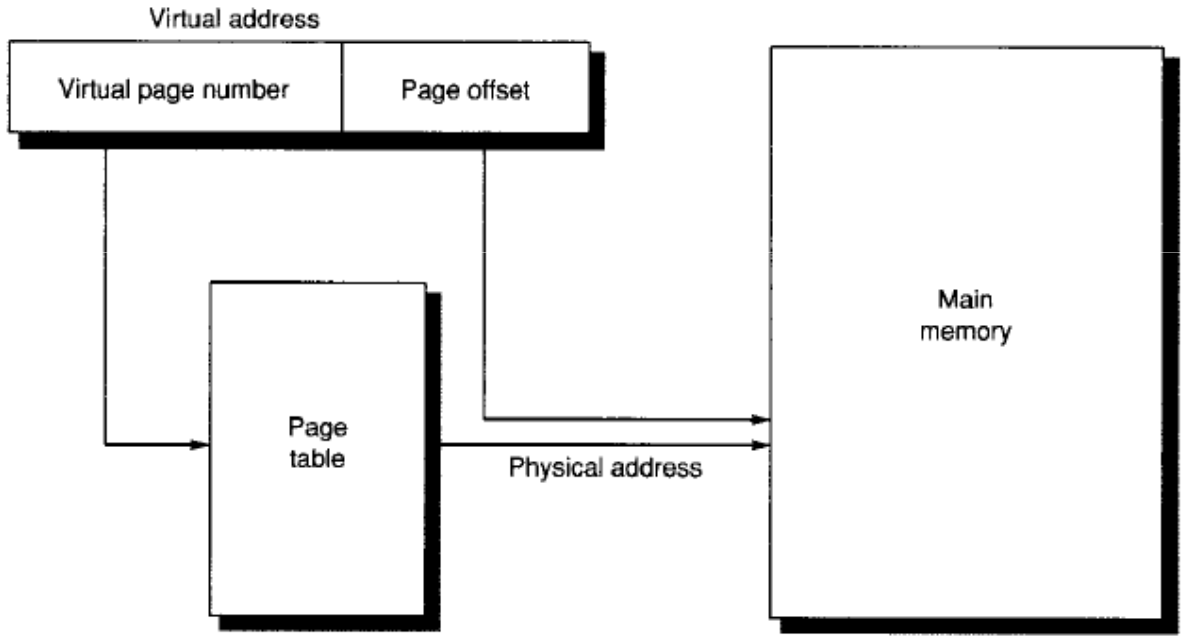
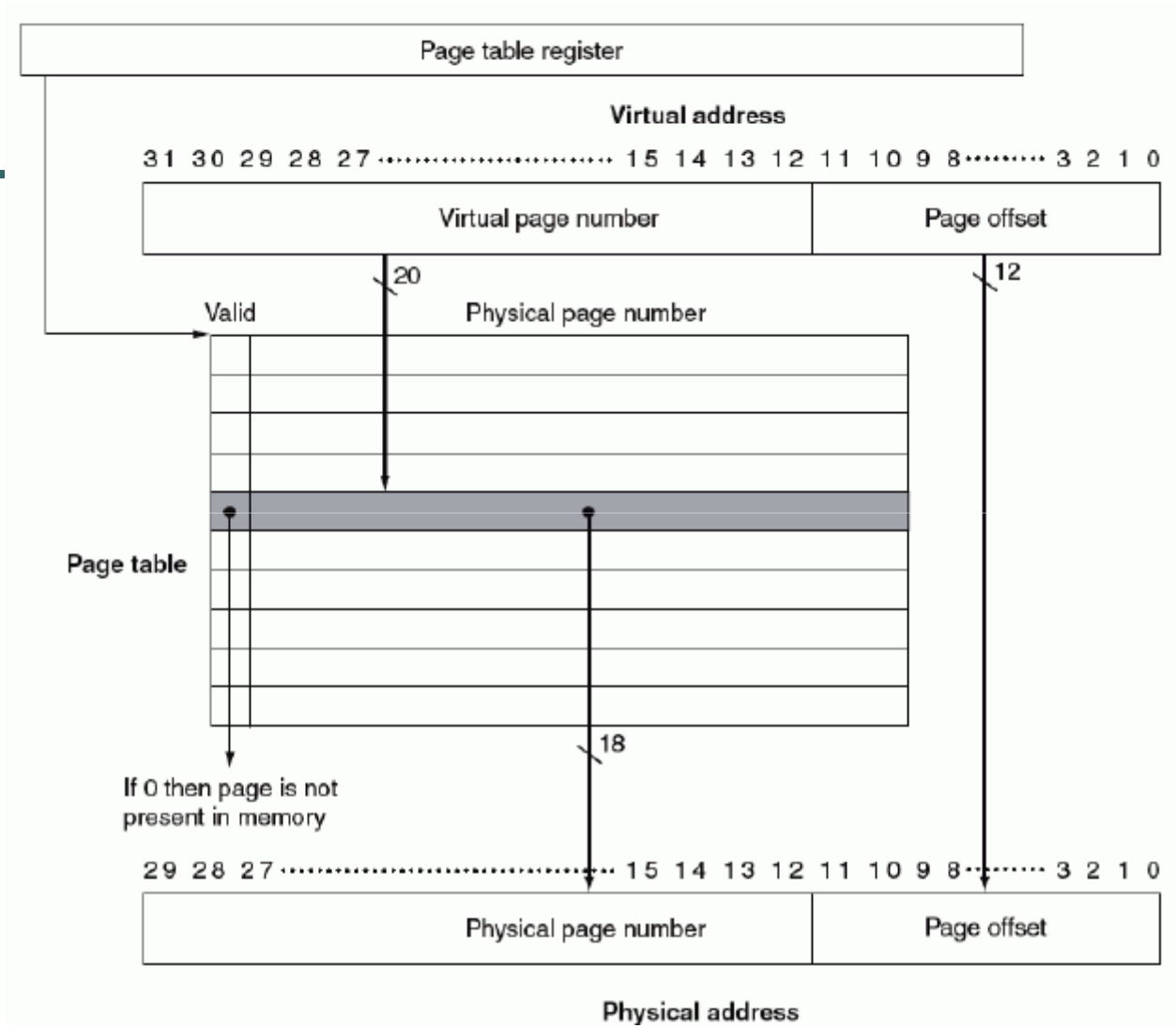
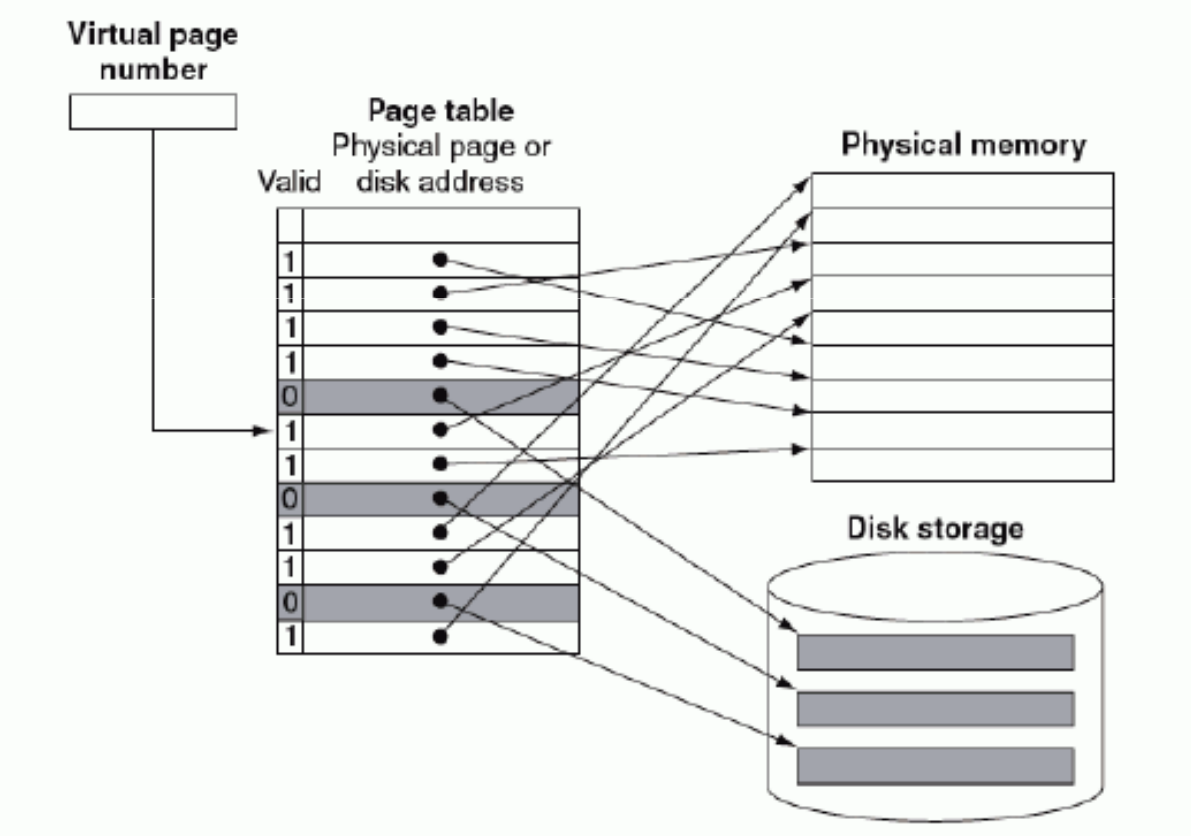


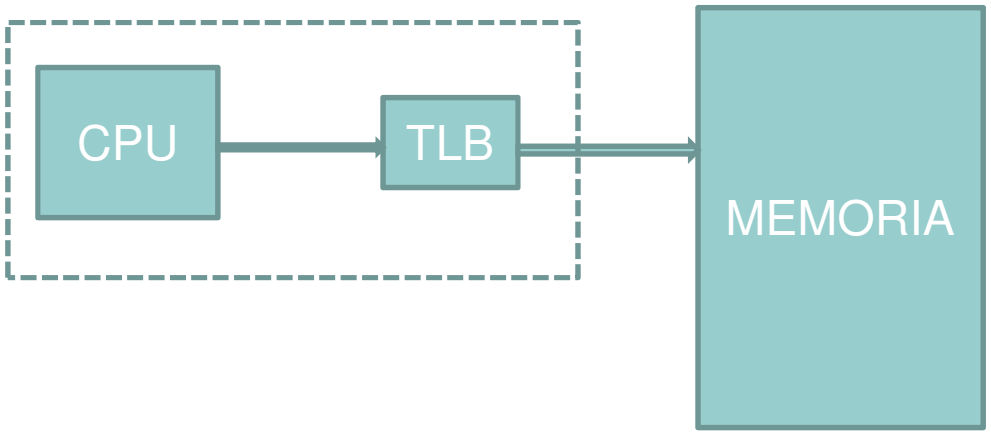
Figure 5.35 The mapping of a virtual address to a physical address via a page table.


Page table register
Virtual address
31 30 29 28 27 ····· 15 14 13 12 11 10 9 8 ····· 3 2 1 0
Virtual page number
Page offset
20
12
Valid
Physical page number
Page table
18
If 0 then page is not present in memory
29 28 27 ····· 15 14 13 12 11 10 9 8 ····· 3 2 1 0
Physical page number
Page offset
Physical address

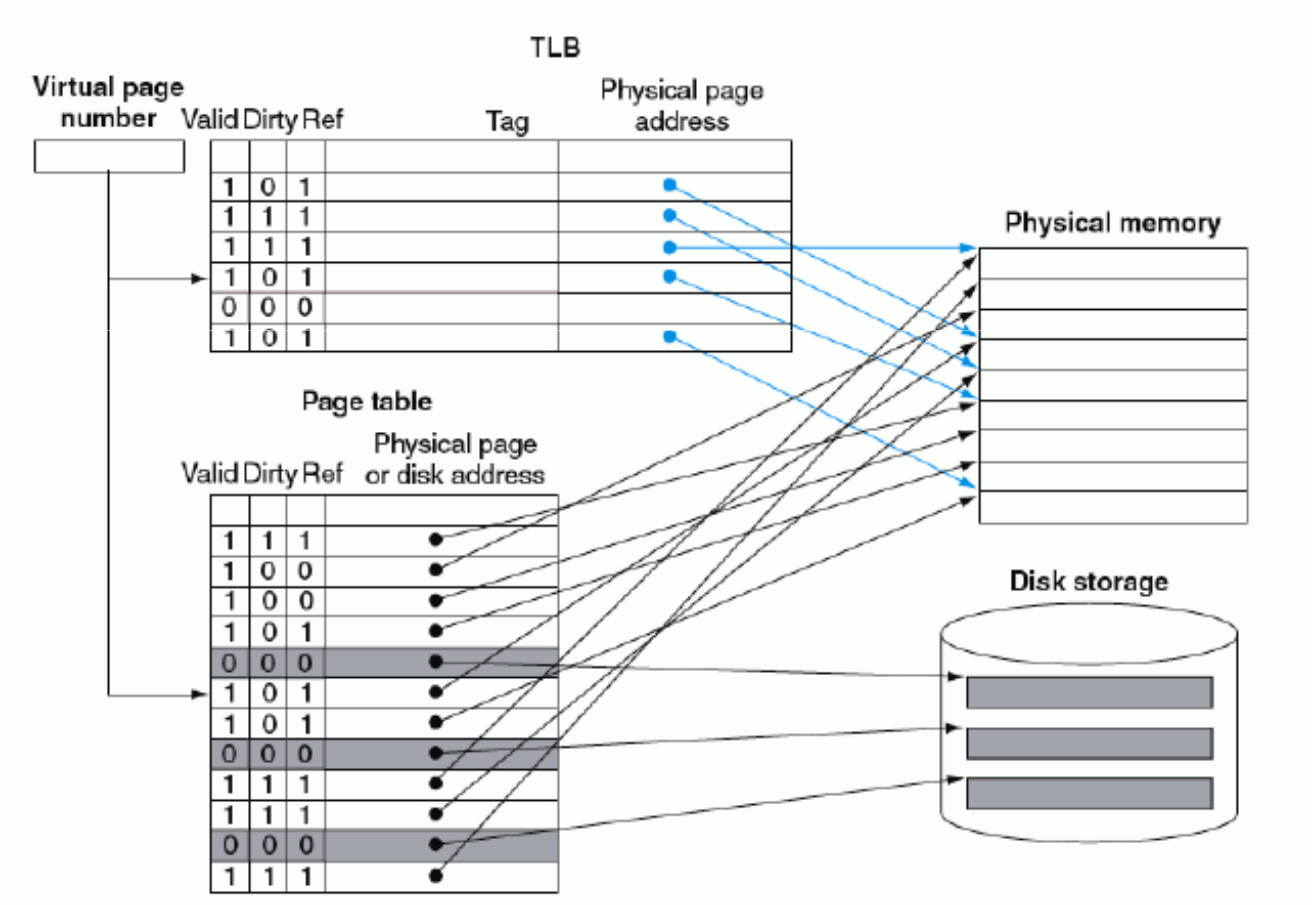
Páginas se alojan en memoria física o disco.



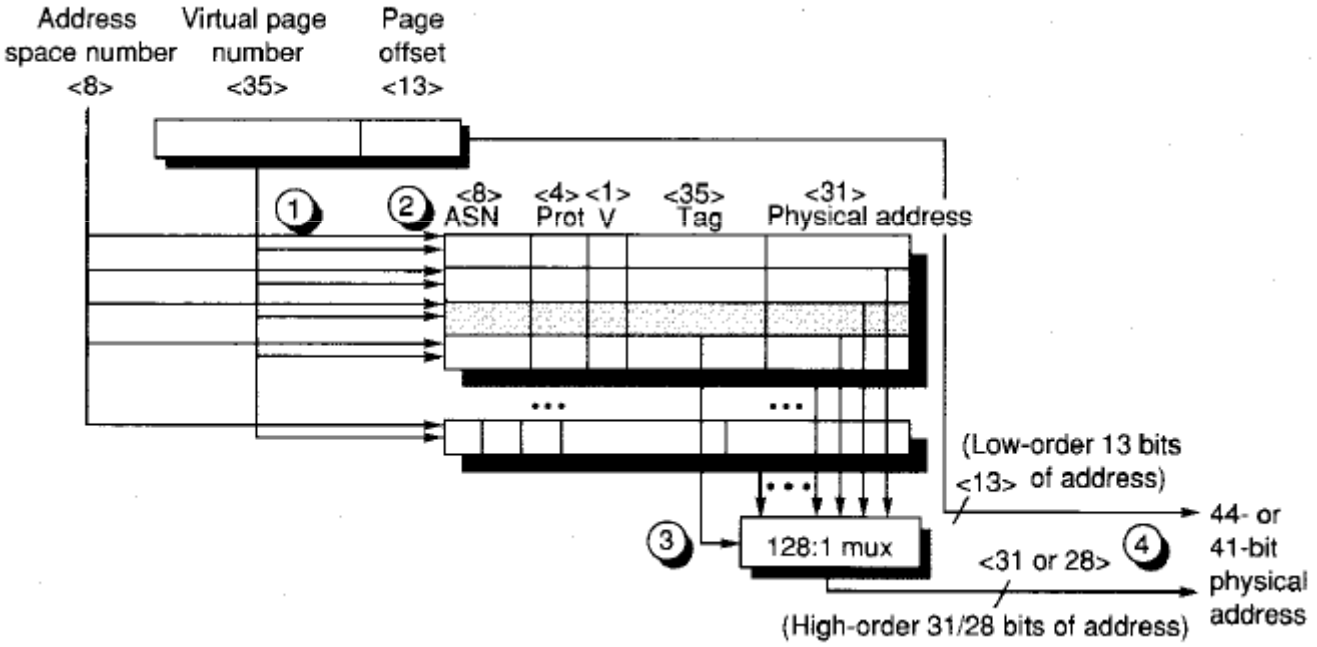
Técnicas de traducción rápida: TLB (Translation Lookaside Buffer) o cache de traducción de páginas.



TLB


TLB
Virtual page number
Valid Dirty Ref
Tag
Physical page address
Physical memory
Page table
Physical page or disk address
Valid Dirty Ref
Disk storage

TLB. Ejemplo

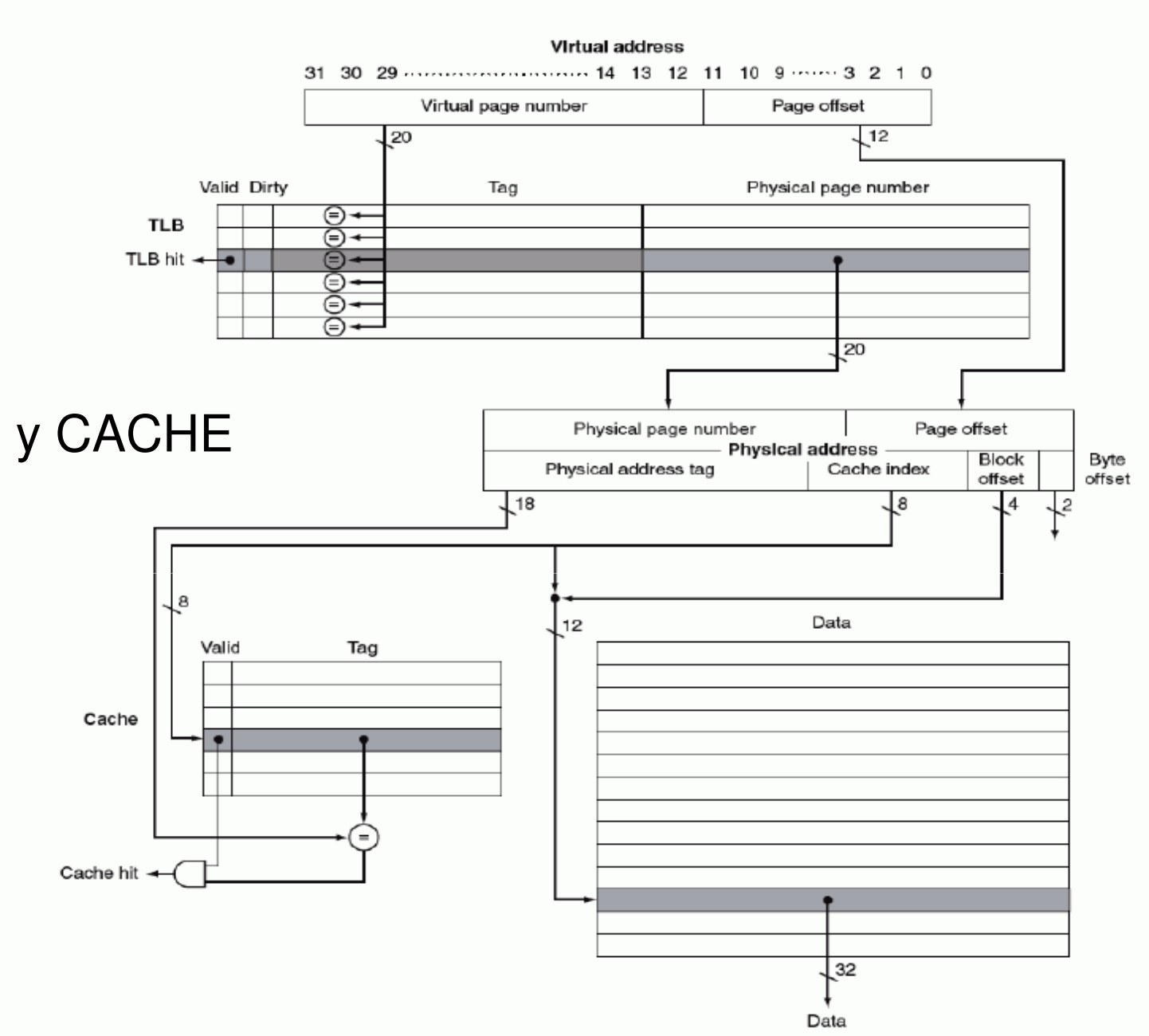

Address space number <8>
Virtual page number <35>
Page offset <13>
1
2
<8> ASN
<4> Prot
<1> V
<35> Tag
<31> Physical address
(Low-order 13 bits of address)
<13>
44- or 41-bit physical address
3
128:1 mux
4
<31 or 28>
(High-order 31/28 bits of address)

Miss en TLB (MIPS)

Register	CP0 register number	Description
EPC	14	Where to restart after exception
Cause	13	Cause of exception
BadVAddr	8	Address that caused exception
Index	0	Location in TLB to be read or written
Random	1	Pseudorandom location in TLB
EntryLo	2	Physical page address and flags
EntryHi	10	Virtual page address
Context	4	Page table address and page number

```
TLBmiss:
  mfc0  $k1,Context    # copy address of PTE into temp $k1
  lw    $k1, 0($k1)    # put PTE into temp $k1
  mtc0  $k1,EntryLo    # put PTE into special register EntryLo
  tlbwr                # put EntryLo into TLB entry at Random
  eret                 # return from TLB miss exception
```

TLB y CACHE


Virtual address
31 30 29 14 13 12 11 10 9 3 2 1 0
Virtual page number
Page offset
20
12
Valid Dirty
Tag
Physical page number
TLB
TLB hit
20
Physical page number
Page offset
Physical address
Physical address tag
Cache index
Block offset
Byte offset
18
8
4
2
8
12
Data
Valid
Tag
Cache
Cache hit
32
Data

Index Virtual – Tag Físico

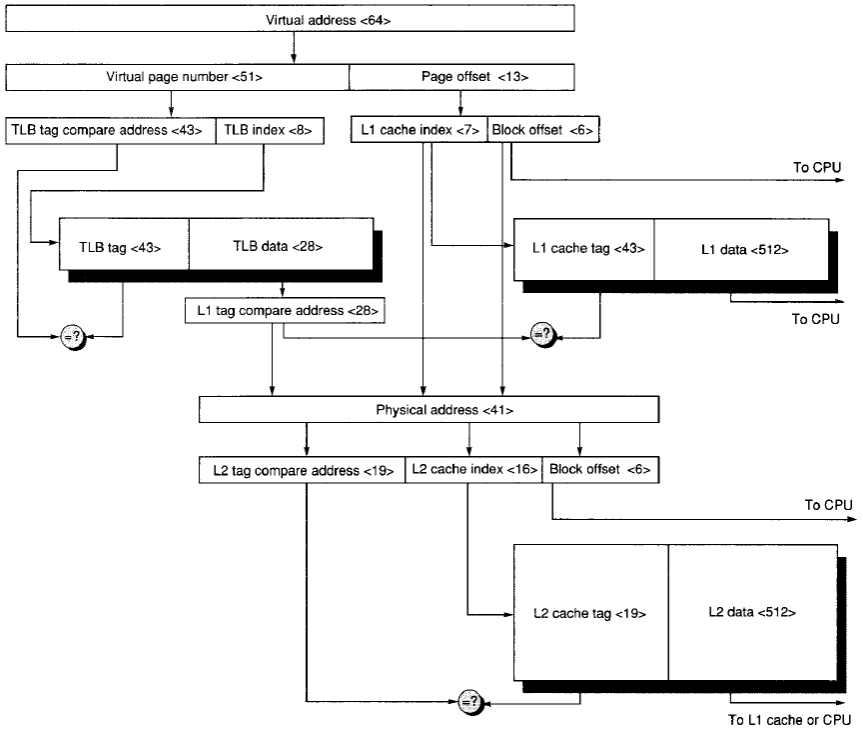
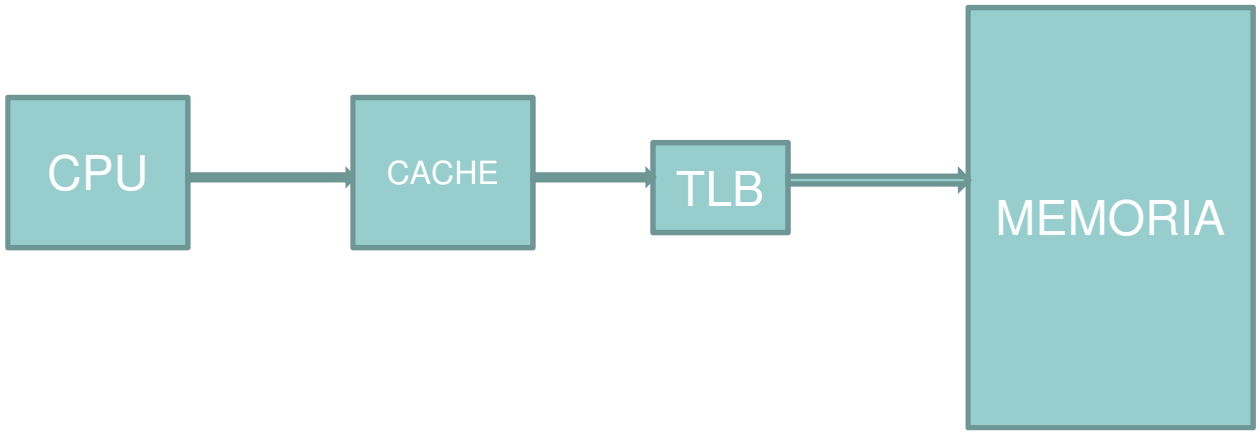


Figure 5.37 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KB, and the L2 cache is a direct-mapped 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache, as in Figure 5.43, is replication of pieces of this figure.

Caches direccionadas por direcciones virtuales



FIN