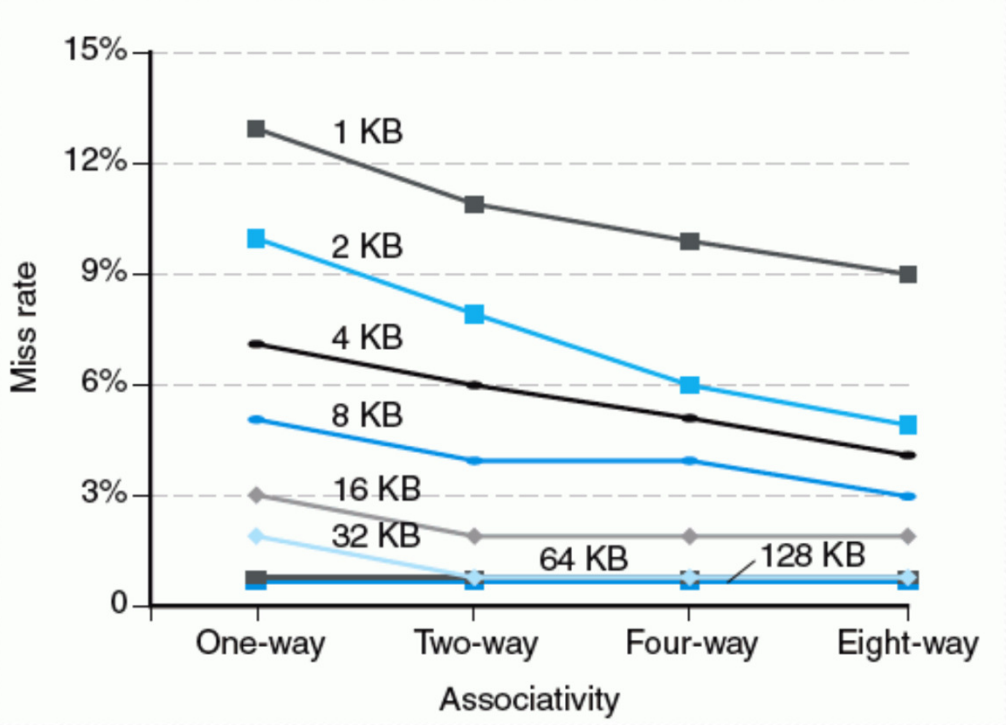
Diseño del caché

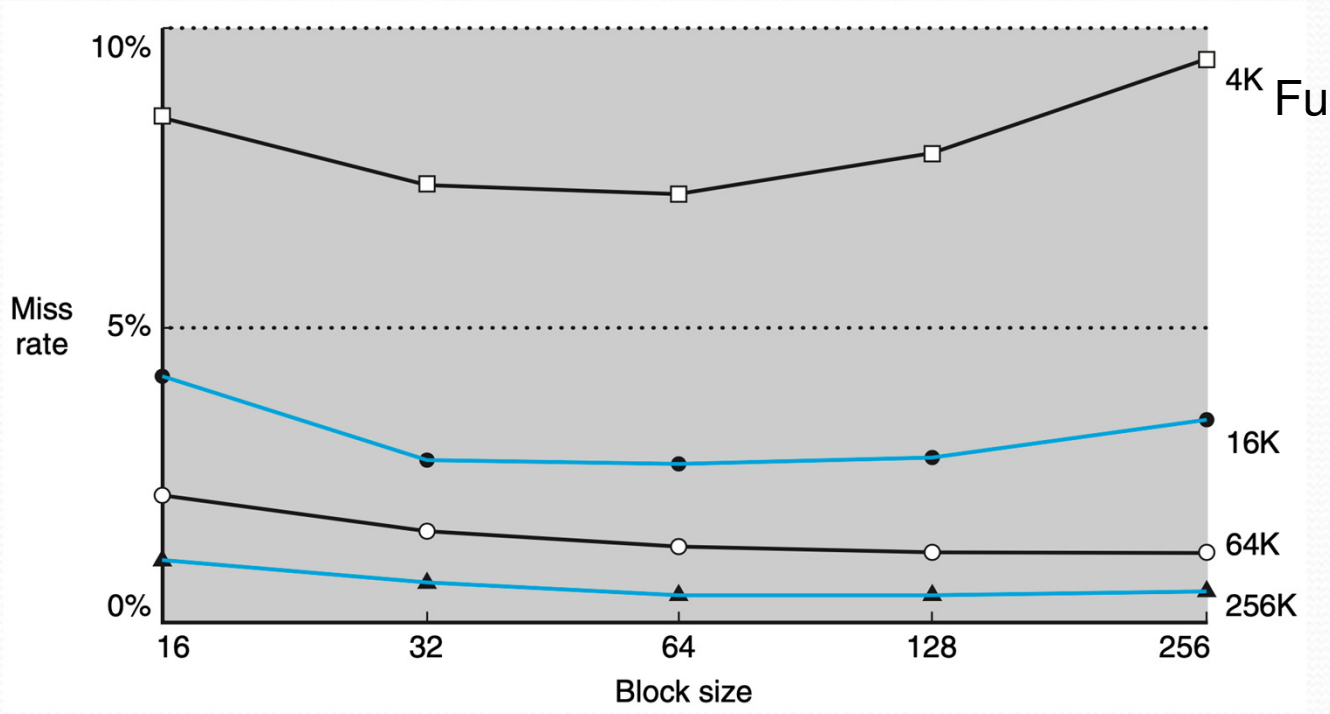
Comportamiento de las fallas de caché

- Las fallas de caché disminuyen conforme se incrementa la asociatividad.



Comportamiento de las fallas de caché

- Las fallas de caché aumentan si los bloques son grandes respecto al tamaño del caché.



Fuente: COD5, p. 391

Origen de las fallas

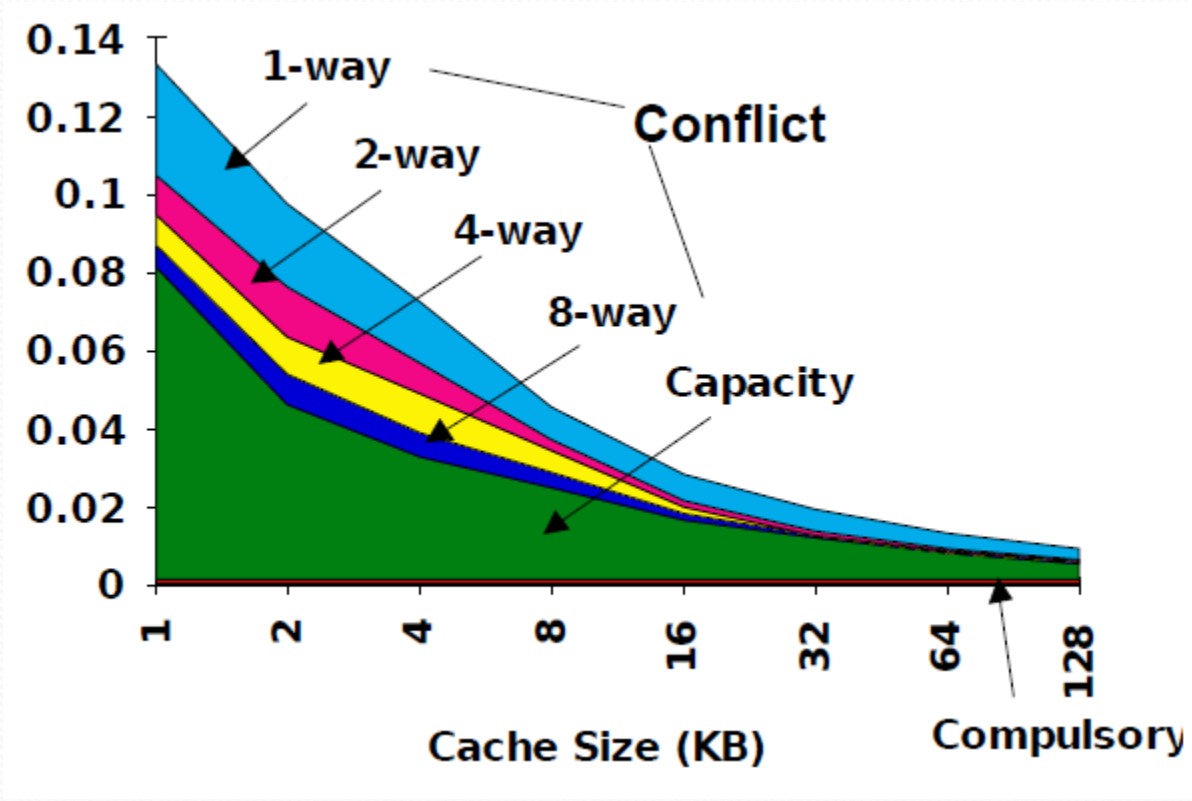
- Las 3 Cs:
  - Fallas obligatorias (compulsory misses). Son causadas por el primer acceso a un dato que nunca ha estado en el caché. También llamadas cold-start misses.
  - Fallas de capacidad (capacity misses). Son causadas cuando el caché no puede guardar todos los bloques necesarios para correr un programa. Por ejemplo, necesitar un bloque que acaba de ser reemplazado.

Origen de las fallas

- Fallas de conflicto (conflict misses). Ocurren en cachés de mapeo directo o set associative cuando múltiples bloques compiten por el mismo conjunto. También llamadas fallas de colisión.

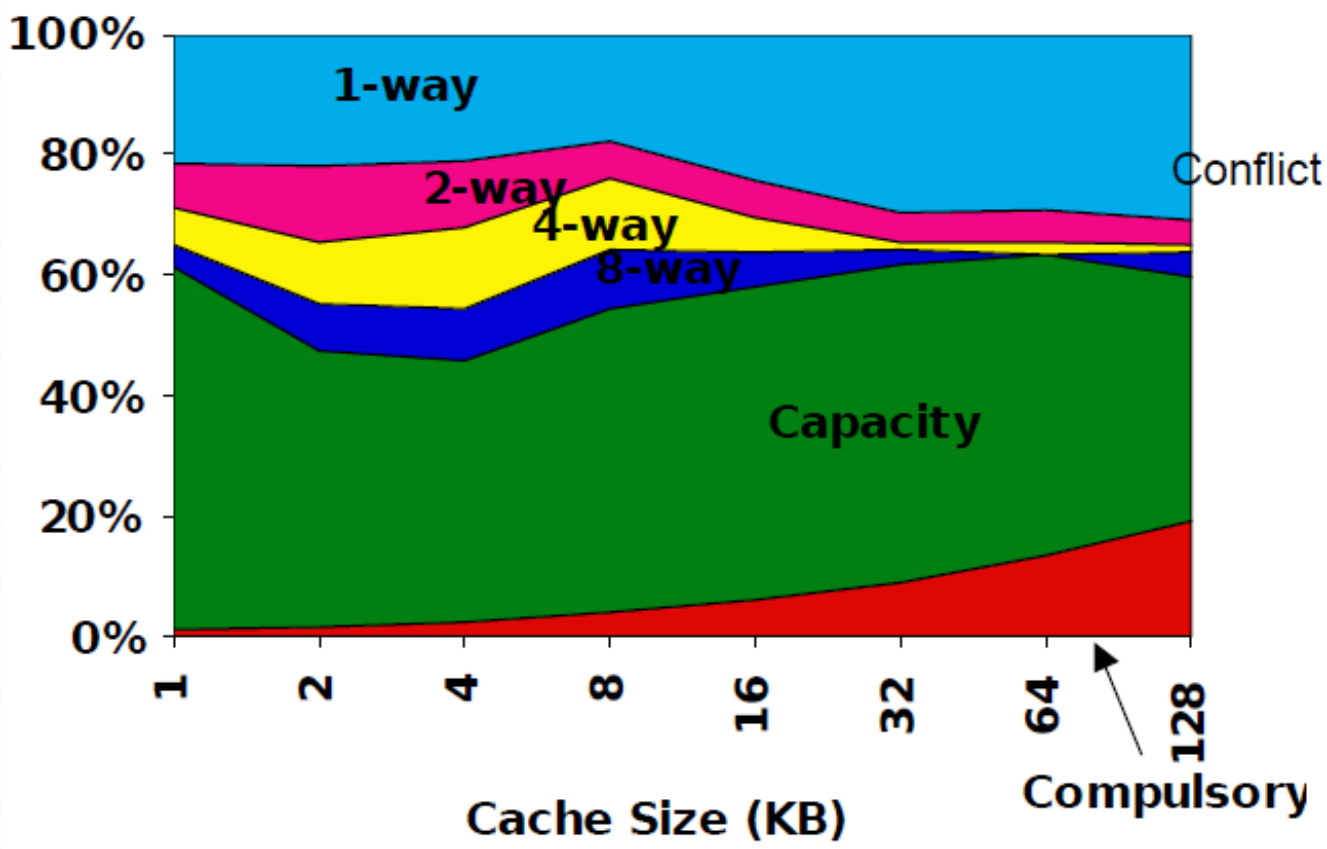
Origen de las fallas

- Las fallas obligatorias son 0.0006%



Origen de las fallas

- Tasa de fallas relativa.



Reto del diseño

Design change	Effect on miss rate	Possible negative performance effect
Increase cache size	decreases capacity misses	may increase access time
Increase associativity	decreases miss rate due to conflict misses	may increase access time
Increase block size	decreases miss rate for a wide range of block sizes due to spatial locality	increases miss penalty. Very large block could increase miss rate

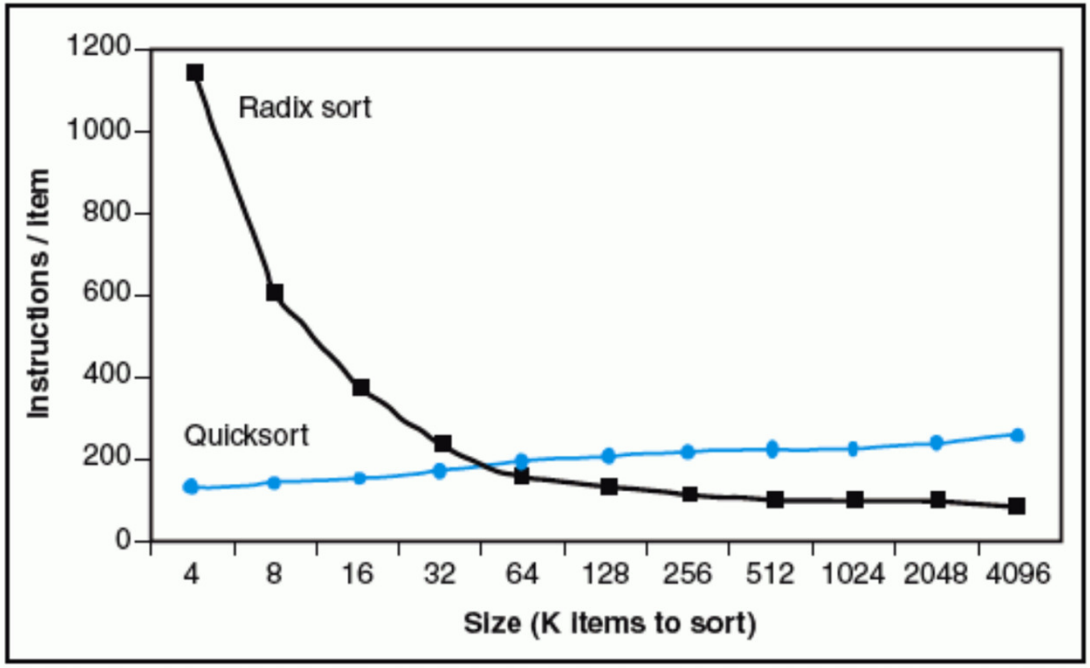
- Tres mejoras:
- Reducir la tasa de fallas
- Reducir el castigo por fallas
- Reducir el tiempo de éxito
- Se pueden escoger solo 2

Algoritmos y el caché

- Un algoritmo puede tener un comportamiento distinto a su comportamiento teórico debido a la presencia del caché.
- Ejemplo: en teoría, Radix Sort es mejor que Quicksort para arreglos grandes.

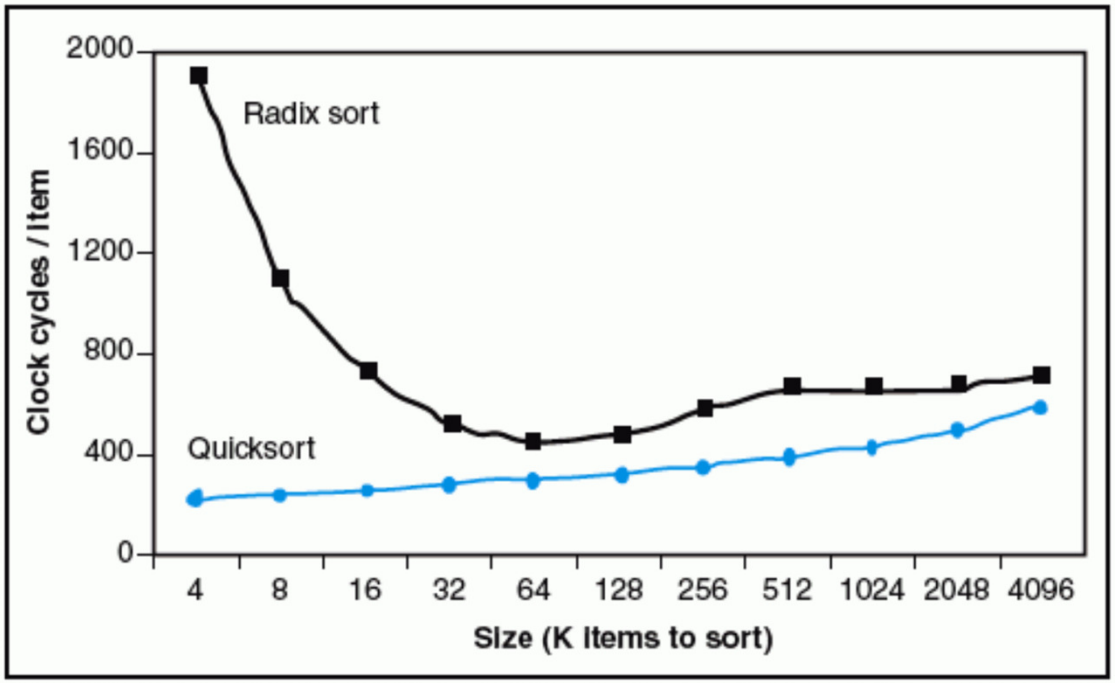
Algoritmos y el caché

- Comportamiento teórico



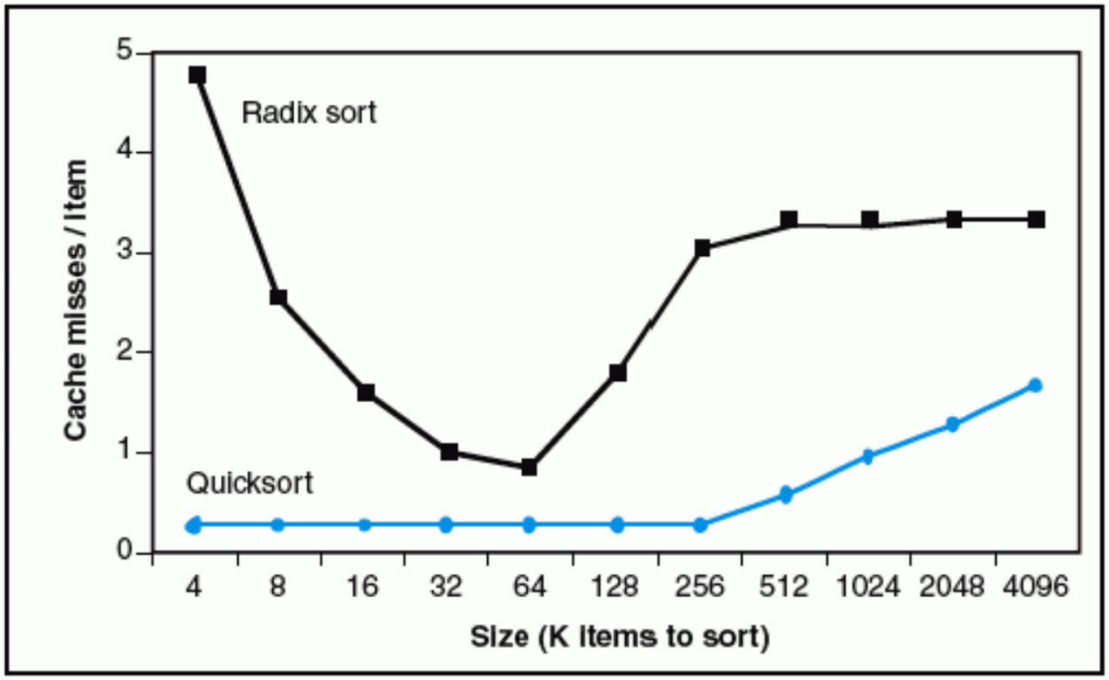
Algoritmos y el caché

- Comportamiento real



Algoritmos y el caché

- Motivo: fallas de caché.



Solución

- Diseñar algoritmos que hagan uso efectivo del caché (principio de locality espacial y/o locality temporal).

Ejemplo 1

```
// Obtiene la suma de un vector
int sumvec(int v[N])
{
    int i, sum = 0;
    for (i = 0; i < N; i++)
        sum += v[i];
    return sum;
}
```

Análisis

- La función tiene locality temporal respecto a las variables locales `i` y `sum`.
- Un compilador las va a guardar en registros.
- Suponer que los enteros ocupan 4 bytes (1 palabra), que el caché tiene bloques de 4 palabras y que inicialmente está vacío.
- Las referencias al arreglo siguen el siguiente patrón:

v[i]	i = 0	i = 1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7
orden, hit / miss	1 **m**	2 h	3 h	4 h	5 **m**	6 h	7 h	8 h

Análisis

- La tasa de fallas es de 25%.
- Es lo mejor que se puede obtener con un caché inicialmente vacío.
- Conclusión: la función tiene un buen locality espacial.

Ejemplo 2

```
// Obtiene la suma de una matriz
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Análisis

- La función tiene locality temporal respecto a las variables locales i, j y sum.
- Un compilador las va a guardar en registros.
- C y Java guardan los arreglos por renglón.
- Suponer que los enteros ocupan 4 bytes (1 palabra), que el caché tiene bloques de 4 palabras y que inicialmente está vacío.
- Las referencias al arreglo siguen el siguiente patrón:

Análisis

a[i][j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 **m**	2 h	3 h	4 h	5 **m**	6 h	7 h	8h
i = 1	9 **m**	10 h	11 h	12 h	13 **m**	14 h	15 h	16 h
i = 2	17 **m**	18 h	19 h	20 h	21 **m**	22h	23 h	24 h
i = 3	25 **m**	26 h	27 h	28 h	29 **m**	30 h	31 h	32 h

- La tasa de fallas es de 25%.
- Es lo mejor que se puede obtener con un caché inicialmente vacío.
- Conclusión: la función tiene un buen locality espacial.

Ejemplo 3

```
// Obtiene la suma de una matriz
   int sum_array_cols(int a[M][N])
   {
      int i, j, sum = 0;
      for (j = 0; i < N; j++)
         for (i = 0; i < M; i++)
            sum += a[i][j];
      return sum;
   }
```

Análisis

- La función tiene locality temporal respecto a las variables locales i, j y sum.
- Un compilador las va a guardar en registros.
- Las referencias al arreglo siguen el siguiente patrón:

a[i][j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 **m**	5 **m**	9 **m**	13 **m**	17 **m**	21 **m**	25 **m**	29 **m**
i = 1	2 **m**	6 **m**	10 **m**	14 **m**	18 **m**	22 **m**	26 **m**	30 **m**
i = 2	3 **m**	7 **m**	11 **m**	15 **m**	19 **m**	23 **m**	27 **m**	31 **m**
i = 3	4 **m**	8 **m**	12 **m**	16 **m**	20 **m**	24 **m**	28 **m**	32 **m**

Análisis

- La tasa de fallas es de 100%.
- Conclusión: la función **no** tiene un buen locality espacial.
- Si el arreglo cupiera en el caché la tasa de éxito mejoraría.

Conclusiones

- La tasa de fallas tiene un impacto en el tiempo de ejecución.
- Para arreglos grandes, sum_array_rows corre 25 veces más rápido que sum_array_cols.
- Los programadores deben estar conscientes del locality de sus programas y escribir programas para tomar ventaja de el.

Recomendaciones

- Enfocarse en los ciclos internos.
- Para maximizar el locality espacial tratar de leer los objetos secuencialmente en el orden en que están guardados en la memoria.
- Para maximizar el locality temporal usar los datos tan frecuente como sea posible una vez que han sido leídos de la memoria.