UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
Facultad de Ingeniería

Seminario: **Multicore Application Programming**

Laboratorio de Intel para la Academia

Proyecto PAPIME PE104911

Elabora: Ariel Ulloa Trejo
Revisión: Ing. Laura Sandoval Montaño
Ing. Andrés Mondragón Contreras

Del autor: Darryl Gove

- Considerado un Senior de Ingeniería de Software, director en el equipo Oracle Solaris Studio.
- Tiene maestría y doctorado en Investigación Operacional por parte de la Universidad de Southampton, UK.
- Ha escrito Solaris Application Programming (Prentice Hall, 2008) y The Developer´s Edge (Sun Microsystems, 2009).
- Escribe regularmente en un blog acerca de programación y optimización en www.darrylgove.com

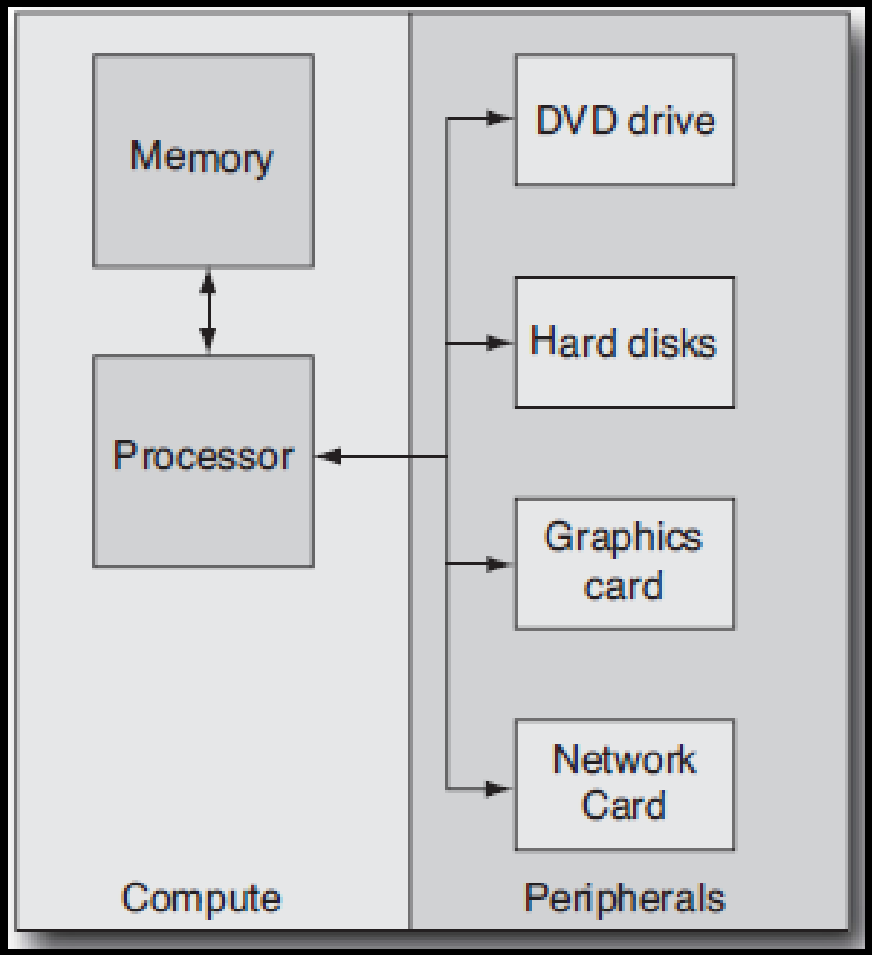
Multicore Application Programming

For Windows, Linux, and Oracle Solaris

1. Hardware, Procesos e Hilos
2. Para obtener rendimiento en la codificación
3. Identificando Oportunidades de Paralelismo
4. Sincronización y Datos Compartidos
5. Utilizando hilos POSIX
6. Hilos en Windows
7. Utilizando Paralelización Automática y OpenMP
8. Sincronización el en código y Recursos Compartidos
9. Ajuste con Procesadores Multicore
10. Otras Tecnologías de Paralelización
11. Observaciones finales

Proyecto PAPIME: PE104911
ARIEL ULLOA TREJO

1. Hardware, Procesos e Hilos



El interior de una Computadora

- Las características del rendimiento de una computadora dependen de la memoria y el procesador.
- En un segundo, el procesador puede transferir gygabytes de datos a la memoria (megabytes a disco duro).
- Para obtener datos, de la memoria el procesador tarda nanosegundos (milisegundos de disco duro).

La motivación para usar procesadores Multicore

- La razón es sencilla: de 1978 a 1987 se mejoró la velocidad de reloj del procesador de 5MHz a 3GHz (cerca de 600 veces). Esta mejora es cada vez más cara, el procesador es más grande, la temperatura aumenta, etc.
- En cambio, si se agrega otro núcleo, se espera que el resultado consista en duplicar el trabajo en el mismo tiempo.

Soportando múltiples hilos en un solo chip

- Core: Parte responsable de la ejecución de instrucciones de un chip.
- Cache: Es el área en el chip que contiene los datos e instrucciones usados recientemente.

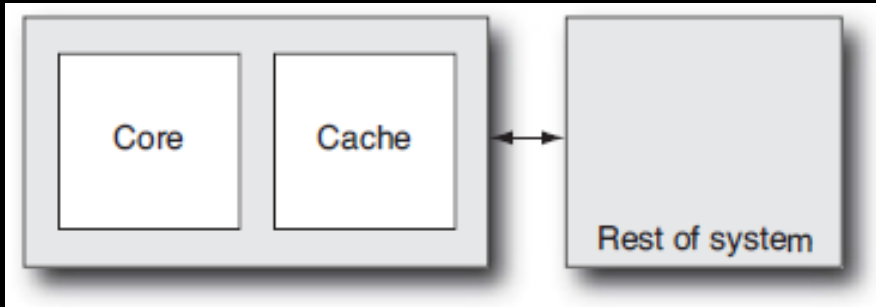


Figure 1.3 Single-core processor

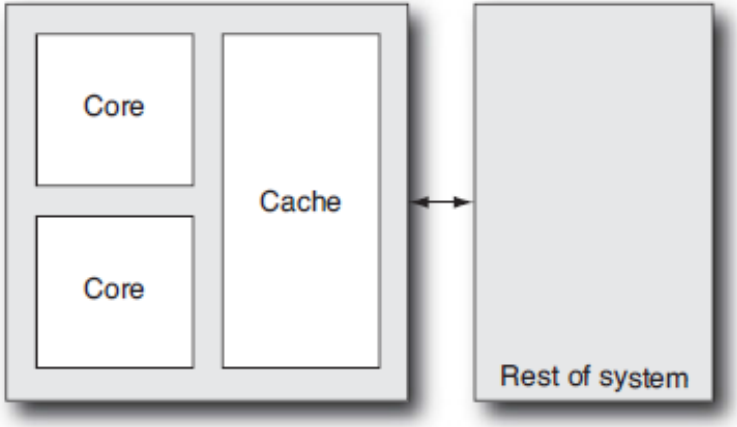


Figure 1.4 Dual-core processor capable of running two simultaneous hardware threads

Soportando múltiples hilos en un solo chip

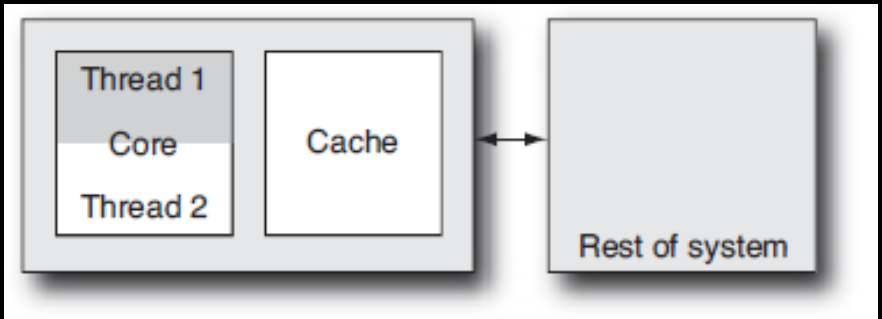


Figure 1.5 Single-core processor with two hardware threads

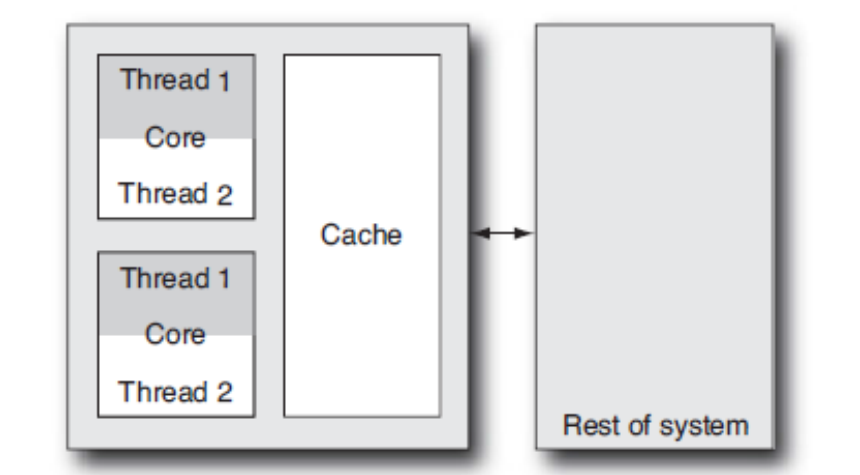


Figure 1.6 Dual-core processor with a total of four hardware threads

(Chip multithreading)

Ventajas:

Puede hacer un switch entre hilos (digamos, 100 ciclos y 100 ciclos).

Se pueden ir a buscar instrucciones simultáneamente.

Cuando no se tiene el dato en cache para un hilo, puede irse a buscar a memoria mientras se ejecuta el otro.

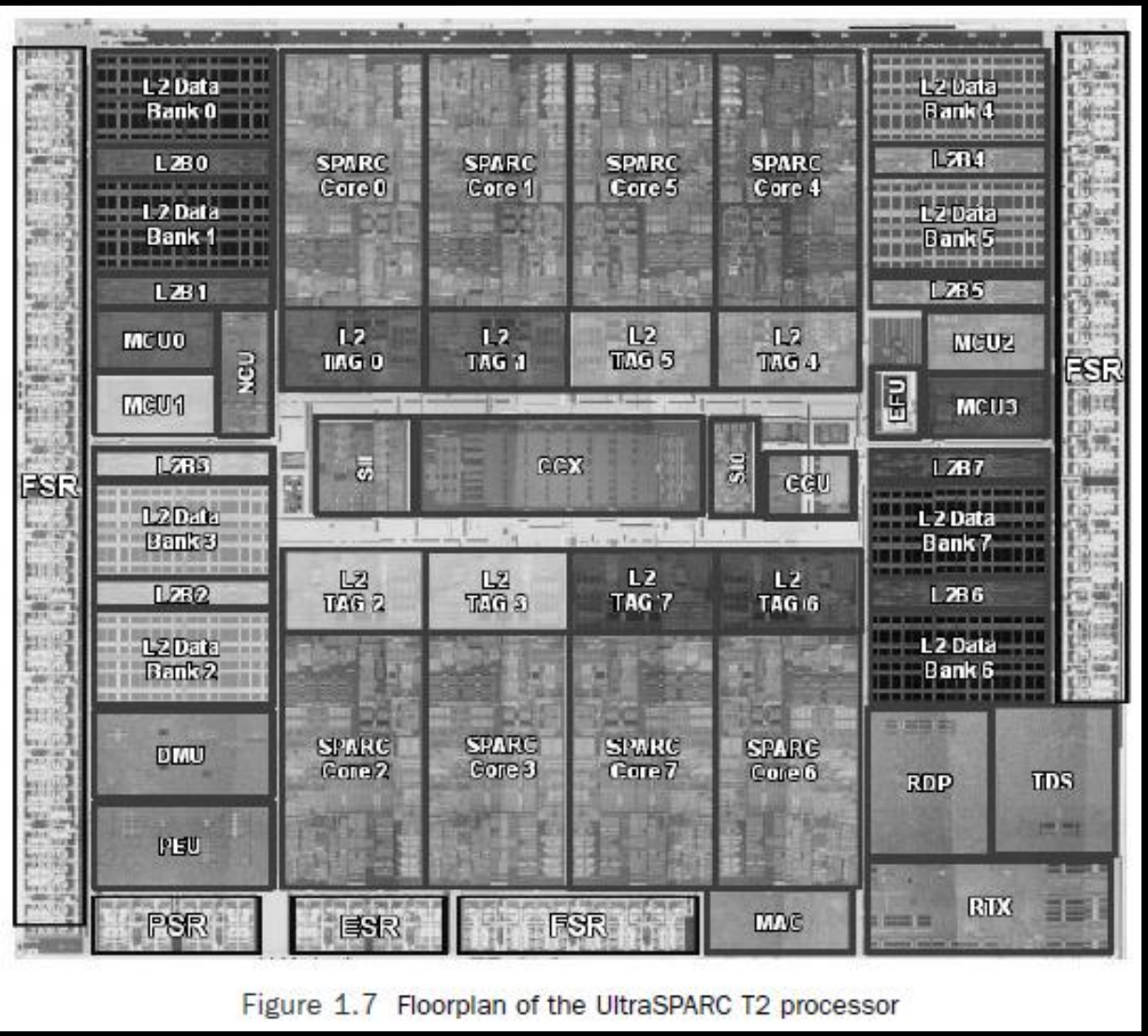


Figure 1.7 Floorplan of the UltraSPARC T2 processor

- Solaris OS: *locality group* (intentará asignar hilos qye no compartan recursos a procesadores virtuales).
- Linux: *affinity* (mantiene a los hilos locales donde se estaban ejecutando)

Diagrama del núcleo de un procesador

Pipeline:

- Las instrucciones se dividen en pasos pequeños: **Buscar instrucción, determinar la instrucción, buscar datos, ejecutar y retirar.**
- ¿Entonces las compañías que venden procesadores nos engañan?

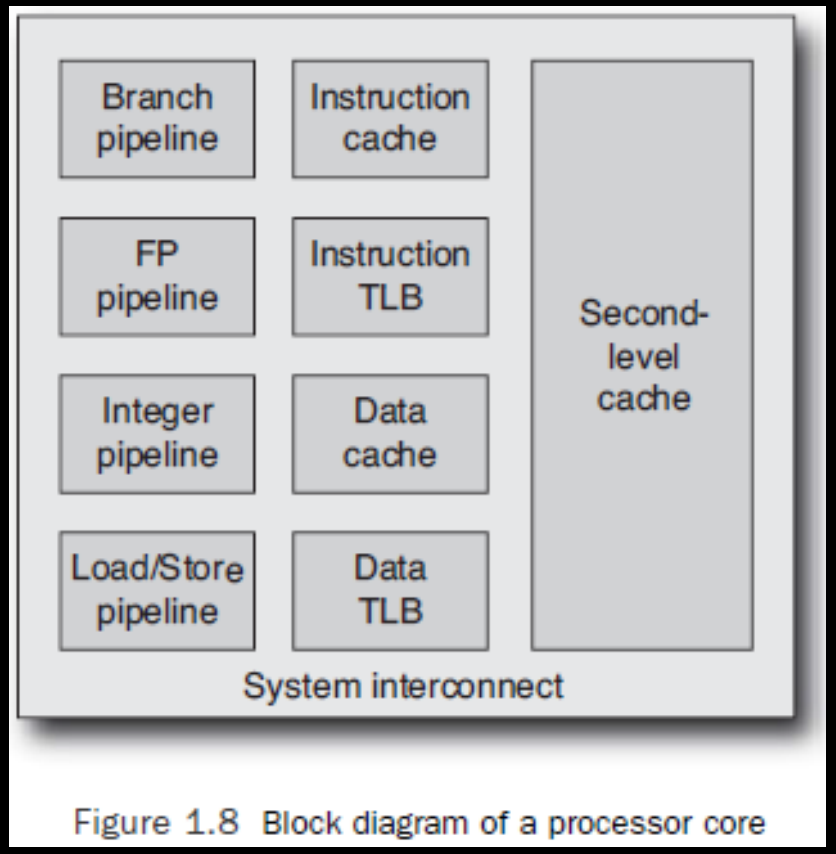


Figure 1.8 Block diagram of a processor core

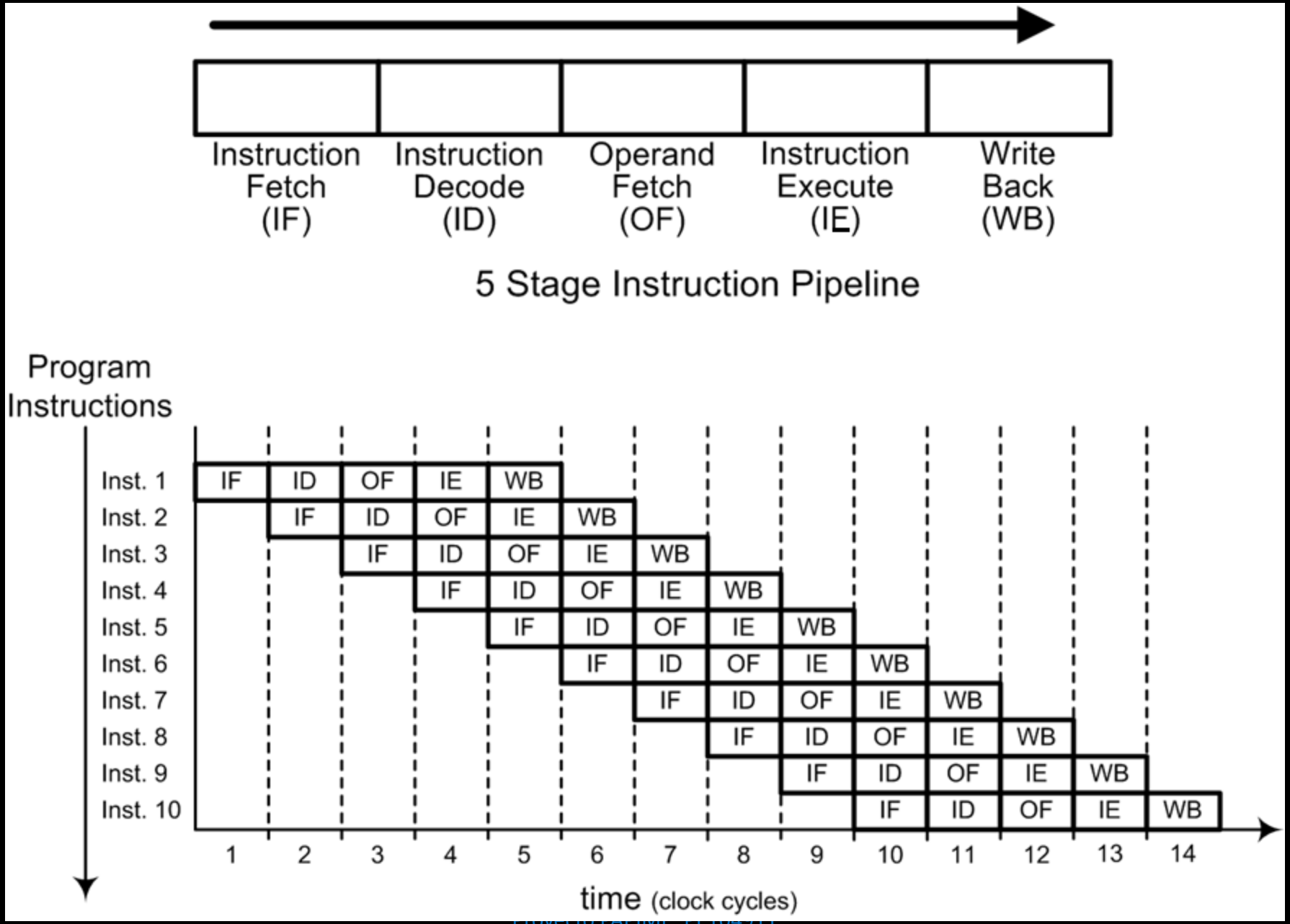

Instruction Fetch (IF)
Instruction Decode (ID)
Operand Fetch (OF)
Instruction Execute (IE)
Write Back (WB)
5 Stage Instruction Pipeline
Program Instructions
Inst. 1 IF ID OF IE WB
Inst. 2 IF ID OF IE WB
Inst. 3 IF ID OF IE WB
Inst. 4 IF ID OF IE WB
Inst. 5 IF ID OF IE WB
Inst. 6 IF ID OF IE WB
Inst. 7 IF ID OF IE WB
Inst. 8 IF ID OF IE WB
Inst. 9 IF ID OF IE WB
Inst. 10 IF ID OF IE WB
1 2 3 4 5 6 7 8 9 10 11 12 13 14
time (clock cycles)

Diagrama del núcleo de un procesador

Cache:

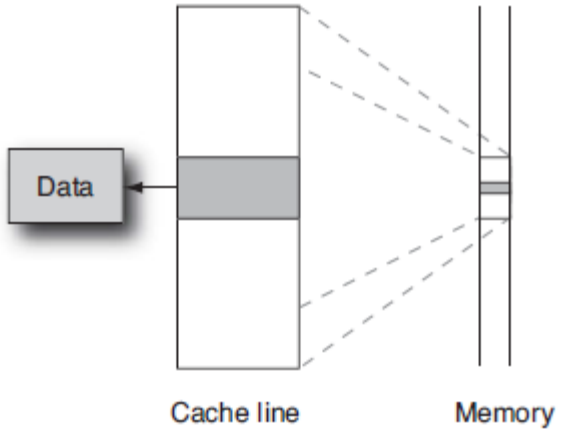


Figure 1.12 Fetching data and surrounding cache line from memory

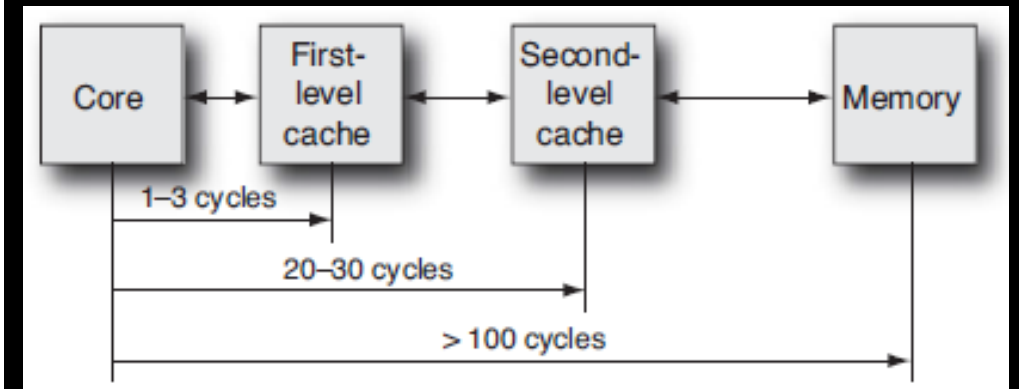


Figure 1.13 Latency to caches and memory

- Cuando el procesador necesita un dato, lo busca en memoria RAM, pero trae consigo otros datos contiguos (generalmente 64 bytes).
- Estos datos se guardan en memoria cache, que trabaja a velocidad del procesador

Usando Memoria Virtual para guardar datos

- Cuando se ejecutan varias aplicaciones, los datos son mantenidos en memoria, pero utiliza la *memoria virtual* en vez de la principal.
- Para lograrlo, los programas en ejecución son divididos en *páginas,* que son almacenados en *marcos de páginas*, ya sea en memoria principal (usados recientemente) o virtual (que no han tenido lecturas/escrituras durante cierto tiempo).
- **Desventajas:**
  - Es basta
- **Ventajas:**
  - Varias a AM limitada.
  - Abrir a zan en cierto momen
  - La mism aciones.
  - Casi es una necesidad para poder ejecutar varios hilos.

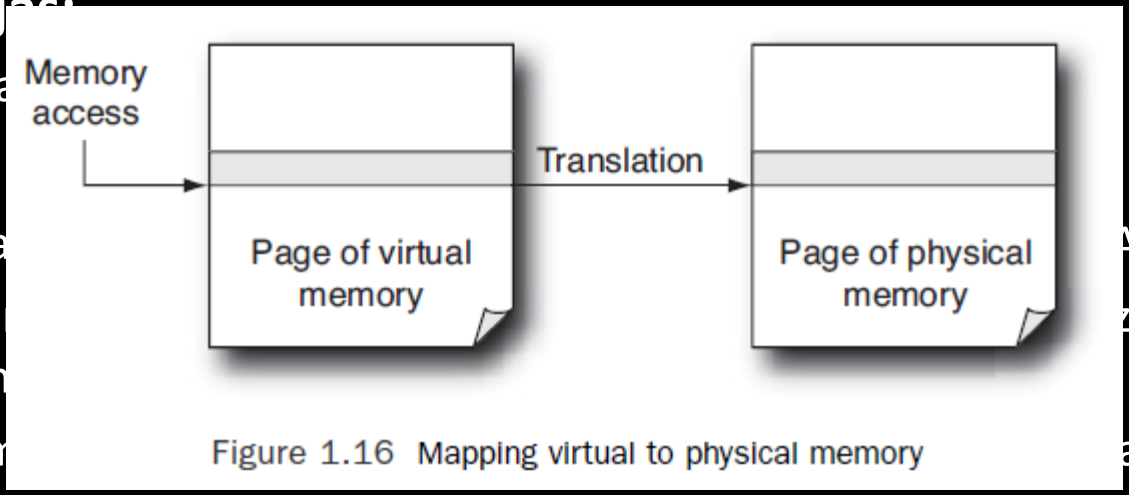


Figure 1.16 Mapping virtual to physical memory

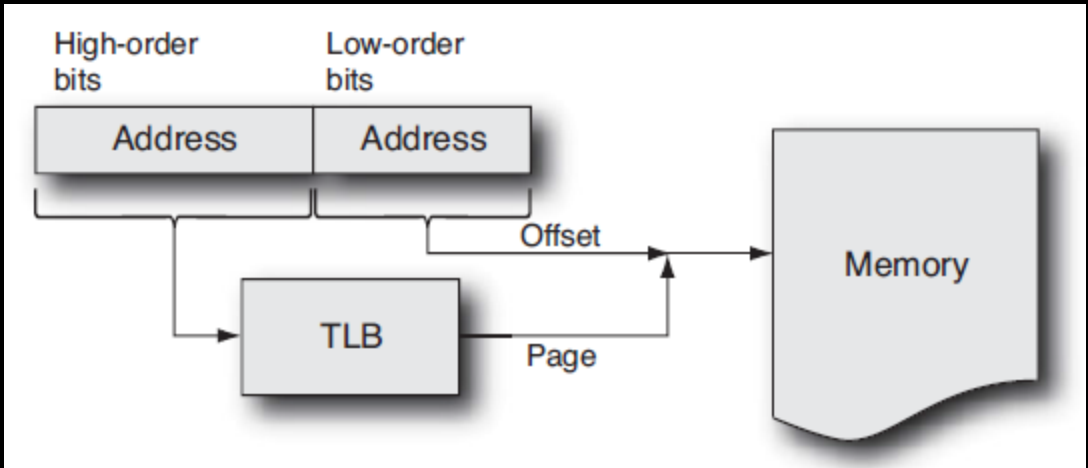


Figure 1.17 Virtual to physical memory address translation

- Para conocer las direcciones físicas, el procesador tiene una parte llamada *translation look-aside buffer*, tanto para traducir direcciones de instrucciones (ITLB) como direcciones de datos (DTLB); cuando las direcciones que el procesador necesita no están en estos buffers, recurre a la *tabla de páginas,* pero esto afecta en el rendimiento del programa.

Características de Sistemas Multiprocesos

- El acceso a memoria se vuelve más complejo en sistemas con varios procesadores: para que un código se ejecute correctamente, debe h); para lograrl escribir en una n más recient
- Proble
  - Dise
  - Late

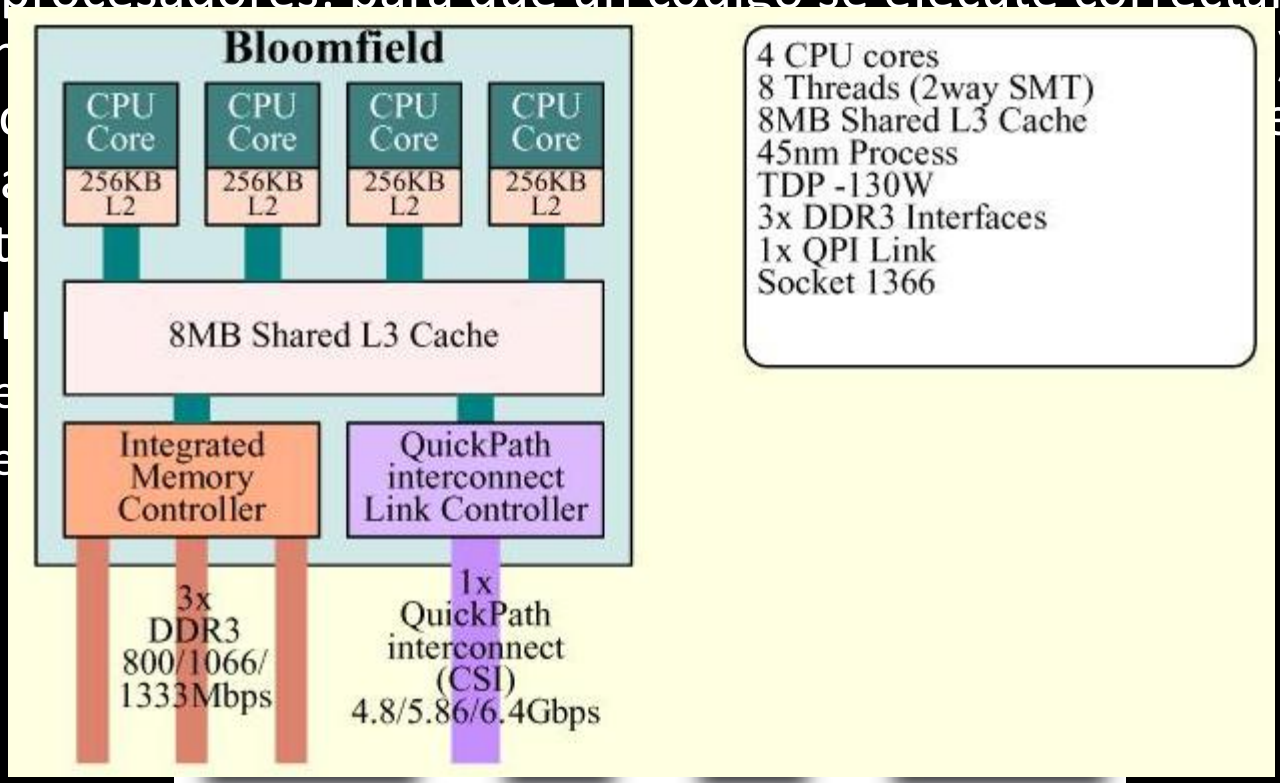


Figure 1.18 Two alternative memory configurations

Cómo la latencia impacta en el desempeño

- En sistemas con un procesador, puede haber latencia de 100ns; con dos, puede duplicarse el tiempo, y si hay más, puede volverse un problema.
- Posibles soluciones:
  - Chip multithreading: Cuando un hilo no cuenta con el dato que necesita, otro hilo puede entrar en ejecución en ese núcleo; no se soluciona la latencia, pero el núcleo sigue trabajando.
  - Out-of-order execution (ejecución fuera de orden): mientras se espera por un dato, el procesador puede ejecutar instrucciones subsecuentes (posteriores) que no lo requieran.
    - Problema:
    - Se requiere de un procesador que sea muy “inteligente”.

Rendimiento de código de 32 versus 64 bits

64 bits

- Se puede tener una dirección de 16 exabytes (EB).
- Cuando se llama a una función, los parámetros se quedan en los registros, evitando el almacenamiento en la pila.
- Los apuntadores (por el cambio en la arquitectura) miden 8 bytes.

32 bits

- Puede direccionar un máximo de 4 GB.
- Cada vez que se llama a una función, los parámetros deben de ser guardados en la pila; acto seguido, obtenerlos de nuevo y pasarlos a registros.
- Loa apuntadores miden 4 bytes.

Listing 1.5 **Data Structure Containing an Array of Pointers to Integers**

```
struct s
{
  int *ptr[8];
};
```

- Cuando se compila:

- 32 bits:
- 8 * 4 bytes = 32 bytes
- Cuando se busca a una de estas estructuras, la siguiente también es traída a caché.

- 64 bits:
- 8 * 8 bytes = 64 bytes
- Sólo se puede traer una estructura.

Diferencias entre procesos e Hilos

Proceso:

- Es un programa en ejecución.
- Consta de un estado: dirección de la instrucción en ejecución, los valores en la memoria y cualquier otro valor único que define al proceso en cualquier momento (privado al proceso).

Hilo:

- Es una tarea.
- Cada proceso puede correr varios hilos.
- También tiene un estado, pero sólo conformado por los valores en sus registros,
- Los hilos comparten algunos recursos.

Proyecto PAPIME: PE104911
ARIEL ULLOA TREJO

Proceso:

- Ventajas:
  - Son independientes (si muere uno, no afecta a los demás).
- Desventajas:
  - Cada proceso requiere su TLB (lo que aumenta la probabilidad de fallo en caché).
  - Compartir datos requiere control explícito (lo que puede ser costoso).

Hilo:

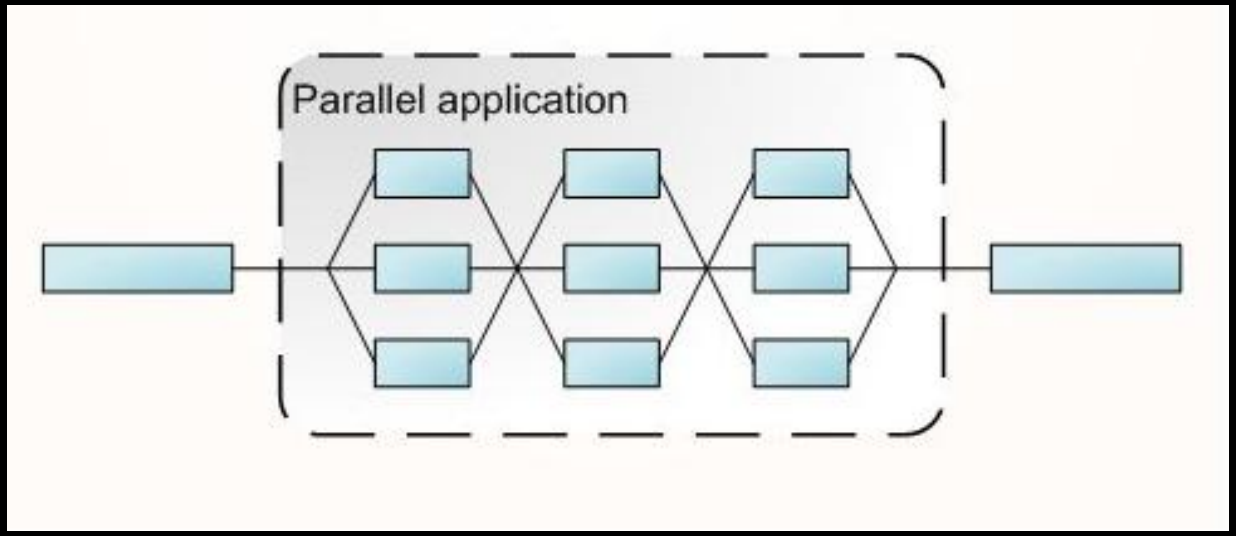
- Ventajas:
  - Es fácil compartir datos.
  - Todos los hilos comparten TLB (disminuye la probabilidad de fallo en caché).
- Desventajas:
  - Puede que si un hilo falla, toda la aplicación falle.

Proyecto PAPIME: PE104911
ARIEL ULLOA TREJO

2. Para obtener rendimiento en la codificación

Rendimiento:

- Es más importante cuántas tareas se completan en cierto tiempo a cuánto dura cada una



Complejidad Algorítmica

- Es una medida de la cantidad de cálculos que un programa llevará a cabo en un algoritmo.
- Eficiencia y estimado del conteo de operaciones.
- Este concepto se entenderá mejor con algunos ejemplos:

Suma de los primeros N números:

Listing 2.1 Sum of the First N Numbers

```
void sum(int N)
{
  int total=0;
  for (int i=1; i<=N; i++)
  {
    total += i;
  }
  printf( "Sum of first %i integers is %i\n", N, total );
}
```

Complejidad: k*N → O(N)

k = tiempo de adición

N = tamaño de la entrada

Listing 2.2 Sum of the First N Factorials

```
int factorial(int F)
{
   int f = 1;
   for (int i=1; i<=F; i++)
   {
      f = f*i;
   }
   return f;
}

void fsum(int N)
{
   int total = 0;
   for (int i=1; i<N; i++)
   {
     total+=factorial(i);
   }
}
```

Suma de los primeros N factoriales

- Dos ciclos anidados.
  - El externo hará N iteraciones.
  - El interno hará un promedio de N/2 iteraciones.

Por lo tanto, habrá en promedio N*N/2 multiplicaciones y N sumas.

La complejidad es tomada sólo con los factores dominantes, que son N*N.

Por lo tanto: $O(N^2)$

Ejemplos terrenales: ordenamiento

- Ordena un arreglo de menor a mayor.
- Si el elemento i es mayor al i+1, los cambia entre sí.

- Se necesitan N comparaciones para poner a cada elemento en su lugar.
- Hay N elementos, por lo que tomará N*N comparaciones en ordenar completamente la lista.

- Por lo tanto: $O(N^2)$

Bubble sort:

Listing 2.3 Implementation of Bubble Sort

```
void bubble_sort(int*array, int N)
{
  int sorted =0;
  while ( !sorted )
  {
    sorted=1;
    for ( int i=0; i < N-1; i++ )
    {
      if (array[i] > array[i+1])
      {
        int temp    = array[i+1];
        array[i+1] = array[i];
        array[i]    = temp;
        sorted=0;
      }
    }
  }
}
```

Quicksort:

Listing 2.4 **Implementation of Quicksort**

```
void quick_sort(int * array, int lower, int upper)
{
  int tmp;
  int mid    = (upper+lower)/2;
  int pivot  = array[mid];
  int tlower = lower
  int tupper = upper;
  while (tlower <= tupper)
  {
    while ( array[tlower] < pivot ) { tlower++; }
    while ( array[tupper] > pivot ) { tupper--; }
    if ( tlower <= tupper )
    {
      tmp           = array[tlower];
      array[tlower] = array[tupper];
      array[tupper] = tmp;
      tupper--;
      tlower++;
    }
  }
  if (lower<tupper) { quick_sort(array,  lower, tupper); }
  if (tlower<upper) { quick_sort(array, tlower,  upper); }
}
```

- Su nombre nos indica que es un algoritmo rápido.
- Consiste en dividir la lista en dos: una lista cuyos elementos son menores a un “pivote” y otra en la que son mayores.
- Se repite haciendo listas más pequeñas (recursivo).

- Razonamiento de complejidad:
  - Para una entrada N, tomará $\log_2(N)$ divisiones para tener N listas.
  - Cada vez que la lista se divide, hay dos ordenaciones (uno para la lista menor y otro para la mayor). Entonces, para cada lista habrá iteración de N elementos

    por lo tanto: $O(N*\log_2(N))$

¿Por qué la complejidad es importante?

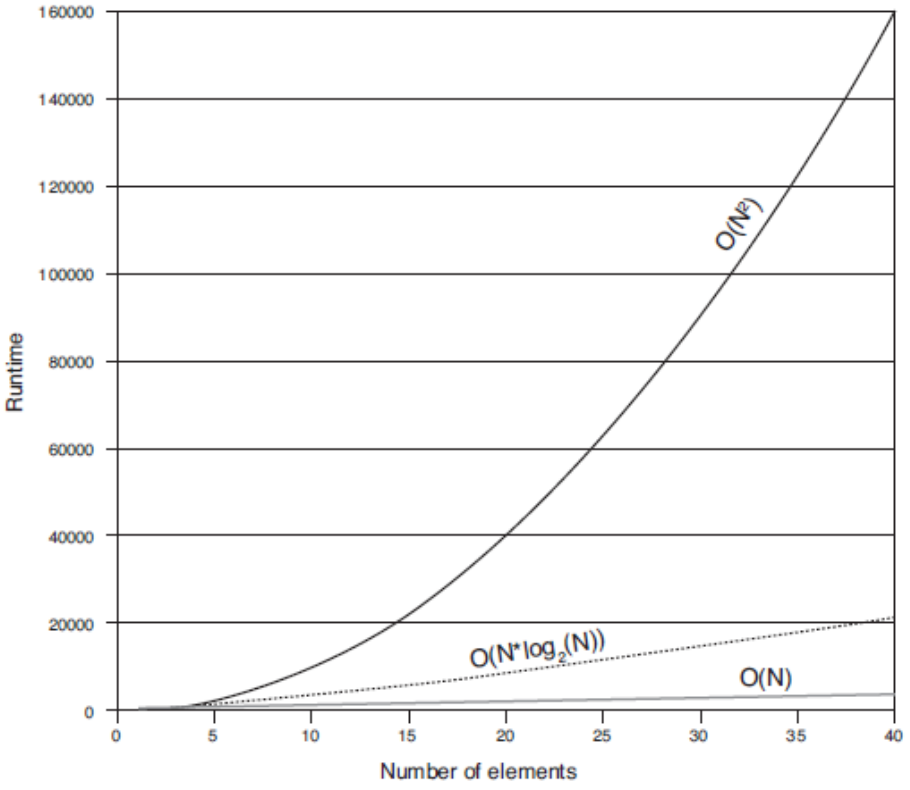


Figure 2.1 Different orders of algorithmic complexity

******Se ha considerado un procesador de 1 GHz (cada instrucción tarda 100 ns en ser ejecutada).**

Table 2.1 **Execution Duration at Different Algorithm Complexities**

Elements	O(1)	O(N)	O(N $\log_2$ N)	O(N^2)
1	100ns	100ns	100ns	100ns
10	100ns	1,000ns	3,322ns	10,000ns
100	100ns	10,000ns	66,439ns	1,000,000ns
1,000	100ns	100,000ns	996,578ns	100,000,000ns
10,000	100ns	1,000,000ns	13,287,712ns	10,000,000,000ns

El impacto de los datos en el Desempeño

- Tres atributos de la estructura de datos, de una aplicación pueden ser considerados como "estructura":
  1. Cómo el código es distribuido entre los códigos fuente.
  2. Cómo los códigos fuente son combinados.
  3. Cómo están organizados los datos en la aplicación.

Cuando un programa necesita un dato, lo trae de memoria a caché. Al hacerlo residente en esta, el acceso es mucho más rápido.

– Menor costo.
– Menor tiempo de espera.
– Trae datos adyacentes ("predice" que quiza los utilice).

Listing 2.5 **Accessor Functions**

```
#include <stdio.h>

int a;

void setvalue( int v ) { a = v; }

int  getvalue() { return a; }

void main()
{
  setvalue( 3 );
  printf( "The value of a is %i\n", getvalue() );
}
```

Listing 2.6 **Pseudosource Code After Inlining Optimization**

```
#include <stdio.h>

int a;

void main()
{
  a = 3;
  printf( "The value of a is %i\n", a );
}
```

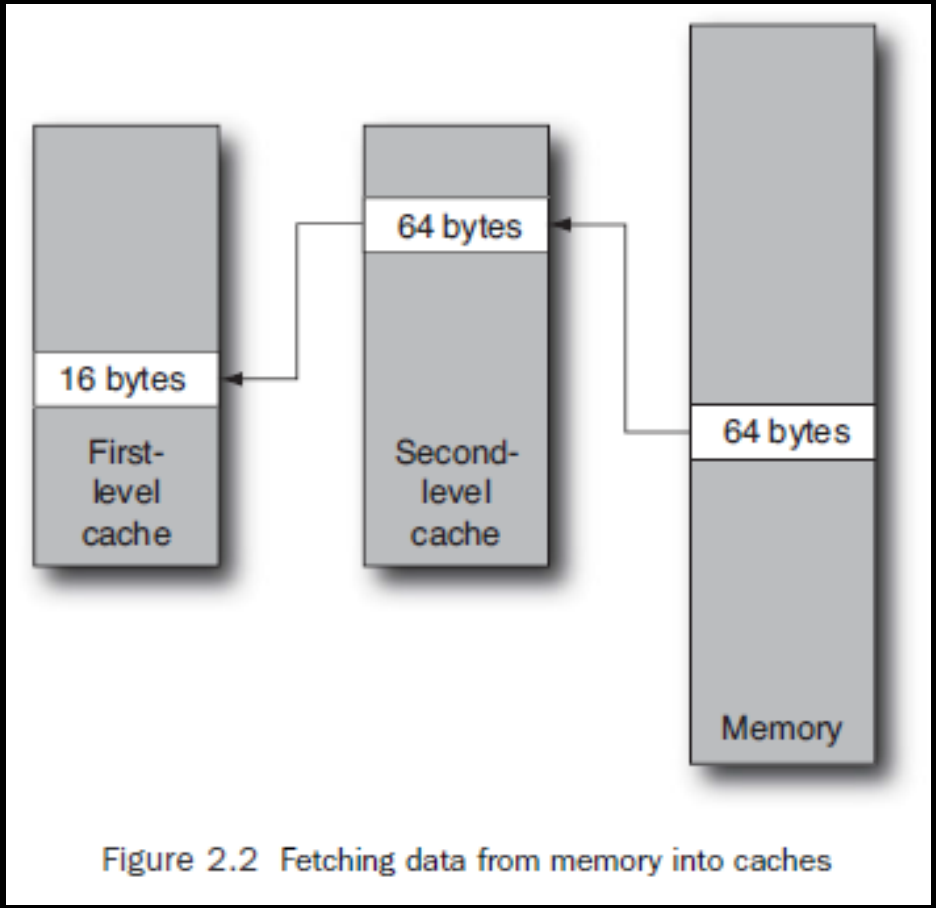


Figure 2.2 Fetching data from memory into caches

- Cuando ocurre un fallo en la caché (el dato requerido no está), se busca en memoria y es instalado en el segundo nivel de caché; esto es muy tardado (cientos de ciclos de reloj), y si el fallo es constante se observarán los efectos de la latencia. El procesador tiene algunas técnicas que la reducen:
  - Ejecución fuera de orden: El procesador "revisa" las próximas instrucciones y busca todos los datos necesarios de forma simultánea.
  - Pre-búsqueda Hardware: Parte del procesador "predice" qué datos se utilizarán y los pone en el segundo nivel de caché antes de ser solicitados (sólo funciona con datos que tienen cierto patrón: arreglos).
  - Pre-búsqueda Software: Revisa las próximas instrucciones y trae los datos correctos a caché (funciona aunque no sea memoria continua), pero consume tiempo del procesador.

Por ejemplo, una mala práctica en C

Listing 2.29 **Noncontiguous Memory Access Pattern**

```
{
   double ** array;
   double total=0;
   …
   for (int i=0; i<cols; i++)
     for (int j=0; j<rows; j++)
       total += array[j][i];
   …
}
```

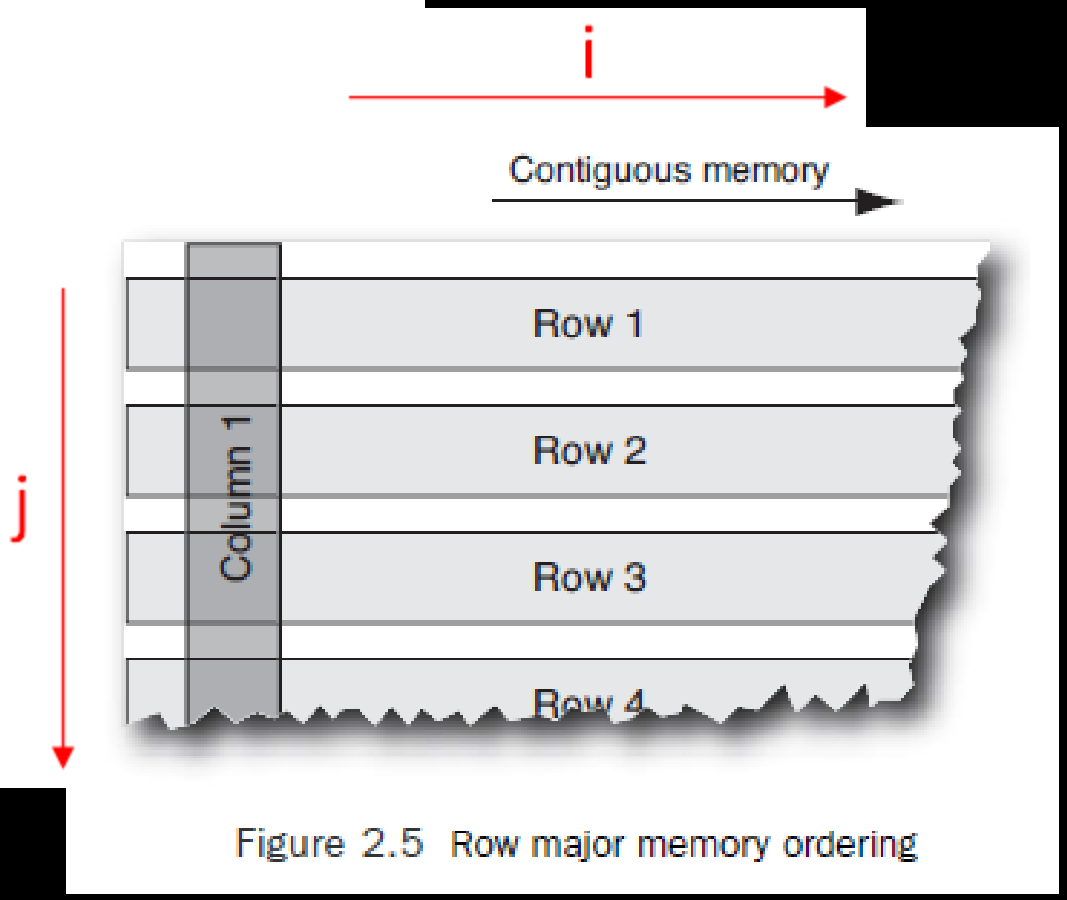


Figure 2.5 Row major memory ordering

Ejemplos de diccionarios

Lista ligada

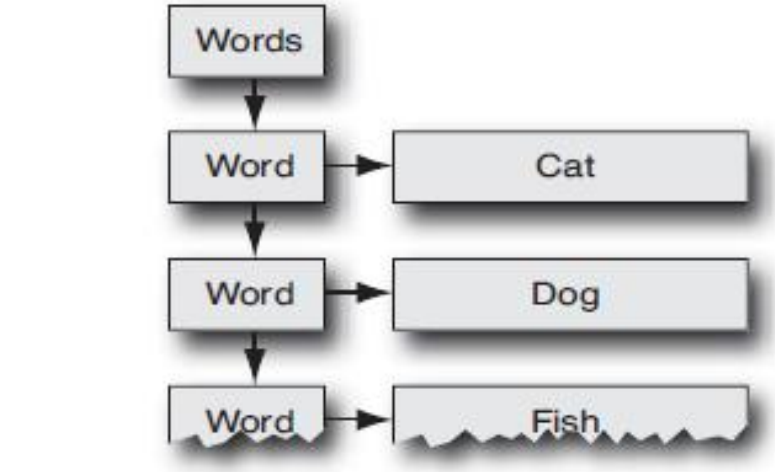


Figure 2.6 Using a linked list to hold an ordered list of words

Arreglos:

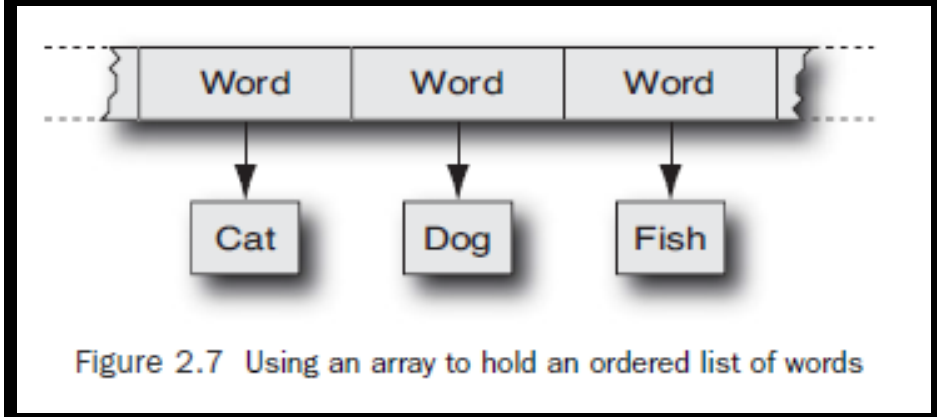


Figure 2.7 Using an array to hold an ordered list of words

¿Cuál es mejor?

Consideraciones:

- **¿Cuánto te tardas en implementarlo?**
- **¿Se tendrá una entrada lo suficientemente grande?**
- **¿Vale la pena?**