Arquitectura de Computadores

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Arquitectura de Computadores

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán
Arturo Olivares Martos

Granada, 2023-2024

Índice general

1. Arquitecturas TLP

En este capítulo, nos centraremos en arquitecturas que permiten ejecutar de forma paralela o concurrente múltiples flujos de instrucciones (o *threads*) que comparten memoria. Se tratan de arquitecturas con paralelismo a nivel de *thread* (*Thread-Level Parallelism*) con una única instancia del Sistema Operativo. Por tanto, cada vez que mencionemos arquitecturas TLP, nos estamos refiriendo a arquitecturas TLP con una única instancia del Sistema Operativo. En este contexto, el SO es el encargado de gestionar los flujos de instrucciones.

Los paradigmas de programación paralela por variables compartidas son los más fáciles de implementar en este tipo de arquitecturas, mientras que las orientadas a paso de mensajes están más relacionadas con arquitecturas TLP con múltiples instancias del SO.

La compartición de memoria que se da trae conceptos como la coherencia del sistema de memoria, consistencia del sistema de memoria o la sincronización entre flujos; conceptos que estudiaremos a lo largo de este capítulo.

1.1. Tipos de Arquitecturas

1.1.1. Objetivos

Esta sección está orientada a:

- Distinguir entre cores multhread, multicores y multiprocesadores.
- Comparar entre cores multithread de grano fino, grueso y cores con multithread simultáneo.

1.1.2. Clasificaciones de arquitecturas TLP

Las arquitecturas TLP con una instancia del SO pueden clasificarse en:

Multiprocesadores. Son capaces de ejecutar en paralelo varios flujos de instrucciones (hilos) en un computador con varios núcleos o procesadores de forma que cada flujo se ejecuta en un núcleo o procesador distinto.

Podemos encontrarnos multiprocesadores en un chip (como los multinúcleos), en una placa o en uno o varios armarios.

Multinúcleos (*multicores*). Pueden ejecutar en paralelo varios flujos de instrucciones en un chip de procesamiento con múltiples núcleos de forma que cada flujo se ejecuta en un núcleo distinto. Un chip multinúcleo no es más que un multiprocesador en un chip.

La denominación de *multicores* proviene de un nombre comercial que dio Intel a sus multiprocesadores en un chip. Además, denominó *procesador* a los chips o encapsulados de procesamiento y *núcleos* (o *cores*) a los procesadores.

Núcleos (o cores) *multithread*. Se trata de un núcleo de procesamiento (un procesador) en el que se ha modificado su arquitectura ILP (*Instruction Level Parallelism*) para poder ejecutar flujos de instrucciones de forma concurrente o en paralelo.

1.1.3. Repaso de arquitecturas ILP

Recordamos lo que eran las arquitecturas ILP (*Instruction Level Parallelism*): son la capacidad por parte de un procesador de ejecutar múltiples instrucciones en paralelo para mejorar el rendimiento del sistema. Las arquitecturas se centran en identificar y aprovechar las dependencias de datos entre las instrucciones para ejecutarlas de forma eficiente.

Etapas de ejecucución

Antes de continuar, recordamos las etapas básicas para la ejecución de una instrucción:

Etapa de captación de instrucciones (*Instruction Fetch*).
Etapa en la que se capta de la caché de instrucciones la siguiente instrucción a ejecutar, incrementando el valor del PC(*Program Counter*).

Etapa de decodificación de instrucciones (*Instruction Decode*).
Etapa en la que se decofica la instrucción captada para determinar su tipo y operaciones a realizar. Se identifican aquí las dependencias de datos y condiciones de control que pueden afectar la ejecución de la instrucicón.

Etapa de ejecución (*Execution*).
Se lleva a cabo la ejecución de la instrucción. Podemos encontrarnos 4 tipos de instrucciónes (en relación a ellos se ejecutará una cosa u otra).

- Operaciones de enteros.
- Operaciones de coma flotante.
- Saltos.
- Escrituras o lecturas de memoria.

Aunque esta última no se realizaría en esta etapa, es importante para el desarrollo que vamos a hacer de arquitecturas ILP.

Etapa de acceso a memoria (*Memmory*).
Se accede a memoria en caso de que sea necesario (depende de la instrucción).

Etapa de almacenamiento de resultados en registros (*Write-Back*).
Se almacenan en los registros del procesador los resultados de la instrucción, si es necesario.

Formas de arquitecturas ILP

Podemos encontrarnos con dos formas principales de paralelizar estas etapas a la hora de desarrollar una estructura ILP:

Escalar segmentada.
Segmentaremos las etapas de ejecución de forma que dispongamos de 5 módulos que sean capaces de procesar su parte de forma paralela. Incluiremos *buffers* auxiliares entre las distintas etapas.

VLIW y superescalar.
Consideraremos simplemente 4 etapas (fusionaremos las de ejecución y memoria en una), de forma que replicaremos los componentes de la unidad funcional para que esta sea capaz de ejecutar al mismo tiempo diversos tipos de instrucciones. También es necesario el uso de *buffers* auxiliares.

Por ejemplo, podemos replicar los componentes de la unidad funcional de forma que podamos ejecutar en paralelo (al mismo tiempo):

- Operaciones con enteros.
- Operaciones de coma flotante.
- Operaciones de lecturas y escrituras en memoria.
- Saltos.

Podemos además tener no sólo una unidad sino varias de las ya mencionadas (2 unidades para operaciones con enteros, ...).

Además, las 3 etapas restantes estarán segmentadas. Podemos tener también unidades superescalares en las que tengamos replicadas además las etapas de captación, decodificación y *write-back*.

1.1.4. Clasificaciones de cores multithread

Ahora vamos a clasificar los *cores multithread*, que no dejan de ser procesadores con arquitectura ILP que se aprovechan para ejecutar a la vez distintos hilos del sistema operativo de forma concurrente o paralela.

***Temporal Multithreading* (TMT)**
Ejecutan distintos hilos de forma concurrente en el mismo core. De esta forma, emite instrucciones de un único hilo en cada ciclo. Podemos pensar que el core se está multiplexando.

La conmutación entre hilos la decide el hardware.

***Simultaneous MultiThreading* (SMT)**
Ejecuta distintos hilos de forma paralela en el mismo core. Pueden llegar a

emitir en un sólo ciclo instrucciones de varios hilos. Para llevar esto a cabo, necesitamos un core superescalar.

No implementa conmutación entre hilos, al no ser necesaria.

Según qué tan seguido intercambiamos los hilos del core, nos encontramos con:

TMT de grano fino (*Fine-grain multithreading*, FGMT).
La conmutación de hilos en el core se realiza en cada ciclo. Presentan un coste de cambio de contexto bajo, no es necesario perder ningún ciclo para realizar los cambios de contexto.

La planificación del siguiente hilo a ejecutar puede ser *round-robin* u otra técnica de planificación (podemos guiarnos por el hilo menos recientemente ejecutado, por accesos a datos, por saltos no predecibles, por operaciones con gran latencia, ...).

TMT de grano grueso (*Coarse-grain multithreading*, CGMT).
La conmutación entre hilos no se realiza en cada ciclo. Presentan un mayor coste por cambios de contexto: pueden perderse entre ninguno y varios ciclos debido a los cambios de contexto.

La planificación puede depender cualquier técnica, como tras intervalos de tiempos prefijados (*timeslice multithreading*), por eventos de cierta latencia (*switch-on-event multithreading*), ...

Clasificación de cores con CGMT con conmutación por eventos

Podemos conmutar los hilos del core de grano grueso de forma:

Estática.
Realizando la conmutación de forma *explícita*, mediante nuevas instrucciones para conmutación añadidas al repertorio; o de forma *implícita*, al detectar instrucciones e carga, almacenamiento, salto, ...

- Como ventaja, destacamos el bajo coste de los cambios de contexto (de 0 o 1 ciclos).
- Como inconveniente, pueden producirse cambios de contexto innecesarios.

Dinámica
La conmutación se realiza típicamente por fallos de caché o por interrupciones (interrupciones por señales).

- Como ventaja, reduce los cambios de contexto innecesarios de la estática.
- Como inconveniente, la sobrecarga que se añade por los cambios de contexto es mayor.

Hardware	CGMT	FGMT	SMT
Registros	Replicado (al menos el PC)	Replicado	Replicado
Almacenamiento	Multiplexado	Cualquiera de las 4	Multiplexado no
Hardware de etapas del cauce	Multiplexado el resto multiplexadas	Captación repartida o compartida, el resto repartidas o compartidas	Unidad funcional comp Replicado
Necesidad de distinguir el hilo de una instruc.	Sí	Sí	Sí
Hardware para conmutar	Sí	Sí	No

Característica

En un núcleo multithread, podemos usar las siguiente modificaciones con el objetivo de ejecutar varios hilos en un mismo core:

- Multiplexado: Hacemos que los hilos se turnen en el uso de una unidad.
- Repartición: Repartimos una unidad entre (al menos) dos hilos, de forma que a uno le asociamos su zona y al otro la suya.
- Compartición: Hay (al menos) dos hilos que acceden a la misma unidad de forma simultánea.
- Replicación: Disponemos de varias unidades, de forma que cada hilo puede hacer uso de una.

Notemos que tanto el precio de implementación como la bondad de la técnica se encuentran en orden creciente (siendo más caro y mejor realizar la replicación de unidades).

Una vez desarrollada la clasificación de cores multithread, podemos mostrar la siguiente tabla a modo de resumen, que nos ayudará a entender mejor cada tipo de multithreading: "CMP" es un Chip multicore, aquel en el que tenemos replicado todo el cauce (esto es, todas las unidades de la etapa de ejecución).

1.1.5. Comparativa de cores multithread

1.2. Coherencia del sistema de memoria

A la hora de usar multiprocesadores, surge un problema fundamental de manera natural: las incoherencias en memoria. A lo largo de esta sección definiremos este problema, dando ejemplo y protocolos que lo resuelven. Cabe mencionar que todos los multiprocesadores salvo los NUMA implementan la coherencia del sistema de memoria por hardware.

1.2.1. Objetivos

Tras esta sección, debería ser capaz de:

- Comparar los métodos de actualización de memoria principal implementados en caché.
- Comparar las alternativas para propagar una escritura en protocolos de coherencia de caché.
- Explicar qué debe garantizar el sistema de memoria para evitar problemas por incoherencias.
- Descibir las partes en las que se puede dividir el análisis o el diseño de protocolos de coherencia.
- Distinguir entre protocolos basados en directorios y protocolos de espionaje (snoopy).
- Explicar el protocolo de mantenimiento de coherencia de espionaje MSI.
- Explicar el protocolo de mantenimiento de coherencia de espionaje MESI.
- Explicar el protocolo de mantenimiento de coherencia MSI basado en directorios con difusión y sin difusión.

1.2.2. Definición del problema

La utilización de memoria caché trae consigo el esquema de jerarquía de memoria y la posibilidad de tener en distintas jerarquías una misma posición de memoria repetida. En sistemas uniprocesador, si tratamos de modificar una posición de memoria, esta se llevará a caché y será modificada en caché, indicando que ha sido modificada. En un cierto momento, la modificación en esta jerarquía de caché será comunicada a jerarquías de memoria mayores, hasta llegar a memoria principal y modificar dicho dato. De esta forma, la próxima vez que se necesite en caché dicha posición, estará actualizada conforme a la última modificación.

Recordamos ahora que cada procesador lleva asociada una memoria caché y, al ser mayor la memoria principal que la caché, necesitamos almacenar en algún sitio qué bloque de memoria principal contiene cada entrada de memoria caché. Esta información se almacena en una tabla asociada a la caché. En dicha tabla hay una entrada por cada dirección de la memoria caché y, en cada entrada, se almacena

0	Dirección de MP	Bits
1	Dirección de MP	Bits
	⋮	⋮
$N-1$	Dirección de MP	Bits

Tabla 1.1: Tabla para una caché de N direcciones de memoria.

qué bloque de memoria principal está cargado y unos bits de información sobre el bloque cargado. Representamos una aproximación a esta tabla en la Figura **??**. En un sistema uniprocesador, nos basta con un bit sucio (que indique si el bloque ha sido modificado o no, para saber si hay que escribir en jerarquías supsriores) y un bit de validez (que indique si el bloques es válido).

En un multiprocesador, cada procesador lleva consigo una memoria caché, de forma que todos los procesadores comparten el espacio de memoria. Puede suceder que dos procesadores distintos trabajen con la misma posición de memoria k (y por tanto, tengan al bloque que contiene a k en sus respectivas cachés). Si uno de los dos procesadores decide modificar k, se modificará la dirección de memoria caché correspondiente, pero el bloque en la caché del otro procesador y en memoria principal quedarán inalterados. Nos acabamos de encontrar con una incoherencia en el sistema de memoria.

Concretando las ideas, una **incoherencia en el sistema de memoria** se produce cuando en el sistema de memoria las copias de una misma dirección no tienen el mismo contenido. Como hemos ya comentado, en sistemas uniprocesdores teníamos este problema: podíamos tener un bloque en memoria caché modificado que no estuviese modificado en memoria principal. En este caso hablamos de incoherencias en distintas jerarquías de memoria. Sin embargo, ahora en multiprocesadores podemos tener incoherencias en la misma jerarquía de memoria, tal y como mencionábamos en el ejemplo anterior.

Las situaciones de incoherencia se deben abordar no permitiendo que se produzcan nunca o bien evitando que causen problemas (que algún componente lea el valor no actualizado de la memoria) en caso de permitirse. Las cachés implementan dos métodos de actualización de memoria principal:

Escritura directa (*write-through*).

Con escritura directa, no se permiten situaciones de incoherencia entre caché y memoria principal al escribir en caché: cada vez que se escribe en un bloque de la caché, el correspondiente bloque de la memoria principal es modificado.

Por tanto, tenemos una escritura en memoria principal por cada escritura, lo que requiere usar la red de comunicación entre procesador y memoria principal. Este sobreuso de la red empeora más aún en multiprocesadores, al tener múltiples procesadores que son susceptibles de modificar datos de forma paralela.

Otro problema que se plantea es desaprovechar los principios de localidad espacial y temporal (al modificar una variable, es posible que se modifique otra

próxima a ella, o que la ya modificada se vuelva a modificar próximamente), por lo que sería mejor esperar a que termine la modificación en curso (por ejemplo, si estamos iterando sobre un vector y modificando sus componentes), antes de escribir en memoria los cambios modificados.

Cabe destacar también que sí pueden producirse con escritura directa situaciones de incoherencia entre cachés (dos cachés tienen el mismo bloque y una lo modifica) y entre caché y memoria principal (cuando un componente modifica la dirección de un bloque de memoria principal que se encuentra en alguna caché del multiprocesador). En sistemas uniprocesadores, tenemos la ventaja de no necesitar un bit sucio.

Posescritura (*write-back*).
En posescritura, cuando una dato es modificado por un procesador, sólo se modifica en la caché correspondiente a dicho procesador. La actualización en memoria principal se produce cuando un bloque modificado (un bloque en el que se ha alterado una dirección de memoria mientras estaba en la caché) es retirado de la caché (por ejemplo, para hacer sitio a otro bloque más necesario). De esta forma, minimizamos el número de accesos a memoria y, por tanto, de uso de la red de conexión entre el procesador y la memoria; aprovechando los principios de localidad espacial y temporal.

De esta forma, se permiten situaciones de incoherencia entre caché y memoria principal (incluso en sistemas uniprocesador). También puede producirse cualquier tipo de incoherencia en este sistema, empeorando la situación que teníamos con escritura directa.

Es necesario además mantener la información sobre qué bloques han sido modificados en caché y cuales no. Esta labor la realiza un bit en la tabla de la caché, llamdo "bit sucio".

Lo usual en cachés es que usen posescritura, debido a sus ventajas frente a escritura directa. A continuación, estaremos hablando siempre de sistemas multiprocesadores, ya que en sistemas uniprocesadores las situaciones de incoherencia ya están resueltas (tanto con escritura directa no permitiendo incoherencias tanto con posescritura, permitiendo incoherencias pero controlando que no provoquen fallos).

1.2.3. Protocolos de coherencia entre cachés

Como ya habrás podido deducir, nuestra tarea ahora es buscar cómo resolver las incoherenias ya mencionadas, y manejar las incoherencias que se permiten para que no provoquen fallos en el sistema. Para evitar situaciones de incoherencias entre cachés, se deben cumplir las siguientes dos condiciones:

1. **Propaganción de escrituras.** Se debe garantizar que todo lo que se escribe en la copia de un bloque en caché se propague a las copias del bloque en otras cachés.

2. **Serialización de escrituras.** Se debe garantizar que las escrituras en una dirección se ven en el mismo orden por todos los procesadores: el sistema de

memoria debe parecer que realiza en serie las operaciones de escritura en la misma dirección. Debe dar la impresión de que estas operaciones sean atómicas.

Esto es de vital importancia en arquitecturas donde no todos los procesadores tienen el mismo (o similar) tiempo acceso a sus cachés, como los NUMA, debemos garantizar que el primer procesador que se dispuso a escribir en memoria sea el primero que lo haga.

Los **protocolos de coherencias de caché** buscan resolver el problema de las incoherencias entre cachés, de forma que cada escritura en caché sea visible para el resto de procesadores, propagando de forma fiable el valor escrito en una dirección. Los protocolos de coherencia usan dos alternativas principales para propagar escrituras a otras cachés:

Escritura con actualización (*write-update*).
Siempre que se modifique una dirección en la copia de un bloque en una caché, se modifica dicha dirección en las copias del mismo bloque de memoria que tengan el resto de procesadores en sus cachés (en caso de tenerlo). Si se cumplen los principios de localidad espacial y temporal, esto provoca una alta sobrecarga en la red de comunicación, al tener que avisar al resto de procesadores la modificación de una variable.

Escritura con invalidación (*write-invalidate*).
Antes de que un procesador modifique un bloque en su memoria caché, invalida el resto de copias del mismo bloque en las cachés de otros procesadores. Posteriormente, es libre de modificar su bloque tantas veces como desee, obteniendo un *acceso exclusivo* al bloque.

Cuando otro procesador quiera acceder a dicho bloque desde su caché, en caso de tenerlo, verá que estará invalidado y deberá solicitarlo a la memoria (en caso de que el bloque se encuentre actualizado en memoria) o al procesador que tenía el acceso exclusivo al bloque. Es necesario por tanto disponer de un bit en la tabla de la caché que indique si un bloque se encuentra invalidado o no.

Notemos que invalidar es más rápido que actualizar, ya que sólo necesitamos recibir la información del bloque invalidado y cambiar un bit, en lugar de reescribir el bloque completo.

Esta práctica sólo permite compartir bloques de memoria mientras sólo se lee del bloque.

Si escribimos varias veces sucesivas sin que otro procesador lea de su bloque correspondiente, con invalidación podemos reducir el número de accesos (transferencias) a la red. Por otra parte, si deseamos que un procesador escriba un dato para que el resto lo lean podría ser más eficiente usar escritura con actualización, que disminuye en este caso los accesos a la red (notemos que con invalidación necesitamos un acceso a la red por cada caché que falle, pero con actualización con un único acceso a la red podemos actualizar todas las cachés). Cabe destacar que esta ventaja de la actualización no se da en arquitecturas que no implementan difusión. Podemos decir que en general, la política de actualización genera un tráfico innecesario cuando los

datos compartidos se leen por pocos procesadores.

La propagación de las actualizaciones o invalidaciones entre los procesadores se puede realizar:

- Con una **difusión** de los paquetes de actualización o invalidación a todas las cachés.
- Con el envío de los paquetes de actualización o invalidación sólo a aquellas cachés con copias del bloque, que son sólo las que necesitan recibirlo. Es decir, realizar un **envío selectivo**.

  Para esta última alternativa, necesitamos mantener una tabla o directorio de memoria, que informe de las cachés que tienen copia de un determinado bloque para poder realizar la comunicación. Esta tabla tendría una entrada por cada dirección de memoria de cada caché y contendría el bloque que contiene, junto con bits de estado.

1.2.4. Protocolos de mantenimiento de coherencia

En una arquitectura UMA, podemos utilizar una red bus para las transferencias entre los procesadores y la memoria que comparten. Los buses implementan la difusión de forma natural. De esta forma, todos los paquetes enviados a la red son visibles por todos los componentes conectados al bus, de forma que toos ven las peticiones en el orden en el que se solicitan (dándose las dos condiciones para evitar situaciones de incoherencias). Todo esto hace que en el caso de los multiprocesadores UMA con red bus no sea necesarios mantener información de la cachés con copias de los bloques, pudiendo suprimir incluso el directorio de memoria del que antes se hablaba.

En este tipo de arquitecturas, es común el uso de **protocolos de espionaje** (*snoopy*), ya que todos los componentes pueden ver (espiar) dicho bus. Cada controlador de caché espía los paquetes del bus y actúa en consecuencia (si por ejemplo otro controlador solicita un paquete invalidado del que nuestra caché tiene acceso exclusivo y no está actualizado en memoria, nuestro controlador invalida la respuesta de la memoria y es él quien responde a la petición del paquete, devolviendo el paquete actualizado). Es por ello sencillo implementar protocolos de coherencia en una arquitectura que use buses. Sin embargo, los protocolos de espionaje no escalan bien debido a retardos que introducen otro tipo de redes[1].

En redes en las que la difusión es costosa de implementar o redes que requieren de una gran escalabilidad, se usan **esquemas basados en directorios**, en los que para reducir el tráfico, se envían los paquetes únicamente a las cachés implicadas (cachés con copia del bloque al que se accede). Es necesario por tanto el uso del directorio de memoria. Sin embargo, también obtenemos un cuello de botella al tener que acceder por cada procesador a dicho directorio. Se obtienen mejores prestaciones distribuyendo el directorio entre los módulos de memoria principal, de forma que el subdirectorio de cada módulo mantenga la información sobre sólo los bloques que

[1] al tener que esperar a que todos los procesadores reciban el paquete enviado.

contiene cada módulo. De esta forma, los diferentes subdirectorios pueden procesar peticiones en paralelo.

Cabe destacar que no son los únicos tipos de protocolos de coherencia, sino que hay también esquemas organizados en **jerarquía**, compuestos de protocolos de espionaje y directorios, que dependen de la red utilizada en cada nivel.

Resumiento toda esta introducción a los protocolos de coherencia realizada, debemos tener claro que para diseñar un protocolo de mantenimiento de coherencia, debemos planificar:

- La política de actualización en memoria principal: escritura directa o posescritura.
- La política de propagación de escrituras entre cachés: escrituras con actualización o con invalidación.
- El comportamiento:
  - Los posibles estados de un bloque en caché y las acciones que el controlador de caché debe realizar ante eventos recibidos.
  - Los posibles estados de un bloque en memoria principal, junto con las acciones que el controlador de memoria principal debe hacer ante eventos.
  - Los paquetes que genera el controlador de memoria principal.
  - Saber relacionar las acciones con eventos y estados.

A continuación, vamos a estudiar cuatro protocolos de mantenimiento de coherencia, dos para multiprocesadores UMA con red bus (MIS y MESI) y dos para multiprocesadores NUMA en una placa (MSI con y sin difusión). Todos ellos usan posescritura y escrituras con invalidación, como cabría esperar.

Los protocolos de espionaje que estudiaremos

1.2.5. Protocolo MSI (*Modified-Shared-Invalid*) de espionaje

1.3. Consistencia del sistema de memoria

1.3.1. Objetivos

Esta sección está orientada a adquirir los conocimientos necesarios para:

- Explicar el concepto de consistencia.
- Distinguir entre coherencia y consistencia.
- Distinguir entre el modelo de consistencia secuencial y los modelos relajados.
- Distinguir entre los diferentes modelos de consistencia relajados.

1.4. Sincronización

1.4.1. Objetivos

Tras esta sección, debería ser capaz de:

- Explicar por qué es necesaria la sincronización en multiprocesadores.
- Describir las primitivas para sincronización que ofrece el hardware.
- Implementar cerrojos simples, cerrojos con etiqueta y barreras a partir de instrucciones máquina de sincronización y ordenación de accesos a memoria.

2. Relaciones de Problemas