Capítulo 2.

Fundamentos teóricos

En este capítulo se hace un pequeño resumen de todos los conceptos teóricos incluidos en el simulador, ya que resultan imprescindibles para la comprensión total de la herramienta.

2.1. Jerarquías de memoria

Desde el inicio de la informática los usuarios y programadores han deseado disponer de una cantidad de memoria ilimitada que fuera lo más rápida posible. Sin embargo, una solución basada en el uso masivo de la memoria más rápida disponible, tendría un coste tan alto que la haría inviable.

Una solución más económica pasa por la implementación de una jerarquía de memoria, que se aproveche de la localidad de los programas y de las relaciones coste/rendimiento de diferentes tecnologías para "*proporcionar un sistema de memoria con un coste casi tan bajo como el del nivel más barato de memoria con una velocidad tan alta como la del nivel más rápido".* [1]

Una jerarquía de memoria se organiza en diferentes niveles, cada uno de ellos más pequeño pero más rápido que el anterior. La información contenida en cada nivel suele ser un subconjunto de la del nivel anterior, de manera que todos los datos de un nivel están a su vez en el nivel inferior.

La importancia de las jerarquías de memoria ha aumentado con el avance en prestaciones de los procesadores, cada vez más rápidos y más dependientes de la velocidad del sistema de memoria.

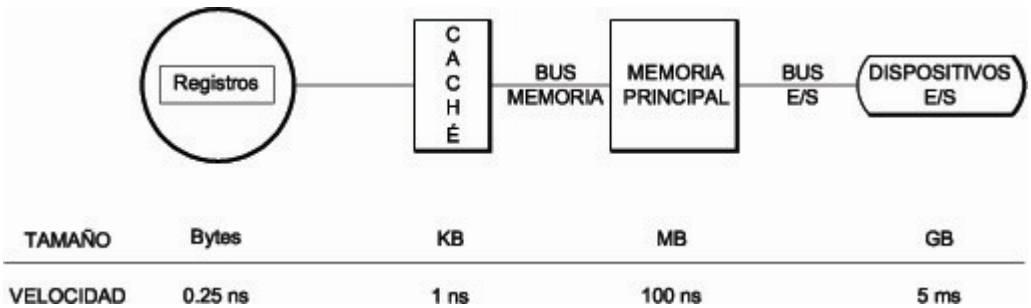

Figura 2.1. Esquema típico de una jerarquía de memoria

[1] Traducción de la cita de la página 390 de Computer Architecture: A Quantitative Approach [1]

2.2. Localidad de referencias

El análisis de los programas muestra que la mayor parte de su tiempo de ejecución se invierte en rutinas en las que muchas de sus instrucciones se ejecutan repetidas veces, de manera que instrucciones de zonas concretas del programa se ejecutan repetidamente durante largos periodos de tiempo mientras que al resto del programa se accede de manera poco frecuente. Este fenómeno se conoce como localidad de referencias y se manifiesta de dos formas:

- *Localidad temporal:* Es probable que si se referencia un elemento, éste vuelva a ser referenciado pronto.

- *Localidad espacial:* Es probable que si se referencia un elemento, los elementos próximos a él también sean referenciados pronto.

2.3. Memorias caché

La memoria principal es un recurso de acceso lento, lo que significa que el procesador debe esperar mucho tiempo cada vez que desea obtener un dato o instrucción desde la memoria. Para paliar el efecto de esta espera y así aumentar las prestaciones, se incluye en la jerarquía una memoria de acceso muy rápido que se intercala entre el procesador y la memoria principal, a la que denominamos caché. Su tamaño suele ser bastante menor que el de la memoria principal y normalmente se implementa utilizando memorias SRAM en lugar de DRAM[2].

Su función principal será almacenar la información usada en el pasado, pues según la propiedad de localidad de referencias, es muy posible que se necesite usarla en un futuro próximo.

2.3.1. Aciertos y fallos de caché

Cuando el procesador solicita un dato a la caché y éste se encuentra en la misma, se produce lo que denominamos "Acierto en caché" mientras que cuando no se encuentra, se produce un "Fallo de caché". Esto provoca que una colección de datos denominada bloque, sea transferida desde la memoria principal hasta la memoria caché. En consecuencia se deduce que, para aumentar el rendimiento, debemos reducir el número de "Fallos de caché".

2.3.2. Medida de prestaciones de las memorias caché

Para medir la influencia de la caché dentro de la jerarquía de memoria vamos a utilizar la fórmula de tiempo de CPU por instrucción:

Tiempo CPU = (Ciclos de reloj para ejecutar una instrucción +
Ciclos de reloj espera por memoria) x Duración del ciclo de reloj

[2] Más información sobre memorias DRAM y SRAM puede encontrarse en Carl Hamacher, Zvonko Vranesic, Safwat Zacky, *Organización de Computadores* [4]

Donde los *Ciclos de reloj para ejecutar una instrucción* representan el tiempo necesario para ejecutar una instrucción en caso de "Acierto de caché" y los *Ciclos de reloj espera por memoria* representan los ciclos que la CPU debe esperar a que el bloque sea transferido desde memoria principal a caché en caso de "Fallo de caché".

Debemos obtener ahora por tanto los *Ciclos de reloj espera por memoria.* Para hacerlo nos valemos de la siguiente formula:

Ciclos de reloj espera por memoria =
(Instrucciones lectura / Instrucciones de programa)
x Frecuencia fallos lectura x Penalización por fallo de lectura
+ (Instrucciones escritura / Instrucciones de programa)
x Frecuencia fallos escritura x Penalización por fallo de escritura

Simplificando los términos obtenemos esta otra fórmula:

Ciclos de reloj espera por memoria =
(Instrucciones acceso a memoria / Instrucciones de programa)
x Frecuencia fallos x Penalización por fallo

Añadiéndolo a la fórmula inicial queda de la siguiente forma:

Tiempo CPU = (Ciclos de reloj para ejecutar una instrucción
+ (Instrucciones acceso a memoria / Instrucciones de programa)
x Frecuencia fallos x Penalización por fallo)
x Duración del ciclo de reloj

Vemos que dado que los *Ciclos de reloj para ejecutar una instrucción*, la relación (*Instrucciones acceso a memoria / Instrucciones de programa)* y la *Duración del ciclo de reloj* son independientes del sistema de memoria, son la *Frecuencia fallos* y la *Penalización de fallos* los parámetros que debemos reducir para incrementar las prestaciones.

2.3.3. Fuentes de fallos de caché

Un estudio del comportamiento de la caché revela que todos los fallos se producen debido a una de estas tres causas:

- *Primera referencia:* En el primer acceso a un bloque es seguro que éste no se encuentra en la memoria caché, ya que debe ser traído desde memoria principal a caché.

- *Capacidad:* Si la caché no puede contener todos los bloques necesarios durante la ejecución de un programa, se presentarán fallos que obligarán a descartar bloques que posteriormente deberán ser traídos de nuevo.

- *Conflicto:* Según el tipo de estrategia de colocación utilizado es posible que se puedan descartar bloques que posteriormente deban ser traídos de nuevo

Una caché eficiente deberá implementar por tanto mecanismos orientados a evitar estas situaciones.

2.3.4. Cachés divididas frente a cachés conjuntas

Un estudio más concienzudo de la localidad de referencias de los programas, muestra que éstos manifiestan diferentes comportamientos según se trate de instrucciones o datos. Por ello, las cachés a veces se dividen en caches de instrucciones y caches de datos, con lo que se tiene la posibilidad de optimizarlas de forma separada (diferentes capacidades, tamaños de bloque y asociatividades).

Aunque el hecho de dividir provoca un incremento en el coste, ya que es necesario duplicar memorias y buses, separando instrucciones y datos, conseguimos eliminar un gran número de conflictos y fijar un espacio en el que direcciones y datos aprovechen mejor la capacidad disponible.

2.3.5. Cachés multinivel

Actualmente los procesadores se van volviendo cada vez más rápidos, con lo que las tecnología necesaria para que la caché se mantenga a la altura y supere el creciente desnivel entre el procesador y la memoria principal es demasiado costosa.

Siguiendo entonces el principio de las jerarquías de memoria se recurre a añadir nuevos niveles entre la caché original y la memoria principal. Surgen entonces los *"niveles de caché",* basados en introducir memorias más pequeñas pero más rápidas del lado del procesador y memorias mayores aunque sensiblemente más lentas del lado de la memoria principal.

De esta manera conseguimos llevar a cabo de forma eficiente el principio de las jerarquías de memoria, "*proporcionar un sistema de memoria con un coste casi tan bajo como el del nivel más barato de memoria con una velocidad tan alta como la del nivel más rápido"*.

En los sistemas actuales de escritorio es corriente encontrar dos niveles de memoria, subiendo hasta tres en el caso de servidores:

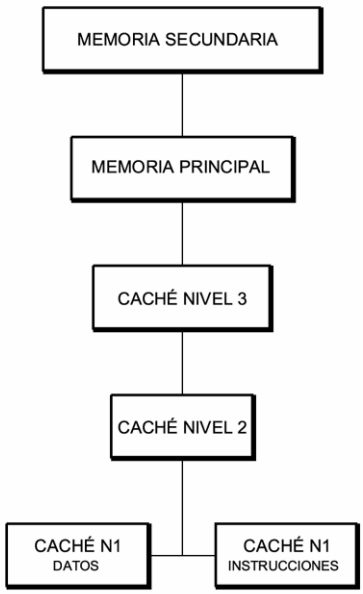

Figura 2.2. Jerarquía de memoria con caché multinivel

Si pasamos de nuevo a evaluar las prestaciones de las memorias caché multinivel, en este caso de dos niveles, nos encontramos con la siguiente fórmula:

Tiempo medio de acceso a memoria =
Tiempo empleado si acierto en nivel 1 + Frecuencia fallos nivel 1
x (Tiempo empleado si acierto en nivel 2 +
Frecuencia fallos nivel 2 x Penalización por fallo en nivel 2)

Cómo vemos, el éxito de la frecuencia de fallos del segundo nivel se mide con las sobras del primer nivel, lo cual da lugar a dos tipos de frecuencia de fallos que debemos reducir para aumentar las prestaciones:

- *Frecuencia local de fallos:* El número total de fallos dividido por el número total de accesos a esta caché.

- *Frecuencia global de fallos:* El número de fallos de la caché dividido por el número total de accesos a memoria generados por el procesador.

2.3.6. Políticas de mapeado

A la hora de decidir que bloques entrarán en memoria caché y cuáles permanecerán en memoria principal, se plantean el siguiente conjunto de estrategias:

- *Estrategias de colocación:* Deciden en que posición debe ser colocado el nuevo bloque dentro de la memoria caché.

- *Estrategias de búsqueda:* Determinan en qué posición se encuentra un determinado bloque de memoria caché.

- *Estrategias de reemplazamiento:* Deciden que bloque debe ser reemplazado cuando la posición es solicitada por un nuevo bloque.

- *Estrategias de coherencia:* Tratan la coherencia entre los datos que han sido modificados cuando el bloque se encontraba en caché y los datos contenidos en la memoria principal.

2.3.6.1. Estrategias de colocación

La organización de la memoria caché determina la posición en que un nuevo bloque puede ser colocado dentro de memoria caché. Así se distinguen tres tipos de estrategias de colocación.

- *Mapeado directo:* Cada bloque de memoria principal tiene un solo lugar donde ser colocado en memoria caché según la fórmula.

(Bloque de memoria principal) MOD (Número de bloques en caché)

- *Memoria completamente asociativa:* Un bloque de memoria principal puede ser colocado en cualquier lugar de la memoria caché.
- *Memoria asociativa por conjuntos:* A un bloque de memoria principal le corresponde un conjunto de bloques en memoria caché, de manera que puede ser colocado en cualquier bloque dentro del conjunto determinado por la fórmula.

  (Bloque de memoria principal) MOD (Número de conjuntos en caché)

  Cuando un conjunto tiene n bloques decimos que tenemos una memoria asociativa por conjuntos de n-vías, es decir, que tiene asociatividad n.

Como se puede deducir, las diferentes estrategias no son más que un incremento de los niveles de asociatividad por conjuntos: de manera que la caché de mapeado directo es simplemente una caché asociativa por conjuntos de una vía, mientras una memoria completamente asociativa con m bloques es una memoria caché asociativa por conjuntos de m vías.

En la siguiente figura se muestra qué posición ocuparía un hipotético bloque 12 según las diferentes organizaciones en una jerarquía compuesta por una caché de 8 bloques y una memoria principal de 32 bloques.

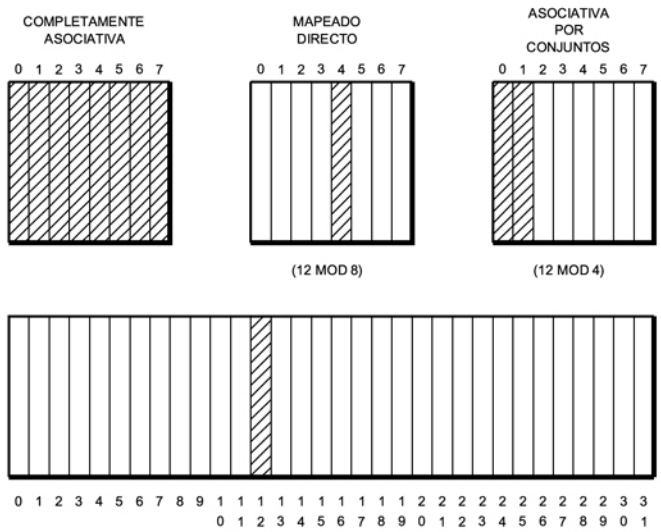

Figura 2.3. Estrategias de colocación

2.3.6.2. Estrategias de búsqueda

Para encontrar un bloque dentro de la caché, debemos encontrar primero la correspondencia entre la dirección de la CPU y la dirección de caché.

Mientras que una dirección de CPU está compuesta por bloque de memoria principal y desplazamiento del bloque, en las cachés, normalmente bastante más pequeñas que la memoria principal y teniendo en cuenta la asociatividad se componen de etiqueta, índice y desplazamiento.

El índice se utiliza para seleccionar el conjunto en mapeado directo y memoria asociativa por conjuntos, mientras que en memoria completamente asociativa no existe.

La etiqueta se incluye en un campo y se utiliza para comparar si el bloque encontrado es el bloque que buscamos.

Según esto podemos deducir que si mantenemos igual el tamaño de la caché, incrementamos la asociatividad, reducimos el tamaño del índice e incrementamos el tamaño de la etiqueta; aceleramos las búsquedas pero ralentizamos las comparaciones. Debemos por tanto encontrar un equilibrio para obtener las máximas prestaciones del sistema de caché.

Figura 2.4. Bloque de memoria caché

2.3.6.3. Estrategias de reemplazamiento

En ocasiones el nuevo bloque que entra en caché se encuentra con que la posición ya está ocupada por un bloque anterior. Debe entonces reemplazarse un bloque, de decidir cual se van a encargar las estrategias de reemplazamiento.

Las estrategias de colocación influyen también en la decisión, así en el mapeado directo la posición de un bloque ya se encuentra determinada, de manera que el bloque antiguo en esa posición debe ser reemplazado con el nuevo.

Con las restantes estrategias de colocación deben tenerse en cuenta otros factores, pues es posible variar la posición del nuevo bloque dentro del conjunto, dando lugar a variadas estrategias de reemplazamiento tales como:

- *Óptima:* Esta estrategia nos dice que el rendimiento óptimo se obtendría cuando el bloque reemplazado fuese aquel que no se va a utilizar durante más tiempo en el futuro. Dado que los ordenadores no pueden conocer el futuro, no es una estrategia implementable en un sistema real.

- *Azar:* Esta estrategia decide el bloque a reemplazar al azar, no es la más eficiente pero su simplicidad de implementación y ausencia de contador o sello de tiempo la hacen válida para ciertos sistemas.

- *FIFO (First in – First out o Primero en entrar – Primero en salir):* Esta estrategia reemplaza aquel bloque que ha estado durante más tiempo en memoria. Se implementa asociando a cada entrada un sello de tiempo que marca el instante en que el bloque ha entrado en memoria.

- *LRU (Least Recently Used o Menos recientemente usado):* Esta estrategia reemplaza aquel bloque que lleva más tiempo en memoria sin ser usado. Se implementa asociando a cada entrada un sello de tiempo que marca el último instante en que el bloque fue usado.

- *Clock:* Según su forma de implementación esta estrategia puede ser una aproximación a FIFO o a LRU.

    - *Aproximación a FIFO:* Reemplaza aquel bloque que ha estado durante más tiempo en memoria. Se implementa utilizando un puntero que determina que bloque debe ser reemplazado avanzando siempre en el sentido de las agujas del reloj, de ahí que se denomine "Clock", reloj en inglés.

    - *Aproximación a LRU:* Reemplaza aquel bloque que ha estado durante más tiempo en memoria sin ser usado. Se implementa, al igual que la aproximación a FIFO, utilizando un puntero que determina qué bloque debe ser reemplazado avanzando siempre en el sentido de las agujas del reloj, pero además añade un bit que marca si el bloque ha sido usado desde la última vez que el puntero pasó por allí, para evitar que un bloque recientemente usado sea reemplazado.

- *LFU (Least Frequently Used o Menos frecuentemente usado):* Esta estrategia reemplaza aquel bloque que ha sido usado el menor número de veces desde que ha entrado en memoria. Se implementa asociando a cada entrada un contador del número de veces que el bloque ha sido usado.

- *NUR (Not used recently o No recientemente usado):* Esta estrategia reemplaza aquel bloque que no ha sido usado recientemente. Para determinar esta situación utiliza dos factores: si el bloque ha sido referenciado y si el bloque ha sido modificado. Se implementa asociando a cada entrada una etiqueta de dos bits, lo que da lugar a cuatro posibles grupos de bloques para ser reemplazados.

    - Bit 1: Bit de referencia – (0 no referenciado y 1 referenciado)
    - Bit 2: Bit de modificación – (0 no modificado y 1 modificado)

    - 00 - Bloque no referenciado y no modificado
    - 01 - Bloque no referenciado pero modificado
    - 10 - Bloque referenciado pero no modificado
    - 11 - Bloque referenciado y modificado

    Los bloques son reemplazados en este mismo orden.

2.3.6.4. Estrategias de coherencia

Dado que los datos contenidos en memoria caché son copias de los datos contenidos en memoria principal, debe existir algún método que nos permita mantener la coherencia entre el contenido de memoria principal y el de la memoria caché. De esto se encargan las estrategias de coherencia, que son de dos tipos:

- *Write Through o Escritura directa:* Una escritura en memoria actualiza tanto la copia en memoria principal como la copia en memoria caché.

    - *Victim Buffer:* Para minimizar el efecto de la lenta escritura en memoria principal o en niveles de caché de diferentes velocidades se suele utilizar una memoria (buffer) la cual permitirá actualizar la memoria principal y los niveles de caché más lentos sin que el procesador tenga que detener la ejecución.

- *Write Back o Escritura retardada:* Una escritura en memoria actualiza sólo la copia en caché, mientras que la copia de la memoria principal se actualizará cuando el bloque de caché sea reemplazado. La copia modificada de caché se marcará con un bit (*Bit dirty o Bit Sucio*) indicando que la copia en memoria principal no es válida.

Además de estas estrategias los sistemas de caché suelen implementar mecanismos para invalidar un determinado bloque de caché en los diferentes niveles de la jerarquía.

2.3.7. Bits de control

Cómo hemos visto, según las estrategias implementadas, cada entrada de memoria caché contará con una serie de bits, denominados bits de control.

Figura 2.5. Bloque de memoria caché con bits de control

2.4. Memoria principal

Una vez estudiada la memoria caché, pasamos al siguiente nivel en la jerarquía, la memoria principal.

Esta memoria desempeña una doble función dentro de la jerarquía, por un lado satisface las demandas de las cachés, y por el otro sirve como interfaz de entrada/salida, ya que es tanto el destino de la entrada como la fuente para la salida.

2.4.1. Medida de prestaciones de la memoria principal

Desde el punto de vista de la caché debemos de tener en cuenta dos parámetros:

- *Tiempo de acceso*: Tiempo desde que se ordena una lectura hasta que se recibe el dato.

- *Duración del ciclo*: Tiempo mínimo transcurrido entre peticiones a memoria.

Este último parámetro se encuentra influenciado por otro, el tiempo de refresco, ya que actualmente las implementaciones de memoria principal se realizan en DRAM[3]. Esto implica que la memoria puede no estar disponible ocasionalmente porque se está enviando una señal que regenere de nuevo los datos.

El conjunto de éstos parámetros suele agruparse en lo que denominamos *latencia.*

Por otro lado, desde el punto de vista de los dispositivos de E/S el factor primordial es el *ancho de banda*, que determina las velocidades de transferencia con los dispositivos de entrada/salida.

Debemos por lo tanto, reducir la *latencia* y aumentar el *ancho de banda* para incrementar las prestaciones.

2.4.2. Anchura de la memoria principal

Las cachés a menudo se organizan con una anchura de una palabra, porque la mayoría de los accesos a la CPU son de ese tamaño, y a su vez la memoria principal tiene el ancho de una palabra para que coincida con la anchura de la caché.

Sin embargo, duplicar o cuadruplicar el ancho de la memoria disminuirá la penalización por fallos, por lo que el coste de implementar un bus más ancho puede verse compensado con el aumento de las prestaciones producido al reducir la *latencia* y aumentar el *ancho de banda*.

2.4.3. Memoria principal entrelazada

Otra posible medida para aumentar el *ancho de banda*, y a su vez reducir la *latencia*, puede ser utilizar una memoria entrelazada, en la que los chips de memoria se organizan en bancos para leer o escribir múltiples palabras a la vez en lugar de una. De esta manera se disminuye de forma significativa la penalización por fallos.

Aunque la implementación hardware de esta solución plantea ciertas limitaciones, y su comportamiento ante escrituras muy seguidas no demasiado bueno, se ve compensado por la mejora cuando se efectúan múltiples accesos secuenciales.

2.5. Memoria secundaria

La memoria principal no puede usarse para cubrir toda la capacidad de almacenamiento necesaria en un ordenador, debido al elevado coste por bit de información almacenada. Se hace por tanto necesario utilizar otras tecnologías que nos permitan disponer de grandes cantidades de memoria aunque éstas sean de acceso más lento. Aparece entonces este nivel en la jerarquía, al que denominamos memoria secundaria.

En este nivel se suelen utilizar discos y cintas magnéticos, discos ópticos y cada vez más, memorias flash en forma de dispositivos fijos o extraíbles. Éstos se fijan a un bus de entrada/salida (IDE, SCSI, FibreChannel, PCI, USB, IEEE1394-FireWire) gestionado por un controlador que organiza las transferencias, comprueba

[3] Más información sobre memorias DRAM y SRAM puede encontrarse en Carl Hamacher, Zvonko Vranesic, Safwat Zacky, *Organización de Computadores* [4]

los errores y evita los conflictos; y que en muchas ocasiones incluye una memoria semiconductora (buffer) capaz de almacenar unos cuantos megabytes de datos, actuando como una memoria caché, que de nuevo aprovecha el principio de localidad de los programas.

2.6. Memoria virtual

Aunque actualmente es poco frecuente, pues ya es corriente contar con grandes cantidades de memoria física, puede suceder que un programa sea demasiado grande para ser alojado en memoria principal. Además suele ocurrir que los ordenadores se encuentren ejecutando múltiples procesos al mismo tiempo, ya que la mayoría de los sistemas operativos actuales son multiusuario y multitarea. En éstas ocasiones la suma del espacio ocupado por todos los procesos puede ser mayor que el tamaño de la memoria principal.

Para solucionar estos problemas y evitar al programador la tarea de identificar que partes de un programa deben convivir en memoria y cuáles no, se ideó un sistema de memoria denominado memoria virtual.

La memoria virtual permite disponer de la capacidad de direccionar un espacio de almacenamiento mayor que el disponible en la memoria principal de un determinado sistema. De esta manera, tenemos la impresión de disponer de más memoria física de la que realmente tenemos.

La memoria virtual se implementa aprovechando principalmente los dos niveles superiores de la jerarquía, la memoria principal y la memoria secundaria. Así, cuando se va a ejecutar un proceso, su código y datos pasan desde la memoria secundaria hasta la memoria principal siguiendo alguna estrategia que permita mantener sólo una pequeña porción del programa en memoria principal.

Además, también es posible dividir la memoria virtual de manera que asignemos porciones a cada usuario, para que pueda ejecutar sus procesos sin interferir con los de los demás usuarios.

Para ello, debe implementar una serie de mecanismos que le permitan gestionar estas tareas eficientemente.

2.6.1. Direcciones virtuales y direcciones reales

La clave de un sistema de memoria virtual está en discernir entre las direcciones virtuales y las direcciones reales:

- *Direcciones virtuales:* Son las direcciones generadas en la CPU por el proceso en ejecución.

- *Direcciones reales:* Son las direcciones físicas disponibles en la memoria principal del ordenador.

El sistema debe, por tanto, implementar un mecanismo que permita traducir entre los dos tipos de direcciones mientras el proceso se encuentra en ejecución.

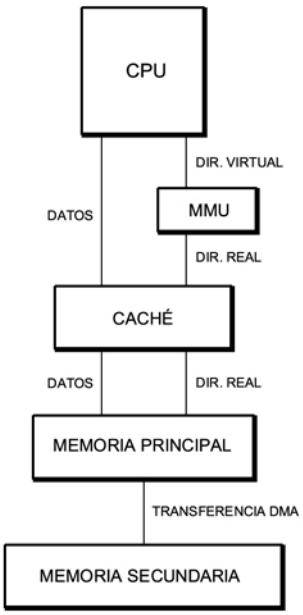

Figura 2.6. Transformación dinámica de direcciones

La tarea de realizar la traducción se suele implementar en una unidad hardware especial denominada MMU (Memory Management Unit o Unidad de Gestión de Memoria) para realizarla lo más rápidamente posible, y evitar la ralentización del sistema. A esta tarea se la suele conocer como *"Transformación dinámica de direcciones"*.

2.6.2. Transformación en bloques

Uno de los mecanismos de traducción dinámica de direcciones más utilizado es el de *"Transformación en bloques"*, que se basa en la idea de que el sistema de memoria virtual debe ser dividido en bloques, para evitar que la información del mapa de traducción sea demasiado voluminosa y difícil de manejar.

De esta manera, la información se agrupa en bloques que contienen elementos de información (instrucciones y datos), de los cuales el sistema conoce exactamente donde han sido colocados en la memoria principal o donde buscarlos en memoria secundaria si no se han cargado todavía.

El tamaño de estos bloques determina la técnica utilizada, distinguiéndose así dos tipos:

- *Paginación:* Compuesta por bloques de tamaño fijo (páginas).
- *Segmentación:* Compuesta por bloques de tamaño variable (segmentos).

La decisión de usar una técnica u otra lleva consigo una serie de ventajas y desventajas, que pasamos a enumerar en la siguiente tabla:

	Paginación	Segmentación
Palabras por dirección	Una	Dos (Segmento y desplazamiento)
¿Visible al programador?	No	Es posible
Reemplazamiento	Trivial, todos los bloques tienen el mismo tamaño	Difícil, debe localizarse una parte no utilizada y contigua en memoria principal
Uso de memoria	Fragmentación interna (Porciones de la página inutilizadas)	Fragmentación externa (Porciones de memoria principal sin usar)
Tráfico de disco	Eficiente (El tamaño de página se ajusta)	No siempre (Algunas transferencias llevan pocos datos)

Tabla 2.1. Paginación frente a segmentación

Dado que cada técnica aporta ventajas y desventajas, en los sistemas actuales se suele implementar un mecanismo mixto de paginación y segmentación (*Segmentación paginada*), en el que los bloques de tamaño variable (segmentos), alojan un grupo de bloques de tamaño variable (páginas). De esta manera, los segmentos tienen tamaños menos dispares, múltiplos del tamaño de página, que disminuyen la fragmentación externa y facilitan el reemplazamiento. Además, como contienen los datos asociados al proceso (segmento de código, segmento de datos) dentro del mismo segmento, aprovechan mejor las transferencias de disco.

En la siguiente sección vamos a dejar a un lado la segmentación, centrándonos en la paginación, ya que proporciona una manera más sencilla de comprender todos los mecanismos involucrados en un sistema de memoria virtual.

2.7. Paginación

Cómo ya se ha descrito, la paginación es un sistema de memoria virtual compuesto por bloques de tamaño fijo denominados páginas.

Dado que un proceso sólo puede ser ejecutado si su página actual se encuentra en memoria principal, las páginas deben ser transferidas desde memoria secundaria hasta memoria principal en bloques, llamados marcos de páginas, que tienen el mismo tamaño que las páginas. De esta manera, a cada página que se traslade a memoria principal, se le asignará un marco de página de entre todos los disponibles. De determinar cuál, se encargarán las estrategias de colocación que veremos más adelante.

Una dirección virtual en un sistema de paginación es un par ordenado (p, d), donde p es el número de página en el sistema de memoria virtual en el que reside el elemento al que se está haciendo referencia y d es el desplazamiento dentro de la página p donde está el elemento referenciado.

2.7.1. Selección del tamaño de página

En un sistema de memoria virtual paginado el primer parámetro que incide en las prestaciones es el tamaño de página. Elegir su tamaño es cuestión de encontrar el equilibrio entre:

- Páginas grandes: favorecen que el tamaño de la tabla de páginas sea menor y la transferencia entre memoria secundaria y memoria principal sea más rápida.

- Páginas pequeñas: aunque incrementan el tamaño de la tabla de páginas, el porcentaje de memoria y ancho de banda malgastados en cada bloque disminuye, pues un bloque demasiado grande provoca que éste no se llene completamente provocando lo que se denomina *"Fragmentación interna"*.

De aquí concluimos que, según la naturaleza de los procesos y la arquitectura concretas de una máquina, puede resultar más ventajoso un tamaño de página grande que otro pequeño, o viceversa.

2.7.2. Mecanismos de traducción de direcciones

Existen varios mecanismos para llevar a cabo la traducción de direcciones virtuales a direcciones reales en los sistemas de memoria virtual paginados, los cuales se detallan a continuación.

2.7.2.1. Traducción de direcciones por transformación directa

Este mecanismo basa su funcionamiento en una tabla de páginas, normalmente alojada en la propia memoria principal.

La tabla de páginas almacena la siguiente información para cada una de las páginas de la memoria virtual de los procesos:

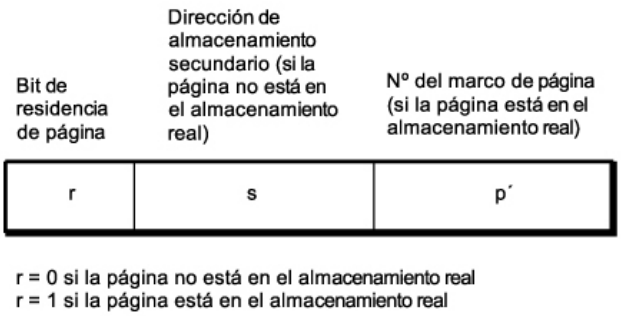

Figura 2.7. Entrada de la tabla de páginas

Como vemos en la siguiente figura, la parte p de la dirección virtual se utiliza para indexar la tabla de páginas, alojada a partir de la dirección b en memoria principal, en busca de la entrada correspondiente para determinar si la página se encuentra o no en la memoria principal.

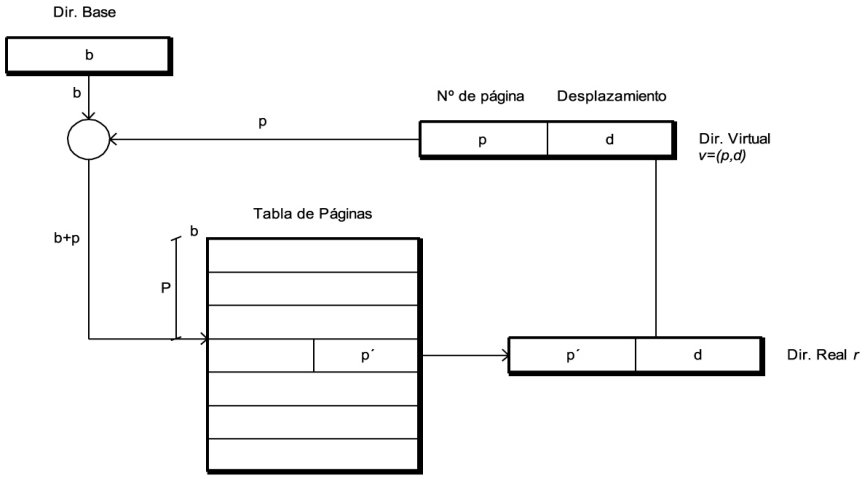

Figura 2.8. Traducción de direcciones por transformación directa

Si el bit de residencia *r* está a:

- *0:* La página no está en memoria principal, de manera que debemos transferirla desde el almacenamiento secundario en la dirección s hasta un marco libre en memoria principal y modificar la entrada de la tabla de páginas. A este caso lo denominamos "Fallo de página".
- *1:* La página está ya cargada en la memoria principal en el marco p'. Debemos entonces acceder a él y sumarle el desplazamiento d para obtener el dato o instrucción. A este caso lo denominamos "Acierto en tabla de páginas".

2.7.2.2. Traducción de direcciones por transformación asociativa

La transformación directa presenta el problema de que el acceso a la tabla de páginas es muy lento, ya que se encuentra en una memoria de acceso aleatorio.

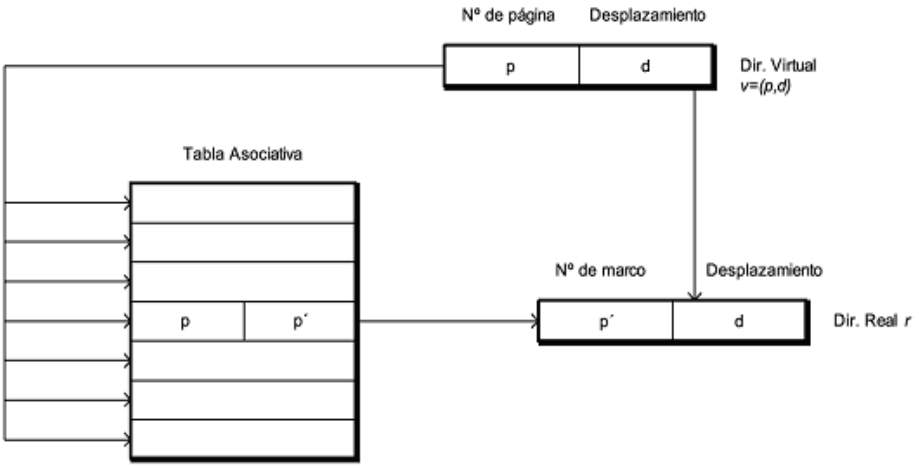

Figura 2.9. Traducción de direcciones por transformación asociativa

Una primera solución puede ser acelerar el acceso, colocando la tabla de páginas en una memoria asociativa en la que se acceda a todas las posiciones al mismo tiempo.

Sin embargo, dado el tamaño normal de las tablas de página, esta solución suele ser demasiado costosa, por lo que no se implementa en la realidad.

2.7.2.3. Traducción de direcciones por transformación asociativa-directa con TLB

Otra posible manera de acelerar las traducciones puede ser recordar las últimas traducciones incluyéndolas en una pequeña memoria asociativa de acceso muy rápido a la que se denomina TLB (Translation Lookaside Buffer o Buffer de traducciones anticipadas).

Esta técnica aprovecha el principio de localidad de los programas para obtener un rendimiento parecido al de la técnica anterior, pero con una memoria de tamaño muchísimo menor.

Para implementar la búsqueda usando una TLB hacemos uso del siguiente algoritmo:

- Extraemos el número de página p de la dirección virtual.
- Buscamos el número de página virtual dentro de la TLB:
    - Si se encuentra, accedemos al marco p' indicado de memoria principal y le sumamos el desplazamiento para obtener el dato o instrucción. A este caso lo denominamos "Acierto en TLB".
    - Si no se encuentra, debemos acceder a la tabla de páginas para obtener el marco p' correspondiente, usando la parte p de la dirección virtual para indexar la entrada correspondiente de la tabla de páginas. Si el bit de residencia *r* está a:
        - *0:* La página no está en memoria principal, de manera que debemos transferirla desde el almacenamiento secundario en la dirección s hasta un marco libre en memoria principal y modificar las entradas de la tabla de páginas y de la TLB. A este caso lo denominamos "Fallo de página".
        - *1:* La página está ya cargada en la memoria principal en el marco p'. Debemos entonces acceder a él y sumarle el desplazamiento d para obtener el dato o instrucción, y a su vez introducir la entrada en la TLB. A este caso lo denominamos "Acierto en tabla de páginas".

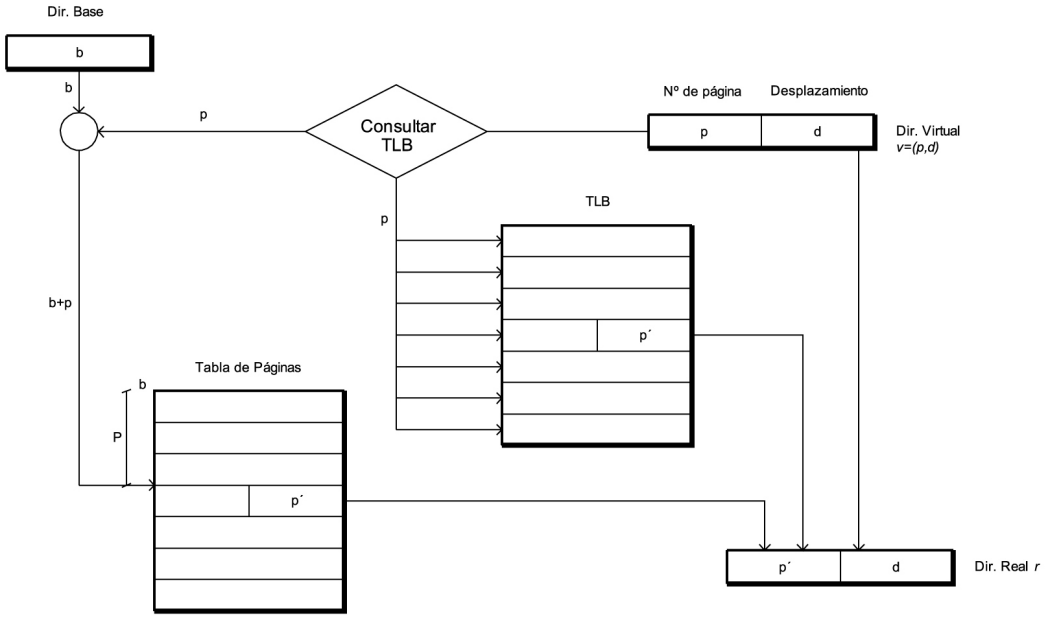

Figura 2.10. Traducción de direcciones por transformación asociativa-directa con TLB

Según el algoritmo anterior vemos que en los casos en que se produce "Fallo de página" o "Acierto en tabla de páginas" el tiempo efectuado en acceder a la TLB ha sido tiempo malgastado, por lo que debemos tratar de encontrar un tamaño de TLB adecuado a la localidad de los programas y la arquitectura concreta de la máquina.

Otro factor a tener en cuenta será la política de reemplazamiento que seguiremos cuando todas las entradas se encuentren llenas. En una TLB se pueden implementar políticas tales cómo:

- *Azar:* Se reemplaza una entrada al azar de la TLB.
- *FIFO (First in – First out o Primero en entrar – Primero en salir):* La entrada a reemplazar es la que lleva más tiempo en la TLB.
- *LRU (Least Recently Used o Menos recientemente usada):* La entrada a reemplazar es la que lleva más tiempo sin usarse.
- *LFU (Least Frequently Used o Menos frecuentemente usada):* La entrada a reemplazar es la que ha sido usada un menor número de veces.

2.7.2.4. Traducción de direcciones por transformación asociativa-directa con TLB dividida

Una posible mejora sobre el esquema anterior puede ser el uso de dos TLB, una para almacenar las traducciones correspondientes a datos y otra para almacenar

las correspondientes a instrucciones. De esta manera podrían aprovecharse mejor las diferentes localidades de referencia de las instrucciones y los datos.

El funcionamiento es similar al caso anterior, pero efectuando la búsqueda en una TLB u otra según se trate de instrucciones o datos.

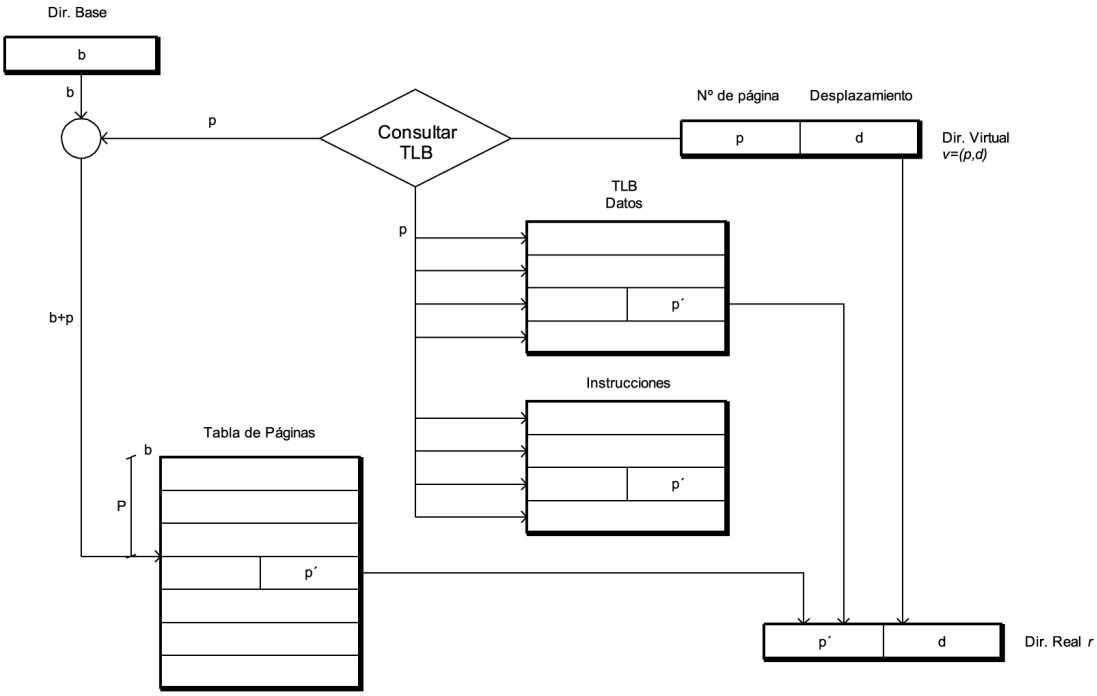

Figura 2.11. Traducción de direcciones por transformación asociativa-directa con TLB dividida

2.7.3. Estrategias de administración del sistema de memoria virtual paginado

A la hora de decidir que páginas entrarán en memoria principal y cuáles permanecerán en memoria secundaria se plantean el siguiente conjunto de estrategias:

- *Estrategias de búsqueda:* Deciden en que casos una nueva página debe ser traída desde memoria secundaria hasta memoria principal.

- *Estrategias de colocación:* Determinan en qué posición se colocará una nueva página en memoria principal.

- *Estrategias de reemplazamiento:* Deciden que página se debe trasladar desde memoria principal a memoria secundaria cuando es necesario el marco en memoria principal.

2.7.3.1. Estrategias de búsqueda

Para que un proceso pueda ejecutarse, es necesario que sus páginas se encuentren cargadas en memoria principal, y en los múltiples niveles de caché, si

estos existen. Por lo tanto, debe existir un mecanismo que determine qué páginas deben ser traídas desde memoria secundaria a memoria principal. Son las estrategias de búsqueda, que se dividen en:

- *Paginación por demanda:* Las páginas son trasladadas desde memoria secundaria a memoria principal cuando son referidas por el proceso en ejecución, presentando el problema de que al principio del programa el procesador debe esperar la transferencia página por página.

- *Paginación anticipada:* El sistema predice qué páginas deben ser trasladadas desde memoria secundaria a memoria principal sin que éstas tengan que ser referenciadas por el proceso en ejecución. Normalmente, para aprovechar la localidad de los programas, el sistema suele traer, además de la página referenciada:

    - La *Página anterior:* Es muy probable que la siguiente página referenciada sea la página anterior (bucles).

    - La *Página siguiente:* Es muy probable que la siguiente página referenciada sea la página contigua (datos contiguos).

    - Las *Páginas anterior y siguiente:* Es la combinación de las dos anteriores.

2.7.3.2. Estrategias de colocación

Cuando una página es trasladada desde memoria secundaria a memoria principal, debe determinarse en qué marco va a ser colocarla, de eso se encargan las estrategias de colocación.

Normalmente, puesto que al inicio de la máquina todas las posiciones de memoria están vacías, y cuando se llena son las estrategias de reemplazamiento las que determinan donde colocar la nueva página; se suele usar la estrategia *FIRST FIT* basada en ir colocando las páginas consecutivamente desde la posición de memoria más baja.

2.7.3.3. Estrategias de reemplazamiento

Normalmente no todas las páginas de un proceso caben en memoria principal, o el proceso en ejecución cambia y es necesario cargar todas sus páginas. Por ello, debe implementarse un mecanismo eficiente para gestionar los casos en los que todos los marcos de página se encuentran ocupados y es necesario trasladar una nueva página desde memoria secundaria hasta memoria principal. De decidir qué pagina reemplazar, se encargan las estrategias de reemplazamiento, que pueden ser de diferentes tipos:

- *Óptima:* Esta estrategia nos dice que el rendimiento óptimo se obtendría cuando la página reemplazada fuese aquella que no se va a utilizar

durante más tiempo en el futuro. Dado que los ordenadores no pueden conocer el futuro, no es una estrategia implementable en un sistema real.

- *Azar:* Esta estrategia decide la página a reemplazar al azar. No es la más eficiente, pero su simplicidad de implementación y ausencia de contador o sello de tiempo la hacen válida para ciertos sistemas.

- *FIFO (First in – First out o Primera en entrar – Primera en salir):* Esta estrategia reemplaza aquella página que ha estado durante más tiempo en memoria. Se implementa asociando a cada entrada un sello de tiempo que marca el instante en que la página ha entrado en memoria.

- *LRU (Least Recently Used o Menos recientemente usada):* Esta estrategia reemplaza aquella página que lleva más tiempo en memoria sin ser usada. Se implementa asociando a cada entrada un sello de tiempo que marca el último instante en que la página fue usada.

- *Clock:* Según su forma de implementación esta estrategia puede ser una aproximación a FIFO o a LRU.

    - *Aproximación a FIFO:* Reemplaza aquella página que ha estado durante más tiempo en memoria. Se implementa utilizando un puntero que determina que página debe ser reemplazada avanzando siempre en el sentido de las agujas del reloj, de ahí que se denomine "Clock", reloj en inglés.

    - *Aproximación a LRU:* Reemplaza aquella página que ha estado durante más tiempo en memoria sin ser usada. Se implementa, al igual que la aproximación a FIFO, utilizando un puntero que determina que página debe ser reemplazada avanzando siempre en el sentido de las agujas del reloj; pero además añade un bit que marca si la página ha sido usada desde la última vez que el puntero pasó por allí, para evitar que una página recientemente usada sea reemplazada.

- *LFU (Least Frecuently Used o Menos frecuentemente usada):* Esta estrategia reemplaza aquella página que ha sido usada el menor número de veces desde que ha entrado en memoria. Se implementa asociando a cada entrada un contador del número de veces que la página ha sido usada.

- *NUR (Not used recently o No recientemente usada):* Esta estrategia reemplaza aquella página que no ha sido usada recientemente. Para determinar esta situación utiliza dos factores: si la página ha sido referenciada y si la página ha sido modificada. Se implementa asociando a cada entrada una etiqueta de dos bits, lo que da lugar a cuatro posibles grupos de páginas para ser reemplazados.

    - Bit 1: Bit de referencia – (0 no referenciado y 1 referenciado)

- Bit 2: Bit de modificación – (0 no modificado y 1 modificado)

- 00 - Página no referenciada y no modificada
- 01 - Página no referenciada pero modificada
- 10 - Página referenciada pero no modificada
- 11 - Página referenciada y modificada

Las páginas son reemplazadas en este mismo orden.

2.7.4. Prestaciones del sistema de memoria virtual

Cómo se ha visto en esta sección, son muchos los mecanismos involucrados en un sistema de memoria virtual, de manera que las características de cada uno influyen en las prestaciones globales del sistema.

Algunas de las técnicas vistas mejoran sensiblemente las prestaciones, aprovechando sobre todo la localidad de los programas, pero añaden complejidad al hardware. De manera que los mecanismos elegidos para implementar un sistema real deben representar el equilibrio entre coste del hardware y prestaciones globales.

2.8. Un poco de historia

Aunque desde un principio se supo que disponer de enormes cantidades de memoria sería una necesidad, y que posiblemente las jerarquías de memoria fueran el camino para cubrirla, la implementación generalizada no se produjo hasta que las máquinas Atlas e IBM 360 demostraron su gran eficacia.

Desde entonces, sólo algunos superordenadores, máquinas embebidas o viejos ordenadores personales, prescinden de las jerarquías de memoria.

Hoy día, gracias a los avances en tecnología de procesadores, los niveles superiores de la jerarquía, la memoria caché de nivel 1 y de nivel 2, suelen ir integrados en la misma pastilla. Gracias a ello, pueden funcionar a la mitad o la misma velocidad que el núcleo del procesador, mientras que los siguientes niveles suelen añadirse en forma de módulos de memoria a la placa madre del sistema.

Son también muchos los avances que se han introducido para mejorar el ancho de banda y la latencia de la memoria principal, apareciendo cada año memorias más rápidas:

- Memorias SDR: desde 66 hasta 133Mhz
- Memorias RAMBUS: 800 y 1066Mhz
- Memorias DDR: desde 266 hasta 400Mhz
- Memorias DDR2: 533Mhz

También los dispositivos de memoria secundaria y buses de E/S están en plena evolución, con la generalización del almacenamiento en memorias flash de gran tamaño y el almacenamiento en unidades compartidas en redes de alta velocidad.

Para demostrar el uso masivo de las jerarquías de memoria en los sistemas actuales basta con observar la siguiente tabla:

	Intel Pentium 3	Intel Pentium 4	AMD Athlon	IBM PowerPC 405	Sun UltraSparc III
Juego de instrucciones	80x86			PowerPC	Sparc v9
Posibles aplicaciones	Sistemas de escritorio y servidores			Sistemas embebidos	Servidores
Velocidad reloj	Hasta 1.4Ghz	Por encima de 3Ghz	Por encima de 2Ghz	266Mhz	900Mhz
Caché L1 instrucciones	16Kb asociativa por cjtos de 2 vías	96KB Trace Caché	64Kb asociativa por cjtos de 2 vías	16Kb asociativa por cjtos de 2 vías	32Kb asociativa por cjtos de 4 vías
Caché L1 datos	16Kb asociativa por cjtos de 2 vías	8Kb asociativa por cjtos de 4 vías	64Kb asociativa por cjtos de 2 vías	8Kb asociativa por cjtos de 2 vías	64Kb asociativa por cjtos de 4 vías
TLB Datos / Instrucciones	32 / 64	128	280 / 288	4 / 8	128 / 512
Tamaño mínimo página	8 KB	8 KB	8 KB	1 KB	8 KB
Caché L2	256Kb asociativa por cjtos de 8 vías	256 / 2048 Kb asociativa por cjtos de 8 vías	256 / 512 Kb asociativa por cjtos de 16 vías	No tiene	8192 Kb mapeado directo
Caché L3	No tiene	En XEON hasta 2048 kb asociativa por cjtos de 8 vías	No tiene	No tiene	No tiene
Tamaño del bloque caché	32	64 / 128	64	32	32
Ancho banda memoria en bits	64	64	64	64	128
Velocidad memoria	133Mhz	400 a 1066 Mhz	100 a 400 Mhz	133 Mhz	150 Mhz

Tabla 2.2. Jerarquías de memoria en los sistemas actuales