Arquitectura y Organización de Computadores

Bibliografía:

- Universidad de Murcia. Diapositivas de Arquitectura y Organización de Computadores.
- Wikipedia, the Free Encyclopedia (https://en.wikipedia.org/). *Dark silicon, SPMD*, *Processor consistency*, *Weak consistency*, *Release consistency*, *Load-link/store-conditional*, *Hypergraph*.
- Sarita Adve, Mark Hill, Barton Miller, Robert H. B. Netzer (1991). *Detecting Data Races on Weak Memory Systems*. ACM SIGARCH Computer Architecture News. 19. 234-243.
- MIPS Technologies Inc. (2008). *MIPS32® Instruction Set Quick Reference* (https://www.cs.duke.edu/courses/fall13/compsci250/MIPS32_QRC.pdf).

Capítulo 1

Introducción

Llamamos **arquitectura** de un computador a:

1. La **ISA** (*Instruction Set Architecture*), el conjunto de instrucciones del procesador. Normalmente hay una ISA base, como x86, ARM o RISC-V, con extensiones.
2. La **microarquitectura**, la implementación de cada instrucción de la ISA.
3. El diseño del *hardware*.

Llamamos **tamaño característico** (*feature size*) al tamaño mínimo de un transistor en un chip. Fue reduciéndose de 10 μm en 1971 a 10 nm en 2019. Así, la **ley de Moore** afirma que se podrá construir un chip con el doble de transistores cada 18–24 meses, y la **ley de Dennard** (*Dennard scaling* o *MOSFET scaling*) afirma que a medida que disminuye el tamaño de los transistores, la densidad de potencia consumida permanece constante.

Al disminuir el tamaño característico, la frecuencia de reloj puede aumentar de forma más o menos proporcional porque los transistores cambian de estado más rápido; la **densidad de integración**, o densidad de transistores, aumenta cuadráticamente, y en general se puede aumentar proporcionalmente el tamaño del chip. Estos transistores se usan para aumentar los recursos del chip mediante paralelismo y cachés, por lo que los programas limitados por CPU aumentaban su velocidad de forma cuadrática sin cambios. Por ejemplo, la empresa de procesadores Intel evoluciona sus diseños en dos pasos: una fase *tick* en la que disminuye el tamaño de los componentes, optimiza el diseño de los núcleos y aprovecha el aumento en densidad de transistores para conseguir mejor rendimiento y eficiencia energética, y una fase *tock* en la que diseña una nueva arquitectura y amplia la ISA.

Actualmente esto plantea varios problemas.

1.1. Problemas

1.1.1. Potencia

La energía debe ser traída al chip mediante pines, distribuida entre las distintas capas y disipada para evitar el sobrecalentamiento. La **potencia de diseño térmico** (**TDP**, *Thermal Design Power*) es la usada como objetivo para la fuente de alimentación y el sistema de

refrigeración. Se divide en **potencia estática**, producto del voltaje, la intensidad de potencia de fuga por transistor y el número de transistores, y **potencia dinámica**, proporcional al nivel de uso, el cuadrado del voltaje y la frecuencia de reloj.

Inicialmente la potencia estática era casi despreciable. Actualmente, al no poder bajar el voltaje por debajo de unos 0,5 V, la potencia dinámica sigue aumentando al aumentar la frecuencia y la potencia estática aumenta aun más rápido porque se sigue aumentando el número de transistores, por lo que la potencia estática es en torno al 25–50 % de la total. Además, deja de cumplirse la ley de Dennard y surge el ***dark silicon***, un límite en el área total de un chip que se puede encender a la vez sin superar la TDP.

La **densidad de potencia** es la potencia disipada por unidad de superficie, y ha ido aumentando, pero el **rendimiento por vatio** también ha aumentado mucho. Las soluciones de enfriamiento para alta densidad de potencia son caras o poco prácticas, por lo que se busca reducir la potencia con técnicas como:

1. Reducir la frecuencia de reloj dinámicamente.
2. **DVFS** (*Dynamic Voltage Frequency Scaling*): Reducir el voltaje o la frecuencia de zonas que no se estén usando.
3. Apagar selectivamente núcleos del procesador.
4. Estados de bajo consumo en RAM, discos duros, etc.

1.1.2. Paralelismo

Disminuir el CPI requiere aumentar el bus de datos y usar segmentación, pero a partir de 10 etapas aparecen muchos conflictos, por lo que se replica el cauce usando ejecución superescalar (varias instrucciones empezando y terminando a la vez) fuera de orden y especulativa. Estas técnicas permiten conseguir **ILP** (*Instruction-Level Parallelism*), en que varias instrucciones se ejecutan a la vez consiguiendo superar la instrucción por ciclo, pero requiere mucho *hardware* y por tanto consume mucha energía.

1.1.3. Acceso a memoria

La mejora en los procesadores es más rápida que en la memoria, y el bus entre ambas tiene muy alta latencia. La ejecución fuera de orden solo puede mitigar este problema parcialmente.

1.1.4. Fiabilidad

Conforme los diseños de procesadores se hacen más grandes es más difícil comprobar su corrección. Además, al reducir el tamaño de los componentes, estos son más sensibles a partículas cargadas en el ambiente capaces de cambiar el valor de los bits, por lo que se crean memorias y cachés con códigos de corrección (**ECC**, *Error Correction Codes*). Surgen problemas de impedancia y capacitancia, y como los transistores reciben y alteran la señal de reloj, al aumentar su número puede haber modificaciones perceptibles en dicha señal, el **sesgo de reloj** o ***clock skew***.

Podemos medir la fiabilidad en **tiempo medio hasta un fallo** (**MTTF**, *Mean Time To Failure*), **tiempo medio para reparaciones** (**MTTR**, *Mean Time To Recover*), **tiempo**

medio entre fallos (**MTBF**, *Mean Time Between Failures*, la suma de los dos anteriores) y **disponibilidad** (el MTTF entre el MTBF).

1.2. Medidas de rendimiento

Podemos medir el rendimiento de un procesador por el tiempo de ejecución de algún programa o su productividad o *throughput* (operaciones de algún tipo por unidad de tiempo). Se puede usar el tiempo de ejecución real, que incluye la sobrecarga de todo el sistema, o el tiempo de CPU usado para el cálculo. La «**aceleración**» o *speedup* de un procesador o un programa respecto a otro (funcionalmente equivalente) es el tiempo de ejecución del segundo entre el del primero. El **ancho de banda** es el trabajo total realizado en un tiempo determinado, y la **latencia** o **tiempo de respuesta** es el tiempo entre el inicio y el final de una operación.

Los ***benchmarks*** son programas usados para comparar el rendimiento. Hay de varios tipos: *kernels* (como multiplicación de matrices), programas de juguete (como ordenación de elementos), *benchmarks* sintéticos (como Dhrystone) o suites de *benchmarks* como SPEC06fp o TPC-C.

La **intensidad aritmética** es el número de operaciones en coma flotante por byte u operando leído de memoria para un cierto algoritmo. El **modelo de rendimiento *Roofline*** consiste en medir el rendimiento de coma flotante en **FLOPS** (operaciones de punto flotante por segundo) en función de la intensidad aritmética, midiendo así tanto el ancho de banda de memoria como el rendimiento de punto flotante.

Algunos aspectos relevantes son la frecuencia de reloj, el número de transistores, el tamaño de la caché de cada nivel, la cantidad de memoria direccionable, el ancho de banda del bus de memoria, el número de núcleos por chip, el tamaño de la litografía y el tamaño de los operandos.

1.3. Paralelismo

Michael J. Flynn clasifica las arquitecturas en:

- **SISD** (*Single Instruction Single Data*): Procesador secuencial sin paralelismo.
- **MISD** (*Multiple Instruction Single Data*): Operaciones secuenciales pero redundantes, útil para sistemas de emergencia.
- **SIMD** (*Single Instruction Multiple Data*): Operaciones **vectorizadas**, en que una sola instrucción ejecuta la misma operación sobre varios datos. Es la usada en GPUs.
- **MIMD** (*Multiple Instruction Multiple Data*): Varios procesadores o **núcleos** realizando operaciones distintas en datos distintos.

1.3.1. Multiprocesadores

Son procesadores con **arquitecturas multinúcleo** (MISD o MIMD). Un **CMP** (***Chip Multiprocessor***) es un multiprocesador en un solo chip.

Usar varios núcleos pequeños en vez de uno grande hace más fácil verificar el diseño y permite reducir el voltaje y la frecuencia con el mismo rendimiento, con lo que la cantidad de

transistores útiles es mayor. Además, se mitigan los problemas eléctricos y, como cada núcleo tiene su propio reloj, desaparece el sesgo de reloj.

Así, la frecuencia de reloj de los procesadores no sigue aumentando, de hecho disminuye, pero la ley de Moore se reinterpreta como que el número de núcleos por chip se puede duplicar cada 2 años.

1.3.2. Acceso a memoria

En un procesador MIMD, la memoria puede ser:

- **Compartida**: Un único espacio de direcciones al que tienen acceso todos los procesadores, y que puede usarse para la comunicación entre estos. El acceso a memoria puede hacerse con un bus compartido o con una red más directa como una malla.
- **Distribuida**: Cada procesador tiene su propia memoria local, y la comunicación se hace por instrucciones de paso de mensajes.

Los clústeres y centros de datos usan memoria distribuida entre los ordenadores, aunque cada ordenador suele tener varios núcleos con memoria compartida y vectorización SIMD.

1.3.3. Límites del paralelismo

Todos los programas paralelos tienen secciones secuenciales, debido a la duplicación de trabajo entre varios procesadores o a la espera del resto de procesadores a que uno acabe. La **ley de Amdahl** afirma que, si en un proceso una parte que ocupa una porción p del tiempo total se hace S veces más rápido, el proceso en total se hace

$$\frac{1}{(1-p)+\frac{p}{S}}$$

veces más rápido. En particular, como las partes paralelas se hacen tantas veces más rápido como el total de procesadores, las partes secuenciales limitan la efectividad del paralelismo con una cota superior inversa a la fracción de código secuencial.

En la práctica nunca se llega al rendimiento obtenido por la ley de Amdahl porque los procesadores tienen que comunicarse. La **ley de Gustafson** afirma que, si un programa se ejecuta en N procesadores iguales con un tiempo secuencial s y un tiempo paralelo en cada uno p, la aceleración por haberlo paralelizado es

$$\frac{s+Np}{s+p},$$

con lo que un problema con parte paralela lo suficientemente grande puede ser paralelizado eficientemente.[1]

Esto cambia los objetivos hacia la formulación de problemas para poder solucionar casos mayores en la misma cantidad de tiempo, de modo que a veces es preferible aumentar la cantidad total de cálculos a cambio de disminuir la parte secuencial.

[1] Realmente esto es lo mismo que la ley de Amdahl y la misma conclusión se puede obtener directamente de la ley de Amdahl, pero a los ingenieros les gusta pensar que es otra cosa así que así sea.

1.3.4. Software paralelo

Podemos usar **paralelismo de datos**, en que se ejecuta una misma tarea en varios datos a la vez, y **paralelismo de tareas**, en que se ejecutan distintas tareas a la vez. No obstante, hay que tener en cuenta que comenzar un hilo o proceso, comunicar datos compartidos, sincronizar o hacer computación redundante tiene un coste en el rango de milisegundos.

Debemos balancear la carga para evitar que haya núcleos rápidos esperando a otros lentos, y gestionar los recursos compartidos de forma segura, considerando estas cuestiones a la hora de decidir la granularidad de las tareas y modelar el rendimiento.

1.4. Tipos de ordenadores

Distinguimos:

1. **Ordenadores empotrados**, usados para controlar dispositivos; pequeños y baratos.
2. **Móviles**, *tablets*, etc. Con más potencia pero enfocados en la eficiencia energética y una baja latencia.
3. **Ordenadores de escritorio**: Enfocados en la relación rendimiento/precio.
4. **Servidores**: Enfocados en la disponibilidad, la escalabilidad y el rendimiento.
5. ***Clusters***: Enfocados en disponibilidad y el rendimiento/precio.
6. **Supercomputadores**: Con gran velocidad en punto flotante, red de interconexión y consumo de energía. Se usan para ciencia, ingeniería, negocios y defensa.

Se considera que el primer supercomputador es el CDC 6600, aunque el término se acuñó para el Cray-1, un ordenador con arquitectura de carga y almacenamiento sin caché de datos ni memoria virtual y con control cableado, pero con registros e instrucciones vectoriales, unidades funcionales muy segmentadas y varios bancos de memoria.

Capítulo 2

Arquitecturas superescalares

Para reducir por debajo de 1 los CPI de un núcleo del procesador, debemos lanzar y ejecutar varias instrucciones en cada ciclo de reloj, pero esto implica más presión sobre la memoria y los registros, más posibilidad de riesgos, más área de silicio y más consumo. Distinguimos:

- **Arquitecturas superescalares**: Lanzan un número variable de instrucciones por ciclo, de 0 a 8.
- **VLIW** (***Very Long Instruction Word***): En cada ciclo se carga una **macro-instrucción**, formada por varias instrucciones independientes agrupadas por el compilador. El *hardware* es más simple, lo que permite una mayor frecuencia de reloj y un menor consumo, pero no se beneficia de las técnicas de planificación dinámica y el código suele ser más grande e incompatible entre distintas versiones del procesador, por lo que actualmente está casi en desuso.

2.1. Planificación

Aunque la mayoría de compiladores reordenan las instrucciones para intentar evitar riesgos, estos todavía pueden producirse.

Un núcleo con arquitectura superescalar tiene planificación **estática** si, cuando encuentra un riesgo, detiene el cauce hasta que el riesgo desaparece, o planificación **dinámica** si reordena las instrucciones para evitar detenciones, lo que aumenta la velocidad y reduce la necesidad del compilador de conocer la microarquitectura a cambio de necesitar más recursos. Con planificación dinámica, la ejecución es fuera de orden, y la terminación puede ser en orden o fuera de orden.

Si la ejecución es fuera de orden, distintas instrucciones pueden producir excepciones a la vez, puede pasar bastante tiempo desde que se produce una excepción hasta que se reconoce y una instrucción puede producir una excepción después de que otra posterior se haya ejecutado. Las excepciones son **precisas** si, cuando se reconocen, el contador del programa apunta a una instrucción, dicha instrucción provocó la excepción si esta es generada por el procesador, las instrucciones anteriores se han completado y las posteriores no han modificado el estado visible del procesador, y son **imprecisas** en otro caso. Hoy en día las excepciones imprecisas

son inviables, y la mayoría de procesadores implementan excepciones precisas impidiendo que una instrucción cambie el estado si hay instrucciones previas sin terminar.

Los núcleos superescalares normalmente tienen un sistema de memoria de alto rendimiento y unidades funcionales redundantes, que administra con técnicas como planificación dinámica, ejecución especulativa para mitigar riesgos de control y renombrado de registros para mitigar las dependencias de nombre.

2.2. Algoritmo de Tomasulo

El **algoritmo de R. M. Tomasulo** de 1971 es un método de planificación dinámica con renombrado de registros. En su versión con terminación en orden, usa:

- Una **cola de instrucciones** a decodificar, que también suaviza los fallos de caché.
- Un ***buffer* de re-ordenación** (**ROB**, *Re-Order Buffer*), un *buffer* FIFO circular que almacena las instrucciones en curso[1] y, cuando terminan de ejecutarse, también el valor calculado por ellas si lo hay, así como las excepciones que surjan.
- Un **estado de los registros**, que indica si el último valor de cada registro se debe tomar del ROB y, en tal caso, de qué entrada.
- **Estaciones de reserva** (**ER**): Tablas asociadas a un grupo de unidades funcionales con las instrucciones por ejecutar por dicho grupo, dados por un código de instrucción, la entrada del ROB correspondiente y, para cada operando, su valor o la entrada del ROB que lo producirá.
- ***Buffer* de carga**: Tabla con una serie de direcciones de memoria a cargar junto con la entrada del ROB en la que cargarla.
- **Bus de resultados** (**CDB**, *Common Data Bus*): Para transmitir los resultados al ROB y los ERs, implementando el adelantamiento.

Las instrucciones se segmentan en 6 etapas:

1. **Obtención de instrucciones** (**IF**, **F**, *Instruction Fetch*): Lee varias instrucciones por ciclo en orden y las añade a la cola de instrucciones. La predicción de saltos se puede hacer aquí o en ID.
2. **Decodificación** (**ID**, **D**, *Instruction Decode*): Decodifica varias instrucciones por ciclo para conocer su tipo, en orden. Si hay una entrada libre en el ROB y una en el ER para la instrucción, añade la instrucción al ROB y al ER. Para cada operando, carga el valor en el ER desde el ROB o el banco de registros según indique el estado del registro o, si este no ha sido calculado, el número de la entrada del ROB donde estará. Finalmente guarda el número de la entrada del ROB en el estado del registro de destino, si lo hay.

   Los algoritmos con terminación fuera de orden usan una **ventana de instrucciones** en vez de un ROB.

[1] Si el procesador usa micro-instrucciones, serán estas las que se añadan al ROB.

3. **Emisión** (***Issue***): Fuera de orden. Para cada ER, si un operando no está disponible, espera a que llegue por el CDB para guardarlo en la ER. En otro caso, si hay una unidad funcional disponible, le envía los operandos para que ejecute la instrucción, y si no la hay espera. Hay que asegurar que dos ERs no puedan enviar a la misma unidad funcional a la vez. Como mejora, se pueden usar técnicas como la predicción de valor o la reutilización de valores. La instrucción se emite a la unidad funcional en el ciclo siguiente en que el dato que faltaba se escribe en el CDB, pero si la unidad no está disponible, puede emitirla en el primer ciclo en que está disponible (dejándola ociosa ese ciclo) o en el anterior según la calidad del procesador.

4. **Ejecución** (**EX**, **X**, *Execute*): Fuera de orden, preferiblemente con varias unidades funcionales de cada tipo totalmente segmentadas, y llevando la cuenta de cuál es el destino de la instrucción. Los almacenamientos solo calculan la dirección. Las cargas se dividen en las etapas X1, en que se calcula la dirección de memoria, y X2, en que se accede a memoria, separadas por el *buffer* de carga. Antes de una carga, se debe esperar a que los almacenamientos anteriores en el ROB terminen, o a que se conozca su dirección de memoria y puede que su valor según si la dirección es o no la misma, para evitar riesgos RAW de memoria.

5. **Pos-escritura** o **escritura** (**WB**, **W**, *Write-Back*): Se difunden los resultados al **bus de resultados**, actualizando el ROB y las ER, y se libera la ER. Los almacenamientos no hacen nada.

6. **Confirmación** (***Commit***): Toma varias instrucciones del ROB por ciclo, en orden. Si una causó una excepción o es un salto especulado[2], vacía el ROB con las instrucciones siguientes, guarda el contador de programa para la siguiente instrucción e invoca a una rutina de manejo de excepciones. En otro caso, si el estado del registro de destino indica la misma entrada del ROB, escribe el resultado al banco de registros y borra el estado del registro; si es un almacenamiento, escribe el dato en memoria, en una dirección calculada en X y normalmente a través de un *buffer* de almacenamiento, y finalmente borra la entrada del ROB.

2.3. Ejecución especulativa

La predicción de saltos es **estática** si siempre predice el mismo resultado para el mismo salto o **dinámica** en otro caso. Algunos mecanismos de predicción dinámica de saltos son la predicción básica de n bits, la predicción con correlación y la predicción híbrida o de contienda. También se suele usar un ***buffer*** **de destino de saltos** (**BTB**, *Branch Target Predictor*), que almacena la dirección de destino de saltos previos para poder cargar las instrucciones de dichas direcciones directamente.[3]

Si la longitud del *pipeline* aumenta, la penalización por fallo de predicción de saltos aumenta al tardar más ciclos en detectar un fallo de predicción, y si el ancho del procesador aumenta, llegan más instrucciones de salto por ciclo (hay una medida de un salto por cada 8 instrucciones). Por ello, si la predicción de saltos es mala, el rendimiento se ve muy afectado.

[2]Se sabe que esto puede permitir ataques *side-channel*.
[3]Se sabe que esto puede permitir ataques *side-channel*.

Para reducir la latencia de memoria se pueden reordenar las instrucciones que acceden a memoria. Si una carga produce un fallo de caché, se pueden ejecutar cargas posteriores, pero en principio, en cualquier caso, no se puede ejecutar una carga hasta que se haya calculado la dirección de todos los almacenamientos anteriores y ninguno de los que están pendientes de confirmar tiene la misma dirección que la carga.

Para mitigar los riesgos de datos, se suele permitir la ejecución especulativa de las cargas aunque no se haya calculado la dirección de los almacenamientos anteriores. Cuando se sabe la dirección de un almacenamiento, se compara con las direcciones de las cargas posteriores y, si hay coincidencia, se descartan los resultados de la carga y las instrucciones posteriores.

2.4. Procesamiento multihilo

Aprovechar al máximo las unidades funcionales de un núcleo con un solo hilo es difícil, por lo que se usa el **paralelismo a nivel de hilo** (**TLP**, *Thread-Level Parallelism*) y se permite ejecutar varios hilos en el mismo núcleo.

Tradicionalmente se reparten los ciclos entre los hilos según una **política de ejecución**:

- **Entrelazado fijo**: Cada hilo toma un ciclo y ejecuta una instrucción, por turnos. Si un hilo no está listo, se pone una **burbuja** en el cauce, un ciclo en que no se hace nada.
- **Entrelazado controlado por el sistema operativo**: El sistema asigna una cantidad de *slots*, mayor al número de hilos, y los reparte entre los hilos disponibles en cada momento.
- **Entrelazado controlado por *hardware***: El *hardware* mantiene una lista de los hilos en ejecución y elige el siguiente a ejecutar según un esquema de prioridades.

El **multihilo simultáneo** (**SMT**, ***Simultaneous Multi-Threading***) permite ejecutar varios hilos en el mismo ciclo, usando uno los recursos que no usa el otro, especialmente unidades funcionales.[4]

El sistema operativo debe manejar más hilos, y hay más conflictos de caché y TLB salvo que se aumente el tamaño. Cada hilo necesita sus propios registros, tanto de propósito general como el contador de programa, el registro para la tabla de páginas y los de manejo de excepciones, por lo que se debe propagar el identificador de hilo por todo el cauce para saber qué registros usar. Se replican las tablas de renombrado de registros, pero se comparten el ROB, las cachés y las tablas de predicción de saltos.

El SMT se puede adaptar para que cuando hay alto TLP el ancho de emisión se comparta entre los hilos y cuando hay poco no se reparta y se dedique al ILP.

2.5. Vectorización

Las aplicaciones multimedia, como de realidad virtual, comunicaciones, procesamiento de imágenes, reconocimiento de voz, etc., así como la compresión y el cifrado, requieren cálculo intensivo, a menudo con restricciones de tiempo real, y es común que apliquen la misma operación a todos los elementos de una lista.

[4] Se sabe que esto puede permitir ataques *side-channel*.

Los procesadores de propósito general no están optimizados para estas aplicaciones, por lo que surgen las **extensiones multimedia** con instrucciones SIMD para explotar el paralelismo de datos con coste mínimo de recursos del chip. En la arquitectura IA-32[5] de Intel y AMD, se crean:

1. **MMX** (*Multi-Media Extensions*): Operaciones con 8 enteros de 8 bits o 4 de 16 bits.
2. **SSE** (*Streaming SIMD Extensions*), SSE2 y SSE4.2: 16 enteros de 8 bits, 8 de 16 bits, 4 de 32 bits, 4 números de punto flotante de 32 bits o 2 de 64 bits.
3. **AVX** (*Advanced Vector Extensions*) y **AVX2**: 4 números enteros o de punto flotante de 64 bits.
4. **AVX-512**: 8 números enteros o de punto flotante de 64 bits.

Intel también crea **IMCI** (*Intel Many-Core Instructions*) para la primera generación de Intel Xeon Phi, con registros de 512 bits. En cualquier caso, los operandos deben estar en posiciones de memoria consecutivas y alineadas al tamaño del registro.

Esto es como lo que hacen los procesadores vectoriales en las GPUs, aunque en general con modos de direccionamiento más sofisticados e instrucciones para mover datos de una parte del registro a otra. Muchas extensiones SIMD soportan instrucciones como `FMA` (*Fused Multiply-Add*), que multiplica dos registros y suma otro al resultado elemento a elemento.

Los compiladores suelen intentar usar instrucciones SIMD para los bucles más internos del programa, pero el programador debe elegir el compilador y las opciones adecuados, reordenar el código para hacer más visible lo que ocurre y, si es necesario, usar *intrinsics*, funciones que el compilador trata de forma especial, o escribir en ensamblador.

Vectorizar es convertir bucles con instrucciones escalares para que usen instrucciones SIMD, y lo suele hacer el compilador tras asegurarse de que esto no modifica los resultados del cálculo, viendo si una iteración accede a los datos producidos en iteraciones previas.[6]

[5] Actualmente obsoleta.

[6] Cuando hay un registro de acumulación, por ejemplo al sumar una lista de números, este se suele convertir a un registro vectorial con varios acumuladores a los que luego se les aplica una reducción. Esto no se puede hacer en punto flotante porque las operaciones de punto flotante no son conmutativas ni asociativas, pero muchos compiladores tienen opciones para no respetar el estándar en este caso y tratar las operaciones como conmutativas y asociativas. Ejemplos son `gcc` (*GNU C Compiler*) con la orden `-ffast-math` y el software privativo `icc` (*Intel Compiler Collection*), solo para arquitecturas Intel, que nunca respeta el estándar.

Capítulo 3

Sincronización entre núcleos

Para aprovechar los recursos de un multiprocesador se suele usar el modelo **STMD**[1] (*Single Thread, Multiple Data*) o **SPMD** (*Single Program, Multiple Data*), una forma de MIMD consistente en ejecutar el mismo programa con distintas entradas en varios procesadores a la vez para obtener resultados más rápido.

3.1. Cachés

ETC

Políticas de escritura

- **Escritura directa** (*write through*): Las escrituras se hacen a la vez en la caché y en memoria. Si el bloque no está en caché, se suele escribir directamente [...] en memoria [...] (*no write allocate*), [...] también se puede traer el bloque a la caché (*write allocate*) [...]
- **Pos-escritura** (*write back*): Las escrituras se hacen sólo en la caché, y sólo se actualiza la [...] memoria al sacar el bloque [...]. Es necesario un **bit de modificación** [...] en cada bloque [...].

En la práctica se usan **cachés multinivel**, con niveles normalmente de L1 a L3, siendo L1 la caché más cercana al procesador y con menor latencia pero menor capacidad. La **política de inclusión** indica si los datos en un nivel de caché están también en los niveles inferiores (de mayor número), y según esta la caché puede ser:

- **Inclusiva**: Todas las líneas en una caché están en las cachés de niveles inferiores, por lo que comprobar si un procesador tiene en su caché una copia de una línea solo requiere comprobar la caché externa.
- **Exclusiva**: Cada línea está en un único nivel de caché, y cuando un nivel requiere una línea de caché, intercambia la línea con la caché inferior.

[1] Nadie lo llama así.

- **Ni inclusiva ni exclusiva** (**NINE**, *Non-Inclusive Non-Exclusive*): Las líneas de un nivel de caché pueden estar o no en el nivel inferior.

Los fallos de caché pueden ser:

- **Obligatorios**, el primer acceso a una línea. Ocurrirían incluso con una caché infinita, aunque se pueden mitigar con *prefetching*.
- **De capacidad**, cuando la línea se expulsó previamente porque la caché no es lo suficientemente grande. Ocurrirían aun si la caché fuera totalmente asociativa.
- **De conflicto**, debidas a la falta de asociatividad. El resto.

Los núcleos en un CMP suelen ser **multiprocesadores simétricos** (**SMP**, *symmetric multiprocessors*), que acceden a una memoria común con latencia de acceso uniforme, pero tienen cachés que pueden ser:

1. Totalmente separadas, una caché por procesador. Es el caso de las cachés L1, de las que hay dos por procesador para datos e instrucciones. Dos cachés separadas pueden tener el mismo dato.
2. Compartidas, para aprovechar mejor el espacio a cambio de un acceso más lento.
3. **NUCA** (*Non-Uniform Cache Access*): Una caché por procesador pero lógicamente compartida, de modo que si un procesador necesita un bloque que no está en su caché se lo pide a las cachés de los otros procesadores.

También hay procesadores con **DSM** (*Distributed Shared Memory*), una arquitectura **NUMA** (*Non-Uniform Memory Access*, acceso a memoria no uniforme) en la que cada núcleo tiene una porción de memoria y acceso a E/S y puede acceder a la porción de otros procesadores mediante una red de interconexión.

3.2. Coherencia de cachés

Un sistema de memoria multiprocesador es **coherente** si todas las copias accesibles de la misma posición de memoria tienen el mismo valor, esto es, si el resultado de cualquier ejecución de un programa es tal que, para una posición de memoria, se puede construir una ordenación secuencial de las operaciones realizadas sobre dicha posición y en la que las operaciones emitidas por un mismo procesador aparecen en el orden en que son emitidas por el procesador al sistema de memoria y el valor devuelto por una operación de lectura es el escrito en la última escritura con la ordenación secuencial.

Equivalentemente, es coherente si el resultado de cualquier ejecución de un programa se puede obtener dividiendo la ejecución en una secuencia de épocas o momentos lógicos de forma que en cada época o bien hay un único procesador leyendo y escribiendo de una dirección o bien ningún procesador escribe en la dirección, y el valor en una dirección al comienzo de una época para cualquier lector es el mismo que el que tenía al final de la última época en que se escribió a la dirección.

En sistemas de memoria compartida, el *hardware* mantiene la coherencia mediante un **protocolo de coherencia**, que debe ser correcto y queremos que sea rápido sin consumir mucha

energía y requiera poco *hardware*. Un procesador mononúcleo puede tener problemas de coherencia por la E/S, pues un controlador DMA puede escribir a una zona de memoria pero el procesador lee de caché. Lo que se hace entonces es desactivar el uso de caché para esa zona. Para procesadores multinúcleo los protocolos pueden ser:

- **Basados en invalidación**: Cuando un procesador modifica su copia, notifica al resto para que descarten la suya. Solo hay que comunicar la primera escritura, pues tras esta no existen más copias hasta que otro nodo vuelva a acceder al dato, y el resto de escrituras a la misma línea de caché son locales.
- **Basados en actualización**: Cuando un procesador modifica su copia, indica su modificación al resto para que la actualicen. No invalida copias de otros nodos, por lo que no aumenta la tasa de fallos, pero usa mucho ancho de banda en actualizaciones que en general no son accedidas de nuevo por otros nodos.

La invalidación o actualización de un bloque en un nivel de caché debe transmitirse a los niveles superiores. La mayoría de sistemas usan protocolos basados en invalidación.

3.2.1. MSI con fisgoneo

Los protocolos ***snoopy*** o basados en **fisgoneo** se basan en que los nodos puedan ver los fallos de caché del resto y actuar en consecuencia, invalidando las copias si hace falta o proporcionando los datos. Esto requiere una red de interconexión (**NoC**, *Network on Chip*) totalmente ordenada que permita la difusión, como un bus o cualquier otra red si se obliga a los mensajes a pasar por un punto de serialización, aunque hay sistemas modernos que usan variantes del fisgoneo que funcionan con redes que no garantizan un orden total, normalmente anillos. Los protocolos basados en fisgoneo funcionan bien cuando hay pocos nodos, pero con muchos nodos la red totalmente ordenada es un cuello de botella.

El protocolo **MSI** es un protocolo basado en invalidación en el que los bloques pueden estar en uno de tres estados:

1. ***Modified***: Los datos están actualizados, la copia en memoria es obsoleta y ninguna otra caché tiene copia.
2. ***Shared***: Los datos están actualizados en caché y memoria, y otras cachés pueden tener copia.
3. ***Invalid***: Bloque inválido.

Un acceso a caché es un **acierto** si no requiere enviar mensajes por la red de interconexión, y es un **fallo** en otro caso.

Los mensajes intercambiados son:

- Del procesador (o nivel de caché superior) a la caché, petición de lectura (`PrRd`) y de escritura (`PrWr`), a las que la caché siempre acaba respondiendo.
- En el bus, petición de lectura (`BusRd`), de lectura exclusiva (con intención de modificar, `BusRdX`) o de invalidación de bloque (`BusUpd`), y envío de datos a otra caché (`Flush`) o a memoria y opcionalmente a otra caché (`FlushM`).

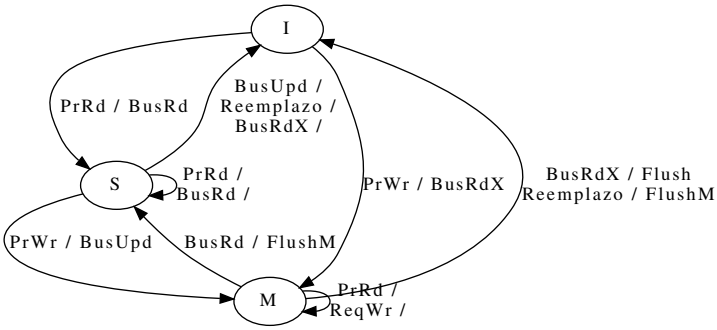

Figura 3.1: Diagrama de estados del protocolo MSI con fisgoneo.

Una **transacción** es un mensaje distinto a `Flush` y a `FlushM`, pues estas pueden ir por otro lado.

El diagrama de estados resultante está en la figura 3.1. Una versión simplificada del protocolo sustituye `Flush` y `FlushM` por un solo mensaje `BusWr`, equivalente a `FlushM`, y `BusUpd` por `BusRdX`.

Cuando se recibe un `BusRd` de un bloque modificado, hacer `FlushM` y compartir el bloque permite que posteriores lecturas por el procesador original resulten en acierto de caché, pero si el nuevo procesador escribe en el bloque, tiene que notificar la actualización para invalidar el bloque y la latencia de la escritura aumenta. Si el patrón de acceso es migratorio; por ejemplo, si se va actualizando un contador compartido, se escribe el bloque continuamente a memoria (más comúnmente a caché L3 compartida), por lo que puede ser mejor hacer `Flush` e invalidar el bloque.

En general es preferible compartir cuando es más probable que el procesador original vuelva a leer y es preferible invalidar cuando es más probable que el patrón de acceso sea migratorio. Algunos procesadores adaptan el protocolo según el tipo de acceso observado.

3.2.2. Directorios

Un directorio es una tabla cuyas entradas están formadas por la etiqueta de un bloque al que asocia bits de presencia, indicando en qué núcleos está el bloque normalmente mediante un vector de bits llamado **código de compartición**, y bits de estado. Se usa como punto de serialización para dar escalabilidad a los protocolos de fisgoneo.

Puede ser centralizado, pero como tiene que ser consultado cada vez que un acceso a memoria falla en una caché local, es mejor que sea distribuido, con varios directorios cada uno encargado de un subconjunto de direcciones de memoria. La latencia de acceso es variable porque el directorio asociado a una dirección puede estar conectado al nodo local de la red o a uno más o menos remoto.

Para que un protocolo de directorio sea escalable debe tener buen rendimiento, minimizando el ancho de banda por fallo especialmente en el camino crítico, y la frecuencia de los fallos, y reduciendo la sobrecarga de memoria asociada al estado del directorio.

El protocolo MSI basado en directorio usa 3 estados posibles para una entrada de directorio: NC (*Non-Cached*), S (*Shared*) y P (*Private*). Los mensajes son:

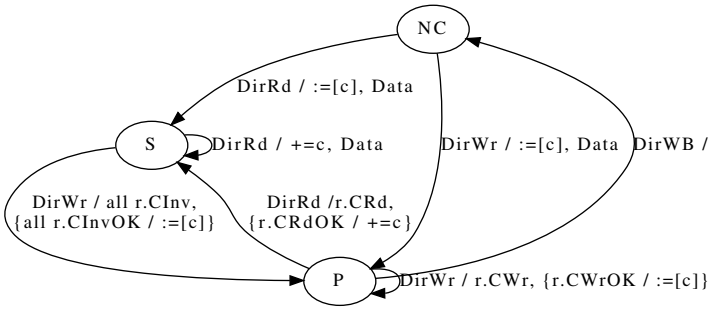

Figura 3.2: Transiciones del directorio en MSI con directorios, donde `c` es la caché peticionaria, `r` es una caché remota que tiene el dato y las acciones a realizar en una transición incluyen modificar el código de compartición y esperar un mensaje para actuar (*{mensaje/acciones}*).

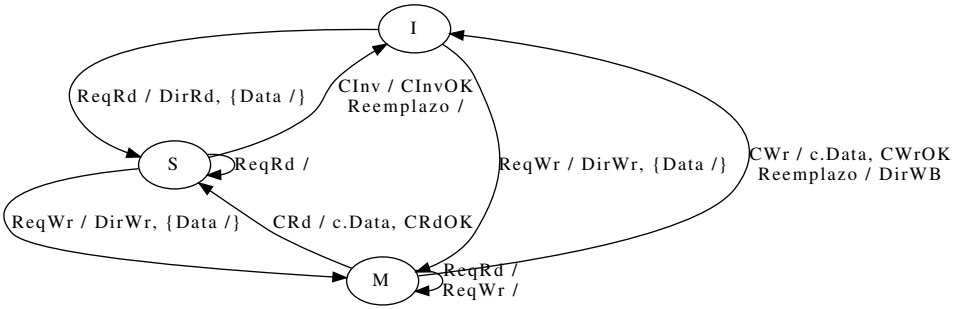

Figura 3.3: Transiciones de las cachés en MSI con directorios, donde `c` es la caché que pide un dato a esta.

1. Del procesador a la caché: Petición de lectura (`PrRd`) y de escritura (`PrWr`).

2. Del directorio a las cachés: Petición de invalidación (`CInv`), de obtener copia para lectura (`CRd`) y de enviar copia para escritura (`CWr`).

3. De las cachés al directorio: Petición de lectura (`DirRd`), escritura (`DirWr`) y volcado en memoria (`DirWB`); confirmación de invalidación (`CInvOK`) y de envío de copia para lectura (`CRdOK`, contiene el dato leído) y escritura (`CWrOK`).

4. Respuesta con datos solicitados (`Data`), de una caché a otra o, si el dato no estaba en ninguna caché, del directorio a la caché, tras solicitar el dato de memoria.

Los diagramas de transición se muestran en las figuras 3.2 y 3.3. A veces se usa otro mensaje `DirUpd` y un `DirUpdOk` para evitar tener que pedir los datos al pasar de S a M. En la práctica queremos que varios procesadores puedan ejecutar transacciones a la vez, incluso hacia la misma línea de caché, lo que supone mucha complejidad.

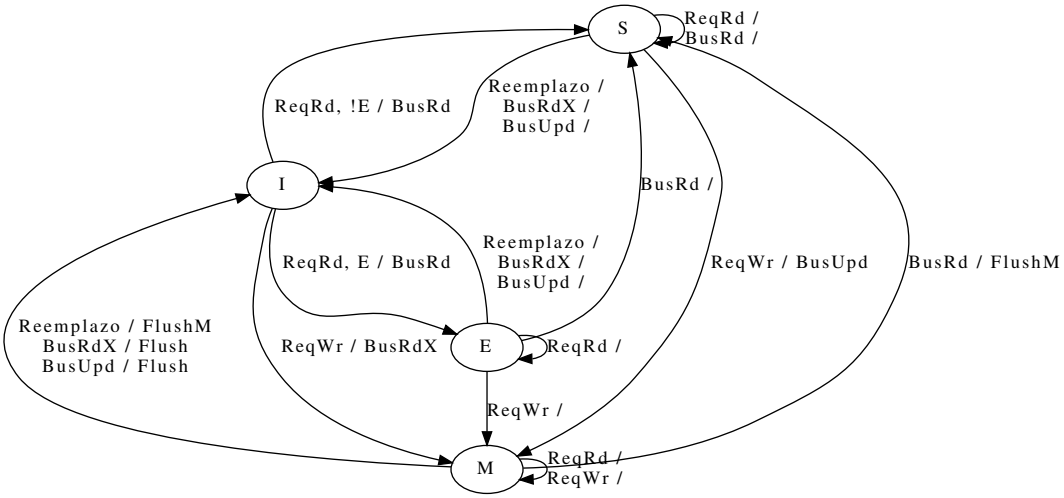

Figura 3.4: Diagrama de transición de MESI basado en fisgoneo. Una señal E se activa cuando ningún nodo ha solicitado el bloque.

3.2.3. Datos compartidos

Cuando varios procesadores usan la misma línea de caché a la vez se dice que comparten la línea. En los protocolos de invalidación, las escrituras frecuentes a una línea compartida degradan el rendimiento. Distinguimos:

- **Verdadera compartición**: Los hilos usan la misma variable. Esto no es un problema si la variable es de sólo lectura, pero si se escribe frecuentemente se deberían usar copias de la variable por hilo si el algoritmo lo permite.
- **Falsa compartición**: Los hilos usan variables distintas en el mismo bloque. La solución es almacenar las variables en bloques distintos o evitar el entrelazamiento de los accesos.

3.2.4. Mejoras a MSI

MSI se puede optimizar con nuevos estados:

1. ***Exclusive***: No se ha modificado el bloque pero ninguna otra caché tiene una copia. Esto es habitual, y codificarlo en otro estado evita enviar mensajes de coherencia para datos privados.
2. ***Owner***: Otras cachés pueden tener copias en estado *Shared*, pero la copia en memoria no está actualizada y esta caché es la encargada de dar los datos en caso de fallo. Esto evita escribir en memoria cuando otro núcleo quiere leer un dato que se ha modificado.

Esto nos da los protocolos **MESI** (MSI con *Exclusive*, figura 3.4), **MOSI** (MSI con *Owner*) y **MOESI** (MSI con *Exclusive* y *Owner*).

3.3. Modelos de consistencia de memoria

El compilador y los núcleos del procesador pueden cambiar el orden de cargas y almacenamientos, por lo que en principio un hilo no puede depender del orden en que otro carga y almacena los valores. Como hace falta un orden para la sincronización, necesitamos un **modelo de memoria**, una especificación de las restricciones en el orden en que las operaciones de memoria de un hilo se hacen visibles al resto. Todos los modelos son coherentes, pero la **consistencia**, la generalización de la coherencia a posiciones de memoria distintas, es rara.

Algunos modelos:

- **Consistencia estricta**: La escritura a una variable por un procesador es vista instantáneamente por el resto. Es determinista y asume un reloj global en el procesador, y aunque es fácil de conseguir en procesadores mononúcleo cuando no hacen E/S, enlentece mucho los sistemas distribuidos.

- **Consistencia secuencial**: Las escrituras no tienen que hacerse visibles instantáneamente, pero son **atómicas**, vistas en el mismo orden por todos los hilos, y en particular las de un mismo hilo se ven en el orden del programa.

  Esto requiere esperar a que las escrituras terminen de propagarse antes de emitir otra operación de memoria, rompiendo optimizaciones como los *buffers* de escritura y el adelantamiento y dificultando mecanismos como la segmentación, lo que se puede aliviar con especulación pero añadiendo mucha complejidad. Además, muchas optimizaciones de compiladores se basan en reordenar el código fuente y en particular los accesos a memoria, o en usar registros. Por eso x86, ARM, MIPS, PowerPC y RISC-V no lo implementan.

En la práctica se usan modelos de consistencia relajados:

- **Consistencia del procesador** o ***Processor Consistency***: Cada hilo ve las escrituras de un mismo procesador en orden.

- **Consistencia débil**: Los accesos a memoria pueden marcarse como de sincronización. Estos accesos tienen consistencia secuencial entre sí y todos los hilos ven el mismo conjunto de operaciones de memoria entre estos accesos.

- **Consistencia de liberación**: Los accesos a memoria pueden marcarse como de **adquisición** (*acquire*) o de **liberación** (*release*). Los accesos de adquisición y liberación tienen consistencia del procesador entre sí, todos los accesos esperan a que los anteriores de adquisición hayan terminado y los de liberación esperan a que todos los anteriores hayan terminado. Normalmente las secciones críticas empiezan con un *acquire* y terminan con un *release*.

  Este es el modelo usado por C, C++ y Java,[2] y ARMv8 y RISC-V tienen modelos similares.

Los accesos a memoria marcados especialmente se suelen traducir en ensamblador como barreras de memoria o *memory fences*. Programar con modelos relajados es más difícil, y se aconseja evitar las **condiciones de carrera**, en que dos hilos acceden a la misma memoria sin ordenación y al menos uno escribe en ella, y usar las primitivas de sincronización del lenguaje de programación, que en general incluyen las *fences* necesarias.

[2] C, C++ y Rust usan el mismo modelo, que también incluye consistencia débil.

3.4. Semáforos

La comunicación y compartición de recursos entre procesos requiere de mecanismos de sincronización, como son:

- Semáforos o cerrojos para la exclusión mutua.
- **Señales** para la sincronización punto a punto, en que uno o más hilos esperan a que otro envíe una señal.
- **Barreras** para sincronización global, que impiden que los hilos sigan avanzando hasta que todos lleguen a cierto punto de su ejecución.

Las implementaciones de exclusión mutua constan de:

1. Un **método de adquisición** para obtener el cerrojo, con poca latencia si no hay más hilos intentando sincronizarse.
2. Un **algoritmo de espera** para esperar a que el cerrojo esté disponible cuando no lo esté, y que consuma poco ancho de banda de la red de interconexión.
3. Un **método de liberación** para notificar que el cerrojo está disponible, que sea suficientemente imparcial y no provoque inanición.

Es importante que el protocolo sea escalable y requiera poca información para funcionar. Estos protocolos se pueden implementar con consistencia secuencial, pero el resultado es complicado e ineficiente, por lo que es más común ampliar el ISA con instrucciones atómicas de sincronización, que actúan como barreras. Algunas son:

- ***Test&set***: Carga una posición de memoria en un registro y establece dicha posición a 1. Un cerrojo implementado como un booleano que está a 1 cuando el cerrojo está bloqueado se bloquearía con `lock: t&s t0, MUTEX; bnez t0, lock` y se desbloquearía con `sw r0, LOCK`. Cuando el cerrojo está ocupado, esto genera invalidaciones continuas, por lo que es preferible usar *test&test&set*, comprobando con una carga normal si el cerrojo está disponible antes de obtenerlo:

  ```
  lock: lw t0, MUTEX
        bnez t0, lock
        t&s t0, MUTEX
        bnz t0, lock
  ```

  Esto aumenta algo la latencia pero reduce el tráfico entre cachés y es más escalable.

- ***Swap***: Intercambia los valores de un registro y una posición de memoria. Se puede usar como `t&s` estableciendo el registro a 1.
- ***Fetch&op***: Lee una posición de memoria un registro y escribe en memoria el valor obtenido al aplicar una cierta operación.
- ***Compare&swap***: Compara el valor en una posición de memoria con el de un registro y, si coinciden, intercambia el contenido de la posición con el de otro registro.

Estas instrucciones son difíciles de implementar y lentas, y una alternativa es el par LL/SC, formado por las instrucciones:

- ***Load Linked*** (LL): Lee una posición de memoria en un registro.
- ***Store Conditional*** (SC): Si, en el bloque (normalmente de caché) de la posición de memoria indicada, nadie ha escrito desde el último LL, escribe el valor de un registro en la posición. Si alguien ha escrito antes, la instrucción falla; ni escribe ni hay invalidaciones.[3] También establece algún registro, como en el que se quería almacenar en el caso de MIPS, para indicar si la instrucción tuvo éxito.

Se ha de ejecutar el mínimo de operaciones posible entre LL y SC para favorecer que SC tenga éxito. Un cerrojo se bloquearía con

```
lock: li t1, 1
      ll t0, MUTEX
      bnez t0, lock
      sc t1, MUTEX
      beqz t1, lock
```

Esta implementación tiene buen rendimiento, pues la espera activa no genera tráfico ni invalidaciones, pero no es imparcial.

3.5. Redes de interconexión

En sistemas con varias CPUs puede haber **interconexión *on-chip***, dentro de un mismo CMP, y ***off-chip***, entre procesadores de distintos chips, como en los clústeres de servidores. Las redes Ethernet suelen tener ancho de banda de $0{,}1, 1, 10, 100\,\text{Gbps}, \ldots$ y latencia de único salto de 25–100 μs, mientras que InfiniBand tiene $20, 40, 54, 80\,\text{Gbps}, \ldots$ y latencia de único salto de 1–3 μs.

Las redes están formadas por enlaces; conmutadores (*switches* o *routers*), que conectan enlaces y pueden tener funciones de encaminamiento, y una interfaz de red en cada núcleo que lo conecta con un conmutador.

La longitud de un cable determina la frecuencia a la que se puede operar y la potencia que puede haber que disipar. Interesa una red con alto ancho de banda (frecuencia por número de hilos) y baja latencia, simple (lo que suele llevar a mejor rendimiento), fiable (que no produzca fallos y soporte un número limitado de ellos) y con bajo coste económico y energético. Queremos estructurarla de forma escalable (que el ancho de banda aumente conforme lo haga el número de nodos) y fácilmente particionable (que se pueda dividir en subsistemas), pues al aumentar el número de núcleos, una mala red de interconexión puede consumir buena parte de la energía y dar lugar a cuellos de botella.

3.5.1. Topologías de red

Un **hipergrafo dirigido** es un par (V, H) formado por un conjunto de **nodos** V y un conjunto de **hiperarcos** $H \subseteq \{(A, B) \in \mathcal{P}(V) \times \mathcal{P}(V) \mid A, B \neq \emptyset\}$. Un **hipergrafo no dirigido** es

[3] En la práctica esto puede ocurrir cuando hay una interrupción, cambio de contexto, otro LL, una escritura a otro bloque, etc.

un par (V, H) donde V es un conjunto de nodos y $H \subseteq \mathcal{P}(V) \setminus \{\emptyset\}$ es un conjunto de **hiperejes**. Identificamos el hipergrafo no dirigido (V, H) con el hipergrafo dirigido $(V, \{(h, h)\}_{h \in H})$. Un **corte de bisección** de un hipergrafo no dirigido finito es un conjunto separador de hiperejes de tamaño mínimo, y el **ancho de la bisección** (*bisection bandwidth*) es dicho tamaño.

Una **hiperred** es una tupla (V, H, ω) donde (V, H) es un hipergrafo y $\omega : H \to \mathbb{R}$ es una función de pesos. Un corte de bisección de una hiperred no dirigida finita es un conjunto separador de hiperejes con mínima suma de los pesos, y el ancho de la bisección es dicha suma.

Una **topología de red** es un hipergrafo dirigido finito (V, H) donde los nodos representan *routers* y un hiperarco (A, B) es un enlace que transmite de los nodos de A a los de B. Su **grado** es el grado máximo de un nodo. Si se añade una función de pesos, el peso de un enlace es su ancho de banda. La topología es **simétrica** si el hipergrafo es no dirigido. Informalmente, es **regular** si los nodos están conectados en un patrón definido, de modo que se pueden asignar coordenadas a los nodos y generalmente se pueden encaminar los mensajes solo según las coordenadas de los extremos, y es **irregular** si los nodos se conectan sin estructura, lo que es más expansible, y se suele encaminar con algoritmos que intentan encontrar el camino más corto.

Algunas topologías regulares simétricas, donde $\mathbb{N}_n \coloneqq \{0, \ldots, n-1\}$:

- **Buses**: $(V, \{V\})$, con ancho de bisección 1 y grado 1.
- **Mallas** n-dimensionales: (V, E) con $V \coloneqq \mathbb{N}_{d_1} \times \cdots \times \mathbb{N}_{d_n}$ y

$$E \coloneqq \{\{(i_1, \ldots, i_n), (i_1, \ldots, i_{k-1}, i_k + 1, i_{k+1}, \ldots, i_n)\}\}_{(i_1, \ldots, i_n) \in V, k \in \mathbb{N}_n, i_k + 1 < d_k}.$$

- **Hipercubos**: Mallas con $d_i = 2$ para todo i.
- **Toros** n-dimensionales: (V, E) con $V \coloneqq \mathbb{N}_{d_1} \times \cdots \times \mathbb{N}_{d_n}$ y

$$E \coloneqq \{\{(i_1, \ldots, i_n), (i_1, \ldots, i_{k-1}, i_k + 1 \bmod d_k, i_{k+1}, \ldots, i_n)\}\}_{(i_1, \ldots, i_n) \in V, k \in \mathbb{N}_n}.$$

- **Anillos**: Toros unidimensionales, con ancho de bisección 2, grado 2 y diámetro $\lfloor \frac{n}{2} \rfloor$ si tiene n nodos.
- **Árboles**, con ancho de bisección 1 y encaminamiento normalmente por el único camino entre fuente y destino.

3.5.2. Redes indirectas

Una red es **directa** si cada conmutador conecta con algún núcleo e **indirecta** en otro caso. Los buses y las mallas son redes directas. Las **redes *cross-bar*** son redes indirectas con topología $(\mathbb{N}_n \times \mathbb{N}_m, \{\{(n, m), (n+1, m)\}, \{(n, m), (n, m+1)\}\}_{(n,m) \in \mathbb{N}_{n-1} \times \mathbb{N}_{m-1}})$, con los nodos en $\mathbb{N}_n \times 0$ conectados a n núcleos de procesador y los nodos en $0 \times \mathbb{N}_m$ conectados a m unidades de memoria. Los conmutadores se llaman ***cross-point switches***.

Una **red multietapa** de n etapas tiene topología

$$(\mathbb{N}_m \times \mathbb{N}_{n+1}, \{\{(i, j), (i, j+1)\}, \{(i, j), (\sigma_j(i), j+1)\}\}_{(i,j) \in \mathbb{N}_m \times \mathbb{N}_n}),$$

donde cada σ_j es una permutación de $\mathbb{N}_m$ y los conmutadores conectados a nodos son los $\mathbb{N}_m \times \{0\}$ y los $\mathbb{N}_m \times \{n\}$, con distintos tipos de nodo.

Una red ***Butterfly*** es una de n etapas con 2^n nodos a cada lado y $\sigma_j(i) = i+2^j$ mód 2^n, y es **bloqueante**, es decir, se puede producir contención en los enlaces según la comunicación entre los nodos de un lado y del otro. Una red **Benes** resulta de «concatenar» dos redes *Butterfly*: tiene $2n$ etapas y 2^n nodos a cada lado y $\sigma_j(i) = i + 2^{\text{mín}\{j,2n-j-1\}}$ mód 2^n. No es bloqueante.

3.5.3. Mecanismo de enrutamiento

El enrutamiento es **determinista** si la ruta que siguen los mensajes solo depende del origen y el destino, y es **adaptativo** si también depende del estado de la red y de los enlaces. Puede ser **de ruta mínima** o **no mínima**, y en el último caso puede tener o no **vuelta atrás** (poder o no producir paseos que no sean caminos). El enrutamiento es **fuente** si el nodo origen calcula todo el camino y es **destino** si cada conmutador decide el siguiente enlace.

En una malla (V, E), el ***dimension-order routing*** consiste en que, si $x, y \in V$ con $x \neq y$, para llevar un mensaje de x a y, tomamos el menor k con $x_k \neq y_k$ y lo llevamos primero a $(x_1, \ldots, x_{k-1}, x_k \pm 1, x_{k+1}, \ldots, x_n)$, tomando $x_k + 1$ si $x_k < y_k$ o $x_k - 1$ si $x_k > y_k$. En un hipercubo esto se llama ***edge-cube routing***.

3.5.4. Estrategia de conmutación

Es la forma en que los datos pasan a través de los conmutadores.

AR
Conmutación de circuitos La red telefónica. [...] Conectan puertos de entrada y salida [...]. [...] El circuito es dedicado y con un ancho de banda fijo [...]. [La latencia es el tiempo de establecer el circuito y propagar los datos.]
Conmutación de mensajes Los *routers* redirigen el mensaje por la interfaz adecuada usando información adicional en los mismos [...], y usan *buffers* para almacenar los mensajes [...] (*store-and-forward*).
Conmutación de paquetes Como la de mensajes, pero los mensajes se dividen en trozos pequeños llamados paquetes, enrutados individualmente [...]. [La latencia es proporcional a la distancia.]

En redes *on-chip* se usan mecanismos de altas prestaciones:

Wormhole Los paquetes se dividen en ***flits***, trozos con el tamaño exacto que cabe en el canal, de los que el primero contiene la cabecera con la dirección de destino. El *switch* encamina el paquete en cuanto recibe la cabecera, y solo almacena partes de este en *buffers* pequeños con baja latencia y poco sensibles a la distancia.

Virtual Cut Through Variante de *wormhole* en la que, si el canal de salida no está disponible, el paquete se almacena. La latencia es similar a la de *wormhole*, no afectada por la distancia, y da más prestaciones cuando hay congestión a cambio de necesitar *buffers* más grandes, capaces de almacenar un paquete completo.

Para el control de flujo en altas prestaciones, cuando se alcanza un límite de uso *stop* procesando *flits* de un canal, se envía una señal al resto para que dejen de enviar y evitar el desbordamiento, y cuando el uso baja a un límite *go*, se les avisa de que pueden volver a enviar.