U.B.A. - Facultad de Ingeniería

66.20/86.37 Organización de Computadoras
Jerarquía de memorias

Práctica

1^{er} cuatrimestre 2020

Introducción

Características deseadas para un sistema de memoria

- La mayor capacidad de memoria posible
- La menor latencia posible
- El mayor ancho de banda posible
- El menor costo posible
- El menor consumo de energía, espacio, etc.

Situación real: cuanto más rápida es una tecnología de memoria, más costosa.

Introducción (cont.)

Solución:

- Memoria organizada de forma jerárquica (en varios niveles)
- Cada nivel es más rápido, más chico y más costoso por byte que el siguiente nivel más bajo
- Todos los datos que se encuentran en un nivel dado, se encuentran también en los niveles inferiores

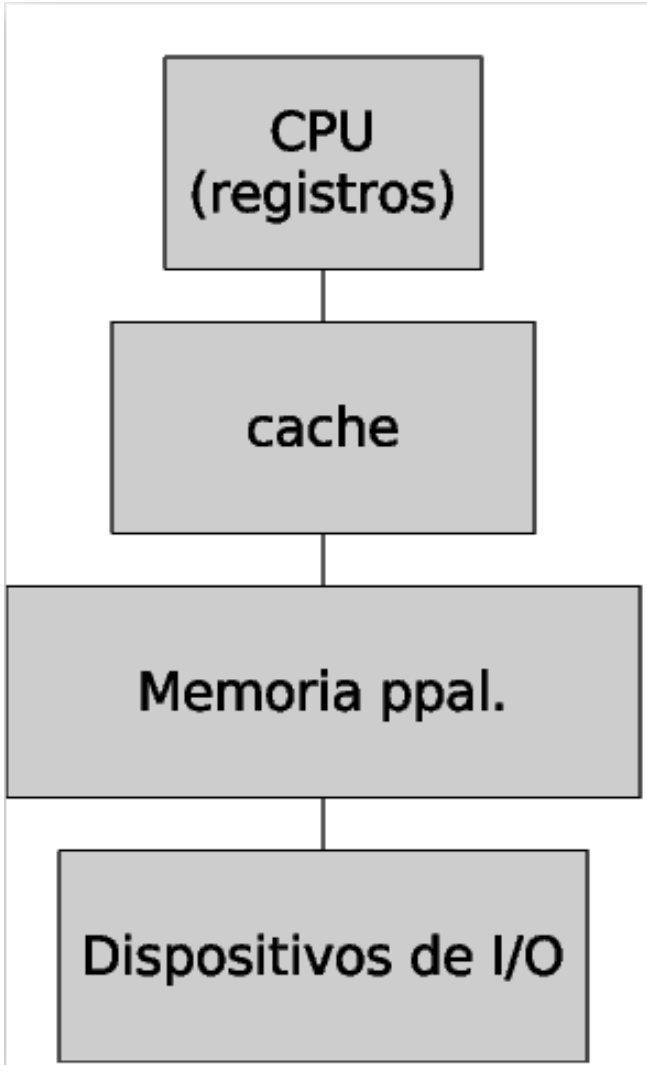
Introducción (cont.)

Se busca con esta organización jerárquica:

- Que el costo sea casi tan bajo como el nivel de memoria de menor costo.
- Que la velocidad sea tan alta como el nivel de memoria más rápido.

La efectividad de esta solución está dada por los principios de localidad espacial y temporal.

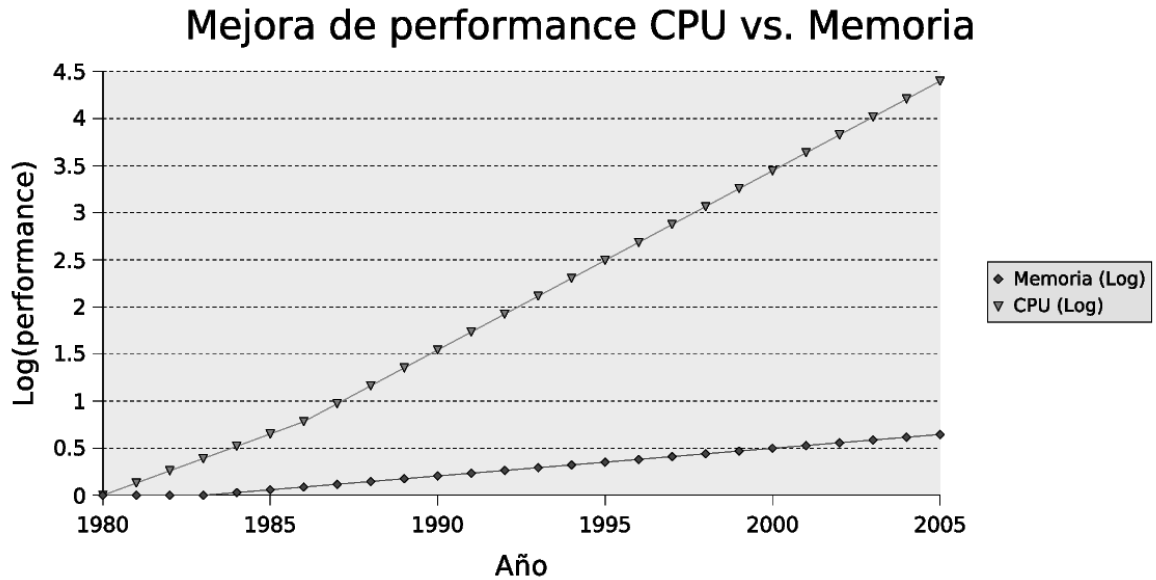
Niveles en una jerarquía de memoria



	Tamaño	Latencia	Bandwidth
CPU (registros)	512B	0.25ns	50000MB/s
cache	64KB	1ns	10000MB/s
Memoria ppal.	1GB	100ns	2500MB/s
Dispositivos de I/O	100GB	5ms	100MB/s

Mejoras performace CPU vs. memoria

- La mejora de performance en los procesadores fue mucho mayor que en las memorias
- Esto acrecentó enormemente la importancia de organizar jerárquicamente el sistema de memoria (para intentar "seguirle el paso" a las CPUs)



Latencia y Bandwidth

- Se busca disminuir la latencia para reducir el tiempo de acceso a un dato cualquiera
- Se busca aumentar el bandwidth para poder poblar la memoria de nivel superior desde la de nivel inferior con la mayor velocidad posible
- Un sistema corriendo múltiples procesos simultáneos realiza muchos context switches por unidad de tiempo, por lo cual se producirán *cache misses* compulsivos (ej: en un server). Este tipo de sistemas requieren un gran ancho de banda para poblar rápidamente niveles superiores con los datos de los inferiores

Caches

Cache es el nombre de los primeros niveles de la jerarquía de memoria luego de los registros, y antes de la memoria principal (RAM).

- Puede haber varios niveles: L1, L2 y recientemente se introdujo L3
- El termino *cache* se utiliza para muchas otras regiones de almacenamiento destinadas a acelerar el acceso a datos aprovechando el principio de localidad (ej: file caches, name caches, etc)

Caches - Conceptos básicos

- **Cache hit / miss**: cuando la CPU encuentra un dato en el cache, tenemos un hit. Sino, es un miss (y el sistema deberá descender en la jerarquía para obtener el dato)
- **Bloque o línea de cache**: cuando se produce un miss, se transfiere entre el cache y el nivel inferior un bloque o línea en lugar de solo el item pedido por el CPU (ppio de localidad: muy probablemente los aproveche)
- **Latencia**: determina cuanto tarda el transferirse el primer word del bloque
- **Bandwidth**: determina el tiempo para cargar el resto de la línea
- **Stall**: un cache miss produce que la CPU se detenga hasta que los datos estén presentes.

Caches - Conceptos básicos (cont.)

- **Memory accesses**
  - cada instrucción requiere un acceso a memoria por fetch (al texto del programa);
  - si es load / store, requiere otro acceso (a datos).

- **Miss Penalty**
  - está dado en cantidad de clock cycles;
  - lo usamos como si fuera una constante;
  - la memoria accedida puede estar ocupada por un request anterior o por un refresco;
  - el número de ciclos varia según la velocidad de clock del CPU, bus y la memoria;
  - por lo tanto considerarlo constante es una **simplificación**.

Caches - Memory Stall Cycles

- **Memory Stall cycles**
  - ciclos de reloj durante los cuales la CPU está detenida esperando los datos requeridos (debido a un cache miss).
  - Para evaluar la performance del cache, puede utilizarse la expresión de *CPU execution time*, incluyendo en la misma los *memory stall cycles*:

$$
\begin{aligned}
\textit{CPU execution time} &= (\textit{CPU clock cycles} + \textit{Memory stall cycles}) \times \\
&\quad \times \textit{Clock cycle time} \\
\textit{Memory stall cycles} &= \textit{Number of misses} \times \textit{Miss penalty} \\
&= \textit{IC} \times \frac{\textit{Misses}}{\textit{Instruction}} \times \textit{Miss penalty} \\
&= \textit{IC} \times \frac{\textit{Memory accesses}}{\textit{Instruction}} \times \textit{Miss rate} \times \textit{Miss penalty}
\end{aligned}
$$

Caches - Memory Stall Cycles (cont.)

- **Miss rate**
  - puede medirse con simuladores de cache (valgrind - cachegrind);
  - algunos procesadores proveen contadores de **misses** y **memory references**.

- **Read / Write Memory References**
  - **miss rate** y **miss penalty** pueden diferir para lectura y escritura;
  - la formula anterior es una simplificación:

$$\begin{aligned} \textit{Memory stall cycles} &= IC \times \frac{\textit{Reads}}{\textit{Instr.}} \times \textit{Read miss rate} \times \textit{Read miss penalty} + \\ &+ IC \times \frac{\textit{Writes}}{\textit{Instr.}} \times \textit{Write miss rate} \times \textit{Write miss penalty} \end{aligned}$$

Caches - Asociatividad

¿Dónde se almacena un bloque o línea en el cache?

- **Direct mapped**: un bloque solo puede almacenarse en un único *block frame*, usualmente:
  $Block\ frame = (Block\ address)\ mod\ (Number\ of\ blocks\ in\ cache)$
- **Fully associative**: un bloque puede almacenarse en cualquier *block frame*;
- **Set associative**: los *block frames* en el cache se agrupan en *sets*. Un bloque determinado se mapea primero a un *set*, y luego se guarda en cualquier *block frame* del *set*, pero siguiendo una *política de reemplazo*. El *set* se elige con:
  $Set = (Block\ address)\ mod\ (Number\ of\ sets\ in\ cache)$

Caches - Asociatividad (cont.)

- **N-way set associative**: cuando hay N bloques en un *set*;
  - éste es el caso general;
  - el caso **fully associative** puede verse como $N = M$, donde M es el número de bloques en el cache;
  - el caso **direct mapped** puede verse como $N = 1$.

Caches - Políticas de reemplazo

- Cuando ocurre un miss, el cache controller tiene que decidir qué bloque tiene que reemplazar;
- Para caches *direct mapped* no hay alternativas;
- Para caches asociativos, hay tres estrategias principales:
  - **Random**: el bloque se almacena en cualquier block frame del set, elegido (pseudo)aleatoriamente;
  - **Least-recently used (LRU)**: se almacena el bloque en el block frame que no fue utilizado por más tiempo (principio de localidad);
  - **FIFO**: es más fácil de implementar que LRU, reemplaza el bloque más viejo en lugar del LRU.

Caches - Cache Writes

Políticas de escritura

- **Write through**
  - la información se escribe al cache y al nivel inferior.
- **Write back**
  - la información solo se escribe al cache;
  - recien cuando ocurre un reemplazo, se escribe al nivel inferior.

Caches - Cache Writes (cont.)

Write Back

- **Dirty bit**:
  - indica si un bloque fue escrito (dirty) o no (clean);
  - en un miss, si el bloque no está *dirty*, no se escribe a nivel inferior (porque la información se repite en los niveles inferiores);
  - este mecanismo permite reducir la frecuencia de escrituras a nivel inferior;

Caches - Cache Writes (cont.)

Ventajas y desventajas

- **Write back**:
  - por tener menor frecuencia de escritura, requiere menor ancho de banda;
  - típicamente, multiples escrituras a un bloque por transferencia hacia el nivel inferior.

- **Write through**:
  - más fácil de implementar;
  - cache siempre limpio (clean), por lo que un read miss no provoca en un write al nivel inferior;
  - los niveles inferiores están siempre actualizados, por lo que la *coherencia* se simplifica (muy importante para multiprocesadores y para I/O).

Caches - Cache Writes (cont.)

Hay dos políticas de operación para un write miss:

- **Write allocate**: se carga el bloque en el cache, y luego se realizan las acciones correspondientes a un write hit.
- **No-write allocate**: el bloque se modifica en el nivel inferior y no en el cache (los write misses no modifican el cache).

Entonces, los bloques no se transfieren a cache en **no-write allocate** hasta que el programa intenta leer estos bloques, pero en **write allocate** los bloques se van a copiar aún cuando solo hay escrituras.

Caches - Cache Writes (cont.)

Write allocate / No-write allocate

Ambas políticas pueden aplicarse a *write-through* y *write-back*. Normalmente:

- **write-back + write allocate**
  (por localidad se espera que las siguientes escrituras caigan en ese bloque en cache)

- **write-through + no-write allocate**
  (aún cuando haya escrituras a ese bloque, los datos deben llegar al nivel inferior)

Caches - Cache Writes (cont.)

- **Write stall**:
  - cuando la CPU debe esperar para que un write through complete;
- **Write buffer**:
  - optimización para reducir el stall;
  - permite al CPU continuar apenas los datos fueron escritos al buffer;
  - los write stalls igual pueden producirse.

Cache - Optimizaciones

- **Optimización de lecturas**
  - Un bloque puede ser leido del cache al mismo tiempo que su tag es leido y comparado;
  - si se detemina que es un hit, la parte del bloque pedida se pasa al CPU inmediatamente;
  - sino, se ignora el valor leido.

- **Optimización de escrituras**
  - No puede modificarse un bloque hasta verificar el tag y ver si es un hit;
  - por este motivo, las escrituras toman más tiempo.