

ÍNDICE

PRESENTACIÓN.....	5
CAPÍTULO 1. EVALUACIÓN DE PRESTACIONES	9
1. Introducción.....	9
2. Tiempo de CPU.....	9
3. Medidas de velocidad de procesamiento y benchmarks	11
4. Estimación de tiempo de CPU	13
5. Ganancia de velocidad y ley de Amdahl	16
6. Cuestiones y Problemas.....	17
CAPÍTULO 2. PROGRAMACIÓN PARALELA	51
1. Introducción.....	51
2. Herramientas de programación paralela.....	51
2.1. Trabajo a realizar por las herramientas de programación paralela o el programador	51
2.2. Nivel abstracción en que sitúan al programador las herramientas.	54
2.3. Estilos de programación paralela	55
2.4. Comunicación/sincronización en herramientas de programación paralela	56
3. Estructuras de procesos o tareas en códigos paralelos	58
4. Evaluación de prestaciones.....	60
5. Problemas.....	64
6. Cuestiones	88
7. Bibliografía.....	92
CAPÍTULO 3. ARQUITECTURAS CON PARALELISMO A NIVEL DE THREAD (TLP)	93
1. Introducción.....	93
2. Arquitecturas TLP	93

3. Coherencia del sistema de memoria	95
3.1. Protocolos de mantenimiento de coherencia	98
3.1.1. Protocolo MSI (Modified-Shared-Invalid) de espionaje	99
3.1.2. Protocolo MESI (Modified-Exclusive-Shared-Invalid) de espionaje ..	103
3.1.3. Protocolos MSI (Modified-Shared-Invalid) para multiprocesadores NUMA.....	103
3.1.3.1. Protocolo MSI para multiprocesadores NUMA sin difusión.....	103
3.1.3.2. Protocolo MSI para multiprocesadores NUMA con difusión	112
4. Consistencia del sistema de memoria	117
5. Sincronización	127
5.1. Cerrojos	130
5.2. Barreras	132
5.3. Instrucciones para implementar sincronización	133
6. Problemas.....	136
7. Cuestiones	159
8. Bibliografía.....	163

CAPÍTULO 4. MICROARQUITECTURAS CON PARALELISMO ENTRE INSTRUCCIONES (ILP).....	165
1. Introducción.....	165
2. Procesadores Superescalares	167
2.1. Procesamiento de instrucciones en un cauce superescalar	167
2.2. Procesamiento de las instrucciones de salto.....	177
3. Procesadores VLIW	180
3.1. Técnicas software para ampliar los bloques básicos	182
3.2. Instrucciones con predicado.....	188
3.3. Procesamiento especulativo.....	190
4. Problemas.....	192
5. Bibliografía.....	237

PRESENTACIÓN

La Arquitectura de Computadores es esencial para el Ingeniero en Informática. No cabe duda de que diseñar un algoritmo de, por ejemplo, minería de datos capaz de procesar N patrones con una complejidad de sólo $O(N)$ es importante para abordar nuevas aplicaciones, que pueden ser muy útiles y demandadas. Pero si estamos ante valores de N de alrededor de miles de billones de datos, aunque una operación elemental consuma una mil millonésima de segundo (sólo un ciclo de reloj en un procesador a un GHz) estaríamos hablando de algo más de once días y medio para completar el programa. Este margen de tiempo podría ser aceptable para la aplicación en cuestión, pero no cabe duda de que es imprescindible disponer de computadores que aprovechen eficientemente las mejoras tecnológicas para procesar y gestionar volúmenes de datos cada vez mayores, en tiempos razonables. Así, si hoy se habla de *Big Data*, o los avances de la neurociencia nos hacen contemplar como algo cada vez más próximo el conocimiento de las claves de la mente humana, se debe en gran medida a que los computadores han alcanzado ciertos niveles en sus prestaciones. Y no solo eso: vemos como algo seguro y evidente que cada vez habrá computadores más veloces. Un Ingeniero Informático debe conocer las estrategias que permiten disponer de computadores más eficientes, ser capaz de generar códigos que las aprovechen, y hacerlas accesibles a los usuarios a través de herramientas fáciles de usar.

Como en cualquier sistema complejo, el estudio del computador se ha abordado a través de capas de abstracción que, desde los elementos hardware constituyentes del computador hasta los niveles más próximos a las aplicaciones, aprovechan los recursos que ofrecen las capas inferiores, ocultando sus detalles al usuario de las superiores. Esta situación ha podido dar a entender que era posible un Ingeniero Informático sin conocimientos acerca de los detalles relacionados con el funcionamiento del computador en sus niveles más próximos al hardware, y ahí puede encontrarse la razón de que normalmente se requiera cierto esfuerzo adicional para motivar el estudio de los temas relacionados con la estructura y la arquitectura del computador. No se repara en que el desconocimiento de esos niveles no solo limita la carrera profesional del Ingeniero Informático, sino que además compromete la competencia con que puede desempeñar su profesión, cualquiera sea su área de actividad, y que constituye una diferencia fundamental entre un Ingeniero Informático y otro titulado que se dedique profesionalmente a la programación. Los conocimientos de la estructura y arquitectura de los computadores no sólo ayudan a realizar programas más rápidos, también a reducir los tiempos de desarrollo.

Un factor importante para motivar la aproximación a una disciplina es facilitar su asimilación, hacer más accesible su aprendizaje. Los modelos más aceptados actualmente indican que el aprendizaje se construye, y que la memoria, esencial en dicho aprendizaje,

no es un almacén donde se introducen cosas que luego se podrán sacar sin más: memorizar es más que almacenar (K. Bain, 2006; N. Carr, 2010). La memorización es la primera etapa de un proceso de síntesis que hace posible entender de forma profunda y personal aquello que se ha recibido (leído, escuchado, visto, etc.), y hace intervenir a toda la mente ya que implica creatividad y juicio. El cerebro transforma los eventos cargados en la memoria a corto plazo en pensamientos memorizados tras un cierto tiempo, necesario para que sea posible esa transición desde la memoria a corto a la de largo plazo (E. R. Kandel, 2007). Estos dos tipos de memoria se basan en procesos biológicos diferentes. De hecho, deben sintetizarse cierto tipo de proteínas para conseguir la memorización a largo plazo, y los resultados de ciertos experimentos neurológicos han puesto de manifiesto que el paso a la memoria a largo plazo no sólo cambia la concentración de neurotransmisores para reforzar ciertas conexiones, sino que también aparecen nuevas conexiones (nuevas sinapsis). Así, la memorización a largo plazo implica cambios anatómicos además de bioquímicos, y deben activarse los genes apropiados para producir las proteínas necesarias y responder al entorno (en este caso, al proceso de aprendizaje). Por tanto, el conocimiento más reciente acerca de la forma en que aprende la mente humana pone de manifiesto la necesidad de aplicar los tópicos estudiados para crear redes que los liguen con las situaciones para las que son útiles. Además, ilustrar la relación entre el cuerpo teórico de la asignatura y la realidad contribuye a motivar el esfuerzo y el interés del estudiante hacia la asignatura.

Este libro propone un acercamiento a los conceptos de Arquitectura de Computadores desde el punto de vista práctico que supone la resolución de problemas y ejercicios relacionados con el paralelismo. Precisamente el compromiso entre el aprovechamiento del paralelismo, en sus diversas facetas y niveles, y la localidad del acceso a los datos que se procesan puede contemplarse como el *leitmotiv* que ha determinado las propuestas de innovación en Arquitectura de Computadores, orientando el desarrollo de nuevas plataformas de cómputo que aprovechen las mejoras que ha aportado la tecnología electrónica a través de circuitos integrados con cada vez más transistores funcionando a frecuencias de reloj más elevadas. Se han fabricado microprocesadores cada vez más potentes gracias al procesamiento paralelo de instrucciones a través de microarquitecturas segmentadas (pipelines) con etapas capaces de procesar varias instrucciones por ciclo (los procesadores segmentados superescalares y VLIW), y también al aprovechamiento del paralelismo de datos. Sin embargo, la fabricación de microprocesadores capaces de doblar su velocidad cada dos años (siguiendo lo marcado por la ley de Moore) a partir de microarquitecturas más complejas que cada vez procesan más instrucciones por ciclo, prácticamente llegó a su saturación a comienzos del siglo XXI dada la cantidad insostenible de energía que disiparían las microarquitecturas al ritmo en que se incrementaba su complejidad y a que la reducción del tamaño de los transistores, a partir de un momento, empezó a ocasionar cierta interacción entre la complejidad de los cauces y la frecuencia de reloj a la que podrían funcionar: los retardos para que las salidas de una etapa pudieran llegar a la siguiente limitaban la frecuencia de reloj que había que utilizar para dar tiempo a los resultados de una etapa a llegar a la siguiente.

Por estos motivos, aparecen los multiprocesadores en un chip o chip de procesamiento multinúcleo (*multicore*) que integran varios procesadores, y para cuyo aprovechamiento es necesario programar en paralelo las aplicaciones. Ya existían computadores con varios procesadores y computadores que aprovechaban el paralelismo de datos en arquitecturas de altas prestaciones (multiprocesadores, multicamputadores, clusters de computadores, etc.) pero ahora, incluso el procesamiento paralelo de flujos de instrucciones se implementa, junto con el paralelismo de instrucciones y de datos, a nivel de chip, y se aprovecha en computadores que están presentes en nuestras actividades cotidianas, desde los computadores personales hasta los teléfonos móviles y otros dispositivos.

Teniendo en cuenta la situación actual de la Arquitectura de Computadores que hemos resumido en el párrafo anterior, en este libro se abordan conceptos esenciales relacionadas con el aprovechamiento de los distintos tipos de paralelismo (de datos, entre instrucciones, *multithread*). Los contenidos se han organizado en cuatro capítulos que bien podrían constituir los cuatro temas en los que se organizaría un curso cuatrimestral (tres créditos ECTS de teoría/problems y tres créditos ECTS de prácticas) de introducción a Arquitectura de Computadores.

El primer capítulo está dedicado a la evaluación de prestaciones. Precisamente la búsqueda de computadores mejores nos lleva a la cuestión de medir y comparar el comportamiento de los computadores. Así, el capítulo contiene problemas relacionados con el tiempo de CPU, y los conceptos de la microarquitectura y del repertorio de instrucciones que lo determinan, y con el análisis de la influencia de los cuellos de botella en las prestaciones y las consecuencias que tiene la ley de Amdahl en la elección de las estrategias de mejora de dichas prestaciones.

El segundo capítulo introduce conceptos de programación paralela. Presenta las herramientas que existen para escribir código paralelo, el trabajo adicional que la escritura de código paralelo plantea a las herramientas y al programador, las estructuras típicas de flujos de instrucciones en los códigos paralelos, y cómo evaluar las prestaciones del código paralelo implementado y qué prestaciones máximas se pueden esperar.

El tercer capítulo se centra en arquitecturas que permiten ejecutar en paralelo o concurrentemente múltiples flujos de instrucciones que comparten memoria, son arquitecturas con paralelismo a nivel de *thread* (*Thread-Level Parallelism*) con una única instancia del Sistema Operativo (SO). Estas arquitecturas TLP debido a la compartición del sistema de memoria, presentan problemas propios, como son coherencia del sistema de memoria, consistencia del sistema de memoria y sincronización entre flujos de instrucciones. La forma particular de abordar estos problemas por parte de las arquitecturas afecta al programador, especialmente al programador de herramientas de programación y de SO.

Finalmente, el capítulo 4 se dedica a los conceptos fundamentales que permiten aprovechar el paralelismo entre instrucciones en los procesadores que han ido apareciendo. Así, se presentan las diferencias esenciales entre las microarquitecturas superescalares,

con planificación de instrucciones fundamentalmente implementada en hardware, y las denominadas VLIW (de *Very Long Instruction Word*), en las que la planificación de instrucciones es responsabilidad del compilador.

Cada uno de esos capítulos se inicia con una presentación de los conceptos fundamentales trabajados en los problemas que se resuelven a continuación. Así se proporciona un libro autoconsistente que, por sí solo, permite acercarse a los conceptos esenciales y entender la resolución de los problemas planteados en relación con ellos. No obstante, aconsejamos la consulta de otros textos donde se exponen más detalladamente alguno de los conceptos (J. Ortega, M. Anguita, A. Prieto 2005), y también recomendamos otros libros en los que se pueden encontrar algunos problemas adicionales a los que aquí se proporcionan.

Referencias

- Ken Bain, 2006: "*Lo que hacen los mejores profesores universitarios*". Publicacions Universitat de València.
- Nicholas Carr, 2010: "*The Shallows. What the Internet is doing to our Brains*". Ed. W.W. Norton (Edición en Castellano en Santillana Ediciones Generales S.L., 2011).
- Eric R. Kandel, 2007: "*En busca de la memoria: una nueva ciencia de la mente*". Katz Editores.
- Julio Ortega Lopera, Mancia Anguita López y Alberto Prieto, 2005: "*Arquitectura de Computadores*". Thomson.

CAPÍTULO 1. EVALUACIÓN DE PRESTACIONES

1. Introducción

Tanto el diseño de nuevos procesadores y arquitecturas de cómputo como el desarrollo de programas que aprovechen los recursos que proporcionan los computadores, hacen necesario evaluar las prestaciones obtenidas al ejecutar programas. Por tanto, uno de los tópicos esenciales en Arquitectura de Computadores es el uso de métricas y la definición de conjuntos de programas, que constituyan un marco de referencia adecuado para la comparación entre computadores, y entre versiones de programas ejecutados en dichos computadores. En este capítulo se incluyen problemas que ponen de manifiesto la relación entre el tiempo de CPU al ejecutar un programa y el número de instrucciones que el procesador ejecuta, la capacidad de la microarquitectura y de la tecnología del procesador para terminar más o menos instrucciones por ciclo, y el tiempo de ciclo del procesador. Además, se ilustra la relevancia de los cuellos de botella en las prestaciones, y el uso de la ley de Amdahl para razonar acerca de dichos cuellos de botella y del efecto de las mejoras de distintas partes del computador en las cargas de trabajo.

2. Tiempo de CPU

El tiempo transcurrido entre que se lanza la ejecución de un programa y se tienen sus resultados es el denominado *tiempo de respuesta*. Este tiempo incluye el denominado *tiempo de CPU* que es el que el procesador dedica a ejecutar instrucciones máquina de su repertorio, tanto en modo de usuario, como las que corresponden a la actividad que debe llevarse a cabo por el sistema operativo para permitir la ejecución del programa. Tanto las instrucciones que se ejecutan como los datos que se procesan se encuentran en la memoria principal del computador (y en los niveles de caché superiores). Dentro del tiempo de respuesta también está incluido el tiempo de espera debido a E/S, o a la ejecución de otros programas con los que se comparte el computador. Si, por ejemplo, se ejecuta el programa en un computador con el sistema operativo Unix, utilizando el comando time se puede tener información de todos estos tiempos. Cuanto menor sea el tiempo de respuesta mayor serán las prestaciones que proporciona el computador en la ejecución del programa, aunque, como se ha dicho, este tiempo también tiene información de la forma en que se están realizando las operaciones de E/S y también influye la carga del computador en un momento dado por la existencia de otros programas. En este capítulo no analizaremos el efecto de las E/S y nos centraremos en el tiempo de CPU para razonar acerca de la eficacia con que se utilizan los recursos del

procesador y su interacción con la jerarquía de memoria (memoria principal y jerarquía de cachés). Por lo tanto, si consideramos despreciable el tiempo de E/S (por ejemplo se consideran programas sin un número apreciable de operaciones de E/S), el tiempo de CPU se puede obtener mediante la expresión:

$$T_{CPU} = Ciclos\ del\ programa \times T_{ciclo} = \frac{Ciclos\ del\ Programa}{F} \quad (1)$$

donde *Ciclos del programa* es el número de ciclos de reloj del procesador que tarda en ejecutarse el programa, T_{ciclo} es el tiempo de un ciclo de reloj del procesador, y F es la frecuencia de reloj, es decir $1/T_{ciclo}$. Dado que el número de ciclos del programa se puede expresar en términos del número de instrucciones máquina del repertorio del procesador que se han procesado, NI , y del número medio de ciclos por instrucción, CPI , la expresión (1) se podría escribir como:

$$T_{CPU} = NI \times CPI \times T_{ciclo} = \frac{NI \times CPI}{F} \quad (2)$$

En un procesador en el que una instrucción máquina dada siempre requiere el mismo número de ciclos para procesarse (cosa que no ocurre en los procesadores segmentados, por ejemplo), el número medio de ciclos por instrucción, CPI , se puede calcular a partir de la expresión (3):

$$CPI = \frac{\sum_{i=1}^W NI_i \times CPI_i}{NI} \quad (3)$$

en la que NI_i es el número de instrucciones del tipo i que tiene el programa ejecutado por el procesador, y CPI_i el número de ciclos del procesador que necesita una instrucción del tipo i para procesarse. Se ha supuesto que hay W tipos de instrucciones diferentes en el programa.

En la expresión (2), el número de instrucciones a ejecutar, NI , depende del número y tipo de operaciones del problema que implementa el programa ejecutado, del repertorio de instrucciones máquina que se utiliza para codificarlas, y de lo eficiente que sea el compilador para usar las instrucciones máquina disponibles. En los denominados *repertorios de instrucciones escalares*, las instrucciones máquina codifican una única operación, por ejemplo la lectura o escritura de un dato, una operación aritmética o lógica sobre sus correspondiente operandos para obtener un resultado, etc. En estos repertorios escalares, cada operación que implementa el programa se codifica mediante una instrucción. No obstante, existen arquitecturas con repertorios específicamente diseñados para aprovechar el *parallelismo de datos* presente en muchas aplicaciones, en las que se repite la misma operación aunque con operandos diferentes. En estos repertorios el número de instrucciones necesario para codificar los programas es menor que si se utilizase un repertorio escalar, y se puede estimar a partir del cociente entre el número

de operaciones del programa, N_{oper} , y el número de operaciones que puede codificar cada instrucción, OPI . Así, se tendría que $NI = N_{oper} / OPI$.

El número medio de ciclos por instrucción, CPI , depende fundamentalmente de las características de la microarquitectura y de su implementación física. No obstante, en microarquitecturas como las de los procesadores segmentados no superescalares, el compilador tiene un papel decisivo en el aprovechamiento que la aplicación hace de los recursos del procesador, y afecta al número medio de ciclos por instrucción. Por tanto, puede tener interés expresar el valor de CPI a partir de características de la microarquitectura en cuestión. Así, se puede utilizar el número medio de ciclos que tienen que transcurrir desde que se emite una o varias instrucciones hasta que se puede realizar otra emisión, CPE , y el número medio de instrucciones que se emiten, IPE , de manera que $CPI = CPE / IPE$. En un procesador no segmentado, en el que las instrucciones se van ejecutando una a una, CPE sería igual al número medio de ciclos de reloj que tarda en procesarse la instrucción dado que hasta que no termine de procesarse una instrucción no se empieza a procesar la siguiente, e IPE sería 1 dado que las instrucciones se emiten una a una. En un procesador segmentado, el valor máximo de CPE es igual a 1 dado que cada ciclo podrían empezar a ejecutarse instrucciones. Por lo tanto, $CPI = 1 / IPE$. Si el procesador segmentado sólo puede empezar a ejecutar una instrucción por ciclo, $CPI = 1$. Esta situación se tendría en un caso ideal, dado que en un procesador segmentado (debido a las dependencias de datos, de control, y las colisiones por el acceso a recursos comunes), no en todos los ciclos pueden empezar a ejecutarse instrucciones. Por lo tanto, CPE suele ser mayor que 1 y lo mismo ocurriría con CPI . Cuanto más próximo a 1 sea CPI , más eficientemente se estará utilizando el cauce del procesador. En el caso de un procesador supercalar o VLIW, se pueden emitir varias instrucciones por ciclo. De hecho, el valor de IPE máximo que podría alcanzar el procesador es una medida de la capacidad paralelismo entre instrucciones (ILP) que aprovecharía el procesador. Así, CPI podría ser menor que 1, y tanto menor cuanto más paralelismo ILP aproveche. Si tenemos en cuenta estos parámetros relacionados con las características del repertorio de instrucciones y de la microarquitectura, la expresión (3) se transformaría en:

$$T_{CPU} = NI \times CPI \times T_{ciclo} = \frac{N_{oper}}{OPI} \times \frac{CPE}{IPE} \times T_{ciclo} \quad (4)$$

El tiempo de ciclo (o la frecuencia) depende fundamentalmente de la tecnología de integración utilizada. No obstante, las características de diseño e implementación de la microarquitectura también afectan al tiempo de ciclo al que pueden funcionar los circuitos.

3. Medidas de velocidad de procesamiento y *benchmarks*

Además del tiempo de CPU, existen otras medidas de prestaciones utilizadas muy frecuentemente. Entre estas están las que se refieren a la velocidad de ejecución de instrucciones u operaciones (usualmente operaciones de datos en coma flotante).

Los MIPS (millones de instrucciones por segundo) miden, como su propio nombre indica, el número (en millones) de instrucciones máquina ejecutadas por unidad de tiempo (en segundos). Se pueden obtener a partir de:

$$MIPS = \frac{NI}{T_{CPU} \times 10^6} = \frac{NI}{NI \times CPI \times T_{ciclo} \times 10^6} = \frac{1}{CPI \times T_{ciclo} \times 10^6} = \frac{F}{CPI \times 10^6} \quad (5)$$

En la expresión (5) se considera que el tiempo transcurrido es el tiempo de CPU, calculado a través de la expresión (2). A medida que los computadores son más rápidos, el número de MIPS que alcanzan crece, y para evitar tener que usar valores de MIPS muy elevados, actualmente es frecuente utilizar medidas como los GIPS, donde el número de instrucciones ejecutado se expresa en miles de millones: se usa 10^9 en lugar de 10^6 en el denominador de la expresión (5).

El uso de los MIPS para comparar las prestaciones de los computadores implica tener en cuenta ciertas cuestiones. Por una parte, los computadores que se comparan deben tener el mismo repertorio de instrucciones máquina. Un mismo programa codificado con instrucciones sencillas, por ejemplo las propias de un repertorio tipo RISC (Computador de Conjunto de Instrucciones Reducido), usualmente genera un número mayor de instrucciones máquina que si se utiliza un repertorio con instrucciones de mayor contenido semántico, como las de los repertorios de tipo CISC (Computador de Conjunto de Instrucciones Complejo). Si, por ejemplo, suponemos que el programa tarda en ejecutarse el mismo tiempo en las dos máquinas, el computador con repertorio más sencillo tendría un mayor valor de MIPS ya que ha ejecutado más instrucciones en el mismo tiempo que el computador con repertorio más complejo. Sin embargo, dado que el tiempo de ejecución ha sido el mismo, las prestaciones son iguales en los dos casos. La cuestión es que los MIPS miden la velocidad con que cada procesador ejecuta las instrucciones de su repertorio. Si comparásemos computadores con el mismo repertorio, tener información de la rapidez con que ejecutan sus instrucciones máquina sí daría información sobre los tiempos de ejecución.

En muchos casos, los MIPS se utilizan para referirse a la *velocidad pico* del procesador, entendida como el ritmo máximo al que el procesador puede procesar sus instrucciones. Así, a partir de la expresión (5), teniendo en cuenta que $MIPS=F/(CPI \times 10^6)$, considerando que $CPI=1/IPC$ (el número de ciclos por instrucción es la inversa del número de instrucciones por ciclo), el valor más pequeño para CPI correspondería al máximo número de instrucciones que el procesador puede terminar por ciclo, IPC . Conociendo la frecuencia a la que funciona el procesador, el valor pico para IPC nos permitiría obtener los *MIPS* pico del procesador. Por otra parte, el valor pico de IPC se puede deducir de las características de la propia microarquitectura. Por ejemplo, si un procesador superescalar que funciona a una frecuencia de 2 GHz tiene recursos para completar dos instrucciones por ciclo como máximo, tendremos que $IPC_{pico}=2$, o lo que es lo mismo $CPI_{pico}=0.5$, y por lo tanto proporcionará 4 GIPS (es decir 2×10^9 ciclos/sg)/(0.5 ciclos x 10^9). En el capítulo 4 se verán más ejemplos de relaciones entre la microarquitectura de un procesador y valores de IPC .

Otra medida similar a los MIPS, pero ahora referida a la capacidad de procesamiento de operaciones en coma flotante, son los MFLOPS, es decir, Millones de Operaciones en Coma Flotante por Segundo, que se obtendrían a partir de la expresión:

$$MFLOPS = \frac{\text{Operaciones en Coma Flotante}}{T_{CPU} \times 10^6} \quad (6)$$

Como en el caso anterior, el aumento de velocidad de los procesadores ha hecho que el número de MFLOPS haya crecido considerablemente. Para evitar el uso de valores muy elevados de MFLOPS se suelen utilizar medidas como los GFLOPS (GigaFLOPS, dividiendo por 10^9) TFLOPS (TeraFLOPS, dividiendo por 10^{12}), PFLOPS (PetaFLOPS, dividiendo por 10^{15}), EFLOPS (ExaFLOPS, dividiendo por 10^{18}), etc. También aquí, si el número de operaciones en coma flotante que se utiliza en la expresión (6) corresponde con el valor pico de operaciones en coma flotante que un procesador puede terminar por unidad de tiempo, se tienen los correspondientes MFLOPS, GFLOPS, TFLOPS, ... pico.

Tanto para evaluar los MIPS como los MFLOPS, si no nos referimos a los valores pico de esas magnitudes, habría que utilizar un programa o un conjunto de programas sobre los que hacer las medidas de tiempo y de número de instrucciones (en el caso de los MIPS), o de operaciones de coma flotante (en el caso de los MFLOPS). Para establecer un marco de referencia común se definen los *benchmarks*, o conjuntos de programas de prueba para evaluar las prestaciones. Existen distintos consorcios que han propuesto diferentes *benchmarks*, en función del tipo de aplicaciones que son de su interés (de cálculo científico, de transacciones, de análisis de datos, etc.), o del tipo de recursos que se intentan evaluar preferentemente (la CPU y la memoria, el hardware de E/S, etc.). Entre los *benchmarks* más populares están los SPEC, TPC, Linpacks, etc. Recientemente, con el auge de las aplicaciones relacionadas con *Big Data*, se han propuesto *benchmarks*, como por ejemplo BigDataBench, que incluyen la resolución de problemas frecuentes en este ámbito. Se pueden consultar las características de muchos de estos *benchmarks*, su área de aplicación, e incluso acceder a los propios programas y resultados de evaluación de computadores en las páginas Web de cada *benchmark*.

4. Estimación de tiempo de CPU

En muchos casos es importante estimar el tiempo de CPU (como se ha indicado anteriormente, aquí no se considerará el efecto de las E/S ni los tiempos de espera asociados a la asignación del procesador a otro proceso) que puede necesitar un programa a partir de las características del computador en el que se ejecuta y del conjunto de operaciones y del patrón de accesos a memoria al ejecutar el programa. Se trata de obtener aproximaciones a los valores mínimos de los tiempos de CPU que nos permitan determinar si la ejecución de un programa se ajustará o no a las restricciones de tiempo que se tengan.

Las distintas expresiones que se han proporcionado hasta ahora para el tiempo de CPU suelen considerar que los datos a los que se accede están en la caché de primer nivel. Es decir se utilizan valores de CPI para las instrucciones de acceso a memoria que incluyen retardos de unos pocos ciclos (uno o dos ciclos) de retardo (o *latencia*) entre que se genera la dirección de memoria y se tiene el dato. Sin embargo, el tiempo de una instrucción de acceso a memoria puede ser bastante mayor si hay que acceder a niveles de caché inferiores o a la memoria principal. Es posible hacer una estimación del tiempo mínimo de CPU utilizando la expresión (2), sin tener en cuenta la existencia de fallos de caché, y considerando que el procesador funciona a pleno rendimiento. Se utilizaría el valor pico de *IPC* (el número de instrucciones que pueden terminarse a pleno rendimiento según las características de la microarquitectura) que correspondería al valor más pequeño posible para *CPI* y, a partir del número de instrucciones máquina del programa, *NI*, y de la frecuencia (o del tiempo de ciclo, T_{ciclo}) a la que funciona el procesador, se tendría el tiempo de CPU mínimo que necesitaría el programa.

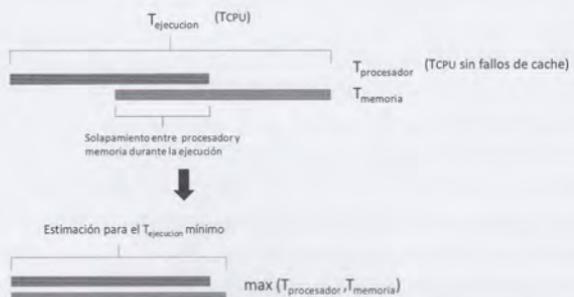


Figura 1. Solapamiento de tiempos para la estimación de prestaciones

No obstante, si se tiene información de las características de la jerarquía de memoria se podría tener una estimación algo mejor del tiempo mínimo de CPU. Como se muestra en la Figura 1, se puede considerar que al ejecutar el programa puede existir solapamiento entre la ejecución de instrucciones en el procesador y el acceso a memoria principal por parte de instrucciones de carga o escritura de datos para las que se producen fallos de caché. En el mejor de los casos se podría suponer que existe un solapamiento total entre el tiempo consumido por las instrucciones del procesador (el tiempo de procesamiento mínimo considerando que no hay faltas de caché) y el tiempo de acceso a memoria principal según las características de la jerarquía de memoria y el patrón de acceso a los datos del programa (que dará lugar a un mayor o menor número de fallos). Por tanto, el tiempo mínimo para ejecutar el programa sería igual al mayor de los dos tiempos, el tiempo de procesamiento en la CPU sin faltas de caché, y al tiempo de acceso a memoria:

$$T_{ejecución_min} = \max(T_{procesador}, T_{memoria}) \quad (7)$$

donde $T_{procesador}$ es el tiempo de CPU, T_{CPU} , sin faltas de caché y $T_{memoria}$ es el tiempo de acceso a la jerarquía de memoria (se tienen en cuenta los fallos de caché) que se obtiene

como el producto del número de accesos a memoria, N_{acceso} , y el tiempo medio por acceso, $t_{memoria}$, es decir, $T_{memoria} = N_{acceso} \times t_{memoria}$. Para estimar el tiempo medio de un acceso a memoria, $t_{memoria}$, hay que tener en cuenta las características de la jerarquía de memoria (tiempos de acceso a los distintos niveles de la jerarquía, características de los ciclos de bus, tamaños de las líneas de caché, etc.) y la tasa de fallos que se producen. Para una jerarquía de memoria de dos niveles en el que el primero corresponde a la memoria caché (caché de nivel L1):

$$t_{memoria} = a_1 \times t_1 + (1 - a_1) \times (t_1 + t_M) \quad (8)$$

donde a_1 es la tasa de aciertos a caché (la probabilidad de que un acceso a memoria esté en caché), t_1 es el tiempo de acceso a la caché de primer nivel, y t_M el tiempo de acceso a la memoria principal. En la expresión anterior se está suponiendo que todos los accesos se producen en primer lugar a caché y cuando la palabra no está ahí, se genera el acceso a memoria principal (caché *look-through*). En (8) no se ha tenido en cuenta el tiempo que se consume cuando, además de reemplazar una línea de caché con información que se trae desde memoria principal, hay que actualizar en memoria principal el contenido de la línea reemplazada (que el procesador ha modificado con respecto a lo que se captó de memoria principal en su momento). Así, en una jerarquía de memoria de dos niveles con una caché L1 con post-escritura (se actualiza la información modificada en caché en el momento del reemplazo de la línea) el tiempo medio de acceso será:

$$t_{memoria} = a_1 \times t_1 + (1 - a_1) \times (t_1 + t_M + p_{reemplazo} \times t_{linea}) \quad (9)$$

donde $p_{reemplazo}$ es la probabilidad de que, cuando se produzca una falta, haya que reemplazar la línea actualizando en memoria la línea reemplazada, y t_{linea} es el tiempo necesario para actualizar la línea. Para estimar la tasa de aciertos, a_1 , se tienen en cuenta los accesos a memoria que genera el programa y se determina en qué casos se producen fallos. A partir del patrón de accesos a memoria y de las características de la caché se podrían estimar también la probabilidad de reemplazo $p_{reemplazo}$. El valor de t_1 es el tiempo de acceso de la caché L1, y los valores de t_M y t_{linea} vienen determinados por la arquitectura de la conexión con la memoria principal, y de las características de los buses de memoria y la tecnología de los circuitos de memoria DRAM. El controlador de memoria genera las señales que deben intercambiarse con las memorias DRAM (las direcciones de fila y de columna de las celdas de memoria, las señales que validan esas direcciones, las que permiten controlar el refresco de las celdas de memoria, etc.) a partir de las que recibe del procesador (o de cualquier otro elemento que pueda realizar accesos a la memoria principal, como por ejemplo un controlador de DMA). Así, una transferencia de datos entre el procesador y la memoria (el controlador de memoria) se lleva a cabo intercambiando una serie de señales de datos, direcciones y control a lo largo de uno o varios ciclos de reloj. Los ciclos necesarios para realizar una transferencia entre el procesador y la memoria (entre el procesador y el controlador de memoria que, a su vez se tiene que comunicar con el propio circuito de memoria) se denomina *ciclo de bus*.

Usualmente, un ciclo de bus comprende varios ciclos de reloj, y ese número puede ser variable en función de que la memoria (a través del controlador de memoria) tarde más o menos tiempo en completar la transacción que el procesador solicita (proporcionar el dato en el caso de una escritura, o terminar la escritura del dato en las celdas de memoria correspondientes). Usualmente, el procesador incorpora diferentes tipos de ciclos de bus para mejorar el ancho de banda de intercambio de información con la memoria. Dado que en sistemas con caché, el acceso a la memoria externa suele implicar el acceso a varias palabras de memoria principal para leer o escribir una línea de caché, existen ciclos de bus específicos, como los *ciclos burst o a ráfagas*, en los que el procesador sólo genera la dirección de una palabra dentro de la línea, e indica, con las correspondientes señales de control, que desea todas las palabras de la línea en la que se incluye la dirección que ha proporcionado. El controlador de memoria se encarga de generar las señales para que la DRAM proporcione esos datos. Como, por otra parte, se trata de palabras que estarán en posiciones contiguas de memoria, se aprovechará eficientemente el diseño de la arquitectura de memoria que favorece los accesos a posiciones consecutivas.

Teniendo en cuenta la expresión (9), para mejorar las prestaciones de la jerarquía de memoria, se podrían reducir los tiempos de acceso a los distintos niveles, reducir la tasa de fallos a los niveles de caché, y/o reducir la penalización cuando se producen fallos. Entre las técnicas para reducir la tasa de fallos, junto a técnicas basadas en la inclusión de recursos como las cachés de víctimas o las pseudo-asociativas, también existen procedimientos que se pueden aplicar desde el nivel de programación, tales como las técnicas de precaptación (que también requieren o pueden beneficiarse de ciertos recursos en el procesador) y otras técnicas de optimización de código como la mezcla de arrays, la fusión de bucles, o las operaciones con submatrices (*blocking*). Existen bastantes textos donde se pueden consultar detalles de estas técnicas, entre ellos [ORT05].

5. Ganancia de velocidad y ley de Amdahl

Uno de los cometidos más importantes de la Arquitectura de Computadores consiste en determinar dónde están los cuellos de botella del computador y proporcionar estrategias para evitarlos y mejorar las prestaciones. Para medir el resultado de una mejora se puede utilizar la ganancia de velocidad, que compara la velocidad de un computador antes y después de mejorar alguno de sus recursos. La expresión correspondiente a la definición de ganancia de velocidad, S_p , es la siguiente:

$$S_p = \frac{V_p}{V_1} = \frac{(W/T_p)}{(W/T_1)} = \frac{T_1}{T_p} \quad (10)$$

en la que V_p es la velocidad de ejecución del programa o conjunto de programas antes de aplicar la mejora y V_p es la velocidad con la mejora. Como la carga de trabajo, W , que se realiza en ambos casos es la misma, los numeradores en ambas medidas de velocidad son iguales, y la ganancia de velocidad se puede expresar en términos del cociente entre

el tiempo que tarda en procesarse la carga de trabajo antes de aplicar la mejora, T_p , y el tiempo de procesamiento de la carga de trabajo tras la mejora, T_p' . Los subíndices 1 y p que se utilizan pretenden dar a entender que se ha aplicado una mejora consistente en hacer p veces más rápido alguno de los recursos del computador.

La ley de Amdahl establece una cota superior a la ganancia de velocidad, S_p , que se puede conseguir al mejorar alguno de los recursos del computador en un factor igual a p , y según la frecuencia con que se utiliza ese recurso en la máquina de partida. Concretamente, la ley de Amdahl establece que:

$$S_p \leq \frac{1}{1 + f \times (p - 1)} \quad (11)$$

donde f es la fracción del tiempo de ejecución antes de aplicar la mejora en un recurso en la que no se utiliza dicho recurso (y por lo tanto, la mejora no tendría ningún efecto). Así, si $f = 1$ (es decir, el recurso mejorado no se utiliza) S_p es menor o igual a 1 y, por lo tanto no se produciría mejora de velocidad alguna. Sólo si $f = 0$ (el recurso mejorado se utiliza durante todo el tiempo T_p), S_p podría alcanzar una valor igual a p . Es decir, solo si $f=0$ se podrá “observar” la mejora de velocidad igual a p en el recurso mejorado.

La justificación de la ley de Amdahl se puede obtener teniendo en cuenta que en el tiempo de ejecución de la carga de trabajo en el recurso sin mejora, T_p , se puede distinguir una parte, f , donde no se utiliza el recurso y otra parte, $(1-f) \times T_p$, en la que se utiliza dicho recurso, es decir:

$$T_1 = f \times T_p + (1 - f) \times T_p \quad (12)$$

Por lo tanto el tiempo de ejecución de la carga de trabajo con el recurso mejorado en un factor p , T_p' , debe cumplir:

$$f \times T_p + \frac{(1 - f)}{p} \times T_p \leq T_p' \quad (13)$$

dado que la parte de tiempo $f \times T_p$, donde no se utiliza el recurso mejorado no se reducirá, y la parte $(1-f) \times T_p$, donde se utiliza el recurso podría reducirse como máximo un factor p . Sin más que sustituir (12) y (13) en (10) se tendría la desigualdad (11) correspondiente a la ley de Amdahl.

6. Cuestiones y Problemas

Cuestión 1. Si le dicen que un ordenador es de 32 GIPS es cierto que ejecutará cualquier programa de 32000 instrucciones en un microsegundo.

Solución

Los GIPS (Giga Instrucciones Por Segundo) se definen como el número de instrucciones que ejecuta un procesador (en miles de millones) divididos por el tiempo que tardan en ejecutarse expresado en segundos. Esta medida es similar a los MIPS, que se utilizaban cuando los procesadores eran menos veloces. Los MIPS se definen también como el cociente entre el número de instrucciones ejecutadas y tiempo de ejecución en segundos, pero ahora el número de instrucciones se expresa en millones:

$$MIPS = \frac{\text{Número de Instrucciones}}{\text{tiempo (segundos)} \times 10^6}$$

Estas medidas de instrucciones máquina por segundo (MIPS y GIPS) suelen utilizarse como medida de las prestaciones de un procesador, pero hay que tener en cuenta que:

- El tiempo que tarda en ejecutarse un programa depende de las instrucciones que lo constituyen. La distribución de instrucciones de operación con enteros o de coma flotante, el número de instrucciones de salto o accesos a memoria pueden hacer que programas con el mismo número de instrucciones tarden tiempos diferentes en ejecutarse. Según esto, para poder comparar medidas de MIPS (o GIPS) habría que especificar también el programa con el que se hace.
- Las características de las instrucciones cambian en repertorios de instrucciones diferentes. Hay repertorios con instrucciones que codifican operaciones sencillas, y otros que contienen instrucciones que implementan varias operaciones. De esta forma, un mismo programa puede ser codificado utilizando un número distinto de instrucciones en repertorios de instrucciones máquina diferentes. Por ejemplo, si un procesador tiene un repertorio de instrucciones de tipo CISC que permite codificar un algoritmo con, por ejemplo, 1.000.000 instrucciones, y otro con un repertorio de tipo RISC lo codifica con 2.000.000, si el primero tarda 1 microsegundo y el segundo 1.5 microsegundos, diríamos que el primero alcanza 1000 GIPS y el segundo 1333 GIPS. Por tanto, si nos fijamos en los GIPS, el segundo procesador parecería más veloz porque ejecuta más instrucciones por segundo, aún habiendo que ha completado el trabajo más lentamente porque ha tardado un 50% más de tiempo que el primer procesador. Así, solo podríamos comparar máquinas cuyos procesadores tengan microarquitecturas que implementen el mismo repertorio de instrucciones máquina. Incluso en este caso deberíamos estar seguros de que los códigos generados por el compilador utilizado en cada caso son iguales.

Teniendo en cuenta los dos puntos anteriores, incluso se ha dicho que el verdadero significado de MIPS realmente era *Meaningless Information of Processor Speed* (información sin sentido de la velocidad del procesador). No obstante, esta medida sí proporciona una información útil si se utiliza para indicar la velocidad máxima a la que un procesador, con un repertorio de instrucciones dado, puede terminar instrucciones. Lo que se denomina *velocidad pico*. Por ejemplo, si un procesador de 1 GHz puede

terminar una instrucción por ciclo de reloj como máximo, su velocidad pico sería $(1 \text{ instrucción/ciclo})/1 \times 10^{-9} \text{ (segundos/ciclo)} = 1 \times 10^9 \text{ instrucciones/segundo}$, o lo que es lo mismo 1 GIPS. Es decir, $(1 \times 10^9 \text{ instrucciones})/(1 \text{ segundo} \times 10^9)$.

En resumen, a no ser que los MIPS (o los GIPS) se utilicen como medida de velocidad pico para comparar procesadores con el mismo repertorio de instrucciones, la única forma de usar esta medida para comparar dos máquinas es que, además de que ambas tengan el mismo repertorio, se use el mismo programa (con las mismas instrucciones máquina generadas por el compilador).

En relación a la pregunta que se plantea, si un programa que ejecuta 32.000 instrucciones tarda 1 microsegundo, tendríamos que, efectivamente, el programa se ha ejecutado a una velocidad de 32 GIPS:

$$GIPS = \frac{32.000 \text{ instrucciones}}{1 \times 10^{-6} \text{ segundos} \times 10^9} = \frac{32 \times 10^3}{1 \times 10^3} = 32$$

No obstante, NO podríamos estar seguros de que cualquier programa de 32000 instrucciones tardaría en ejecutarse un microsegundo porque dependería de las características de las instrucciones que lo compongan.

Sin ir más lejos, el número de instrucciones que constituyen un programa (número estático de instrucciones) puede ser distinto del número de instrucciones que finalmente ejecuta el procesador (número dinámico de instrucciones), ya que puede haber instrucciones de salto, bucles, etc. que hacen que ciertas instrucciones del código se ejecuten más de una vez, y otras no se ejecuten nunca.

Por otro lado, según el tipo de instrucciones que constituyan el programa y las dependencias entre dichas instrucciones, pueden variar los tiempos que tardan en ejecutarse las instrucciones.

Problema 1. En el bucle siguiente, los arrays $a[]$, $b[]$, $c[]$, y $d[]$, son números en coma flotante de 64 bits, y $n=2 \times 10^{10}$:

```
for (i = 0 ; i < n ; i++)
    d[i] = a[i] + b[i] + c[i];
```

Si el programa se ejecuta en un procesador a 2 GHz que puede terminar dos operaciones en coma flotante por ciclo, ¿cuál es el tiempo mínimo que tardaría en ejecutarse?. ¿Cuántos GFLOPS de velocidad pico tiene el procesador?.

Solución

El tiempo mínimo que tardaría en ejecutarse el programa se puede estimar teniendo en cuenta el número máximo de instrucciones en coma flotante por ciclo que puede terminar el programa. Obviamente, teniendo en cuenta que el programa debe ejecutar

más instrucciones (las de carga de memoria, control del bucle, etc.) y que hay que realizar accesos a memoria que, en ciertos casos darán lugar a faltas de caché y consumirán su tiempo correspondiente, la ejecución del programa consumirá más tiempo que el que obtengamos considerando sólo la ejecución de operaciones en coma flotante, pero lo que pretendemos es obtener una estimación del tiempo mínimo de ejecución.

$$T_{CPU}(\text{min}) \approx N_{\text{float}} \times CPI_{\text{float}} \times T_{\text{ciclo}} \approx \frac{N_{\text{float}}}{IPC_{\text{float}} \times F}$$

donde N_{float} es el número de operaciones en coma flotante, CPI_{float} es el número de ciclos por operación en coma flotante, que es igual a la inversa del número de operaciones o instrucciones en coma flotante por ciclo, IPC_{float} , y T_{ciclo} es tiempo por ciclo, es decir, la inversa de la frecuencia de reloj, F .

Por tanto, dado que $N_{\text{float}}=2 \times 10^{10}$, que $IPC_{\text{float}}=2$ (terminan dos operaciones en coma flotante por ciclo), y que $F=2 \times 10^9$ ciclos/segundo obtenemos:

$$T_{CPU}(\text{min}) \approx \frac{2 \times 10^{10} \text{ op. com. flot.}}{2 \left(\frac{\text{op. com. flot.}}{\text{ciclo}} \right) \times \left(2 \times 10^9 \frac{\text{ciclos}}{\text{segundo}} \right)} = 5 \text{ segundos}$$

En cuanto al número de GFLOPS pico del procesador, dado que puede terminar como máximo dos operaciones en coma flotante por ciclo, es decir $IPC_{\text{float}}=2$ operaciones en coma flotante por ciclo, tendremos que:

$$\begin{aligned} GFLOPS_{\text{pico}} &= IPC_{\text{float}} \left(\frac{\text{op. com. flot.}}{\text{ciclo}} \right) \times F \left(\frac{\text{ciclos}}{\text{segundo}} \right) \times 10^{-9} = \\ &= 2 \left(\frac{\text{op. com. flot.}}{\text{ciclo}} \right) \times \left(2 \times 10^9 \frac{\text{ciclos}}{\text{segundo}} \right) \times 10^{-9} = 4 \end{aligned}$$

Por tanto, el procesador tiene 4 GFLOPS de velocidad pico (se terminan 2 operaciones en coma flotante por ciclo y cada ciclo es de 0.5 ns).

Problema 2. En un procesador sin segmentación de cauce, se plantean dos alternativas en el repertorio de instrucciones máquina para implementar los saltos condicionales:

- ALT1. Utilizar dos instrucciones: una instrucción de comparación COMP actualiza un código de condición y es seguida por una instrucción de salto BRANCH que comprueba esa condición.
- ALT2. Utilizar una sola instrucción: esa instrucción (COMP+BRANCH) incluiría la funcionalidad de las instrucciones COMP y BRANCH de la alternativa ALT1.

Indique qué alternativa es la mejor teniendo en cuenta que hay un 30% de instrucciones BRANCH en los programas con la primera alternativa (ALT1); que las instrucciones BRANCH (de ALT1) y COMP+BRANCH (de ALT2) necesitan 4 ciclos mientras que todas

las demás necesitan sólo 3; y que el ciclo de reloj de la ALT1 es un 15% menor que el de la ALT2, dado que al incluirse en la alternativa ALT2 una instrucción de mayor funcionalidad (la instrucción COMP+BRANCH) la microarquitectura del procesador es algo más compleja en este caso y el hardware correspondiente algo más lento (los diseños de las microarquitecturas utilizan el mismo número de ciclos para las instrucciones del mismo tipo en las dos alternativas, pero el ciclo de reloj es mayor en la microarquitectura del procesador de la ALT2).

Solución

Para resolver este problema determinaremos el tiempo medio de CPU para cada una de las alternativas y los compararemos. La alternativa para la que ese tiempo medio de CPU sea menor será la mejor. El tiempo medio de CPU se puede expresar como:

$$T_{CPU} = CPI \times NI \times T_{CICLO}$$

donde CPI es el número medio de ciclos por instrucción, NI es el número de instrucciones, y T_{CICLO} el tiempo de ciclo del procesador. La situación en ALT1 se resume en la Tabla 1

Tabla 1. Situación de la ALT1

Tipo de Instrucción	Número de Instrucciones	Ciclos de cada tipo de instrucción
BRANCH	$0.3NI_1$	4
Resto	$0.7NI_1$	3
	$NI_1 = 0.3NI_1 + 0.7NI_1$	$CPI_1 = 0.3 \times 4 + 0.7 \times 3 = 3.3$

y tendríamos que calcular:

$$T_{CPU}(1) = CPI_1 \times NI_1 \times T_{CICLO}(1)$$

El valor de CPI_1 se obtiene sumando, para todos los tipos de instrucciones, los productos del número de ciclos que necesita cada instrucción en ALT1 y la frecuencia con que aparece dicha instrucción:

$$CPI_1 = \left(\frac{0.3 \times NI_1}{NI_1} \times 4 \right) + \left(\frac{0.7 \times NI_1}{NI_1} \times 3 \right) = 3.3$$

y, por tanto

$$T_{CPU}(1) = 3.3 \times NI_1 \times T_{CICLO}(1)$$

A continuación se calcula el tiempo medio de CPU para la segunda alternativa, ALT2:

$$T_{CPU}(2) = CPI_2 \times NI_2 \times T_{CICLO}(2)$$

En realidad, lo que se va a hacer es expresarlo en términos del número medio de ciclos por instrucción, el número de instrucciones, y el tiempo de ciclo de la alternativa ALT1 para poder comparar ambas expresiones y determinar cuál es la mejor.

Así, para determinar el número de instrucciones que habría en la ALT2 hay que tener en cuenta lo siguiente:

- En los códigos de ALT2 tendremos $0.3NI_1$ instrucciones COMP+BRANCH dado que habrá que utilizar una de esas instrucciones para expresar el salto condicional, que en ALT1 se hacía utilizando sus instrucciones BRANCH.
- En los códigos de ALT2 no se tendrán las instrucciones COMP, que en los códigos de ALT1 estaban asociadas a instrucciones BRANCH para implementar el salto condicional, ya que esta función se encuentra incluida en las instrucciones COMP+BRANCH que se utiliza en ALT2. Esto quiere decir que habrá $0.4NI_1$ instrucciones distintas de las instrucciones COMP+BRANCH en ALT2, es decir $0.7NI_1 - 0.3NI_1$ (restamos a las instrucciones que había en ALT1 las COMP que desaparecen en ALT2).

Así, el número de instrucciones que quedan en ALT2 son

$$NI_2 = 0.3NI_1 + (0.7NI_1 - 0.3NI_1) = 0.7NI_1$$

y, dado que el número de ciclos de las instrucciones BRANCH de ALT1 y las COMP+BRANCH de ALT2 es igual a 4 y que el resto de instrucciones mantienen el número de ciclos que necesitan, el valor de CPI_2 es:

$$CPI_2 = \left(\frac{0.3 \times NI_1}{0.7 \times NI_1} \times 4 \right) + \left(\frac{0.4 \times NI_1}{0.7 \times NI_1} \times 3 \right) = \left(\frac{0.3}{0.7} \times 4 \right) + \left(\frac{0.4}{0.7} \times 3 \right) = 3.43$$

La situación de la ALT2 se resume en la Tabla 2

Tabla 2. Situación de la ALT2

Tipo de Instrucción	Número de Instrucciones	Ciclos de cada tipo de instrucción
COMP+BRANCH	$0.3NI_1$	4
Resto	$(0.7NI_1 - 0.3NI_1) = 0.4NI_1$	3
	$NI_2 = 0.3NI_1 + 0.4NI_1 = 0.7NI_1$	$CPI_2 = 0.43 \times 4 + 0.57 \times 3 = 3.43$

Además, se tiene que el ciclo de reloj de ALT2 es más largo que el de ALT1. Concretamente, se dice en el enunciado que el ciclo de ALT1 es un 15% menor que el de ALT2, lo que implica que:

$$T_{CICLO}(1) = 0.85 \times T_{CICLO}(2) \rightarrow T_{CICLO}(2) = (1/0.85) \times T_{CICLO}(1) = 1.18 \times T_{CICLO}(1)$$

Sustituyendo $NI(2)$, $CPI(2)$, y $T_{CICLO}(2)$ en la expresión de $T_{CPU}(2)$ se tiene que:

$$\begin{aligned} T_{CPU}(2) &= CPI_2 \times NI_2 \times T_{CICLO}(2) = 3.43 \times 0.7NI_1 \times 1.18T_{CICLO}(1) \\ &= 2.83 \times NI_1 \times T_{CICLO}(1) \end{aligned}$$

Así, como $T_{CPU}(1) > T_{CPU}(2)$ se tiene que ALT2 mejora a la ALT1. Es decir, un repertorio con instrucciones más complejas, al reducir el número de instrucciones de los programas, puede mejorar las prestaciones aunque al implementar esta opción la mayor

complejidad del procesador lo haga algo más lento. Sin embargo, esto no tiene que ser así en todos los casos. Depende de la reducción en el número de instrucciones que se produzca, y de la velocidad con que el nuevo procesador ejecute las instrucciones, tal y como se ilustra en el siguiente problema.

Problema 3. Qué ocurriría en el caso de las alternativas ALT1 y ALT2 del problema anterior: (a) si hubiera solamente un 20% de instrucciones BRANCH en ALT1, y (b) si el tiempo reloj de ciclo de reloj de ALT2 fuese un 45% mayor que el de ALT1. (NOTA: solo cambia la condición que se indica en cada nuevo caso, las condiciones de las que no se dice nada se mantienen iguales a las indicadas en el problema anterior).

Solución

Compararemos las ALT1 y ALT2 para las nuevas condiciones que se indican en los apartados (a) y (b).

Solución para el apartado (a). Las Tablas 3 y 4 muestran, respectivamente, las nuevas situaciones en cada una de las alternativas *ALT1* y *ALT2*

Tabla 3. Situación de la ALT1 en (a)

Tipo de Instrucción	Número de Instrucciones	Ciclos de cada tipo de instrucción
BRANCH	$0.2NI_1$	4
Resto	$0.8NI_1$	3
	$NI_1=0.2NI_1+0.8NI_1$	$CPI_1=0.2\times 4+0.8\times 3=3.2$

Tabla 4. Situación de la ALT2 en (b)

Tipo de Instrucción	Número de Instrucciones	Ciclos de cada tipo de instrucción
COMP+BRANCH	$0.2NI_1$	4
Resto	$(0.8NI_1-0.2NI_1)=0.6NI_1$	3
	$NI_2=0.2NI_1+0.6NI_1=0.8NI_1$	$CPI_2=0.25\times 4+0.75\times 3=3.25$

Por lo tanto tendremos que los nuevos valores de los tiempos medios de CPU para las alternativas ALT1 y ALT2 serán, respectivamente:

$$T_{CPU}(1) = 3.2 \times NI_1 \times T_{CICLO}(1)$$

y

$$T_{CPU}(2) = CPI_2 \times NI_2 \times T_{CICLO}(2) = 3.25 \times 0.8NI_1 \times 1.18T_{CICLO}(1) \\ = 3.068 \times NI_1 \times T_{CICLO}(1)$$

En este caso, se sigue obteniendo mejora, aunque es menor que la obtenida anteriormente. Si se compara lo sucedido con la alternativa ALT2 en esta situación (a) en relación con lo que ocurría en el problema anterior es que el porcentaje de instrucciones COMP+BRANCH que se introducen es menor. Al tratarse de las instrucciones más lentas

(necesitan 4 ciclos) el incremento que se produce en *CPI* es menor (3.25 frente a 3.43) y aunque hay una menor reducción en el tamaño de los códigos a ejecutar (ahora los códigos tienen $0.8 \times NI_1$ en lugar de $0.7 \times NI_1$) el producto del *CPI* por el número de instrucciones sigue siendo lo suficientemente bajo como para compensar el incremento en el tiempo de ciclo de reloj ($3.25 \times 0.8 = 2.6$ es mayor que $3.43 \times 0.7 = 2.4$ que se tenía en ALT2, pero el valor $2.6 \times 1.18 = 3.068$ que se tiene ahora en ALT2 sigue siendo menor que el valor de 3.2 que se tiene en ALT1).

Solución para el apartado (b). En este caso, no cambian las condiciones relacionadas con el número de instrucciones de cada tipo que se tiene, lo único que sucede es que el tiempo de ciclo del procesador en ALT2 es un 45% más alto, en lugar de un 18%. Por lo tanto:

$$T_{CICLO}(2) = 1.45 \times T_{CICLO}(1)$$

y, sustituyendo, se tendría que:

$$\begin{aligned} T_{CPU}(2) &= CPI_2 \times NI_2 \times T_{CICLO}(2) = 3.43 \times 0.7NI_1 \times 1.45T_{CICLO}(1) \\ &= 3.48 \times NI_1 \times T_{CICLO}(1) \end{aligned}$$

Dado que se tenía que

$$T_{CPU}(1) = 3.3 \times NI_1 \times T_{CICLO}(1)$$

en este caso ALT2 no mejora a ALT1. Es decir, la reducción en el número de instrucciones que habría que ejecutar no compensa el ligero incremento que se produce en el tiempo medio de ciclos por instrucción (de 3.3 a 3.43) y el incremento en el tiempo de ciclo (un 45% más lento en lugar de un 18% más lento). Se puede extraer como conclusión de este problema y del anterior que, aunque una estrategia pueda ser razonable para mejorar las prestaciones, el resultado final depende de las condiciones en las que se aplique (perfil de instrucciones de los programas que ejecuten) y de las características del diseño hardware (aunque la frecuencia a la que funcione el procesador sea menor, importa cuánto menor sea).

Problema 4. Considere un procesador no segmentado con una arquitectura de tipo LOAD/STORE en la que las operaciones sólo utilizan como operandos registros de la CPU. Para un conjunto de programas representativos de su actividad se tiene que el 40% de las instrucciones son operaciones con la ALU (con 4 CPI), el 20% LOADs (con 4 CPI), el 15% STOREs (con 3 CPI) y el 25% BRANCHs (con 4 CPI).

Además, un 20% de las operaciones con la ALU utilizan operandos en registros, que no se vuelven a utilizar. ¿Se mejorarían las prestaciones si, para sustituir ese 20% de operaciones, se añaden instrucciones con un dato en un registro y otro en memoria, teniendo en cuenta que para ellas el valor de CPI es 5 y que ocasionarían un incremento de un ciclo en el CPI de los BRANCHs, pero que no afectan al ciclo de reloj?

Solución

En primer lugar se calcula el tiempo de CPU para la situación inicial. Para ello, se tiene que:

$$T_{CPU}(1) = CPI(1) \times NI(1) \times T_{ciclo}(1)$$

donde

$$CPI(1) = 0.4 \times 4 + 0.20 \times 4 + 0.15 \times 3 + 0.25 \times 4 = 3.85$$

y, por tanto:

$$T_{CPU}(1) = 3.85 \times NI(1) \times T_{ciclo}(1)$$

En la nueva situación se tiene la siguiente distribución de instrucciones y ciclos para cada una de ellas:

Tabla 5. Situación de la nueva alternativa del problema 5

Instrucción	Número	CPI
Operaciones ALU r-r	$(0.40 - 0.20 \times 0.40) \times NI(1)$	4
LOADs	$(0.20 - 0.20 \times 0.40) \times NI(1)$	4
Operaciones ALU r-m	$0.20 \times 0.40 \times NI(1)$	5
STOREs	$0.15 \times NI(1)$	3
BRANCHs	$0.25 \times NI(1)$	5
Total	$(1 - 0.2 \times 0.4) \times NI(1) = 0.92 \times NI(1)$	4.197

En la Tabla 5 se muestra que, al introducir las nuevas instrucciones de operación con la ALU con uno de los operandos en memoria:

- Se reduce el 20% de las $0.40 \times NI(1)$ instrucciones de operación con la ALU y operandos en registros a las que las nuevas instrucciones sustituyen.
- Se reducen $0.20 \times 0.40 \times NI(1)$ LOADs, ya que según se indica en el enunciado, esos LOADs sólo están en el programa para cargar uno de los operandos de las operaciones con la ALU, que no se vuelven a utilizar nunca más.
- Hay que contabilizar las $0.20 \times 0.40 \times NI(1)$ nuevas instrucciones de operación con la ALU que se introducen en el repertorio (y que sustituyen a las correspondientes operaciones con la ALU y operandos en registros).
- Como resultado, al sumar todas las instrucciones que se tienen en la nueva situación, el número total de instrucciones se reduce (lógicamente, ya que se han ahorrado instrucciones LOADs), siendo igual a

$$NI(2) = 0.92 \times NI(1)$$

Teniendo en cuenta la nueva distribución de instrucciones y sus nuevos CPIs, se tiene que:

$$CPI(2) = \frac{0.40 - 0.20 \times 0.40}{0.92} \times 4 + \frac{0.20 - 0.20 \times 0.40}{0.92} \times 4 + \frac{0.20 \times 0.40}{0.92} \times 5 + \frac{0.15}{0.92} \times 3 \\ + \frac{0.25}{0.92} \times 5 = 4.197$$

Como el tiempo de ciclo no varía se tiene que:

$$T_{CPU}(2) = CPI(2) \times NI(2) \times T_{ciclo}(2) = 4.197 \times 0.92 \times NI(1) \times T_{ciclo}(1)$$

Es decir,

$$T_{CPU}(2) = 3.86 \times NI(1) \times T_{ciclo}(1)$$

Se puede ver en este caso que, aunque por muy poco, $T_{CPU}(1) < T_{CPU}(2)$, y por lo tanto no se mejoran las prestaciones. Si el porcentaje de instrucciones sustituidas fuese un 40% en lugar de un 20%, en ese caso, se puede ver que $NI(2) = 0.84 \times NI(1)$

$$CPI(2) = \frac{0.40 - 0.40 \times 0.40}{0.84} \times 4 + \frac{0.20 - 0.40 \times 0.40}{0.84} \times 4 + \frac{0.40 \times 0.40}{0.84} \times 5 + \frac{0.15}{0.84} \times 3 \\ + \frac{0.25}{0.84} \times 5 = 4.31$$

y, por tanto:

$$T_{CPU}(2) = CPI(2) \times NI(2) \times T_{ciclo}(2) = 4.31 \times 0.84 \times NI(1) \times T_{ciclo}(1)$$

Ahora,

$$T_{CPU}(2) = 3.62 \times NI(1) \times T_{ciclo}(1)$$

y la segunda opción mejora a la primera. Como conclusión, se puede indicar que una determinada decisión de diseño puede suponer una mejora en el rendimiento del computador correspondiente según sean las características de las distribuciones de instrucciones en los programas que constituyen la carga de trabajo característica del computador.

Por tanto, queda clara también la importancia que, para evaluar las prestaciones de los computadores, tiene la definición de conjuntos de *benchmarks* que representen de manera fiable los porcentajes de instrucciones de cada tipo que existen en los programas.

Problema 5. La empresa DataNimbus estima que debe adquirir un nuevo computador con una velocidad pico de 100 TFLOPS para alcanzar los niveles de tiempos de respuesta requeridos en su nueva generación de algoritmos para aplicaciones Big Data. Se ha decidido configurar la máquina a base de nodos HP ProLiant SL230s Gen8. Concretamente, cada uno de estos servidores tiene dos procesadores Sandy

Bridge Intel® Xeon® E5-2670 a 2.60GHz con 8 núcleos/procesador (se puede ver en, por ejemplo, http://h18000.www1.hp.com/products/quickspecs/14231_na/14231_na.pdf):
(a) ¿Cuántos nodos (servidores HP ProLiant SL230s) se necesitan para configurar la máquina de 100 TFLOPS?; **(b)** Clasifique el nuevo servidor que se pretende adquirir, sus nodos, sus encapsulados y sus núcleos dentro de la clasificación de Flynn y dentro de la clasificación que usa como criterio el sistema de memoria; y **(c)** ¿Cuál es el número máximo de operaciones de coma flotante por ciclo de cada core del Intel Xeon E5-2670?

NOTA: La velocidad pico de cada núcleo se puede determinar a partir de la lista TOP500 (<http://www.top500.org/>), en cuya edición de Noviembre de 2012 se tiene que, por ejemplo, el equipo número 13 (el Yellowstone del National Center for Atmospheric Research), que utiliza el procesador Intel® Xeon® E5-2670 a 2.6 GHz, tiene una velocidad pico de 1,503,590 GFLOPS, y contiene 72,288 núcleos.

Solución

(a) Como sabemos, por la información que nos proporciona la lista TOP500 (y que se indica en la NOTA del problema), que la máquina Yellowstone del National Center for Atmospheric Research con el procesador Intel® Xeon® E5-2670 a 2.6 GHz tiene una velocidad pico de 1,503,590 GFLOPS y contiene 72,288 núcleos, la velocidad pico de un núcleo para operaciones en coma flotante (número máximo de GFLOPS que puede proporcionar dicho núcleo) es:

$$1,503,590 \text{ GFLOPS} / 72,288 \text{ núcleos} = 20.8 \text{ GFLOPS/núcleo}$$

Dado que hay que alcanzar 100 TFLOPS = 100×10^3 GFLOPS, el número de núcleos que necesitamos es $(100 \times 10^3 \text{ GFLOPS}) / (20.8 \text{ GFLOPS/núcleo}) = 4,807.7$ núcleos, es decir 4,808 núcleos.

Como el servidor HP ProLiant SL230s que se utiliza en cada nodo tiene 16 núcleos (2 microprocesadores con 8 núcleos cada uno), el número de nodos necesario sería:

$$(4,808 \text{ núcleos}) / (16 \text{ núcleos/nodo}) = 300.5 \rightarrow 301 \text{ nodos}$$

(b) Desde el punto de vista de la taxonomía de Flynn es un computador MIMD. Cada uno de los microprocesadores que hay en el nodo tiene 8 núcleos que comparten la memoria local del microprocesador y por tanto son multiprocesadores UMA. Esos dos microprocesadores se interconectan en el nodo constituyendo un multiprocesador NUMA dado que cada uno de ellos tiene su memoria principal local. Los nodos están interconectados a través de una red y configuran un computador NORMA (también suele denominársele *cluster*).

(c) Como la velocidad pico de un núcleo es igual al producto de la frecuencia de reloj y el número de instrucciones por ciclo (en este caso operaciones en coma flotante por ciclo) tenemos que, en cada núcleo:

$$20.8 \text{ GFLOPS} = (\text{Operac}_\text{Coma}_\text{Flotante/ciclo}) \times \text{Frecuencia (ciclos/s)}$$

$$20.8 \text{ GFLOPS} = (\text{Operac_Coma_Flotante/ciclo}) \times 2.6 \times 10^9 \text{ (ciclos/s)}$$

y, despejando

$$(\text{Operac_Coma_Flotante/ciclo}) = (20.8 \times 10^9 \text{ Operac_Coma_Flotante/s}) / (2.6 \times 10^9 \text{ ciclos/s}) = 8$$

Por tanto, el número máximo de operaciones en coma flotante que puede terminar el núcleo por ciclo es 8.

Problema 6. Un núcleo de procesamiento (*core*) sin segmentación de cauce tiene una arquitectura de tipo LOAD/STORE en la que las operaciones sólo utilizan como operandos registros de la microarquitectura. El núcleo se encuentra integrado en un SoC (*System on Chip*) que ejecuta aplicaciones en las que, en promedio, el 30% de las instrucciones son operaciones con la ALU (4 CPI), el 25% LOADs (4 CPI), el 15% STOREs (3 CPI) y el 20% BRANCHs (4 CPI). Se ha diseñado un nuevo compilador que permite reducir un 10% el número de instrucciones de operación con la ALU y otro 10% el de instrucciones LOAD. Si el núcleo funciona a una frecuencia de reloj de 500 MHz, indique: (a) ¿Cuál es el número de MIPS que se tienen con el compilador antiguo y con el nuevo?; (b) ¿Cuál es la ganancia de velocidad que se consigue al utilizar el nuevo compilador?; (c) ¿Qué pasaría si el nuevo compilador solo fuese capaz de reducir un 20% las instrucciones STORE, dejando igual todas las demás?.

Solución

Para resolver el apartado (a) se utilizará la expresión de los MIPS (NI es el número de instrucciones y F la frecuencia de reloj):

$$\begin{aligned} MIPS &= \frac{NI(\text{instr})}{T_{CPU}(\text{segundos}) \times 10^6} = \\ &= \frac{NI(\text{instr})}{\frac{NI(\text{instr}) \times CPI(\text{ciclos/instr}) \times T_{ciclo}(\text{segundos/ciclo}) \times 10^6}{F(\text{ciclos/segundo})}} \\ &= \frac{NI(\text{instr})}{CPI(\text{ciclos/instr}) \times 10^6} \end{aligned}$$

Con el primer compilador tenemos las condiciones que se indican en la Tabla 6:

Tabla 6. Situación de partida

Instrucción	Número	CPI
Operaciones ALU r-r	$0.40 \times NI(1)$	4
LOAD	$0.25 \times NI(1)$	4
STORE	$0.15 \times NI(1)$	3
BRANCH	$0.20 \times NI(1)$	4
Total	$NI(1)$	

En este caso

$$CPI(1) = (0.40 + 0.25 + 0.20) \times 4 + 0.15 \times 3 = 3.85$$

y por tanto

$$MIPS(1) = \frac{F(\text{ciclos/segundo})}{CPI(1)(\text{ciclos/instr}) \times 10^6} = \frac{500 \times 10^6 (\text{ciclos/segundo})}{3.85 (\text{ciclos/instr}) \times 10^6} = 129.87$$

Con el nuevo compilador tendremos:

Tabla 7. Situación con el nuevo compilador

Instrucción	Número	CPI
Operaciones ALU r-r	0.9×0.40×NI(1)	4
LOAD	0.9×0.25×NI(1)	4
STORE	0.15×NI(1)	3
BRANCH	0.20×NI(1)	4
Total	0.935×NI(1)	

En este caso tenemos que

$$CPI(2) = \frac{(0.36 + 0.225 + 0.20) \times 4 + 0.15 \times 3}{0.935} = 3.839$$

y por tanto

$$MIPS(2) = \frac{F(\text{ciclos/segundo})}{CPI(1)(\text{ciclos/instr}) \times 10^6} = \frac{500 \times 10^6 (\text{ciclos/segundo})}{3.839 (\text{ciclos/instr}) \times 10^6} = 130.24$$

Es claro que con las optimizaciones que aplica el nuevo compilador se consigue un número mayor de MIPS (130.24 frente a 129.87)

(a) Para resolver el segundo apartado se aplican las expresiones del tiempo de CPU en los dos casos y realizamos el cociente entre el tiempo de CPU de la opción de partida y el de la opción mejorada debido al uso del nuevo compilador:

$$S = \frac{T_{CPU}(1)}{T_{CPU}(2)} = \frac{NI(1) \times CPI(1) \times T_{CICLO}(1)}{NI(2) \times CPI(2) \times T_{CICLO}(2)}$$

Teniendo en cuenta que $T_{CICLO}(1) = T_{CICLO}(2)$, y sustituyendo, obtenemos que:

$$S = \frac{T_{CPU}(1)}{T_{CPU}(2)} = \frac{NI(1) \times CPI(1)}{NI(2) \times CPI(2)} = \frac{NI(1) \times 3.85}{0.935 \times NI(1) \times 3.839} = 1.072$$

Es decir, se ha conseguido un 7.25% de mejora en las prestaciones.

(b) La situación en este caso vendría dada por la Tabla 8

Tabla 8. Situación de partida

Instrucción	Número	CPI
Operaciones ALU r-r	$0.40 \times NI(1)$	4
LOAD	$0.25 \times NI(1)$	4
STORE	$0.8 \times 0.15 \times NI(1)$	3
BRANCH	$0.20 \times NI(1)$	4
Total	$0.97 \times NI(1)$	

En este caso tenemos que

$$CPI(2) = \frac{(0.40 + 0.25 + 0.20) \times 4 + 0.8 \times 0.15 \times 3}{0.97} = 3.87$$

y por tanto

$$MIPS(2) = \frac{F(\text{ciclos/segundo})}{CPI(1)(\text{ciclos/instr}) \times 10^6} = \frac{500 \times 10^6 (\text{ciclos/segundo})}{3.87 (\text{ciclos/instr}) \times 10^6} = 129.19$$

Ahora, con las optimizaciones que aplica el nuevo compilador, se consigue un número menor de MIPS (129.19 frente a 129.87), a pesar de que los nuevos programas tardarían menos dado que sigue existiendo ganancia de velocidad:

$$S = \frac{T_{CPU}(1)}{T_{CPU}(2)} = \frac{NI(1) \times CPI(1)}{NI(2) \times CPI(2)} = \frac{NI(1) \times 3.85}{0.97 \times NI(1) \times 3.87} = 1.0256$$

Es decir, se ha conseguido un 2.56% de mejora en las prestaciones (algo menos que en el caso anterior). Eso significa que el tiempo de ejecución de los programas con el nuevo compilador es menor que en el caso de partida. Es lógico porque lo que consigue el compilador, tanto en este caso como en el caso anterior, es reducir el número de instrucciones que se ejecutan (y no hay cambios en el repertorio de instrucciones ni en la frecuencia de reloj, solo disminuye el número de instrucciones).

Sin embargo, los MIPS que se tienen ahora han disminuido respecto al caso de partida: los MIPS pueden reducirse a pesar de que las prestaciones aumenten. Esto se debe a que los MIPS miden lo rápido que se ejecutan las instrucciones, más que el tiempo de ejecución. Se pueden ejecutar instrucciones muy rápidamente, pero si hay que ejecutar más instrucciones, el tiempo puede ser mayor en el caso de un valor de MIPS mayor. Así, en la primera mejora, se reduce el número de instrucciones más lentas (4 ciclos) y el porcentaje de instrucciones más rápidas que se ejecutan en los nuevos programas es mayor. Por eso los MIPS también aumentan con el aumento de las prestaciones. En el segundo caso, correspondiente a la pregunta (c), se eliminan instrucciones más rápidas (con 3 ciclos), aumenta el porcentaje de instrucciones lentas en los códigos, por lo que los MIPS se reducen a pesar de que sigue existiendo mejora de prestaciones.

Problema 7. Un procesador superescalar de 64 bits a 1 GHz capaz de finalizar tres instrucciones por ciclo ejecuta el programa que se indica a continuación:

(1) start: ld	f0, a	// f0=a
(2)	add r8, r0, r2	// r8=r2 (r0=0)
(3)	addi r6, r8, #2048	// r6=r8+2048
(4)	add r12, r0, r4	// r12=r4 (r0=0)
(5) loop: ld	f2, 0(r8)	// f2=m(r8)
(6)	multd f2, f0, f2	// f2=f0*f2
(7)	ld f4, 0(r12)	// f4=m(r12)
(8)	addd f4, f2, f4	// f4=f2+f4
(9)	sd 0(r12), f4	// m(r12)=f4
(10)	addi r8, r8, #8	// r8=r8+8
(11)	addi r12, r12, #8	// r12=r12+8
(12)	sub r16, r6, r8	// r16=r6-r8
(13)	bnez r16, loop	// Si r16!=0 salta

En el programa, a es un número real, r0 es un registro que siempre está a cero, r2 contiene la dirección a partir de la cual empieza un array, X, de números reales de 64 bits, y r4 contiene la dirección a partir de la que empieza otro array también de números reales de 64 bits. ¿Qué hace el programa? ¿Cuál es el tiempo mínimo que tardaría en ejecutarse?

Solución

El programa se encarga de calcular el producto de un escalar (el número a) por un array/vector $X[i]$ ($i=1, \dots, N$) y sumarlo a un array/vector $Y[i]$ ($i=1, \dots, N$). Para ello:

- La instrucción (1) carga el escalar a en el registro f0; en la instrucción (2) el registro r8 se carga con el valor de r2 y por lo tanto apunta al comienzo del array X. Después, la instrucción (3) carga en r6 la suma de r8 y 2048 y la instrucción (4) carga en r12 la dirección de comienzo del array Y, que estaba en r4.
- Despues empieza el bucle, en la línea (5) donde está la etiqueta loop. Las instrucciones (5) a (9), respectivamente, realizan la carga de un elemento de X a f2, lo multiplican por f0 que es donde está el número a, cargan en f4 un elemento de Y y le suman el resultado de la multiplicación que está en f2, para posteriormente almacenar el resultado en la posición de memoria de la que se leyó el correspondiente elemento de Y. Por lo tanto las instrucciones (5) a (9) realizan el cálculo $Y[i] = a * X[i] + Y[i]$ donde $X[i]$ e $Y[i]$ están almacenados en memoria.
- A continuación, se incrementan los registros r8 y r12 para que apunten, respectivamente, al siguiente componente del array X (instrucción (10)) y al siguiente

componente del array Y (instrucción (11)). Dado que los números son de 64 bits habrá que restar 8 de $r8$ y $r12$ ya que 64 bits son 8 bytes y las direcciones de memoria apuntan a bytes.

- Para terminar el bucle, la instrucción (12) resta $r6$ y $r8$ y pone el resultado en $r16$. Así, la instrucción (13) comprueba si $r16$ es igual a cero. En el caso de que $r16$ sea igual a cero no se produce el salto a la etiqueta loop y acaba el programa. Eso quiere decir que el número de iteraciones que se hacen es igual al número que añadíamos a $r8$ en la instrucción (3), es decir 2048, dividido por 8 dado que en cada iteración $r8$ se incrementa precisamente en 8. Por tanto el número de iteraciones (es decir el valor de N) es:

$$N = \frac{2^{11}}{2^3} = 2^8 = 256$$

El tiempo mínimo de ejecución de este programa se puede estimar teniendo en cuenta que el número de instrucciones que se ejecutan es $NI = 4 + 256 \times 9 = 2,308$

Por otro lado, el número máximo de instrucciones por ciclo que puede terminar el procesador es 2 ($IPC=2$) y por lo tanto el número mínimo de ciclos por instrucción sería $CPI = 1/IPC = 1/2 = 0.5$

Teniendo en cuenta que la frecuencia de reloj es 1 GHz = 10^9 ciclos/segundo, el tiempo de ciclo sería $T_{ciclo} = 10^{-9}$ segundos/ciclo, lo que es lo mismo, 1 nanosegundo.

Teniendo en cuenta todo:

$$T_{CPU}(\text{min}) = NI \times CPI \times T_{ciclo} = 2,308 \text{ (inst)} \times 0.5 \left(\frac{\text{ciclos}}{\text{inst}} \right) \times 1 \left(\frac{\text{ns}}{\text{ciclo}} \right) = 1,154 \text{ ns}$$

Problema 8. La Figura 2 muestra un diagrama de ciclos correspondiente a la ejecución de una secuencia de instrucciones en un procesador segmentado. Cada una de las instrucciones pasa por las etapas de captación (IF), decodificación (ID), ejecución (intEX o fmulEX según la instrucción), acceso a memoria (MEM) y almacenamiento de resultado en registros (WB). También se indican los ciclos de espera de resultados previos (R-Stall) o de liberación de etapas (Stall).

Si el procesador utiliza un reloj a 500 MHz, indique cuál es el número medio de ciclos por instrucción para la secuencia de instrucciones ejecutada.

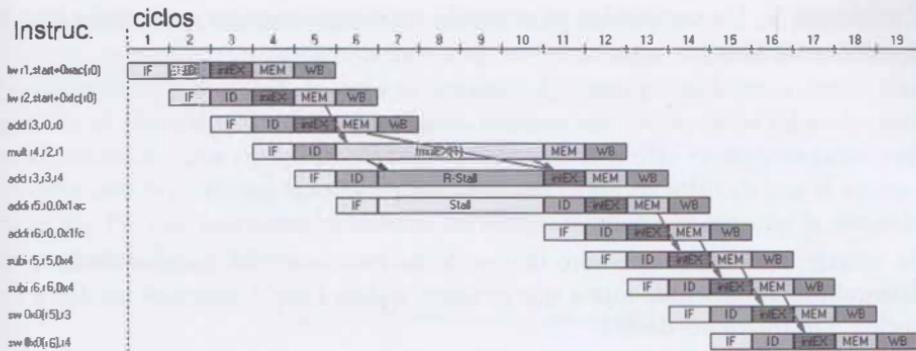


Figura 2. Ejecución de una secuencia de instrucciones en un procesador segmentado

Solución

La secuencia de instrucciones tarda 19 ciclos en ejecutarse en el procesador segmentado. Teniendo en cuenta que la frecuencia de reloj es de 500 MHz, el tiempo de ciclo es igual a:

$$T_{ciclo} = \frac{1}{Frecuencia} = \frac{1}{500 \times 10^6 \left(\frac{\text{ciclos}}{\text{seg}} \right)} = 2 \times 10^{-9} \left(\frac{\text{seg}}{\text{ciclo}} \right) = 2 \text{ nseg}$$

El tiempo que tarda en ejecutarse la secuencia de instrucciones es $T_{CPU} = 19 \times 2 = 38$ nseg, y, como el número de instrucciones que se ejecuta es $NI = 11$, dado que

$$T_{CPU} = NI \times CPI \times T_{ciclo}$$

se puede obtener el valor de CPI que se pide:

$$CPI = \frac{T_{CPU}}{NI \times T_{ciclo}} = \frac{38 \text{ nseg}}{11 \text{ inst} \times 2 \left(\frac{\text{nseg}}{\text{ciclo}} \right)} = 1.73 \left(\frac{\text{ciclos}}{\text{inst}} \right)$$

Es decir, las instrucciones tardan un promedio de 1.73 ciclos en ejecutarse. Hay que tener en cuenta que en un procesador segmentado, en el caso ideal, se terminaría de ejecutar una instrucción cada ciclo, y por tanto tendríamos que cada instrucción tardaría un ciclo. Aquí se tiene que se ejecuta menos de una instrucción por ciclo ($1/1.73=0.58$). Este funcionamiento por debajo del caso ideal se puede comprobar en la Figura 2: en los ciclos 1 a 4 no se terminan instrucciones (debido al tiempo de latencia de inicio hasta que termina de ejecutarse la primera instrucción), y en los ciclos 8 a 11 debido a la dependencia entre las instrucciones de multiplicación (que necesita más de un ciclo para ejecutarse) y la de suma que le sigue.

Por otra parte, hay que tener en cuenta que hasta el ciclo (5) no termina ninguna instrucción. Es el denominado tiempo de latencia de inicio (necesario para llenar el cauce).

Problema 9. Un compilador ha generado un código máquina optimizado para el siguiente programa:

```
for (i=0; i<N; i++)
    if ((i%2) == 0)
        par=par+c*x[i];
    else
        impar=impar-c*x[i];
```

sin utilizar instrucciones de salto dentro de las iteraciones del bucle(es decir, se ha desenrollado el bucle), de forma que el nuevo código tiene 7 instrucciones fuera del bucle y 9 instrucciones dentro.

El computador donde se ejecuta dispone de un procesador superescalar de 32 bits a 2 GHz capaz de terminar dos instrucciones por ciclo (que pueden ser instrucciones que ejecutan operaciones de coma flotante), con dos cachés internas (una para datos y otra para instrucciones) de 512 KBytes cada una, mapeo directo, política de actualización de postescritura, líneas de 32 bytes, y latencia de un ciclo de reloj. La memoria principal del computador tiene una latencia de 32 nanosegundos y ciclos burst 8-1-1-1 para el acceso a través de un bus de memoria de 250 Mhz. (a) ¿Cuántos GFLOPS pico tiene el procesador?. (b) ¿Cuál es el tiempo mínimo que tarda en ejecutarse el programa para $N=2^{11}$. (c) ¿Cuántos GFLOPS alcanza en este programa?.

(NOTA: Considere la situación más favorable para los datos en memoria y caché, que sea compatible con las características de la máquina y el programa; las variables par, impar, c, y $x[]$ son números de 32 bits en coma flotante).

Solución

El programa con desenrollado sería:

```
for (i=0; i<N; i=i+2) {
    par=par+c*x[i];
    impar=impar-c*x[i+1];
}
```

por lo que el número de iteraciones es $\text{Iteraciones} = N / 2 = 2^{11}/2=2^{10}$ y, como se nos indica que el programa en ensamblador tiene 7 instrucciones fuera del bucle y 9 dentro, el número de instrucciones ejecutadas:

$$NI = 9 \times 2^{10} + 7 = 9,223$$

Se pasa ahora a responder a cada una de las preguntas del problema.

(a) La velocidad en GFLOPS pico del procesador, dado que nos dicen en el enunciado que el procesador puede terminar *2 operaciones de coma flotante por ciclo como máximo*, es:

$$GFLOPS_{pico} = 2 \left(\frac{\text{op. coma. flotante}}{\text{ciclo}} \right) \times 2 \times 10^9 \left(\frac{\text{ciclos}}{\text{s}} \right) \times \left(\frac{1}{10^9} \right) = 4$$

(b) Podemos hacer una estimación del tiempo mínimo que tardaría en procesarse el programa teniendo en cuenta no solo el trabajo del procesador. Así, teniendo en cuenta la información que se nos da sobre la jerarquía de memoria podríamos contabilizar también el efecto de los accesos a memoria, evaluando el efecto de las faltas de caché en el procesador, que tendría que detenerse para esperar el dato en algunos casos, pero en otros podría continuar ejecutando instrucciones de forma solapada con el acceso a memoria. Por ello, una buena estimación del tiempo mínimo que consume la ejecución del programa, $T_{ejecucion}$, sería:

$$T_{ejecucion} = \max(T_{mem}, T_{procesador})$$

donde T_{mem} es una estimación del tiempo de acceso a los datos en memoria y $T_{procesador}$ el tiempo mínimo que tardaría el procesador en ejecutar el programa. La Figura 3 ilustra este razonamiento.

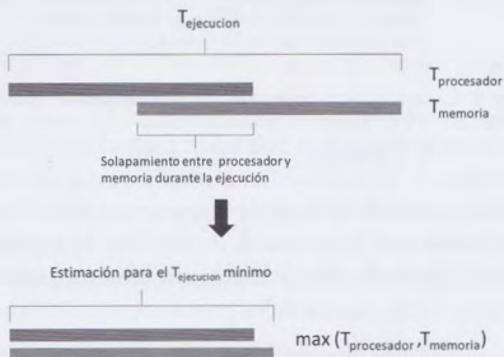


Figura 3. Estimación del tiempo mínimo de ejecución del programa

Ahora procedemos a calcular cada uno de los tiempos:

$$T_{procesador} = NI \times CPI \times T_{ciclo} = 9,223 \text{ (inst.)} \times (1/2) \left(\frac{\text{ciclos}}{\text{inst.}} \right) \times 0.5 \left(\frac{\text{nseg}}{\text{ciclo}} \right) \\ = 2,305.75 \text{ nseg}$$

donde se ha tenido en cuenta que el procesador puede terminar dos instrucciones por ciclo como máximo, y que la frecuencia de reloj es de 2 GHz, lo que significa un tiempo de ciclo de 0.5 nanosegundos.

En cuanto a la estimación del tiempo de acceso a los datos de memoria, la Figura 4 resume las características principales de la jerarquía de cachés y memoria principal del computador.

Los accesos a memoria para cargar el código se desprecian porque el código ocupa sólo dos líneas de caché: $7 \times 4 \text{ bytes/inst.} + 9 \times 4 \text{ bytes/inst.} = 28 + 36 = 64 \text{ bytes}$ y si se divide por 32 bytes/línea obtenemos que $64 \text{ Bytes} / 32 \text{ (Bytes/línea)} = 2 \text{ líneas}$ para el código.

Se producirían, por tanto, sólo dos faltas de caché, una por cada una de las líneas que se cargarían en la caché de instrucciones y se mantendrían ahí durante la ejecución del programa.

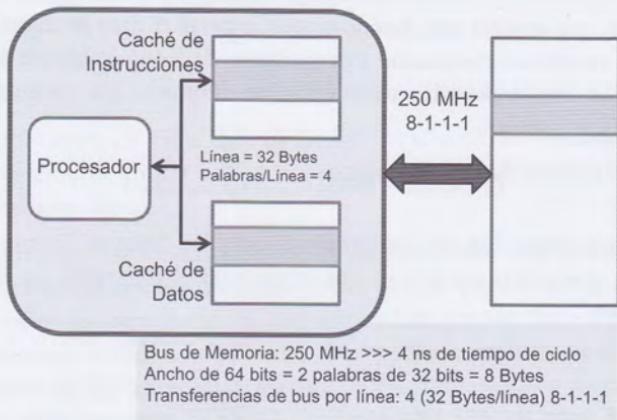


Figura 4. Características relevantes de la jerarquía de memoria

En el acceso a los datos, se producirán más fallos. De hecho, se producirán tantos fallos como líneas ocupe el array $x[]$ almacenado en memoria, ya que una vez que se cargue una línea, se accede a todos los datos de esa línea de forma consecutiva. También supondremos que puesto que los accesos al dato c (que estaría en una línea aparte) solo darían lugar a un fallo de acceso y se puede despreciar dicho fallo, frente al número de fallos para acceder a $x[]$ (lo mismo que hemos hecho con los fallos para el acceso a la caché de instrucciones).

Por tanto, el número de fallos que se producen (como hemos dicho, igual al número de líneas que ocupa el array $x[]$) es:

$$\text{Fallos} = \frac{4 \left(\frac{\text{Bytes}}{\text{dato}} \right) \times 2^{11} \text{datos}}{2^5 \left(\frac{\text{Bytes}}{\text{línea}} \right)} = 2^8 \text{líneas}$$

Teniendo en cuenta que se accede a 2^{11} datos, la tasa de fallos se puede calcular como:

$$\text{Tasa de Fallos} (1 - a) = \frac{\text{Fallos}}{\text{Accesos}} = \frac{2^8}{2^{11}} = \frac{1}{8} = 0.125$$

y la tasa de aciertos es $a = 7/8 = 0.875$.

A partir de los datos de la jerarquía de memoria que se nos da en el enunciado y se resumen en la Figura 4 se puede calcular el tiempo medio de acceso a un dato:

$$t_{memoria} = a \times t_{cache} + (1 - a) \times t_{memoria.principal} = 0.875 \times 0.5 + 0.125 \times (0.5 + 44) = 6.0 \text{ nseg.}$$

donde se ha considerado que el tiempo de acceso a caché es de un ciclo del procesador (es decir 0.5 nseg.) y que cuando hay que acceder a memoria se emplea un ciclo burst para traer una línea de caché. Dado que un ciclo burst tiene $8+1+1+1=11$ ciclos de bus, y que el bus de memoria es de 250 MHz y, por tanto tiene un ciclo de 4 ns, el tiempo de acceso a memoria principal sería $11 \times 4 = 44$ nanosegundos. A este tiempo hemos añadido los 0.5 nanosegundos que costaría llevar el dato al procesador desde la caché (estos 0.5 nseg. se podrían no tener en cuenta si se supone que el dato se pasa directamente al procesador cuando se trae desde memoria principal).

Considerando que se tienen 2^{11} accesos a memoria, tenemos que:

$$T_{memoria} = 2^{11} \times t_{memoria} = 2^{11} \times 6.0 = 12,288 \text{ nseg.}$$

de forma que

$$T_{ejecucion} = \max(T_{mem}, T_{procesador}) = \max(12,288 \text{ nseg.}, 2305.75 \text{ nseg.}) = 12,288 \text{ nseg.}$$

(c) En cuanto a los GFLOPS que se obtienen al ejecutar el programa:

$$GFLOPS = \frac{2 \times (\text{op. coma. flotante/elemento del array}) \times 2^{11} \text{ elementos}}{12288 \times 10^{-9} (\text{segundos}) \times 10^9} = 0.33$$

Como se puede ver, el número de GFLOPS obtenido es bastante menor que los 4 GFLOPS pico que el procesador alcanzaría si funcionase a pleno rendimiento (terminando de ejecutar dos instrucciones cada ciclo de reloj).

Problema 10. Un computador tiene un procesador superescalar de 32 bits a 2 GHz capaz de terminar dos instrucciones de coma flotante por ciclo, con dos cachés L1 internas (una para datos y otra para instrucciones) de 128 KBytes cada una, mapeo directo, política de actualización de postescritura, líneas de 32 bytes, y latencia de un ciclo de reloj. La memoria principal del computador tiene una latencia de 30 ns. y ciclos burst 6-1-1-1 a través de un bus de memoria de 64 bits a 200 Mhz.

Justifique cuál de los dos códigos alternativos siguientes (CÓDIGO 1 y CÓDIGO 2) permite calcular de forma más eficiente (tiempo mínimo estimado más pequeño), para $N=2^{18}$, el valor de

$$\sum_{i=0}^{N-1} (x^2(i) - X^2) \text{ donde } X = \frac{1}{N} \left(\sum_{i=0}^{N-1} x(i) \right)$$

CÓDIGO 1:

```
S=0; R=0;
for (i=0; i<N; i++) S=S+x[i];
for (i=0; i<N; i++) R=R+x[i]*x[i];
R=R-(S*S/N2); // R es el resultado
```

CÓDIGO 2:

```

S=0; R=0;
for (i=0;i<N;i++) {
    S=S+x[i];
    R=R+x[i]*x[i];
}
R=R- (S*S/N2); // R es el resultado

```

¿Qué conclusiones puede extraer del resultado?

NOTAS:

- El CÓDIGO 1 tiene 10 instrucciones fuera de los bucles (cuatro de coma flotante, una de almacenamiento y cinco de inicialización) en el primero de los bucles hay 6 instrucciones por iteración (cuatro instrucciones para controlar el bucle, una instrucción de carga del dato y una instrucción de coma flotante) y en el segundo hay 7 instrucciones por iteración (cuatro instrucciones para controlar el bucle, una instrucción de carga del dato y dos instrucciones de coma flotante).
- El CÓDIGO 2 tiene 10 instrucciones fuera del bucle (cuatro de coma flotante, una de almacenamiento y cinco de inicialización) y dentro del bucle hay 8 instrucciones por iteración (cuatro instrucciones para controlar el bucle, una instrucción de carga del dato y tres instrucciones de coma flotante).
- Tenga en cuenta que al iniciar la ejecución de los programas las cachés están vacías, y considere la situación más favorable para la ubicación de los datos en memoria y caché (compatible con las características de la máquina y el programa). En particular, el array $x[]$ está almacenado en memoria a partir de una dirección que corresponde al inicio de un bloque o línea de caché. Los datos del array $x[]$ son datos de 32 bits en coma flotante.

Solución

Primero haremos la estimación para el CÓDIGO 1, luego para el CÓDIGO 2 y compararemos los resultados.

Estimación para el CÓDIGO1: Dado que el tiempo debido a los accesos a memoria puede solaparse en mayor o menor grado con el tiempo de ejecución del código por parte del procesador, una buena estimación del tiempo mínimo de procesamiento del código es la que corresponde a un solapamiento total de ambos tiempos. En ese caso, el mayor de ellos es el que determina el tiempo mínimo de procesamiento.

$$T_1 \geq \max (T_{\text{memoria}}, T_{\text{procesador}})$$

Por lo tanto, el problema se podría resolver estimando el tiempo del procesador, $T_{\text{procesador}}$, y el del acceso a memoria para los datos que se necesitan, T_{memoria} .

Tenemos que:

$$NI = 10 + 6 \times N + 7 \times N = 10 + 6 \times 2^{18} + 7 \times 2^{18} = 3,407,882$$

$CPI = 0.5$ (se supone que el procesador puede terminar 2 instrucciones por ciclo)

$T_{ciclo} = 0.5 \text{ ns}$ (la frecuencia de reloj es de 2 GHz)

Por tanto:

$$T_{procesador} = NI \times CPI \times T_{ciclo} = 3,407,882 \times 0.5 \times 0.5 = 851,970.5 \text{ ns} = 0.85 \text{ ms}$$

Para estimar el tiempo de memoria se tiene en cuenta que:

Tamaño de caché = 128 KBytes = 2^{17} Bytes

Tamaño de bloque o línea de caché = 32 Bytes = 2^5 Bytes

Tamaño del Array = 2^{18} elementos $\times 2^2$ Bytes/elemento = 2^{20} Bytes

Por lo tanto, el array es mayor que la caché ($2^{20}/2^{17}=8$ veces mayor) y al leerlo se irán escribiendo nuevos bloques sobre los que se habían leído previamente. Si se vuelve a leer nuevamente el array se generarán las mismas faltas que la primera vez que se leyó.

El tiempo necesario para acceder al array (es decir para traerlo a memoria caché desde memoria principal) es igual al número de bloques (o líneas) del array multiplicado por el tiempo que se tarda en traer un bloque de memoria principal a memoria caché:

$$\begin{aligned} \text{Tiempo de lectura de bloque} &= 9 \text{ ciclos del bus de memoria} \times t_{ciclo_bus_memoria} = 9 \times 5 \text{ ns} \\ &= 45 \text{ ns} \end{aligned}$$

Esto se debe a que cada bloque consta de 32 bytes, y se accede a él a través de un ciclo burst (6-1-1-1) mediante el que se realizan cuatro transferencias de 64 bits (8 bytes $\times 4 = 32$ bytes que tiene la línea). En el ciclo burst que nos indican (6-1-1-1), la primera transferencia tarda 6 ciclos y las tres restantes 1 ciclo (gracias a la localidad que se tiene por acceder a los elementos de un bloque, que están en posiciones consecutivas). Por lo tanto, la transferencia de una línea necesita $6+1+1+1=9$ ciclos y como el bus de memoria es de 200 MHz, el tiempo de ciclo es de 5 ns.

El número de bloques que tienen que leerse es igual al número de faltas de caché (cada vez que un acceso se realiza a un nuevo bloque que no se había traído se genera una falta). Como los bloques (que van a marcos de bloque o líneas de caché) tienen un tamaño de $32 = 2^5$ bytes, en el array hay:

$$(2^{20} \text{ Bytes/array}) / (2^5 \text{ Bytes/Bloque}) = 2^{15} \text{ Bloques/array}$$

Así, 2^{15} es el número de faltas (como se ha dicho, el número de bloques que hay que traerse de memoria principal a caché). Puesto que en el primer código hay que traerse el array dos veces (una en cada bucle), tendremos que el número de transferencias de bloques es de $2 \times 2^{15} = 2^{16}$. Por tanto:

$$\begin{aligned} T_{memoria} &= 2^{16} \text{ lecturas de bloques} \times 45 \text{ ns} \text{ (tiempo de lectura de un bloque)} = 2,949,120 \text{ ns} \\ &= 2.95 \text{ ms} \end{aligned}$$

Por lo tanto, para el primer código:

$$T1 \geq \max(T_{memoria}, T_{procesador}) = \max(0.85 \text{ ms}, 2.95 \text{ ms}) = 2.95 \text{ ms}$$

También se podría haber estimado el tiempo de acceso a memoria considerando el tiempo medio de un acceso a memoria y multiplicando por el número de accesos, tal y como se hizo en el ejercicio 10). En este caso se tendría un tiempo algo mayor porque estaríamos contabilizando los accesos a memoria que no dan lugar a faltas y que ya estarían considerados dentro del tiempo del procesador. A continuación se ilustra la forma de hacer ese cálculo para este caso:

El número de faltas es igual 2^{15} bloques que tiene el array multiplicado por 2 dado que hay que acceder dos veces al mismo (una en cada bucle). Además, el número de accesos a datos es 2^{18} multiplicado por 2 bucles. Por lo tanto la tasa de fallos es:

$$(1-a) = 2^{16}/2^{19} = 1/8 = 0.125$$

y la tasa de aciertos:

$$a = 1 - 0.125 = 0.875$$

Así:

$$t_{memoria} = t_{caché} \times a + t_{MP} \times (1-a) = 0.5 \times 0.875 + (45 + 0.5) \times 0.125 = 6.125 \text{ ns}$$

Teniendo en cuenta el número de accesos totales (2^{19}):

$$T_{memoria} = 6.125 \text{ ns} \times 2^{19} = 3,211,264 \text{ ns} = 3.21 \text{ ms}$$

Como ya se ha indicado, en este caso, la estimación del tiempo de memoria es algo mayor porque aquí estamos contabilizando también el tiempo de acceso a los datos cuando están en caché, y esto se ha contabilizado ya en el tiempo de procesamiento de las instrucciones. Si lo estimamos de esta forma $T1=3.21 \text{ ms}$.

Estimación para el CÓDIGO2. Igual que antes, el tiempo mínimo de procesamiento para este segundo caso, $T2$, se puede estimar teniendo en cuenta que:

$$T2 \geq \max(T_{memoria}, T_{procesador})$$

Para estimar el tiempo del procesador, $T_{procesador}$, del segundo código tenemos que:

$$NI = 10 + 8 \times N = 10 + 8 \times 2^{18} = 2,097,162$$

$CPI = 0.5$ (se supone que el procesador puede terminar 2 instrucciones por ciclo)

$T_{ciclo} = 0.5 \text{ ns}$ (la frecuencia de reloj es de 2 GHz)

Por tanto:

$$T_{procesador} = NI \times CPI \times T_{ciclo} = 2,097,162 \times 0.5 \times 0.5 = 524,290.5 \text{ ns} = 0.52 \text{ ms}$$

Para obtener el tiempo de memoria se tiene en cuenta que, como en el caso anterior:

$$\text{Tamaño de caché} = 128 \text{ KBytes} = 2^{17} \text{ Bytes}$$

$$\text{Tamaño de Bloque/línea de caché} = 32 \text{ Bytes} = 2^5 \text{ Bytes}$$

$$\text{Tamaño del Array} = 2^{18} \text{ elementos} \times 2^2 \text{ Bytes/elemento} = 2^{20} \text{ Bytes}$$

Por lo tanto, el array es mayor que la caché ($2^{20}/2^{17}=8$ veces mayor), y al leerlo se irán escribiendo nuevos bloques sobre los que se habían leído previamente. No obstante, en este caso hay que tener en cuenta que solo hay que leer el array una vez.

Así, como se hizo para la primera opción:

$$\begin{aligned} \text{Tiempo de lectura de bloque} &= 9 \text{ ciclos del bus de memoria} \times t_{\text{ciclo_bus_memoria}} = 9 \times 5 \text{ ns} \\ &= 45 \text{ ns} \end{aligned}$$

El número de bloques que tienen que leerse es igual al número de faltas de caché (cada vez que un acceso se realiza a un nuevo bloque que no se había traído se genera una falta). Como los bloques (que van a marcos de bloque o líneas de caché) tienen un tamaño de $32 = 2^5$ bytes, en el array hay:

$$(2^{20} \text{ Bytes/array})/(2^5 \text{ Bytes/Bloque}) = 2^{15} \text{ Bloques/array}$$

Así, 2^{15} es el número de faltas o lo que es igual, el número de bloques que hay que traerse de memoria principal a caché. A diferencia de lo que ocurría en el primer código sólo hay que traerse el array una vez (hay una lectura de memoria en el único bucle). Por tanto:

$$\begin{aligned} T_{\text{memoria}} &= 2^{15} \text{ lecturas de bloques} \times 45 \text{ ns (tiempo de lectura de un bloque)} = 1,474,560 \text{ ns} \\ &= 1.47 \text{ ms} \end{aligned}$$

Así, para el segundo código:

$$T_2 \geq \max (T_{\text{memoria}}, T_{\text{procesador}}) = \max (0.52 \text{ ms}, 1.47 \text{ ms}) = 1.47 \text{ ms}$$

Igual que en la primera alternativa, también se podría haber estimado el tiempo de acceso a memoria considerando el tiempo medio de un acceso a memoria y multiplicando por el número de accesos, tal y como se muestra a continuación.

Ahora el número de faltas es igual 2^{15} bloques. Es la mitad que en la primera alternativa, pero también el número de accesos a datos es la mitad: 2^{18} (hay una instrucción de acceso a memoria en cada iteración del bucle. Por lo tanto la tasa de fallos es la misma que en el caso anterior:

$$(1-a) = 2^{15}/2^{18} = 1/8 = 0.125$$

y la tasa de aciertos también será igual:

$$a = 1 - 0.125 = 0.875$$

Así:

$$t_{memoria} = t_{caché} \times a + t_{MP} \times (1-a) = 0.5 \times 0.875 + (45 + 0.5) \times 0.125 = 6.125 \text{ ns}$$

y teniendo en cuenta el número de accesos totales (que ahora es 2^{18}):

$$T_{memoria} = 6.125 \text{ ns} \times 2^{18} = 1,605,632 \text{ ns} = 1.61 \text{ ms}$$

Como antes, la estimación del tiempo de memoria de esta forma da un resultado algo mayor pues también incluimos el tiempo de acceso a los datos cuando están en caché. Así, si consideremos la segunda estimación del tiempo de acceso a memoria, tendremos que $T2=1.61 \text{ ms}$ (un 8.7% mayor que con la primera forma de estimar el tiempo de memoria).

Conclusión. En primer lugar, para los dos códigos se puede ver que lo que determina el tiempo final (y constituye el cuello de botella) para resolver este problema, es el tiempo de acceso a la memoria. Hay que acceder a un array de un tamaño mayor que la caché, que ocasiona un número considerable de faltas (la tasa de aciertos de la caché es de 0.875, lo que puede considerarse relativamente bajo).

Por otro lado, el tiempo estimado para la primera opción es aproximadamente igual al doble del estimado para la segunda. Esto se debe a que en el primer caso se realizan dos accesos al array, y al tener este un tamaño varias veces el de la caché, posteriores accesos al array volverán a generar faltas porque los primeros bloques del array se han ido sustituyendo por los que se necesitan a continuación. Por tanto, en estas situaciones es mejor hacer todas las operaciones que se puedan con los datos de los bloques que se vayan trayendo de memoria principal a memoria caché. Es lo que se hace en el segundo código: incrementar la localidad de los accesos a memoria.

Problema 11. Ha aparecido en el mercado una nueva versión de un procesador en la que la única mejora con respecto a la versión anterior es una unidad de coma flotante mejorada que permite reducir el tiempo de las instrucciones de coma flotante a una cuarta parte del tiempo que consumían antes.

Suponga que en los programas que constituyen la carga de trabajo habitual del procesador las instrucciones de coma flotante consumen un promedio del 20% del tiempo del procesador antiguo: (a) ¿Cuál es la máxima ganancia de velocidad que puede esperarse en los programas si se sustituye el procesador de la versión antigua por el nuevo?. (b) ¿Cuál es la máxima ganancia de velocidad que podría conseguirse en los programas debido a nuevos aumentos en la velocidad de las operaciones en coma flotante?. (c) ¿Cuál debería ser el porcentaje de tiempo de cálculo con datos en coma flotante en los programas (en la versión antigua del procesador) para que la máxima ganancia a la que se refiere el apartado (b) anterior, sea 8?. (d) Con el porcentaje de operaciones en coma flotante obtenido en (c), ¿cuánto debería reducirse el tiempo de las operaciones en coma flotante con respecto a la situación inicial para que la ganancia máxima sea 4?.

Solución

Dado que el tiempo de las instrucciones de coma flotante se reduce a una cuarta parte, la mejora de velocidad en el recursos es $p=4$ (la inversa de la reducción del tiempo es $t/4$, siendo t el tiempo de las operaciones en coma flotante en la versión antigua del procesador).

(a) Como el porcentaje de instrucciones de coma flotante es el 20%, la fracción a la que se puede aplicar la mejora es:

$$(1-f) = 0.20, \text{ y por tanto, } f = 0.80$$

Sustituyendo estos valores en la expresión de la ley de Amdahl se tiene:

$$S \leq \frac{p}{1 + f \times (p - 1)} = \frac{4}{1 + 0.80 \times (4 - 1)} = \frac{4}{1 + 2.40} = 1.17$$

(b) El valor de la máxima ganancia se obtiene calculando el límite cuando la mejora del recurso (en este caso la ganancia, o mejora en velocidad, de la unidad en coma flotante) tiende a infinito:

$$S_{\max(f=cte)} = \lim_{p \rightarrow \infty} \frac{p}{1 + f \times (p - 1)} = \frac{1}{f} = \frac{1}{0.80} = 1.25$$

Así, si por ejemplo quisiésemos obtener una ganancia de 2 tendríamos que debería verificarse que

$$2 = \frac{p}{1 + 0.80 \times (p - 1)}$$

y, despejando tendríamos $2 + 1.6(p-1) = p$, es decir $0.6p = -0.4$. Sería imposible de conseguir porque p tendría que ser menor que cero, $p = -0.4/0.6$ (p es un cociente de tiempos y siempre es positivo).

(c) Para que la ganancia máxima (es decir cuando la mejora, p , tiende a infinito) sea igual a 8, la fracción de tiempo que no se reduciría con la mejora se obtendría a partir de:

$$S_{\max(f=cte)} = \frac{1}{f} = 8$$

Por lo que $f=1/8=0.125$, y la fracción de tiempo correspondiente a operaciones en coma flotante (que es donde podría aprovecharse la mejora) tendría que ser $(1-f)=1-0.125=0.875$. Es decir, al menos un 87.5% del tiempo de ejecución secuencial debería deberse a las operaciones en coma flotante.

(d) Si, al utilizar el valor de f anterior, es decir $f=0.125$, se deseara obtener una mejora de velocidad de 4 en los programas, la mejora de velocidad en las operaciones de coma flotante (es decir p) se puede obtener a partir de la ley de Amdahl:

$$S \leq 4 = \frac{1}{0.125 + \frac{0.875}{p}}$$

Por lo que $0.5 + (3.5/p) = 1$, y $p = 3.5/0.5 = 7$. Es decir, las operaciones en coma flotante deberían ser siete veces más rápidas.

Problema 12. En un programa que se ejecuta en un procesador no segmentado que funciona a 200 MHz, hay un 25% de instrucciones LOAD que necesitan 4 ciclos, un 15% de instrucciones STORE que necesitan 3 ciclos, un 40% de instrucciones con operaciones en coma flotante que necesitan 8 ciclos, y un 20% de instrucciones de salto que necesitan 4 ciclos. (a) Si en las instrucciones de coma flotante, el tiempo debido a la operación supone 4 ciclos (el resto de ciclos de la instrucción corresponden a la captación, decodificación y acceso a los datos, y almacenamiento de resultados) determine cuál es la ganancia que se puede obtener si se mejora el diseño de la unidad que realiza las operaciones en coma flotante y se reduce su tiempo a la mitad de ciclos. (b) ¿Cuál es la máxima ganancia de velocidad que se puede conseguir por mejoras en la unidad que ejecuta las operaciones en coma flotante?

Solución

(a) Las características del programa que describe el problema se muestra en la Tabla 9.

Tabla 9. Situación de partida

Instrucción	Número	CPI
Coma flotante	$0.40 \times NI_1$	8
LOAD	$0.25 \times NI_1$	4
STORE	$0.15 \times NI_1$	3
BRANCH	$0.20 \times NI_1$	4
Total	NI_1	

El tiempo de partida es

$$T_{CPU}(1) = CPI_1 \times NI_1 \times T_{CICLO}(1)$$

donde, $T_{CICLO}(1) = 1/(200 \times 10^6) = 5 \times 10^{-9} \text{ seg} = 5 \text{ nseg}$, y el valor de CPI_1 se obtiene sumando, para todos los tipos de instrucciones, los productos del número de ciclos que necesita cada instrucción y la frecuencia con que aparece dicha instrucción:

$$CPI_1 = \left(\frac{0.4 \times NI_1}{NI_1} \times 8 \right) + \left(\frac{(0.25 \times NI_1) + (0.20 \times NI_1)}{NI_1} \times 4 \right) + \left(\frac{0.15 \times NI_1}{NI_1} \times 3 \right) = 5.85$$

y, por tanto

$$T_{CPU}(1) = 5.85 \times NI_1 \times 5 = 29.25 \times NI_1 \text{ nseg}$$

A continuación se calcula el tiempo medio de CPU para la mejora, teniendo en cuenta que, dado que el tiempo dedicado al cálculo de la operación en coma flotante pasaría de 4 ciclos a 2 ciclos, el *CPI* de las instrucciones en coma flotante pasaría de 8 ciclos a 6 ciclos y, así:

$$CPI_2 = \left(\frac{0.4 \times NI_1}{NI_1} \times 6 \right) + \left(\frac{(0.25 \times NI_1) + (0.20 \times NI_1)}{NI_1} \times 4 \right) + \left(\frac{0.15 \times NI_1}{NI_1} \times 3 \right) = 5.05$$

Dado que ni el número de instrucciones ni la frecuencia de reloj cambian:

$$T_{CPU}(2) = CPI_2 \times NI_2 \times T_{CICLO}(2) = 5.05 \times NI_1 \times 5 = 25.25 \times NI_1 \text{ nseg}$$

De esta forma, la ganancia obtenida es:

$$S = \frac{T_{CPU}(1)}{T_{CPU}(2)} = \frac{29.25}{25.25} = 1.16$$

(b) La máxima ganancia de velocidad que se podría alcanzar al reducir el tiempo de las operaciones en coma flotante se produciría cuando el tiempo que se consumiera en estas operaciones fuese igual a cero (algo que no sería físicamente posible, pero se trata de obtener la máxima ganancia por este tipo de mejoras). En esta situación, las instrucciones de coma flotante necesitarían 4 ciclos en lugar de los 8 ciclos de partida, con lo que

$$CPI_{min} = \left(\frac{0.4 \times NI_1}{NI_1} \times 4 \right) + \left(\frac{(0.25 \times NI_1) + (0.20 \times NI_1)}{NI_1} \times 4 \right) + \left(\frac{0.15 \times NI_1}{NI_1} \times 3 \right) = 4.25$$

y el tiempo de CPU mínimo sería (teniendo en cuenta nuevamente que no cambian ni el número de instrucciones ni el tiempo de ciclo):

$$T_{CPU}(min) = CPI_{min} \times NI_1 \times T_{CICLO}(1) = 4.25 \times NI_1 \times 5 = 21.25 \times NI_1 \text{ nseg}$$

y la ganancia máxima que se obtendría es:

$$S_{max} = \frac{T_{CPU}(1)}{T_{CPU}(min)} = \frac{29.25}{21.25} = 1.38$$

Esta ganancia máxima también se podría obtener a partir de la ley de Amdahl:

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

donde p es la ganancia de velocidad del recurso que se ha mejorado (en este problema la unidad de operación en coma flotante), y f es la fracción del tiempo de ejecución del programa antes de la mejora donde no se puede aplicar esta mejora. Como nos piden la ganancia máxima que se conseguiría por mejoras en la unidad de coma flotante, debemos considerar que p tiende a infinito, de forma que

$$S_{max} = \lim_{p \rightarrow \infty} \frac{p}{1 + f \times (p - 1)} = \frac{1}{f}$$

y solo tendríamos que calcular el valor de f :

$$f = \frac{T_{CPU}(1)^{(no\ mejorable)}}{T_{CPU}(1)}$$

El tiempo de CPU no mejorable, $T_{CPU}^{(no\ mejorable)}$, sería el que consumen todas las etapas de las instrucciones de coma flotante menos las que corresponden a la ejecución de la propia operación en coma flotante (es decir cuatro ciclos) y el tiempo que consumen todas las demás. Así,

$$\begin{aligned} CPI_1^{(no\ mejorable)} &= \\ &= \left(\frac{0.4 \times NI_1}{NI_1} \times (8 - 4) \right) + \left(\frac{(0.25 \times NI_1) + (0.20 \times NI_1)}{NI_1} \times 4 \right) \\ &+ \left(\frac{0.15 \times NI_1}{NI_1} \times 3 \right) = 4.25 \end{aligned}$$

y el tiempo de CPU no mejorable (teniendo en cuenta que tampoco cambian ni el número de instrucciones ni el tiempo de ciclo):

$$\begin{aligned} T_{CPU}(1)^{(no\ mejorable)} &= CPI_1^{(no\ mejorable)} \times NI_1 \times T_{CICLO}(1) = 4.25 \times NI_1 \times 5 \\ &= 21.25 \times NI_1 \text{ nseg} \end{aligned}$$

por lo que obtenemos nuevamente una mejora máxima del 38%:

$$S_{max} = \frac{1}{f} = \frac{T_{CPU}(1)}{T_{CPU}(1)^{(no\ mejorable)}} = \frac{29.25}{21.25} = 1.38$$

Problema 13. En un programa que se ejecuta en un procesador no segmentado que funciona a 500 MHz, hay un 15% de instrucciones LOAD que necesitan 4 ciclos, un 5% de instrucciones STORE que necesitan 3 ciclos, un 60% de instrucciones con operaciones con la ALU que necesitan 6 ciclos, y un 20% de instrucciones de salto que necesitan 4 ciclos. Si en las instrucciones con la ALU, la operación de la ALU consume 3 ciclos, determine cuál es la ganancia que se puede obtener si se mejora el diseño de la ALU de forma que se reduce el tiempo de ejecución de las operaciones a un ciclo. ¿Cuántas instrucciones con la ALU más debería tener el programa para conseguir una ganancia igual a 1.4 con la mejora indicada? ¿Qué pasaría si se quisiera conseguir una ganancia de 1.5?

Solución

La situación de la que se parte se muestra en la Tabla 10:

Tabla 10. Situación de partida del problema 13

Instrucción	Número	CPI
Operaciones con ALU	$0.60 \times NI_1$	6
LOAD	$0.15 \times NI_1$	4
STORE	$0.05 \times NI_1$	3
BRANCH	$0.20 \times NI_1$	4
Total	NI_1	

El tiempo de partida es

$$T_{CPU}(1) = CPI_1 \times NI_1 \times T_{CICLO}(1)$$

donde, $T_{ciclo}(1) = 1/(500 \times 10^6) = 2 \times 10^{-9} \text{ seg} = 2 \text{ nseg}$, y el valor de CPI_1 se obtiene sumando, para todos los tipos de instrucciones, los productos del número de ciclos que necesita cada instrucción y la frecuencia con que aparece dicha instrucción:

$$CPI_1 = \left(\frac{0.6 \times NI_1}{NI_1} \times 6 \right) + \left(\frac{(0.15 \times NI_1) + (0.20 \times NI_1)}{NI_1} \times 4 \right) + \left(\frac{0.05 \times NI_1}{NI_1} \times 3 \right) = 5.85$$

y, por tanto

$$T_{CPU}(1) = 5.15 \times NI_1 \times 2 = 10.3 \times NI_1 \text{ nseg}$$

Con la mejora, el tiempo dedicado al cálculo de la operación con la ALU pasaría de 3 ciclos a 1 ciclos, y por tanto el CPI de las instrucciones con la ALU pasaría de 6 ciclos a 4 ciclos y, así:

$$CPI_2 = \left(\frac{0.6 \times NI_1}{NI_1} \times 4 \right) + \left(\frac{(0.15 \times NI_1) + (0.20 \times NI_1)}{NI_1} \times 4 \right) + \left(\frac{0.05 \times NI_1}{NI_1} \times 3 \right) = 3.95$$

Dado que ni el número de instrucciones ni la frecuencia de reloj cambian:

$$T_{CPU}(2) = CPI_2 \times NI_2 \times T_{CICLO}(2) = 3.95 \times NI_1 \times 2 = 7.9 \times NI_1 \text{ nseg}$$

De esta forma, la ganancia obtenida es:

$$S = \frac{T_{CPU}(1)}{T_{CPU}(2)} = \frac{10.3}{7.9} = 1.304$$

Para resolver la segunda cuestión del problema utilizaremos la ley de Amdahl, que establece que la ganancia de velocidad está acotada según se indica en la expresión:

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

Así, para poder alcanzar una mejora de 1.4, tiene que verificarse que, al menos:

$$\frac{p}{1 + f \times (p - 1)} = 1.4$$

En primer lugar hay que tener en cuenta que con la mejora que se ha aplicado, las instrucciones con la ALU tardan 4 ciclos en lugar de 6. Por lo tanto, $p=6/4=1.5$, y sustituyendo en la expresión anterior, se tendría:

$$\frac{1.5}{1 + f \times 0.5} = 1.4$$

y al despejar tendríamos que $f = 0.142$ (si el valor de f fuese menos que ese valor se tendría una cota superior para la ganancia de velocidad que sería mayor que 1.4).

A continuación determinaremos el número de instrucciones que tendríamos que añadir. Para ello, consideramos la Tabla 11, donde el número de instrucciones de operaciones con la ALU que tendría el nuevo programa con respecto al inicial crece en $X \times NI_1$ (es decir, X es el porcentaje de instrucciones que se introducen, con respecto al número de instrucciones del programa inicial).

Tabla 11. Situación en el nuevo programa (respecto al inicial)

Instrucción	Número	CPI
Operaciones con ALU	$0.60 \times NI_1 + X \times NI_1$	6
LOAD	$0.15 \times NI_1$	4
STORE	$0.05 \times NI_1$	3
BRANCH	$0.20 \times NI_1$	4
Total	$NI_1 + X \times NI_1$	

Con esta situación, el tiempo de partida sería

$$T_{CPU}(2) = CPI_2 \times NI_2 \times T_{CICLO}(2) = \\ = \left(\frac{(0.6 + X) \times 6 + 0.35 \times 4 + 0.05 \times 3}{1 + X} \right) \times ((1 + X) \times NI_1) \times T_{CICLO}(1)$$

De este tiempo $T_{CPU}(2)$, la parte a la que no se le puede aplicar la mejora es la parte que corresponde a las instrucciones LOAD, STORE, y BRANCH, es decir:

$$T_{CPU}^*(2) = \left(\frac{0.35 \times 4 + 0.05 \times 3}{1 + X} \right) \times ((1 + X) \times NI_1) \times T_{CICLO}(1)$$

y por tanto, f vendrá dada por:

$$f = \frac{0.35 \times 4 + 0.05 \times 3}{(0.6 + X) \times 6 + 0.35 \times 4 + 0.05 \times 3} = \frac{1.55}{5.15 + 6 \times X} \leq 0.142$$

Por lo que, despejando se tiene que $0.819 \leq 0.852 \times X$, con lo que el número de instrucciones que hay que añadir es $X \times NI_1$, con $X \geq 0.961$. Es decir, hay que añadir casi tantas instrucciones de operación con la ALU como instrucciones teníamos en el programa inicial.

No se podría conseguir una ganancia de 1.5 dado que la mejora de velocidad de las instrucciones con la ALU es precisamente igual a 1.5. Por tanto, en la expresión de la ley de Amdahl tendríamos:

$$1.5 \leq \frac{1.5}{1 + f \times 0.5}$$

Sólo si $f=0$ se consigue la igualdad. Únicamente si todas las instrucciones son instrucciones de operación con la ALU (o tuviéramos un programa de tamaño infinito con un número infinito de operaciones con la ALU) para que f fuese igual a 0 podríamos conseguir esa ganancia de 1.5.

Problema 14. Suponga el siguiente bucle en el que los arrays $a[]$ y $b[]$, y la variable S son números reales:

```
for (i=0; i<N; i++) S=S+a[i]*b[i];
```

(a) Si debe ejecutarse en 0.50 segundos para $N=10^9$. ¿Cuántos GFLOPS debe proporcionar el programa como mínimo, suponiendo que el producto y la suma en coma flotante tienen un coste similar?. (b) Si el procesador funciona a 2 GHz ¿Cuántas operaciones en coma flotante tiene que terminar por ciclo?. (c) Suponiendo que el programa en ensamblador tiene $6 \times N + 4$ instrucciones, ¿Cuántas instrucciones por ciclo debe terminar el procesador si en el 20% de los 0.50 segundos que tarda el programa no se termina ninguna instrucción debido a accesos a memoria, por fallos de caché?. (d) Utilice la ley de Amdahl para estimar la máxima ganancia de velocidad que se podría conseguir en la ejecución de ese programa incrementando el número de instrucciones por ciclo que puede terminar el procesador.

Solución

El bucle calcula el producto escalar de dos vectores $a[]$ y $b[]$, que se carga en la variable S . El número de operaciones en coma flotante que hay que realizar es $2 \times N$ (una suma y un producto en coma flotante por iteración, que se nos dice que tienen el mismo coste).

(a) Dado que conocemos el número de operaciones en coma flotante que hay que ejecutar y el tiempo que se tarda en hacerlo, se pueden calcular los GFLOPS:

$$GFLOPS = \frac{\text{Operaciones en Coma Flotante}}{T \times 10^9} = \frac{2 \times 10^9}{0.5 \text{seg} \times 10^9} = 4$$

(b) Como el procesador funciona a 2 GHz, el número de ciclos que se completan en 0.5 segundos es igual a:

$$\text{Ciclos} = 2 \times 10^9 (\text{ciclos/seg}) \times 0.5 (\text{seg}) = 1 \times 10^9$$



Puesto que debe ejecutar 2×10^9 operaciones en coma flotante, tenemos que el número de operaciones por ciclo que el procesador debe ser capaz de terminar:

$$\frac{2 \times 10^9 (\text{op. coma. flotante})}{1 \times 10^9 (\text{ciclos})} = 2 (\text{op. coma. flotante/ciclo})$$

(c) Dado que en el 20% del tiempo de ejecución no se completan instrucciones, eso significa que las $6 \times N + 4$ instrucciones se han completado en $0.8 \times 0.5 = 0.4$ segundos. En estos 0.4 segundos, el número de ciclos que se completa es

$$\text{Ciclos} = 2 \times 10^9 (\text{ciclos/seg}) \times 0.4 (\text{seg}) = 8 \times 10^8$$

Dado que tenemos $6 \times 10^9 + 4$ instrucciones, el número de instrucciones se obtendría a partir del cociente:

$$\frac{6 \times 10^9 + 4 (\text{instrucciones})}{8 \times 10^8 (\text{ciclos})} \approx 7.5 (\text{instrucciones/ciclo})$$

Es decir, el procesador debería ser capaz de terminar 7.5 instrucciones por ciclo durante el tiempo en el que terminan instrucciones. Es decir, su microarquitectura debe ser capaz de terminar 8 instrucciones por ciclo, o más. Se trata de un número muy elevado para las capacidades usuales en los procesadores con paralelismo de instrucciones.

(d) Se puede suponer que el incremento en el número de instrucciones por ciclo no reduciría el tiempo en el que el procesador no termina instrucciones por los accesos a memoria principal que se producen (las faltas de caché serían las mismas y no se producen cambios en la jerarquía de memoria). Por lo tanto el valor de f en la ley de Amdahl sería el 20% del tiempo de ejecución inicial:

$$f = \frac{0.2 \times 0.50 \text{seg}}{0.50 \text{seg}} = 0.2$$

y la ganancia de velocidad sería, como mucho

$$S_{\max} = \frac{1}{f} = \frac{1}{0.2} = 5$$



CAPÍTULO 2. PROGRAMACIÓN PARALELA

1. Introducción

Este capítulo introduce al lector en la programación paralela. En concreto aborda: (1) las herramientas que existen para escribir código paralelo y el trabajo adicional que la escritura de código paralelo plantea a las herramientas y al programador (Sección 2), (2) las estructuras típicas de flujos de instrucciones en los códigos paralelos (Sección 3) y (2) cómo evaluar las prestaciones del código paralelo implementado y qué prestaciones máximas se pueden esperar (Sección 4). Se pueden encontrar más detalles de estos aspectos básicos de la programación paralela en J. Ortega, M. Anguita, 2005. Los problemas introducen los conceptos y, con ellos dificultades, de forma progresiva.

2. Herramientas de programación paralela

En esta sección se presentan los aspectos particulares de las herramientas de programación paralela: trabajo adicional que deben realizar frente a una herramienta de programación secuencial, modelos de programación que ofrecen, nivel de abstracción en que pueden situar al programador y comunicaciones/sincronizaciones colectivas que ofrecen al programador.

2.1. Trabajo a realizar por las herramientas de programación paralela o el programador

La programación paralela requiere que la herramienta de programación utilizada o el programador o ambos realicen el siguiente trabajo no requerido en programación secuencial:

- **Localización/detección** de paralelismo implícito en una aplicación o, dicho de otra forma, **descomposición** de la aplicación en unidades de cómputo independientes, a las que llamaremos **tareas**. Es recomendable terminar la descomposición con un grafo dirigido en el que los vértices representen tareas y los arcos dependencias entre ellas (comunicación de datos). En la Figura 1(a) se puede ver, a la derecha, el grafo que se ha obtenido a partir del código secuencial que aparece, a la izquierda, en la misma figura. El código calcula el número Pi usando integración numérica. Del grafo se deduce que el programador ha visto que las iteraciones del bucle se pueden hacer independientes si cada iteración escribe en una variable `sum` distinta. Después del bucle habría que sumar todas estas variables. Generar el grafo agiliza al programador el trabajo de obtener el código paralelo. En particular, facilita ver: (1) las tareas que se pueden ejecutar en paralelo

(en el ejemplo sería las `intervalos-1` iteraciones del bucle que están representadas en el grafo por los círculos en línea continua) y (2) las dependencias que hay entre ellas (i.e. el orden en el que se deben ejecutar). Además permite encontrar fácilmente el número de tareas que como máximo se puede ejecutar en paralelo (`intervalos-1` en el ejemplo). Este número, llamado **grado de paralelismo** del código, es el número de flujos de instrucciones (y procesadores) máximo necesario en la ejecución paralela.

- **Asignación** de las tareas o **carga de trabajo** (código y datos) a flujos de instrucciones (procesos y/o *threads* del Sistema Operativo -SO) y asignación de flujos de instrucciones a procesadores (o unidades de procesamiento) para que se puedan ejecutar en paralelo. En realidad, teniendo en cuenta que por lo general no es rentable (1) usar más flujos de instrucciones que procesadores y (2) que los flujos cambien de procesador cuando están en ejecución, la asignación a flujos podría ser también la asignación a procesadores. Los flujos pueden estar ya asociados a procesadores antes de la asignación, así se considera en este capítulo de introducción a la programación paralela. La asignación puede ser estática o dinámica, y explícita (la hace el programador) o implícita (la hace la herramienta de programación). Con una asignación **estática** el reparto de tareas entre flujos es la misma en todas las ejecuciones, mientras que con una asignación **dinámica** en cada ejecución el reparto puede ser distinto y sólo se conoce el reparto definitivo al finalizar la ejecución. En el código paralelo MPI de la Figura 1(c) el reparto de tareas es estático y lo ha realizado el programador de forma explícita. El flujo de instrucciones (en este caso proceso del SO) `iproc=0` realiza en todas las ejecuciones de este código la iteración 0, `nproc`, $2 \times nproc \dots$, el proceso `iproc=1` realiza 1, `nproc+1`, $2 \times nproc+1$, etc., `nproc` es el número total de procesos que están ejecutando el código. El reparto de las iteraciones del bucle entre los procesos se ha realizado en turno rotatorio (*Round-Robin*). Para una distribución dinámica se requiere ejecutar código extra y añadir comunicación/sincronización que introducen un retardo adicional (se pueden encontrar más detalles en J. Ortega, M. Anguita. 2005). La asignación dinámica es la única posible cuando el número de tareas no se conoce antes de la ejecución ni en ningún momento de la ejecución, entonces se tienen que ir asignando a flujos conforme van apareciendo. En la implementación OpenMP de la Figura 1(b) la asignación de flujos (en este caso *threads* del SO) la realiza la herramienta, por tanto está implícita. Las implementaciones de OpenMP de los compiladores de Intel y de GNU usan por defecto una asignación estática.
- **Comunicación/sincronización** entre los flujos de instrucciones. Los flujos están colaborando en la ejecución del programa, por lo que es normal que unos produzcan datos que necesiten otros para poder realizar las tareas asignadas. Hay que enviar a los flujos los datos que necesitan para operar que se obtienen en otros flujos. En el código MPI de la Figura 1 el programador ha utilizado la función de comunicación colectiva `MPI_Reduce()` y, en el código OpenMP, ha añadido la cláusula `reduction()` a la directiva `for` para que la herramienta inserte el código necesario para realizar una comunicación colectiva de reducción (Sección 2.4).

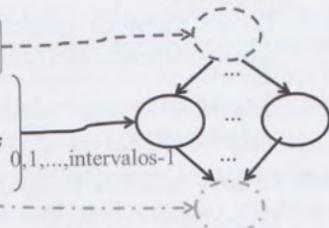
(a) Cálculo de Pi con C

```

main(int argc, char **argv) {
    double ancho, sum=0;
    int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    for (i=0;i< intervalos; i++){
        x = (i+0.5)*ancho;
        sum = sum + 4.0/(1.0+x*x);
    }
    sum* = ancho;
    ...
}

```

Grafo de dependencias entre tareas



(b) Cálculo de Pi con OpenMP/C

```

#include <omp.h>
#define NUM_THREADS 4
main(int argc, char **argv) {
    double ancho,x, sum=0; int intervalos, i;
    intervalos = atoi(argv[1]);
    ancho = 1.0/(double) intervalos;
    omp_set_num_threads(NUM_THREADS); Crear y Terminar
    #pragma omp parallel Comunicar y sincronizar
    Localizar {
        #pragma omp for reduction(+:sum) private(x)
        for (i=0;i< intervalos; i++) {
            x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
        }
        sum* = ancho;
        ...
    }

```

(c) Cálculo de Pi con MPI/C

```

#include <mpi.h>
main(int argc, char **argv) {
    double ancho,x,sum,lsum=0; int intervalos,i,nproc,iproc;
    if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1); Enrolar
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    intervalos=atoi(argv[1]);
    ancho=1.0/(double) intervalos;
    for (i=iproc; i<intervalos; i+=nproc) {
        x = (i+0.5)*ancho; lsum+= 4.0/(1.0+x*x);
    }
    lsum*= ancho;
    MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD); Comunicar/sincronizar
    MPI_Finalize(); Desenrolar
    ...
}

```

Figura 1. (a) Código secuencial C para el cálculo de Pi y códigos paralelos en C usando (b) OpenMP (API+Directivas+Funciones), estándar de-facto para programación con el estilo de variables compartidas y (c) MPI (API+funciones), estándar de-facto para el estilo de paso de mensajes. Las modificaciones realizadas al código secuencial (a) para obtener código paralelo se han destacado en negrita en (b) y (c)

Las herramientas permiten de forma **implícita** (lo implementa la propia herramienta) o **explícita** (en este último caso lo implementa el programador):

Labor 1. Localizar/detectar paralelismo, es decir, descomponer la aplicación en tareas independientes (*decomposition*)

Labor 2. Asignar las tareas, es decir, la carga de trabajo (código + datos), a flujos (*scheduling*).

Labor 3. Crear y terminar flujos de instrucciones, o enrolar y desenrolar en un grupo flujos creados previamente.

Labor 4. Comunicar/sincronizar flujos de instrucciones.

Labor 5. Asignar flujos de instrucciones a unidades de procesamiento (*mapping*).

La última labor la puede realizar el SO o el propio hardware, dependiendo del sistema de cómputo.

2.2. Nivel de abstracción en que sitúan al programador las herramientas

Cuanto mayor sea el nivel de abstracción en el que sitúe la herramienta de programación al programador menos labores de las enumeradas en la Sección 2.1 tendrá que realizar éste. Hay herramientas que permiten escoger entre distintos niveles de abstracción. La labor más difícil para la herramienta es la primera: localizar/detectar el paralelismo. Cuantas más labores haga el programador mayores prestaciones se pueden llegar a conseguir si el programador conoce la arquitectura del sistema de cómputo. Las herramientas de programación paralela se pueden clasificar teniendo en cuenta el nivel de abstracción en el que sitúan al programador. Teniendo en cuenta este criterio se podrían agrupar en las siguientes clases (se enumeran desde el mayor al menor nivel de abstracción):

- *Compiladores paralelos* (ej. compiladores de Intel). Generan código paralelo a partir de código secuencial. El programador sólo tiene que implementar el código secuencial, no tiene que realizar ninguna de las labores requeridas para generar código paralelo.
- En un segundo nivel de abstracción se pueden situar los *lenguajes paralelos* (ej. Occam, Ada, Java), y las *API de funciones y Directivas* (ej. OpenMP, OpenACC). Los lenguajes paralelos tienen construcciones particulares del lenguaje y bibliotecas de funciones, y requieren un compilador exclusivo. Las API (*Application Programming Interface*) en este nivel constan principalmente de un conjunto de directivas y una biblioteca de funciones que se añaden a un compilador de un lenguaje secuencial habitual (ej. C/C++ o Fortran). En este nivel el programador debe al menos detectar el paralelismo implícito en la aplicación (Labor 1). Por ejemplo, en el código paralelo con OpenMP de la Figura 1(b) el programador indica a la herramienta con la directiva `for` que las iteraciones del bucle `for` las puede repartir entre flujos. El programador no hace el reparto (no hace la asignación, Labor 2). La herramienta puede eximir al programador también de asignar flujos a unidades de procesamiento

(Labor 5), y de saber cómo se crean/terminan flujos (Labor 3) o detalles del código para comunicación/sincronización (Labor 4).

- *API de funciones* (ej. Pthreads, MPI). Las API en este nivel consisten principalmente en una biblioteca de funciones que se añade a un compilador de un lenguaje secuencial habitual (ej. C/C++ o Fortran). El programador explícitamente debe realizar la asignación de tareas (Labor 1). También debe participar en todas las labores mencionadas excepto probablemente en la última. En el ejemplo con MPI de la Figura 1(c) se puede ver que el programador ha tenido que repartir explícitamente las iteraciones del bucle entre los flujos modificando la sentencia `for` (Labor 2) y que ha tenido que añadir funciones para enrolar/desenrolar (Labor 3) y comunicar/sincronizar (Labor 4).
- *Lenguajes paralelos para arquitecturas de propósito específico* (ej. CUDA). Consisten en construcciones del lenguaje y en bibliotecas de funciones. Requieren un compilador exclusivo. El programador debe participar en las cuatro primeras labores requeridas y podría participar en la última, y además debe tener un conocimiento mayor de las arquitecturas destino para poder escribir el código paralelo.

2.3. Estilos de programación paralela

Cada herramienta de programación paralela ofrece al programador un modelo de programación particular. Estos modelos se pueden encuadrar en uno de los siguientes paradigmas o estilos de programación paralela: paso de mensajes (ej. MPI, que es un estándar de-facto para paso de mensajes, y PVM), variables compartidas (ej. OpenMP, que es un estándar de-facto para variables compartidas, y Pthread) o paralelismo de datos (ej. HPF –*High Performance Fortran*). Las arquitecturas paralelas se caracterizan por el estilo de programación más cercano a su implementación hardware: paso de mensajes para multicomputadores, variables compartidas para multiprocesadores y núcleos multithread, y paralelismo de datos para procesadores que ejecutan una instrucción en paralelo en múltiples unidades de procesamiento (*SIMD-Single Instruction Multiple Data*). Teniendo esto en cuenta es fácil de comprender qué caracteriza a cada uno de estos modelos de programación:

- **Paso de mensaje.** El programador debe tener en cuenta que los flujos de instrucciones no comparten memoria, cada uno de ellos tiene su espacio de direcciones en particular. La herramienta debe ofrecer el medio para copiar datos de un espacio de direcciones a otro.
- **Variables compartidas.** El programador debe tener en cuenta que los flujos de instrucciones comparten memoria, la comunicación entonces entre flujos se puede realizar a través de variables. La herramienta debe ofrecer (si no lo hace de forma implícita) el medio para sincronizar los flujos con el fin de que las comunicaciones a través de variables compartidas se realicen sin problemas: la sincronización debe conseguir que un consumidor de un dato lo lea de una variable cuando el productor lo haya colocado en dicha variable, no antes.

- **Paralelismo de datos.** El programador debe tener en cuenta que todos los flujos deben ejecutar a la vez la misma instrucción pero con datos distintos.

2.4. Comunicación/sincronización en herramientas de programación paralela

Las herramientas de programación paralela incluyen código que permite implementar una **comunicación uno-a-uno** o envío de datos desde un emisor (productor) a un receptor (consumidor). También pueden ofrecer al programador código para implementar de forma eficiente **comunicaciones colectivas**, que son comunicaciones en las que intervienen múltiples flujos enviando y/o recibiendo. En la práctica, en estas últimas intervienen todos los flujos que están colaborando en la ejecución del trozo de código en el que se usa, aunque la herramienta puede permitir deshabilitar la participación de alguno de ellos. El uso de estas comunicaciones colectivas de la herramienta evita que el programador tenga que implementarlas usando múltiples comunicaciones uno-a-uno, por lo que ahorra trabajo al programador y hace que el programa fuente paralelo que escribe sea más corto. MPI tiene funciones que implementan comunicaciones colectivas; por ejemplo, en la Figura 1(b) se usa `MPI_Reduce()` para reducir las sumas parciales calculadas por los procesos en `lsum` a un único valor, que se guarda en la variable `sum` (2º parámetro de la función) del proceso `iproc=0` (6º parámetro). Para hacer la reducción se usa una operación de suma (5º parámetro, `MPI_SUM`). OpenMP, por su parte, incluye código para realizar comunicaciones colectivas cuando el programador usa ciertas cláusulas o directivas (ej. la cláusula `reduction()` en la Figura 1(c)).

Las comunicaciones colectivas se pueden clasificar en las siguientes clases (intervienen el grupo de flujos que están colaborando en la ejecución del trozo de código, cuatro en los ejemplos de la Figura 2):

- Comunicaciones uno-a-todos. Uno de los flujos de control envía y todos reciben (puede recibir o no el que envía): difusión (*broadcast*), dispersión (*scatter*).
- Comunicaciones todos-a-uno. Todos los flujos de control envían y uno recibe (puede enviar o no el que recibe): acumulación (*gather*), reducción (*reduction*).
- Comunicaciones todos-a-todos. Todos los flujos de control envían a todos y todos reciben de todos: todos difunden (*all-broadcast*), todos dispersan (*all-scatter*).
- Comunicaciones múltiples uno-a-uno: permutaciones (rotación, baraje, etc., todos los flujos envían a uno y todos reciben de uno), desplazamientos.
- Comunicaciones colectivas compuestas de varias de las anteriores: todos combinan (*all-reduction*), recorrido (*scan*) y barreras (*barrier*). Esta última es un punto en el código en el que todos los flujos se esperan entre sí. Cuando todos han llegado a la barrera, continúan a la vez con la ejecución del código que hay detrás de la barrera. Se compone de una reducción y una difusión (en el Capítulo 3 se pueden ver implementaciones de barrera para un modelo de programación con variables compartidas).

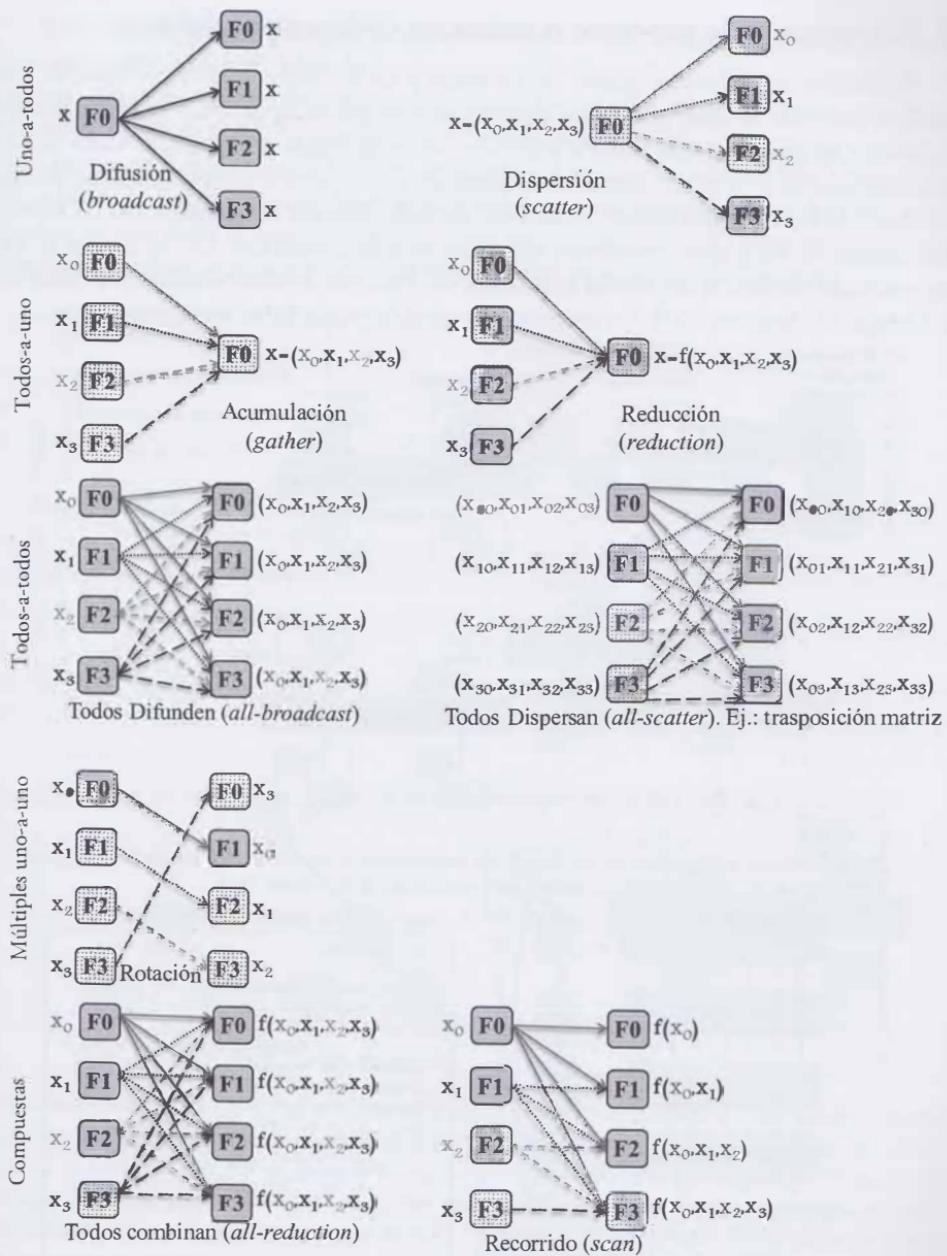


Figura 2. Funciones de comunicación colectivas. Intervienen cuatro flujos F0, F1, F2 y F3. La operación $f(\cdot)$ que reduce múltiples valores a uno es conmutativa y asociativa. Cuando en los ejemplos un flujo recibe o envía un dato (x_i o x_{ij}) se podría también recibir o enviar múltiples datos (x_i o x_{ij} podrían ser vectores).

3. Estructuras de procesos o tareas en códigos paralelos

Analizando la estructura (grafo) de las tareas y de los flujos de instrucciones de los códigos paralelos, se observa que hay ciertos patrones que se repiten. Las estructuras más habituales en programación paralela se pueden ver en la Figura 3 y Figura 4: *master-slave*, descomposición de dominio, segmentada (*pipeline*), y la estructura de tareas/fluxos en forma de árbol de la alternativa divide y vencerás, también llamada descomposición recursiva. Las flechas en los grafos representan comunicación/sincronización. En las figuras se ha representado también la asignación a procesadores. Para más detalles consulte, por ejemplo, J. Ortega, M. Anguita, 2005. En un programa paralelo puede haber varias estructuras.

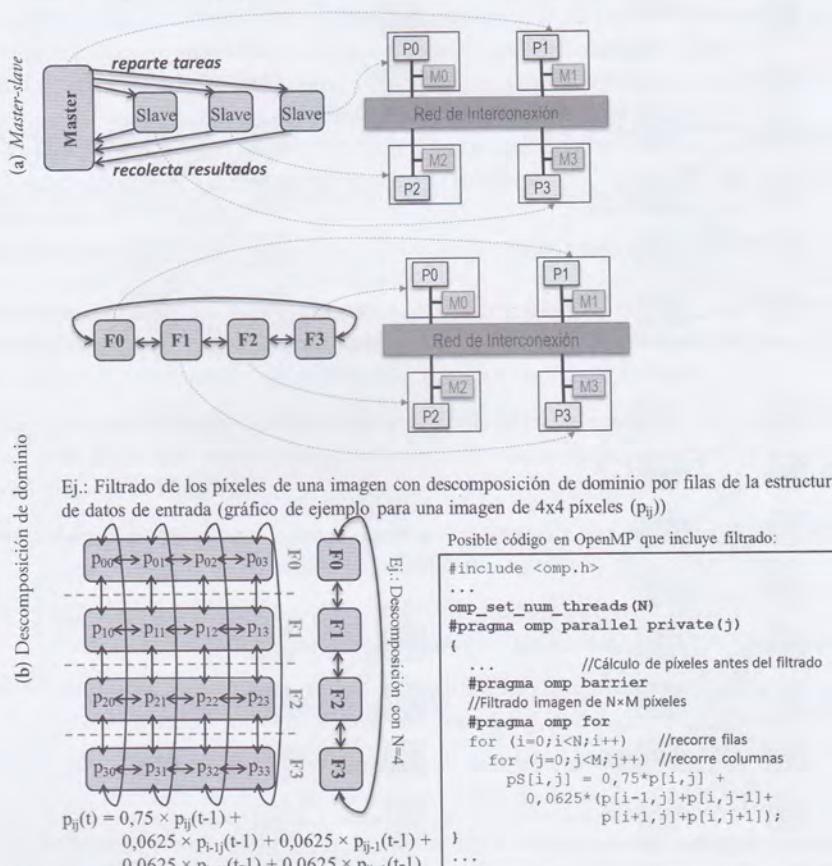
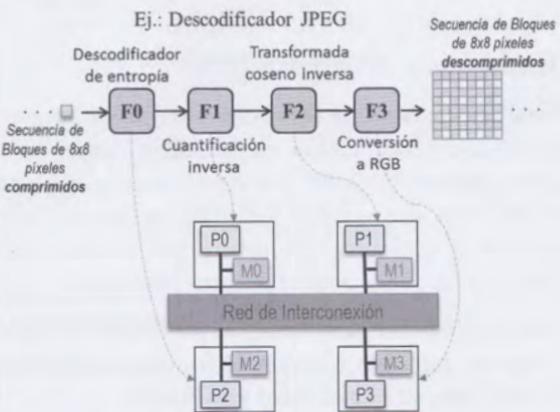


Figura 3. Estructura de tareas o fluxos en códigos paralelos suponiendo cuatro fluxos (cuatro procesadores): (a) *Master-slave* y (b) Descomposición de dominio. Se presenta el grafo de fluxos y la asignación de fluxos a procesadores. En (b) se presenta además la posible descomposición de dominio y código OpenMP para un ejemplo: filtrado de los píxeles de una imagen. La descomposición de la matriz de píxeles se ha realizado por filas (líneas discontinuas).

En el caso de descomposición de dominio (Figura 3(b)) el trabajo a realizar por cada proceso se determina dividiendo las estructuras de datos de entrada o las de salida (o ambas) en trozos, tanto como flujos. Si se divide las entradas, el trabajo a asignar al flujo i será el que utilice las entradas del trozo i . Si se divide las salidas, el trabajo a asignar al flujo i será el que genere las salidas del trozo i . Las comunicaciones que aparecen en el grafo de la Figura 3(b) son las más habituales con descomposición de dominio pero podría incluso darse el caso de no haber ninguna comunicación. Es frecuente encontrar implementaciones que combinen *master-slave* y descomposición de dominio.

(a) Estructura segmentada

```
#include <omp.h>
...
omp_set_num_threads(4);
...
#pragma omp parallel
{
...
    for (i=0;i<N;i++) {
        ...
        #pragma omp sections
        {
            #pragma omp section
            { F0(); }
            #pragma omp section
            { F1(); }
            #pragma omp section
            { F2(); }
            #pragma omp section
            { F3(); }
        }
    }
}
```



(b) Divide y vencerás

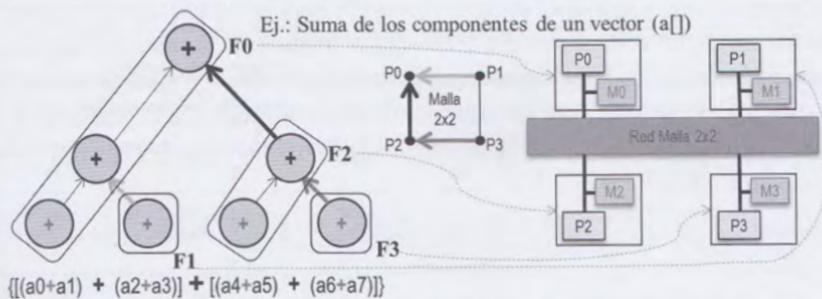


Figura 4. Estructura de tareas o flujos en códigos paralelos suponiendo cuatro flujos (cuatro procesadores): (a) Segmentada (*pipeline*) y (b) Divide y vencerás o descomposición recursiva. En (a) se presenta el grafo de flujos y su asignación a procesadores, y una posible estructura de código OpenMP para segmentación, en este caso con cuatro etapas. En (b) se presenta un grafo de tareas en forma de árbol, típico con divide y vencerás, para un ejemplo (la suma de un vector $a[]$) y, además, la agrupación de tareas en flujos típica con un árbol de tareas y la asignación a procesadores suponiendo una red de interconexión malla.

La estructura de cauce segmentado (Figura 4(a)) se utiliza cuando se aplica a un flujo de datos de entrada las mismas funciones en secuencia (una detrás de otra). Por ejemplo, en un decodificador JPEG (ver ejemplo de la figura) se aplica a bloques de 8x8 píxeles de una imagen las siguientes funciones en orden: primero una decodificación de entropía,

al resultado de esta función se aplica una cuantificación inversa, a la salida de ésta la transformada del coseno inversa y, por último, se hace una conversión a RGB.

Divide y vencerás (Figura 4(b)) se usa cuando el problema a resolver se puede dividir en subproblemas que son instancias más pequeñas del problema original, de forma que combinando los resultados de los subproblemas se resuelve el problema original. Los subproblemas se pueden dividir a su vez, dando lugar a una estructura en forma de árbol para las tareas. Por ejemplo, la suma de 8 números se puede dividir en dos sumas de 4 números, y la suma de 4 números en dos sumas de 2 números (Figura 4(b)).

4. Evaluación de prestaciones

Finalizada la escritura del código paralelo, el programador querrá evaluar las prestaciones del código resultante y, en particular, comprobar si ha conseguido las prestaciones mínimas que requiere la aplicación programada. Hay aplicaciones en las que el resultado de la ejecución sólo es útil si ésta termina con suficiente antelación como para que se pueda usar el resultado. Esto ocurre, por ejemplo, en aplicaciones de predicción del tiempo o en las que se predice peligros potenciales.

En programación paralela, se usa para evaluar las prestaciones de un código paralelo, su tiempo de respuesta (tiempo de ejecución, *walk-clock time*, *elapsed time*, *real time*) y, adicionalmente, su escalabilidad y eficiencia.

Con un estudio de escalabilidad se pretende evaluar en qué medida aumentan las prestaciones de un código paralelo conforme se incrementa el número de procesadores usados en la ejecución. El incremento o **ganancia en prestaciones**, también denominado ganancia en velocidad (*speed-up*), que se obtiene utilizando p procesadores se calcula dividiendo el tiempo de ejecución secuencial T_s por el tiempo de ejecución en paralelo usando los p procesadores $T_p(p)$:

$$S(p) = \frac{\text{Prestaciones}(p)}{\text{Prestaciones}(1)} = \frac{T_s}{T_p(p)} \quad (1)$$

El tiempo de ejecución en paralelo depende del tiempo que supone la ejecución en paralelo de las tareas presentes en el código secuencial (tiempo de cálculo paralelo, T_c) y del tiempo de **sobrecarga** (*overhead*) introducido por la parallelización (T_o):

$$T_p(p, n) = T_c(p, n) + T_o(p, n) \quad (2)$$

En esta expresión se ha tenido en cuenta que en el tiempo también influye el tamaño del problema n ; por ejemplo, si el código suma los componentes de un vector, n será el número de componentes. El tiempo de sobrecarga depende del tiempo que supone la creación y terminación de los flujos de instrucciones (depende de p), del tiempo de comunicación/sincronización (puede depender de p y/o n) y de cálculos o funciones

no presentes en la versión secuencial (ver ejemplos en Figura 1). También en alguna bibliografía se considera dentro de la sobrecarga la penalización por no lograr repartir la carga de trabajo de forma equilibrada entre los flujos.

En la Figura 5 se pueden ver varias curvas de escalabilidad. Los modelos de código a partir de los que se obtienen las gráficas (3), (5) y (6) de esta figura se pueden ver en la Figura 6 con las correspondientes expresiones de ganancia en prestaciones.

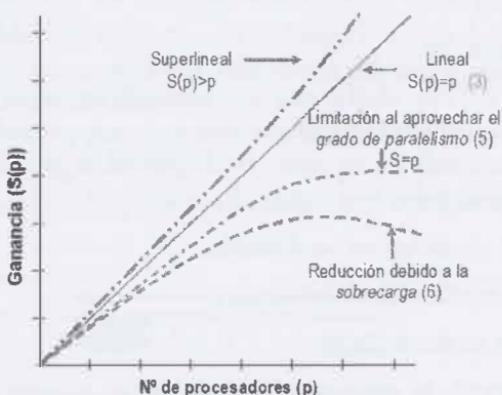
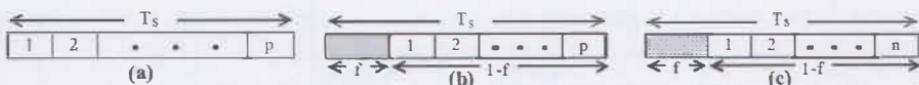


Figura 5. Curvas de escalabilidad



Modelo código	Fracción no paral. en T_s	Grado paralelismo	Sobrecarga (Overhead)	Ganancia en función del número de procesadores p con T_s constante
(a)	0	ilimitado	0	$S(p) = \frac{T_s}{T_p(p)} = \frac{T_s}{T_s/p} = p$ Ganancia lineal (3)
(b)	f	ilimitado	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$ Ley de Amdahl (4)
(c)	f	n	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \xrightarrow{p=n} \frac{1}{f + \frac{(1-f)}{n}}$ (5)
(b)	f	ilimitado	Incrementa linealmente con p	$S(p) = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_O(p)}{T_s}} \xrightarrow{p \rightarrow \infty} 0$ (6)

Figura 6. Ganancia en prestaciones para diferentes modelo de código: (a) Todo el código se puede repartir por igual entre los p procesadores disponibles, sea cual sea p (i.e. el grado de paralelismo es ilimitado), (b) hay una fracción f del código secuencial (del tiempo de ejecución secuencial T_s) que no se puede parallelizar y el resto $(1-f)$ se puede repartir entre los p procesadores disponibles, sea cual sea p , (c) hay una fracción f del código secuencial que no se puede parallelizar y el resto $(1-f)$ se puede dividir hasta n trozos (como ocurre en un trozo de código con un bucle de n iteraciones).

En el mejor caso se esperaría obtener un tiempo de ejecución de T_s/p con p procesadores para un código secuencial con un tiempo de ejecución de T_s . Para llegar a este tiempo se debe:

1. Poder repartir todo el código entre los procesadores disponibles, independientemente de cuál sea este número (grado de paralelismo ilimitado).
2. Hacer el reparto asignando a cada procesador la misma cantidad de trabajo (**carga equilibrada**).
3. No añadir sobrecarga.

La escalabilidad para este caso ideal es una línea recta ((3) en Figura 5 y Figura 6)). En la práctica la curva de escalabilidad no será una línea recta, aunque podrá parecerlo para un rango de procesadores (desde 0 a un valor). Por lo general, la ganancia estará por debajo de la ideal, porque puede haber (ver Figura 6):

1. una fracción f de código no paralelizable,
2. un reparto no equilibrado de trabajo, y/o
3. una sobrecarga no despreciable.

También por lo general, la ganancia dejará de crecer a partir de un número de procesadores, debido a que llegará un momento en que no se podrá repartir el trabajo entre más procesadores ((5) en Figura 5 y Figura 6)). Incluso puede que a partir de un número de procesadores la ganancia comience a decrecer debido a que la sobrecarga se puede incrementar con el número de procesadores ((6) en Figura 5 y Figura 6)).

Aunque el grado de paralelismo fuese ilimitado y no hubiera sobrecarga, la ganancia estaría limitada por la parte que no se puede parallelizar (modelo de código (4) en Figura 6), la ganancia en prestación máxima va a estar limitada por la fracción del código no paralelizable. El tiempo de esta fracción ($f \times T_s$ en la figura) permanecerá constante. Así que, incluso en el caso ideal de que se consiga hacer 0 el tiempo de ejecución de la parte parallelizable (añadiendo procesadores) y la sobrecarga, el tiempo de ejecución paralelo nunca podrá ser inferior al tiempo que supone la ejecución de la parte no paralelizable y, por tanto, la ganancia en prestaciones nunca podrá ser superior a $1/f$ ($= T_s / f \times T_s$). Este razonamiento fue formalizado por Gene M. Amdahl en su conocida ley, que se suele representar con la expresión (4) de la Figura 6. La **ley de Amdahl** dice que la ganancia en prestaciones que se puede conseguir aplicando paralelismo a un código secuencial está limitada por la fracción no paralelizable del mismo. Esta ley se ha generalizado (Capítulo 1), se puede usar en cualquier sistema al que se aplica una mejora: la ganancia en prestaciones que se puede conseguir aplicando una mejora a un sistema está limitada por la parte (fracción) del sistema que no usa la mejora.

La ley de Amdahl da una visión pesimista de las ventajas de la parallelización, ya que dice que la ganancia en prestaciones (y la escalabilidad) está limitada, y que este límite depende de la fracción de código no paralelizable. Gene M. Amdahl era escéptico con

respecto a la viabilidad del paralelismo masivo, como puso de manifiesto en su artículo de 1967 (G. M. Amdahl, 1967). Pero en la práctica la fracción de código secuencial que limita la escalabilidad se puede disminuir en muchas aplicaciones aumentando el tamaño del problema que resuelve el código. Por ejemplo, en un código basado en un bucle (que se corresponde con el modelo de la Figura 6(c)) es muy probable que el bucle recorra operando alguna estructura vectorial o matricial cuyo tamaño varíe en función del tamaño del problema que se resuelve. Aumentando el tamaño se mejora generalmente la calidad de los resultados. En el código basado en un bucle que calcula Pi (Figura 1(a)), por ejemplo, aumentar el tamaño del problema (número de iteraciones del bucle) supone calcular Pi con mayor precisión. Esto lleva a reflexionar que la paralelización de un código puede tener dos objetivos:

- Disminuir el tiempo de ejecución.
- Aumentar el tamaño del problema a resolver para con ello mejorar la calidad del resultado.

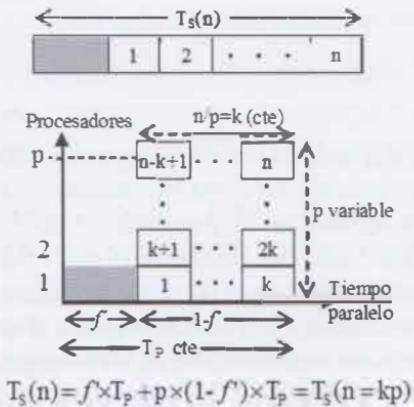


Figura 7. Modelo de código secuencial con una parte no paralelizable y un número de tareas a paralelizar n que se puede incrementar aumentando el tamaño del problema (ej. código basado en un bucle). En el que el tiempo de ejecución paralelo, T_p , se mantiene constante conforme se incrementa el número de procesadores p . Para mantener constante T_p el tamaño del problema se incrementa para conseguir que n/p sea constante.

Si el objetivo, al aumentar el número de procesadores p , es mejorar la calidad de los resultados aumentando el tamaño del problema, se puede obtener una ganancia en prestaciones para el modelo de código de la Figura 7 (código basado en un bucle como los códigos de la Figura 6(c) y Figura 1) que no está limitada y que depende linealmente de p :

$$S(p) = \frac{T_s(n = kp)}{T_p} = \frac{f' \times T_p + p \times (1 - f') \times T_p}{T_p} = f' + p \times (1 - f') \quad (7)$$

Esta expresión se denomina **ganancia escalable** en J. L. Gustafson, 1988. Mientras que Gene M. Amdahl asume que el tiempo de ejecución secuencial (o tamaño del problema) se mantiene constante conforme se incrementa el número de procesadores, concluyendo que la ganancia en prestaciones está limitada debido a la parte del código no paralelizable, John L. Gustafson mantiene constante el tiempo de ejecución paralelo y muestra que la ganancia puede crecer con pendiente constante conforme se incrementa el número de procesadores.

Para evaluar en qué medida las prestaciones que ofrece un sistema para un código paralelo se acercan a las prestaciones máximas que idealmente debería ofrecer dado los procesadores disponibles en el mismo, se usa la siguiente expresión de **eficiencia** (se ha tenido en cuenta la expresión (1)):

$$E(p) = \frac{\text{Prestacion es}(p, n)}{p \times \text{Prestacion es}(1, n)} = \frac{S(p)}{p} \quad (8)$$

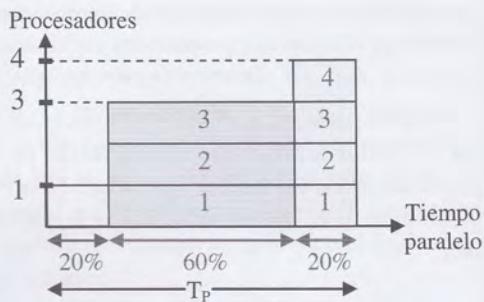
5. Problemas

Problema 1. Un programa tarda 40 s en ejecutarse en un multiprocesador. Durante un 20% de ese tiempo se ha ejecutado en cuatro procesadores (núcleos); durante un 60%, en tres; y durante el 20% restante, en un procesador (considerar que se ha distribuido la carga de trabajo por igual entre los procesadores que colaboran en la ejecución en cada momento, y despreciar sobrecarga). (a) ¿Cuánto tiempo tardaría en ejecutarse el programa en un único procesador? (b) ¿Cuál es la ganancia en velocidad obtenida con respecto al tiempo de ejecución secuencial? (c) ¿Y la eficiencia?

Solución

Datos del ejercicio:

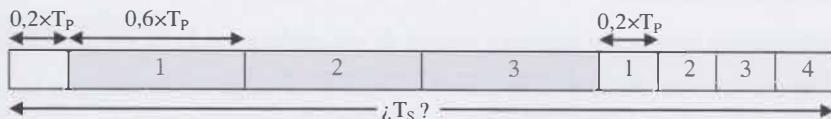
En el gráfico de la derecha se representan los datos del ejercicio. Estos datos son la fracción del tiempo de ejecución paralelo (T_p) que supone el código no paralelizable (20% de T_p , es decir, $0,2 \times T_p$), la fracción que supone el código paralelizable en 3 procesadores ($0,6 \times T_p$) y la que supone el código paralelizable en 4 procesadores ($0,2 \times T_p$). Al distribuirse la carga de trabajo por igual entre los procesadores utilizados en cada instante, los trozos asignados a 3 procesadores suponen todos el mismo tiempo (por eso se han dibujado en el gráfico de



igual tamaño) e, igualmente, los trozos asignados a 4 procesadores también suponen todos el mismo tiempo.

(a) Tiempo de ejecución secuencial, T_s :

Debido a que sólo se usa un procesador (núcleo), los trozos de código representados en la gráfica anterior se deben ejecutar secuencialmente, es decir, uno detrás de otro (en algún orden), como se ilustra el gráfico siguiente:



$$T_s = 0,2 \times T_p + 3 \times 0,6 \times T_p + 4 \times 0,2 \times T_p = (0,2 + 1,8 + 0,8) \times T_p \\ = 2,8 \times T_p = 2,8 \times 40 \text{ s} = 112 \text{ s}$$

(b) Ganancia en velocidad, $S(p)$:

$$S(4) = \frac{T_s}{T_p(4)} = \frac{2,8 \times T_p(4)}{T_p(4)} = 2,8$$

(c) Eficiencia, $E(p)$:

$$E(4) = \frac{S(p)}{p} = \frac{S(4)}{4} = \frac{2,8}{4} = 0,7$$

Problema 2. Un programa tarda 20 s en ejecutarse en un procesador P1, y requiere 30 s en otro procesador P2. Si se dispone de los dos procesadores para la ejecución del programa (despreciamos sobrecarga):

(a) ¿Qué tiempo tarda en ejecutarse el programa si la carga de trabajo se distribuye por igual entre los procesadores P1 y P2?

(b) ¿Qué distribución de carga entre los dos procesadores P1 y P2 permite el menor tiempo de ejecución utilizando los dos procesadores en paralelo? ¿Cuál es este tiempo?

Solución

Datos del ejercicio:

$$T^{p1} = 20 \text{ s}$$

$$T^{p2} = 30 \text{ s}$$

(a) Tiempo de ejecución paralelo distribuyendo la carga por igual, T_p :

$$T_p^{P1}(1/2) = 20s/2 = 10 \text{ s} \text{ y } T_p^{P2}(1/2) = 30s/2 = 15 \text{ s}$$

$$T_p^{P1,P2} = \max(20s/2, 30s/2) = 15 \text{ s}$$

En el entorno de trabajo heterogéneo que conforma P1 y P2, si la carga de trabajo se distribuye por igual entre los procesadores, el tiempo de ejecución en paralelo lo determinará el procesador que acabe más tarde de ejecutar su mitad. La distribución de carga de trabajo no se ha hecho de forma que los procesadores empiecen y terminen a la vez (no se ha equilibrado).

(b) Distribución con menor tiempo y tiempo de ejecución paralelo en ese caso, T_p :

El mejor caso se obtiene si se puede distribuir la carga de forma que los dos procesadores empiecen y terminen a la vez, el tiempo de ejecución sería:

$$T_p^{P1}(x) = T_p^{P2}(1-x) \Rightarrow 20s \times x = 30s \times (1-x) \Rightarrow 2 \times x = 3 \times (1-x) \Rightarrow 5 \times x = 3 \Rightarrow x = 3/5$$

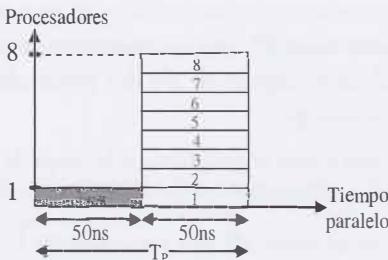
$$3/5 = 0,6 \text{ para P1 y } 2/5 = 0,4 \text{ para P2}$$

$$T_p^{P1}(3/5) = 20s \times 3/5 = 12 \text{ s}$$

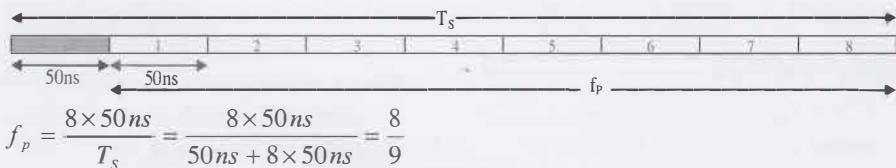
Problema 3. ¿Cuál es fracción de código paralelo de un programa secuencial que, ejecutado en paralelo en 8 procesadores, tarda un tiempo de 100 ns, durante 50 ns utiliza un único procesador y durante otros 50 ns utiliza 8 procesadores (distribuyendo la carga de trabajo por igual entre los procesadores)?

Solución

Datos del programa:



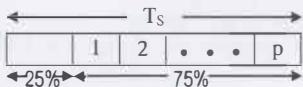
Fracción de código paralelizable del programa secuencial, f_p :



Problema 4. Un 25% de un programa no se puede paralelizar, el resto se puede distribuir por igual entre cualquier número de procesadores. (a) ¿Cuál es el máximo valor de ganancia de velocidad que se podría conseguir al paralelizarlo en p procesadores, y con infinitos? (b) ¿A partir de cuál número de procesadores se podrían conseguir ganancias mayores o iguales que 2?

Solución

Datos del ejercicio:



(a) Ganancia en velocidad con p procesadores y máxima ideal, $S(p)$:

$$S(p) = \frac{T_S}{T_p} = \frac{T_S}{0,25 \times T_S + \frac{0,75 \times T_S}{p}} = \frac{1}{0,25 + \frac{0,75}{p}}$$

$$S(p) = \frac{1}{0,25 + \frac{0,75}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{0,25} = 4$$

(b) Número de procesadores a partir del cual se obtienen ganancias mayores o iguales que 2:

$$S(p) = \frac{1}{0,25 + \frac{0,75}{p}} \geq 2 \Rightarrow 1 \geq 0,5 + \frac{1,5}{p} \Rightarrow 0,5 \geq \frac{1,5}{p} \Rightarrow p \geq \frac{1,5}{0,5} = 3$$

Problema 5. En la Figura 8, se presenta el grafo de dependencia entre tareas para una aplicación. La figura muestra la fracción del tiempo de ejecución secuencial que la aplicación tarda en ejecutar las tareas del grafo. Suponiendo un tiempo de ejecución secuencial de 60 s, que las tareas no se pueden dividir en tareas de menor granularidad y que el tiempo de comunicación es despreciable, obtener el tiempo de ejecución en paralelo y la ganancia en velocidad en un computador con:

- (a) 4 procesadores.
- (b) 2 procesadores.

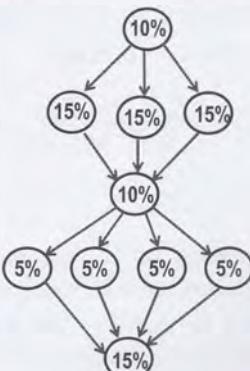
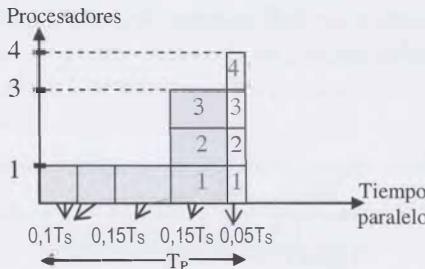


Figura 8. Grafo de tareas del ejercicio 5.

Solución

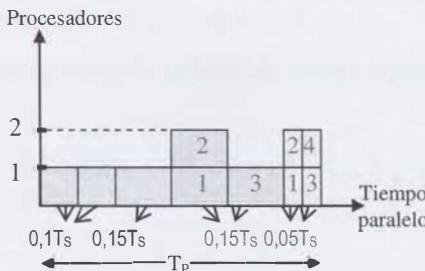
(a) Tiempo de ejecución paralelo, T_p y ganancia en velocidad, $S(p)$, para $p=4$:



$$T_p(4) = (0,1 + 0,1 + 0,15) \times T_s + 0,15 \times T_s + 0,05 \times T_s = (0,35 + 0,15 + 0,05) \times T_s \\ = 0,55 \times T_s = 0,55 \times 60s = 33s$$

$$S(4) = \frac{T_s}{T_p(4)} = \frac{T_s}{0,55 \times T_s} = \frac{1}{0,55} \approx 1,82$$

(b) Tiempo de ejecución paralelo, T_p y ganancia en velocidad, $S(p)$, para $p=2$:



$$T_p(2) = (0,1 + 0,1 + 0,15) \times T_s + 2 \times 0,15 \times T_s + 2 \times 0,05 \times T_s = (0,35 + 0,3 + 0,1) \times T_s \\ = 0,75 \times T_s = 0,75 \times 60s = 45s$$

$$S(4) = \frac{T_s}{T_p(4)} = \frac{T_s}{0,75 \times T_s} = \frac{1}{0,75} \approx 1,33$$

Problema 6. Un programa se ha conseguido dividir en 10 tareas. El orden de precedencia entre las tareas se muestra con el grafo dirigido de la Figura 9. La ejecución de estas tareas en un procesador supone un tiempo de 2 s. El 10% de ese tiempo es debido a la ejecución de la tarea 1; el 15% a la ejecución de la tarea 2; otro 15% a la ejecución de 3; cada tarea 4, 5, 6 o 7 supone el 9%; un 8% supone la tarea 8; la tarea 9

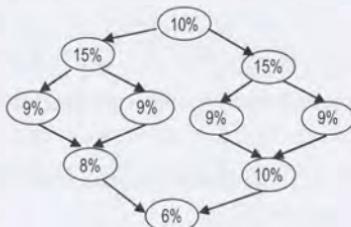
un 10%; por último, la tarea 10 supone un 6%. Se dispone de una arquitectura con 8 procesadores para ejecutar la aplicación. Se considera que el tiempo de comunicación se puede despreciar. Conteste a las siguientes preguntas:

(a) ¿Qué tiempo tarda en ejecutarse el programa en paralelo?

(b) ¿Qué ganancia en velocidad se obtiene con respecto a su ejecución secuencial?

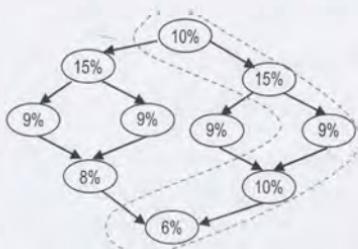
Solución

Datos del ejercicio: $T_s = 2$ s y $p = 8$



(a) Tiempo de ejecución paralelo, T_p :

Hay procesadores suficientes como para poder aprovechar todo el grado de paralelismo del programa. En este caso el grado es 4, ya que 4 es el número máximo de tareas que se pueden ejecutar en paralelo. El tiempo de ejecución paralelo dependerá del camino más largo (que supone mayor tiempo de ejecución) en el grafo de dependencia entre tareas. Como suponemos despreciable el tiempo de comunicación, el camino más largo va a depender sólo del tiempo de cálculo de las tareas. En este caso hay dos caminos que llevan al mayor tiempo de ejecución, uno de ellos está englobado en línea discontinua en el siguiente grafo, ambos caminos pasan por la tarea 9.



$$T_p(8) = T_p(4) = (0,1 + 0,15 + 0,09 + 0,1 + 0,06) \times T_s = 0,5 \times T_s = 0,5 \times 2s = 1s$$

(b) Ganancia en velocidad, $S(p)$:

$$S(8) = S(4) = \frac{T_s}{T_p(4)} = \frac{T_s}{0,5 \times T_s} = \frac{1}{0,5} = 2$$

Problema 7. Se quiere paralelizar el siguiente trozo de código:

```
{Cálculos antes del bucle}
for( i=0; i<w; i++) {
    Código para i
}
{Cálculos después del bucle}
```

Los cálculos antes y después del bucle suponen un tiempo de t_1 y t_2 , respectivamente. Una iteración del ciclo supone un tiempo t_i . En la ejecución paralela, la inicialización de p procesos supone un tiempo $k_1 p$ (k_1 constante), los procesos se comunican y se sincronizan, lo que supone un tiempo $k_2 p$ (k_2 constante) ($k_1 p + k_2 p$ es la sobrecarga).

- Obtener una expresión para el tiempo de ejecución paralela del trozo de código en p procesadores (T_p).
- Obtener una expresión para la ganancia en velocidad de la ejecución paralela con respecto a una ejecución secuencial (S_p).
- ¿Tiene el tiempo T_p con respecto a p una característica lineal o puede presentar algún mínimo? ¿Por qué? En caso de presentar un mínimo, ¿para qué número de procesadores p se alcanza?

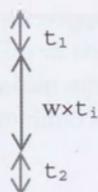
Solución

Datos del ejercicio:

```
{Cálculos antes del bucle}
for( i=0; i<w; i++) {
    Código para i
}
{cálculos después del bucle}
```

Sobrecarga: $T_0 = k_1 p + k_2 p = (k_1 + k_2) p$

Llamaremos t a $t_1 + t_2$ ($= t$) y k a $k_1 + k_2$ ($= k$)



- Tiempo de ejecución paralelo, T_p :

$$T_p(p, w) = t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p$$

w/p se redondea al entero superior

(b) Ganancia en prestaciones, $S(p, w)$:

$$S(p, w) = \frac{T_s}{T_p(p, w)} = \frac{t + w \times t_i}{t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p}$$

(c) ¿Presenta mínimo? ¿Para qué número p ?

$$T_p(p, w) = t + \left\lceil \frac{w}{p} \right\rceil \times t_i + k \times p \quad (9)$$

En la expresión (9), el término englobado en línea continua, o tiempo de cálculo paralelo, tiene tendencia a decrecer con pendiente que va disminuyendo conforme se incrementa p (debido a que p está en el divisor), y el término englobado con línea discontinua o tiempo de sobrecarga ($k \times p$), crece conforme se incrementa p con pendiente k constante (ver ejemplos en Figura 10 y Figura 11). Las oscilaciones en las figuras del tiempo de cálculo paralelo se deben al redondeo al entero superior del cociente w/p , pero la tendencia es que T_p va decreciendo. Dado que la pendiente de la sobrecarga es constante y que la pendiente del tiempo de cálculo decrece conforme se incrementa p , llega un momento en que el tiempo de ejecución en paralelo pasa de decrecer a crecer.

Se puede encontrar el mínimo analíticamente igualando a 0 la primera derivada de T_p . De esta forma se obtiene los máximos y mínimos de una función continua. Para comprobar si en un punto encontrado de esta forma hay un máximo o un mínimo se calcula el valor de la segunda derivada en ese punto. Si, como resultado de este cálculo, se obtiene un valor por encima de 0 hay un mínimo, y si, por el contrario, se obtiene un valor por debajo de 0 hay un máximo. Para realizar el cálculo se debe eliminar el redondeo de la expresión (9) con el fin de hacer la función continua (después se comentará la influencia del redondeo en el cálculo del mínimo):

$$T_p(p, w) = t + \frac{w}{p} \times t_i + k \times p$$

$$T'_p(p, w) = 0 - \frac{w}{p^2} \times t_i + k \Rightarrow \frac{w}{p^2} \times t_i = k \Rightarrow p = \sqrt{\frac{w \times t_i}{k}} \quad (10)$$

El resultado negativo de la raíz se descarta. En cuanto al resultado positivo, debido al redondeo, habrá que comprobar para cuál de los naturales próximos al resultado obtenido (incluido el propio resultado si es un número natural) se obtiene un menor tiempo. Debido al redondeo hacia arriba de la expresión (9), se deberían comprobar necesariamente:

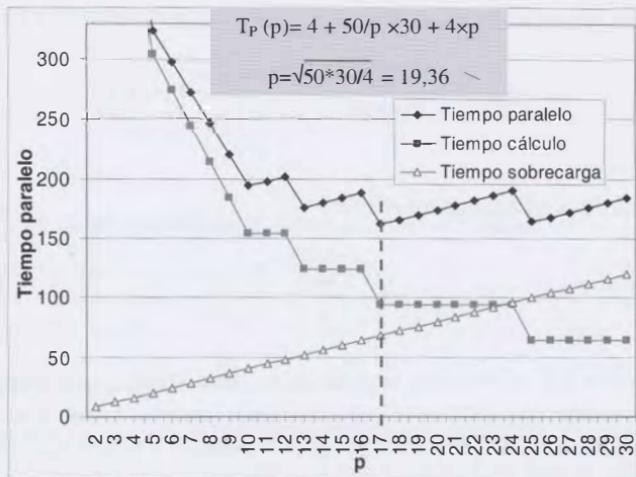


Figura 10. Tiempo de ejecución paralelo para un caso particular en el que $w=50$, t_i es 30 unidades de tiempo, t son 4 unidades de tiempo y k son 4 unidades de tiempo. El mínimo se alcanza para $p=17$.

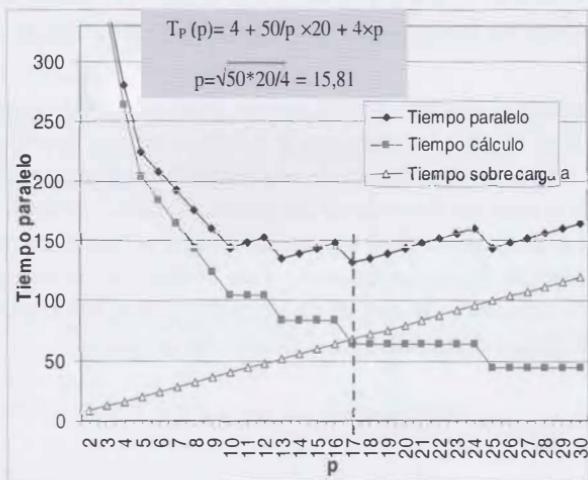


Figura 11. Tiempo de ejecución paralelo para un caso particular en el que $w=50$, t_i es 20 unidades de tiempo, t son 4 unidades de tiempo y k son 4 unidades de tiempo. El mínimo se alcanza para $p=17$.

1. El natural p' menor o igual y más alejado al resultado p generado con (10) para el que

$$\left\lceil \frac{w}{p'} \right\rceil = \left\lceil \frac{w}{p} \right\rceil$$

Observar, que en el ejemplo de la Figura 10, aunque de (10) se obtiene 19,36, el p con menor tiempo es 17 debido al efecto del redondeo.

- 2 El natural p' mayor y más próximo al p generado con (10) para el que

$$\left\lceil \frac{w}{p'} \right\rceil = \left\lceil \frac{w}{p} \right\rceil - 1$$

Observar, que en el ejemplo de la Figura 11, aunque de (10) se obtiene 15,81, el p con menor tiempo es 17 debido al redondeo.

En cualquier caso, el número de procesadores que obtengamos debe ser menor que (que es el grado de paralelismo del código).

La segunda derivada permitirá demostrar que se trata de un mínimo:

$$T_p(p, w) = t + \frac{w}{p} \times t_i + k \times p$$

$$T''_p(p, w) = + \frac{2 \times p \times w \times t_i}{p^4} + 0 = \frac{2 \times w \times t_i}{p^3} > 0$$

La segunda derivada es mayor que 0, ya que w, p y t_i son mayores que 0; por tanto, hay un mínimo.

Problema 8. Supongamos que se va a ejecutar en paralelo la suma de n números en una arquitectura con p procesadores (p y n potencias de dos) utilizando un grafo de dependencias en forma de árbol (divide y vencerás) para las tareas.

- Dibujar el grafo de dependencias entre tareas para n = 16 y p igual a 8. Hacer una asignación de tareas a flujos de instrucciones.
- Obtener el tiempo de cálculo paralelo para cualquier n y p con $n > p$ suponiendo que se tarda una unidad de tiempo en realizar una suma.
- Obtener el tiempo comunicación del algoritmo suponiendo (1) que las comunicaciones en un nivel del árbol se pueden realizar en paralelo en un número de unidades de tiempo igual al número de datos que recibe o envía un proceso en cada nivel del grafo de tareas (tenga en cuenta la asignación de tareas a procesos que ha considerado en el apartado (a)) y (2) que los procesadores que realizan las tareas de las hojas del árbol tienen acceso sin coste de comunicación a los datos que utilizan dichas tareas.
- Suponiendo que el tiempo de sobrecarga coincide con el tiempo de comunicación calculado en (c), obtener la ganancia en prestaciones.
- Obtener el número de procesadores para el que se obtiene la máxima ganancia con n números.

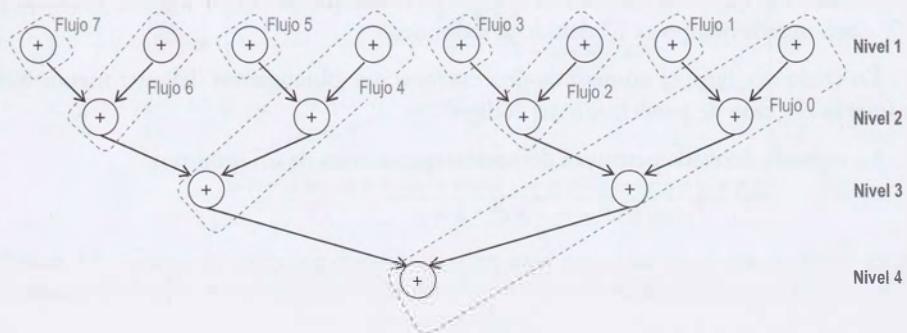
Solución

Datos del ejercicio:

Suma de n números en paralelo con $p=2^p$ y $n=2^N$

(a) Grafo de tareas para $n=16$ y $p=2$ y asignación a procesos.

El grafo sería el siguiente:



Los flujos de instrucciones aparecen en el grafo con línea discontinua. El grado de paralelismo es 8, se puede aprovechar, por tanto, por los 8 procesadores disponibles.

(b) Tiempo de cálculo paralelo, T_c , para $n>p$.

Datos: Cada suma supone una unidad de tiempo.

Primero se va a obtener el tiempo de cálculo paralelo para el ejemplo del grafo del apartado (a). Cada tarea del grafo tiene que realizar una suma, lo que supone una unidad de tiempo. Todas las tareas del nivel 1 del árbol se pueden ejecutar en paralelo al no haber dependencias entre ellas (no hay arcos del grafo entre ellas), esta ejecución supone 1 unidad de tiempo. También todas las tareas del nivel 2 se pueden ejecutar en paralelo en una unidad, al igual que las tareas del nivel 3. La tarea de nivel 4 será la última en ejecutarse y supondrá una unidad. En total se consume 4 unidades:

$$T_c(16,8) = 4 = 1 + \lg_2 8$$

En general, para un $p=n/2$ sería (habrá $1+\lg_2 p$ niveles de tareas en el árbol binario):

$$T_c(p, n; n = 2p) = 1 + \lg_2 p$$

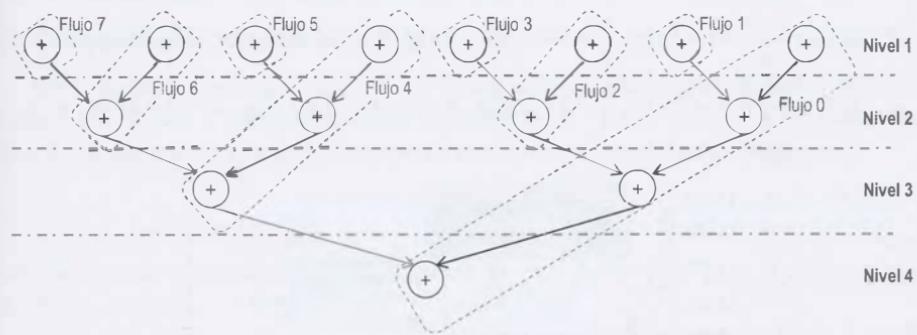
Para un $p>n$ ($n=kp$, con k potencia de 2), teniendo en cuenta que el máximo grado de paralelismo aprovechable sería igual al número de procesadores (flujos), se tendrán tantas tareas en el primer nivel del árbol como procesadores. En total habrá $1+\lg_2 p$ niveles de tareas en el árbol binario. Cada una de las tareas del primer nivel tendría asignada la suma de n/p números (todas el mismo número al ser n divisible por p), por lo que la ejecución en paralelo de estas tareas supondrá $n/p-1$ unidades de tiempo.

Para cada uno de los niveles restantes, la ejecución en paralelo de las tareas supondrá una unidad, ya que únicamente realizan la suma de dos números:

$$T_C(p, n; n = kp) = \frac{n}{p} - 1 + \lg_2 p$$

(c) Tiempo de comunicación/sincronización, $T_{C/S}$, para $n > p$.

La comunicación en cada uno de los niveles del árbol supone 1 unidad de tiempo, teniendo en cuenta el enunciado y que sólo se recibe un dato en cada comunicación. A continuación se han destacado en el grafo los niveles de comunicación trazando líneas discontinuas horizontales, las flechas entre flujos que atraviesan estas líneas (en gris) son comunicaciones a través de la red:



El número de niveles de comunicación en el ejemplo es 3, en general será $\lg_2 p$:

$$T_{C/S}(p) = \lg_2 p$$

(d) Ganancia en prestaciones, $S(p, n)$

$$S(p, n) = \frac{T_S(n)}{T_p(p, n)} = \frac{n - 1}{\frac{n}{p} - 1 + 2 \times \lg_2 p}$$

(e) Número de procesadores para el que se obtiene la máxima ganancia con n números.

Analíticamente se podría encontrar el mínimo de la función para T_p igualando a 0 la primera derivada:

$$T_p(p) = \frac{n}{p} - 1 + 2 \times \lg_2 p$$

$$T'_p(p) = -\frac{n}{p^2} + \frac{2}{p \times L2} = 0 \Rightarrow \frac{n}{p} = \frac{2}{L2} \Rightarrow p = \frac{n \times L2}{2} \approx 1,3863 \times n \approx \frac{n}{2,8854} \quad (11)$$

$$T''_p(p) = \frac{2 \times p \times n}{p^4} - \frac{2}{L2 \times p^2} = \frac{2 \times n}{p^3} - \frac{2}{L2 \times p^2} > 0 \Rightarrow \frac{2 \times n}{p^3} > \frac{2}{L2 \times p^2} \Rightarrow \frac{n}{p} > \frac{1}{L2} \Rightarrow L2 \times n > p$$

El valor de p obtenido, $n \times L2/2$, es menor que $n \times L2$ (L nota logaritmo neperiano), luego se trata de un mínimo y no de un máximo.

En la Figura 12 se ha representado para $n=32$ el tiempo paralelo, T_p , el tiempo de cálculo de las tareas en paralelo, T_c , y el tiempo de sobrecarga, T_o , en función de p (suponiendo que la función es aplicable para cualquier valor de p). En nuestro caso p es potencia de 2. El mínimo que se obtiene de la expresión (11) está entre dos potencias de 2 ($n/2$ y $n/4$), se tendrá que obtener cuál de las dos supone un tiempo menor; en el programa paralelo de este ejercicio, realmente, los dos suponen el mismo tiempo. Este tiempo se puede calcular fácilmente evaluando T_p en $n/2$ y $n/4$:

$$T_p(n/2) = \frac{2 \times n}{n} - 1 + 2 \times \lg_2 \frac{n}{2} = 1 + 2 \times (\lg_2 n - \lg_2 2) = 1 + 2 \times (\lg_2 n - 1) = 2 \times \lg_2 n - 1$$

$$T_p(n/4) = \frac{4 \times n}{n} - 1 + 2 \times \lg_2 \frac{n}{4} = 3 + 2 \times (\lg_2 n - \lg_2 4) = 3 + 2 \times (\lg_2 n - 2) = 2 \times \lg_2 n - 1$$

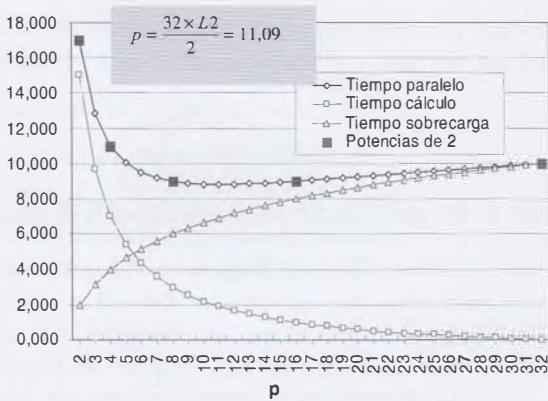
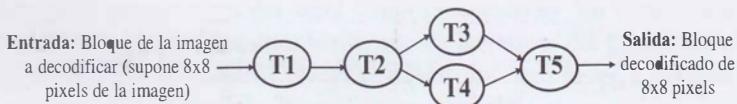


Figura 12. Tiempo de ejecución paralelo suponiendo que la expresión obtenida para el tiempo paralelo se puede aplicar a todo p . Los puntos válidos para este ejemplo son aquellos en los que p es potencia de 2 (representados con cuadrados). El mínimo, para el ejemplo, se alcanza en 8 y en 16.

Problema 9. Se va a paralelizar un decodificador JPEG en un multiprocesador. Se ha extraído para la aplicación el siguiente grafo de tareas que presenta una estructura segmentada (o de flujo de datos):



La tareas 1, 2 y 5 se ejecutan en un tiempo igual a t , mientras que las tareas 3 y 4 suponen $1,5t$. El decodificador JPEG aplica el grafo de tareas de la figura a bloques de la imagen, cada uno de 8×8 píxeles. Si se procesa una imagen que se puede dividir en n bloques de 8×8 píxeles, a cada uno de esos n bloques se aplica el grafo de tareas de la figura. Obtener la mayor ganancia en prestaciones que se puede conseguir paralelizando el decodificador JPEG en (suponga despreciable el tiempo de comunicación/sincronización): (a) 5 procesadores, y (b) 4 procesadores. En cualquier de los dos casos, la ganancia se tiene que calcular suponiendo que se procesa una imagen con un total de n bloques de 8×8 píxeles.

Solución

Para obtener la ganancia se tiene que calcular el tiempo de ejecución en secuencial para un tamaño del problema de n bloques, $T_s(n)$, y el tiempo de ejecución en paralelo para un tamaño del problema de n y los p procesadores indicados en (a) y (b), $T_p(p, n)$.

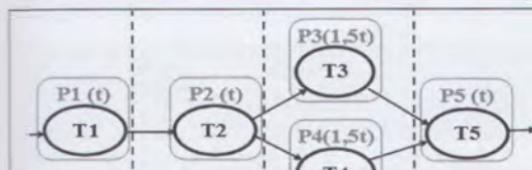
Tiempo de ejecución secuencial $T_s(n)$:

Un procesador tiene que ejecutar las 5 tareas para cada bloque de 8×8 píxeles. El tiempo que dedica el procesador a cada bloque es de $3t$ (tareas 1, 2 y 5) + $3t$ (tareas 2 y 4) = $6t$, luego para n bloques el tiempo de ejecución secuencial será el siguiente:

$$T_s = n \times (6t)$$

(a) Tiempo de ejecución paralelo y ganancia en prestaciones para 5 procesadores $T_p(5, n)$, $S(5, n)$.

Tabla 1. Asignación de tareas a procesadores (P1, P2, P3, P4 y P5), ocupación de las etapas del bucle para los primeros bloques procesados y tiempo de procesamiento del primer bloque y de los siguientes para 5 procesadores



Asignación de tareas a 5 procesadores P1, P2, P3, P4 y P5

Etapa 1 (t)	Etapa 2 (t)	Etapa 3 (1,5t)	Etapa 4 (t)	Terminado	Tiempo procesamiento
T1 (P1)	T2 (P2)	T3 (P3) y T4 (P4)	T5 (P5)		
0 - Bloque 1 - t					
1 - Bloque 2 - 2t	t - Bloque 1 - 2t				Procesamiento Bloque 1 (4,5t)
2 - Bloque 3 - 3t	2t - Bloque 2 - 3t	2t - Bloque 1 - 3,5t			
3 - Bloque 4 - 4t	3t - Bloque 3 - 4t	3,5t - Bloque 2 - 5t	3,5t - Bloque 1 - 4,5t		
4 - Bloque 5 - 5t	4t - Bloque 4 - 5t	5t - Bloque 3 - 6,5t	5t - Bloque 2 - 6t	Bloque 1	$t + t + 1,5t + t = 4,5t$
5 - Bloque 6 - 6t	5t - Bloque 5 - 6t	6,5t - Bloque 4 - 8t	6,5t - Bloque 3 - 7,5t	Bloque 2	$4,5t + 1,5t = 6t$
6 - Bloque 7 - 7t	6t - Bloque 6 - 7t	8t - Bloque 5 - 9,5t	8t - Bloque 4 - 9t	Bloque 3	$6t + 1,5t = 7,5t$
				Bloque 4	$7,5t + 1,5t = 9t$
T_entrada_etapa - Bloque x - T_salida_etapa					

Cada tarea se asigna a un procesador distinto, por tanto todas se pueden ejecutar en paralelo (ver asignación de tareas a procesadores en la Tabla 1). El tiempo de ejecución en paralelo en un cauce segmentado consta de dos tiempos: el tiempo que tarda en procesarse la primera entrada (ver celdas con fondo gris en la tabla) más el tiempo que tardan cada uno del resto de bloques (entradas al cauce) en terminar, una detrás de otra. Este último depende de la etapa más lenta, que en este caso es la etapa 3 (1,5t frente a t en el resto de etapas). El tiempo y la ganancia con la asignación de la Tabla 1 son:

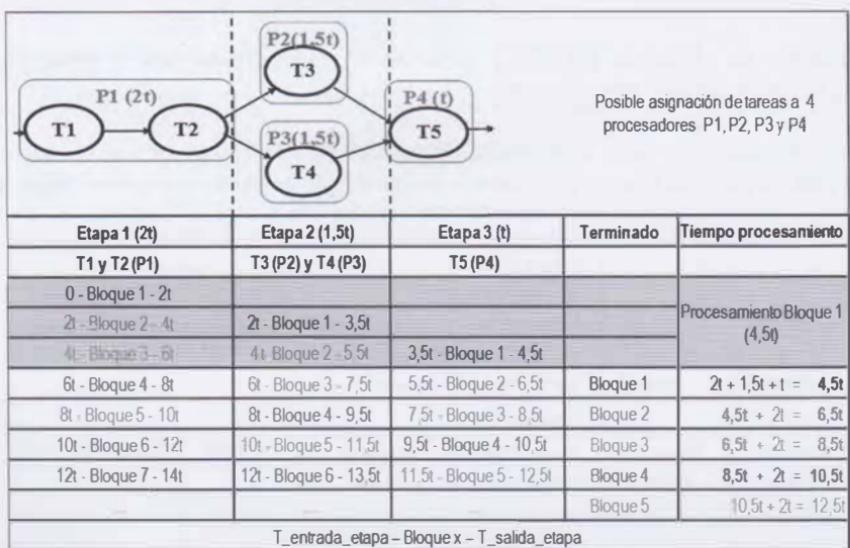
$$T_p(5, n) = 4,5t + (n - 1) \times 1,5t = 3t + n \times 1,5t$$

$$S(5, n) = \frac{T_s(n)}{T_p(5, n)} = \frac{n \times 6t}{3t + n \times 1,5t} \xrightarrow{n \rightarrow \infty} 4$$

La ganancia se aproxima a 4 para n suficientemente grande.

(b) Tiempo de ejecución paralelo y ganancia en prestaciones para 4 procesadores $T_p(4, n)$, $S(4, n)$.

Tabla 2. Asignación de tareas a procesadores (P1,P2,P3 y P4), ocupación de las etapas del cauce para los primeros bloques procesados y tiempo de procesamiento del primer bloque y de los siguientes para 4 procesadores



Se deben asignar las 5 tareas a los 4 procesadores de forma que se consiga el mejor tiempo de ejecución paralelo, es decir, el menor tiempo por bloque una vez procesado el primer bloque. Una opción que nos lleva al menor tiempo por bloque consiste en unir la Etapa 1 y 2 del cauce segmentado del caso (a) en una única etapa asignando T1 y T2 a un procesador (ver asignación de tareas a procesadores en la Tabla 2). Con esta asignación la etapa más lenta supone 2t (Etapa 1); habría además una etapa de 1,5t

(Etapa 2) y otra de t (Etapa 3). Si, por ejemplo, se hubieran agrupado T3 y T4 en un procesador la etapa más lenta supondría $3t$. El tiempo y la ganancia con la asignación de la tabla 2 son:

$$T_p(4, n) = 4,5t + (n - 1) \times 2t = 2,5t + n \times 2t$$

$$S(4, n) = \frac{T_s(n)}{T_p(4, n)} = \frac{n \times 6t}{2,5t + n \times 2t} \xrightarrow{n \rightarrow \infty} 3$$

La ganancia se aproxima a 3 para n suficientemente grande.

Problema 10. Se quiere implementar un programa paralelo para un multicomputador que calcule la siguiente expresión para cualquier x (es el polinomio de interpolación de Lagrange):

$$P(x) = \sum_{i=0}^n (b_i \cdot L_i(x)),$$

donde

$$L_i(x) = \frac{(x - a_0) \cdot \dots \cdot (x - a_{i-1}) \cdot (x - a_{i+1}) \cdot \dots \cdot (x - a_n)}{k_i} = \frac{\prod_{\substack{j=0 \\ j \neq i}}^n (x - a_j)}{k_i} \quad i = 0, 1, \dots, n$$

$$k_i = (a_i - a_0) \cdot \dots \cdot (a_i - a_{i-1}) \cdot (a_i - a_{i+1}) \cdot \dots \cdot (a_i - a_n) = \prod_{\substack{j=0 \\ j \neq i}}^n (a_i - a_j) \quad i = 0, 1, \dots, n$$

Inicialmente k_i , a_i y b_i se encuentra en el nodo i y x en todos los nodos. Sólo se van a usar funciones de comunicación colectivas. Indique cuál es el número mínimo de funciones colectivas que se pueden usar, cuáles serían, en qué orden se utilizarían y para qué se usan en cada caso.

Solución

Los pasos ((1) a (5)) del algoritmo para $n=3$ y un número de procesadores de $n+1$ serían los siguientes:

Pr.	Situación Inicial		(1) Resta paralela $A_i = (x - a_i)$ (($x - a_i$) se obtiene en P_i)		(2) Todos reducen A_i con resultado en B_i : $B_i = \prod_{i=0}^n A_i$	
P0	a_0	x	k_0	b_0	$(x - a_0)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$
P1	a_1	x	k_1	b_1	$(x - a_1)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$
P2	a_2	x	k_2	b_2	$(x - a_2)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$
P3	a_3	x	k_3	b_3	$(x - a_3)$	$(x - a_0) \cdot (x - a_1) \cdot (x - a_2) \cdot (x - a_3)$

Pr.	(3) Cálculo de todos los $L_i(x)$ en paralelo $L_i = B_i / (A_i \cdot k_i)$ ($L_i(x)$ se obtiene en P_i)	(4) Cálculo en paralelo $C_i = b_i \cdot L_i$ ($b_i \cdot L_i(x)$ se obtiene en P_i)	(5) Reducción del contenido de C_i con resultado en P_0 ($P(x)$ se obtiene en P_0)
P0	$(x-a_1) \cdot (x-a_2) \cdot (x-a_3) / k_0$	$b_0 \cdot (x-a_1) \cdot (x-a_2) \cdot (x-a_3) / k_0$	$P = \sum_{i=0}^n (C_i) = \sum_{i=0}^n (b_i \cdot L_i)$
P1	$(x-a_0) \cdot (x-a_2) \cdot (x-a_3) / k_1$	$b_1 \cdot (x-a_0) \cdot (x-a_2) \cdot (x-a_3) / k_1$	
P2	$(x-a_0) \cdot (x-a_1) \cdot (x-a_3) / k_2$	$b_2 \cdot (x-a_0) \cdot (x-a_1) \cdot (x-a_3) / k_2$	
P3	$(x-a_0) \cdot (x-a_1) \cdot (x-a_2) / k_3$	$b_3 \cdot (x-a_0) \cdot (x-a_1) \cdot (x-a_2) / k_3$	

Como se puede ver en el trazado del algoritmo para $n=3$ mostrado en las tablas, se usan un total de 2 funciones de comunicación colectivas (pasos (2) y (5) en la tabla). En el paso (2) del algoritmo se usa una operación de “todos reducen” para obtener en todos los procesadores los productos de todas las restas $(x-a_i)$. En el paso (5) y último se realiza una operación de reducción para obtener las sumas de todos los productos $(b_i \cdot L_i)$ en el proceso 0.

Problema 11. (a) Escribir un programa secuencial con notación algorítmica (se podría escribir en C) que determine si un número de entrada, x , es primo o no. El programa debe imprimir si el número es o no primo. Tendrá almacenados en un vector, NP, M números primos. Determine qué números primos se deberán almacenar si el máximo número de entrada x es MAX_INPUT. ¿De qué orden es el número de operaciones que hay que realizar en el código implementado? Justificar respuestas.

(b) Escribir una versión paralela del programa anterior para un multicomputador usando un estilo de programación paralela de paso de mensajes. El proceso 0 tiene inicialmente el número x y el vector NP en su memoria e imprimirá en pantalla el resultado. Considerar que la herramienta de programación ofrece funciones send() / receive() para implementar una comunicación uno-a-uno, en particular, con función send(buffer, count, datatype, idproc, group) no bloqueante y receive(buffer, count, datatype, idproc, group) bloqueante. Al ser la función receive() bloqueante, si no han llegado aún los datos cuando el flujo ejecuta la función, éste espera en la función hasta que se reciben, así se consigue la sincronización necesaria para que la comunicación se lleve a cabo. En las funciones send() / receive() se especifica:

- group: identificador del grupo de procesos que intervienen en la comunicación.
- idproc: identificador del proceso al que se envía o del que se recibe.
- buffer: dirección a partir de la cual se almacenan los datos que se envían o los datos que se reciben.
- datatype: tipo de los datos a enviar o recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- count: número de datos a transferir de tipo datatype.

¿Cuál es el orden de complejidad del código implementado? (considere sólo cálculos, no tenga en cuenta el orden de complejidad de las comunicaciones o de otro tipo de sobrecarga). Justificar respuesta.

Solución

(a) Programa secuencial que determina si x es o no primo.

Se van a implementar dos versiones, la segunda versión tiene un menor orden de complejidad.

Versión 1

Pre-condición

x : número de entrada $\{2, \dots, \text{MAX_INPUT}\}$. MAX_INPUT : máximo valor de la entrada

M : número de primos entre 2 y MAX_INPUT (ambos incluidos).

NP : vector con M componentes (los M º primos desde 2 hasta MAX_INPUT estarán ubicados desde $\text{NP}[0]$ hasta $\text{NP}[M-1]$)

Pos-condición: Imprime en pantalla si el número de entrada x es primo o no

Código

Versión 1

```
if (x>NP[M-1]) {
    print("%u supera el máximo primo a detectar %u \n", x, NP[M-1]);
    exit(1);
}

for (i=0; x>NP[i]; i++) {};

if (x==NP[i]) printf("%u ES primo\n", x);
else printf("%u NO es primo\n", x);
```

Orden de complejidad:

El número de iteraciones del bucle depende de en qué posición se encuentre el número primo x en NP . Teniendo en cuenta el teorema de los números primos:

$$\lim_{x \rightarrow \infty} \left(\frac{\pi(x)}{x/\ln x} \right)^n = 1 \quad (\pi(x) \text{ es el número de primos menores o iguales que } x)$$

el número de iteraciones y , por tanto, el orden de complejidad se podría aproximar por $O(x/\ln x)$.

Versión 2

Para implementar esta versión se ha tenido en cuenta que, si un número x no es primo, alguno de sus factores (sin contar el 1) debe ser menor o igual que \sqrt{x} . Entonces,

para encontrar si x es primo hay que dividir entre 2, 3, ..., $\lfloor \sqrt{x} \rfloor$. En realidad, bastaría hacer las divisiones entre los números primos menores o iguales que $\lfloor \sqrt{x} \rfloor$. Por ejemplo: para probar si 227 es primo sabiendo que $\sqrt{227} = 15.0665\ldots$ basta con ver si es divisible entre 2, 3, 5, 7, 11 y 13 (en este caso no lo es, 227 es primo).

Pre-condición

x: número de entrada {2,...,MAX_INPUT}. MAX_INPUT: máximo valor de la entrada
 M: número de primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$ (ambos incluidos)
 NP: vector con los M nº primos desde 2 hasta $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$
 xr: raíz cuadrada de x

Pos-condición: Imprime en pantalla si el número x es primo o no

Código

Versión 2

```

xr = sqrt(x); //sqrt(x) devuelve la raíz cuadrada de x
if (xr>NP[M-1]) {
    print("La raíz cuadrada de %u supera la del máximo primo a detectar
    (%u) \n", xr, NP[M-1]);
    exit(1);
}

for ( i=0 ; ((NP[i]<xr) && (x % NP[i])); i++) {} ; // % = módulo

if (x % NP[i]) printf("%u ES primo", x);
else printf("%u NO ES primo", x);
  
```

Orden de complejidad: El peor caso se obtiene cuando x es primo.

- Si el número es primo, el número de iteraciones del bucle depende de en qué posición se encuentra el número primo mayor que $\lfloor \sqrt{x} \rfloor$ en NP. Teniendo en cuenta el número de primos entre 2 y $\lfloor \sqrt{x} \rfloor$ según el teorema de los números primos, el orden de complejidad sería de $O(\lfloor \sqrt{x} \rfloor / L(\lfloor \sqrt{x} \rfloor))$.
 - Si el número no es primo, el bucle recorre el vector hasta que encuentra el primer primo que divide a x . Lo encontrará antes de llegar al número primo mayor que $\lfloor \sqrt{x} \rfloor$.
- (b)** Programa paralelo para multicomputador que determina si x es o no primo.

Se van a repartir las iteraciones del bucle entre los procesadores del grupo. Todos los flujos (procesos) ejecutan el mismo código. Se escribirá una versión paralela para cada versión secuencial del apartado (a).

Pre-condición

x: número de entrada {2,...,MAX_INPUT}; se supone que $x \leq \text{MAX_INPUT}$
 xr (versión 2): almacena la raíz cuadrada de x

M (versión 1): número de primos entre 2 y MAX_INPUT (ambos incluidos)

M (versión 2): número de primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$ (ambos incluidos)

NP (versión 1): vector con M+num_procesos componentes (los M nº primos desde 2 hasta MAX_INPUT estarán ubicados desde NP[0] hasta NP[M-1]; a NP[M]...NP[M+num_procesos-1] se asignará x+1 en el código)

NP (versión 2): vector con M+num_procesos componentes (los M nº primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$ estarán ubicados desde NP[0] hasta NP[M-1]; a NP[M]...NP[M+num_procesos-1] se asignará xr+1 en el código)

grupo: identificador del grupo de procesos que intervienen en la comunicación.

num_procesos: número de procesos en grupo.

idproc: identificador del proceso (dentro del grupo de num_procesos procesos) que ejecuta el código

tipo: tipo de los datos a enviar o recibir

b, baux: variables que podrán tomar dos valores 0 o 1

Pos-condición: Imprime en pantalla si el número x es primo o no

Código

Versión 1	Versión 2
<pre> // x es menor que MAX_INPUT //Difusión de x y NP if (idproc==0) for (i=1; i<num_procesos; i++) send(NP,M,tipo,i,grupo); send(x,1,tipo,i,grupo); } else { receive(NP,M,tipo,0,grupo); receive(x,1,tipo,0,grupo); } //Cálculo paralelo, asignación estática i=id_proc; NP[M+i]=x-1; while (x<NP[i]) do { i=i+num_procesos; } b=(x==NP[i])?1:0; //Comunicación resultados if (idproc==0) for (i=1; i<num_procesos; i++) { receive(baux,1,tipo,i,grupo); b = b baux; } else send(b,1,tipo,0,grupo); //Proceso 0 imprime resultado if (idproc==0) if (b) printf("%u ES primo", x); else printf("%u NO ES primo", x); </pre>	<pre> // x es menor que MAX_INPUT //Difusión de x y NP if (idproc==0) for (i=1; i<num_procesos; i++) send(NP,M,tipo,i,grupo); send(x,1,tipo,i,grupo); } else { receive(NP,M,tipo,0,grupo); receive(x,1,tipo,0,grupo); } //Cálculo paralelo, asignación estática i=id_proc; xr = sqrt(x); NP[M+i]=xr+1; while ((NP[i]<=xr)&&(x % NP[i])) do { i=i+num_procesos; } b=(NP[i]>xr)?1:0; //Comunicación resultados if (idproc==0) for (i=1; i<num_procesos; i++) { receive(baux,1,tipo,i,grupo); b = b && baux; } else send(b,1,tipo,0,grupo); // Proceso 0 imprime resultado if (idproc==0) if (b) printf("%u ES primo", x); else printf("%u NO ES primo", x); </pre>

Orden de complejidad Versión 1:

- Teniendo en cuenta el número de primos entre 2 y x , según el teorema de los números primos, y teniendo en cuenta el reparto de las iteraciones del bucle entre los procesos, el orden de complejidad se puede aproximar por $O(x)$ ($\text{num_procesos} \times Lx$). Se consigue una distribución equilibrada de las tareas a realizar repartiendo las iteraciones en turno rotatorio (*Round-Robin*). Con esta distribución, el proceso 0 ejecuta las iteraciones para i igual a 0, num_procesos , $2 \times \text{num_procesos}$, $3 \times \text{num_procesos}$, ...; el proceso $idproc$ para i igual a 1, $\text{num_procesos} + idproc$, $2 \times \text{num_procesos} + idproc$, $3 \times \text{num_procesos} + idproc$, ..., etc. Algunos procesos pueden ejecutar sólo una iteración más que otros.

Orden de complejidad Versión 2:

- Si el número es primo, el número de iteraciones del bucle depende de en qué posición se encuentra el número primo mayor que $\lfloor \sqrt{x} \rfloor$ en NP . Teniendo en cuenta el número de primos entre 2 y $\lfloor \sqrt{x} \rfloor$ y teniendo en cuenta el reparto de las iteraciones del bucle entre los procesos en turno rotatorio, el orden de complejidad se aproximaría por $O(\lfloor \sqrt{x} \rfloor / (\text{num_procesos} \times L \lfloor \sqrt{x} \rfloor))$.
- Si el número no es primo, algunos procesos recorren con el bucle el vector hasta que encuentran un primo que divide a x , el resto (no todos van a encontrar un factor de x) lo recorre hasta encontrar el número primo mayor que $\lfloor \sqrt{x} \rfloor$. $O(\lfloor \sqrt{x} \rfloor / (\text{num_procesos} \times L \lfloor \sqrt{x} \rfloor))$.

Problema 12. (a) Escribir una versión paralela del programa secuencial del ejercicio 11 suponiendo que la herramienta de programación ofrece las funciones colectivas de difusión y reducción. En la función de difusión, `broadcast(buffer, count, datatype, idproc, group)`, se especifica:

- `group`: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo reciben.
- `idproc`: identificador del proceso que envía.
- `buffer`: dirección de comienzo en memoria de los datos que difunde `idproc` y que almacenará, en todos los procesos del grupo, los datos difundidos.
- `datatype`: tipo de los datos a enviar/recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits ...).
- `count`: número de datos a transferir de tipo `datatype`.

En la función de reducción, `reduction(sendbuf, recvbuf, count, datatype, oper, idproc, group)`, se especifica:

- `group`: identificador del grupo de procesos que intervienen en la comunicación, todos los procesos del grupo envían.

- `idproc`: identificador del proceso que recibe.
- `recvbuf`: dirección en memoria a partir de la cual se almacena el escalar resultado de la reducción de todos los componentes de todos los vectores `sendbuf`.
- `sendbuf`: dirección en memoria a partir de la cual almacenan todos los procesos del grupo los datos de tipo `datatype` a reducir (uno o varios).
- `datatype`: tipo de los datos a enviar y recibir (entero de 32 bits, entero de 64 bits, flotante de 32 bits, flotante de 64 bits, ...).
- `oper`: tipo de operación de reducción. Puede tomar los valores `OR`, `AND`, `ADD`, `MUL`, `MIN`, `MAX`
- `count`: número de datos de tipo `datatype`, del buffer `sendbuffer` de cada proceso, que se van a reducir.

(b) ¿Qué estructura de procesos/tareas implementa el código paralelo del apartado anterior? Justifique su respuesta.

Solución

(a) Código paralelo

Pre-condición

`x`: número de entrada $\{1, \dots, \text{MAX_INPUT}\}$. Se cumple que `x` es menor o igual que `MAX_INPUT`

`xx`: almacena la raíz cuadrada de `x`.

`MAX_INPUT`: máximo valor de la entrada

`M`: número de primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$ (ambos incluidos)

`grupo`: identificador del grupo de procesos que intervienen en la comunicación

`idproc`: identificador del proceso

`tipo`: tipo de los datos a enviar o recibir

`num_procesos`: número de procesos en el grupo

`NP` (versión 2): vector con $M + \text{num_procesos}$ componentes (los M nº primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$ estarán ubicados desde `NP[0]` hasta `NP[M-1]`; a `NP[M]` ... `NP[M+num_procesos-1]` se asignará `xx+1` en el código)

`b`: variable que podrá tomar dos valores 0 o 1

Pos-condición:

Imprime en pantalla si el número `x` es primo o no

Código:

```

// x no es mayor que MAX_INPUT

//difusión del vector NP y de x
broadcast(NP, M, tipo, 0, grupo);
broadcast(x, 1, tipo, 0, grupo);

//Cálculo paralelo, asignación estática
xr = sqrt(x); i=id_proc; NP[M+i]=xr+1;
while ((NP[i]<=xr)&&(x % NP[i])) do {
    i=i+num_procesos;
}
b=(NP[i]>xr)?1:0;

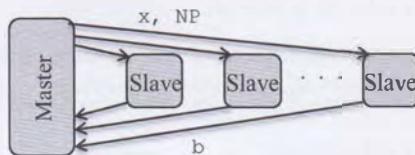
//Comunicación resultados
reduction(b, b, 1, tipo, AND, 0, grupo);

// Proceso 0 imprime resultado
if (idproc==0)
    if (b) printf("%u ES primo", x);
    else    printf("%u NO ES primo", x);

```

(b) Estructura de procesos/tareas

Se podría decir que tiene una estructura *Master-Slave* (ver figura más abajo), puesto que el proceso 0 se encarga de distribuir los datos (el trabajo) entre el resto de procesos del grupo y recolectar resultados para, combinándolos, obtener e imprimir el resultado definitivo. Además, se reparte el trabajo haciendo una descomposición de dominio, es decir, dividiendo NP en trozos y asignando el trabajo asociado a cada trozo a un proceso distinto. Los `num_procesos` esclavos realizan el mismo trabajo.



Problema 13. (a) Escribir una versión paralela del programa secuencial del Problema 11 para un multiprocesador usando el modelo de programación paralela de variables compartidas de OpenMP; en particular usar la directivas de trabajo compartido `for`, `sections/section` y/o `single` para distribuir las tareas entre los *threads* en lugar de realizar una asignación explícita como en los códigos realizados con paso de mensajes. (b) ¿Cuál es el orden de complejidad del código implementado? (considere sólo cálculos, no tenga en cuenta el orden de complejidad de las comunicaciones o de otro tipo de sobrecarga). Razonar respuestas.

Solución

(a) Se van a implementar dos versiones paralelas con OpenMP, teniendo en cuenta las dos versiones secuenciales del Problema 11.

Pre-condición

x: número de entrada {2,...,MAX_INPUT}. MAX_INPUT: máximo valor de la entrada
 xr (versión 2): almacena la raíz cuadrada de x
 M (versión 1): número de primos entre 2 y MAX_INPUT (ambos incluidos)
 M (versión 2): número de primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$ (ambos incluidos)
 NP (versión 1): vector con los M nº primos desde 2 hasta MAX_INPUT
 NP (versión 2): vector con los M nº primos entre 2 y $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor$
 b: variable que podrá tomar dos valores 0 o 1

Pos-condición: Imprime en pantalla si el número x es primo o no

Para poder usar la directiva `for` en un bucle debe ajustarse a la forma que se indica en las especificaciones de OpenMP (la última versión es la 4.5). Esta directiva no se puede aplicar a los bucles usados en los códigos de los ejercicios anteriores que encuentran si un número es o no primo. Los códigos de este ejercicio se han implementado con un bucle `for` al que se puede aplicar la directiva `for`.

Versión 1

```
if (x>NP[M-1]) {
    print("El número %u supera el máximo primo a detectar (%u) \n", x, NP[M-1]);
    exit(1);
}

b=0;
#pragma omp parallel for
for (i=0;i<M;i++) {
    if (NP[i]==x) b=1; //sólo un thread hace b=1
}

if (b) printf("%u ES primo", x);
else printf("%u NO es primo", x);
```

Versión 2

```
xr = sqrt(x); //sqrt(x) devuelve la raíz cuadrada de x
if (xr>NP[M-1]) {
    print("La raíz cuadrada de %u supera la del máximo primo a
detectar (%u) \n", xr, NP[M-1]);
    exit(1);
}

b=1;
#pragma omp parallel for reduction(&&:b)
for (i=0;i<M;i++) {
    b = b && (x % NP[i]);
}

if (b) printf("%u ES primo", x);
else printf("%u NO es primo", x);
```

(b) Versión 1: $O(\text{MAX_INPUT}/(\text{num_threads} \times \text{LMAX_INPUT}))$. Los threads se reparten las M iteraciones del bucle, como M es el número de primos entre 2 y MAX_INPUT , teniendo en cuenta el teorema de los números primos, M se aproxima por $\text{MAX_INPUT}/\text{LMAX_INPUT}$

Versión 2: $O(\lfloor \sqrt{\text{MAX_INPUT}} \rfloor / (\text{num_threads} \times \lfloor \sqrt{\text{MAX_INPUT}} \rfloor))$. Los threads se reparten las M iteraciones del bucle, como M es el número de primos entre 2 y $\sqrt{\text{MAX_INPUT}}$, teniendo en cuenta el teorema de los números primos, M se aproxima por $\lfloor \sqrt{\text{MAX_INPUT}} \rfloor / \lfloor \sqrt{\text{MAX_INPUT}} \rfloor$

6. Cuestiones

Cuestión 1. Indique las diferencias entre OpenMP y MPI.

Solución

OpenMP es un estándar de facto para la programación paralela con el estilo de variables compartidas mientras MPI es un estándar de facto para la programación con el estilo de paso de mensajes. OpenMP es una API basada en directivas del compilador y funciones, mientras que MPI es una API basada en funciones de biblioteca, que sitúa al programador en menor nivel de abstracción que OpenMP.

Cuestión 2. Ventajas e inconvenientes de una asignación estática de tareas a flujos (procesos/threads) frente a una asignación dinámica.

Solución

- Ventajas de la asignación estática frente a la dinámica:
 - La asignación estática requiere menos instrucciones extra que la dinámica.
 - Elimina la comunicación/sincronización necesaria para asignar tareas a flujos durante la ejecución.
- Inconvenientes de la asignación estática frente a la dinámica:
 - No se puede utilizar en aquellas aplicaciones en las que no hay un momento ni antes ni durante la ejecución en el que se sepa con seguridad el número de tareas en total que se deben ejecutar; las tareas a realizar van apareciendo en tiempo de ejecución y en distinto momento; es decir, no aparecen todas a la vez.
 - Difícil conseguir un equilibrado de la carga cuando la plataforma (hardware/software) es heterogénea (i.e. el tiempo de cálculo de los núcleos/procesadores no es igual) y/o no uniforme (i.e. el tiempo de comunicación depende de los nodos que se quieren comunicar, no es igual para todos los nodos de la plataforma)

- Difícil conseguir un equilibrado de la carga cuando las tareas a repartir entre flujos requieren distinto tiempo y, más aún, si no se sabe cuánto va a ser ese tiempo.

Cuestión 3. ¿Qué se entiende por escalabilidad lineal y por escalabilidad superlineal (Figura 5)? Indique las causas por las que se puede obtener una escalabilidad superlineal.

Solución

Para obtener una escalabilidad lineal se debe obtener una ganancia en prestaciones conforme se añaden recursos (por ejemplo, procesadores) igual al número de recursos utilizados. Se llama lineal porque la gráfica es una línea recta. Representa la escalabilidad ideal que se esperaría conseguir. Pero en algunos códigos paralelos se observa ganancias por encima de la lineal. Cuando esto ocurre se dice que la escalabilidad es superlineal.

La escalabilidad superlineal se puede deber al hardware y/o al código que se ejecuta. Al añadir un nuevo recurso (por ejemplo, un chip de procesamiento) realmente en la práctica se añaden varios recursos de distinto tipo (por ejemplo, en el caso de añadir un nuevo chip de procesamiento se añade, además de nuevos núcleos de procesamiento, más caché), si la aplicación se beneficia de más de uno de los tipos de recursos añadidos la ganancia puede resultar por encima de la lineal. También se puede explicar por la aplicación que se ejecuta; por ejemplo, hay aplicaciones que consisten en explorar una serie de posibilidades para encontrar una solución al problema. La exploración en paralelo puede hacer que se llegue antes a comprobar la posibilidad que lleva a una Solución

Cuestión 4. Enuncie la ley de Amdahl en el contexto de procesamiento paralelo.

Solución

La ganancia en prestaciones que se puede conseguir al añadir procesadores está limitada por la fracción del tiempo de ejecución secuencial que supone la parte del código no paralelizable.

Cuestión 5. Deduzca la expresión matemática que se suele utilizar para caracterizar la ganancia escalable. Defina claramente y sin ambigüedad el punto de partida que va a utilizar para deducir esta expresión y cada una de las etiquetas que utilice.

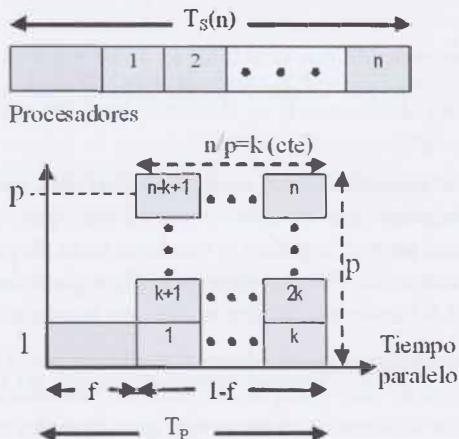
Solución

Punto de partida:

Se parte de un modelo de código en el que el tiempo de ejecución secuencial no permanece constante al variar el número de procesadores, lo que permanece constante es el tiempo de ejecución paralelo y en el que hay (como se representa en la figura de abajo):

- Un trozo no paralelizable (la fracción notada por f en la figura)

- Otro trozo (el resto) que se puede paralelizar repartiéndolo por igual entre los procesadores disponibles.



Las etiquetas de la figura se describen a continuación:

- T_p es una constante que representa el tiempo de ejecución paralelo que permanece constante conforme varía p ya que k es constante
- k es una constante igual a al grado de paralelismo del código secuencial dividido entre el número de procesadores (n/p). Cuando aumenta p aumenta también n para mantener constante k y T_p
- $T_s(n=kp)$ es el tiempo de ejecución secuencial que varía conforme varía p
- f es la fracción del tiempo de ejecución paralelo del código que supone la parte no paralelizable
- $(1-f)$ es la fracción del tiempo de ejecución paralelo del código que supone la ejecución de la parte paralelizable
- p representa el número de procesadores disponibles

Deducción:

La ganancia en prestaciones para este modelo de código ideal sería (se supone 0 el tiempo de sobrecarga):

$$S(p) = \frac{T_s(n = kp)}{T_p} = \frac{f \times T_p + (1-f) \times T_p \times p}{T_p} = f + (1-f) \times p$$

Según esta expresión la ganancia crece de forma lineal conforme varía p con una pendiente constante de $(1-f)$.

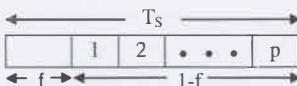
Cuestión 6. Deduzca la expresión que caracteriza a la ley de Amdahl. Defina claramente el punto de partida y todas las etiquetas que utilice.

Solución

Punto de partida:

Se parte de un modelo de código secuencial (como se representa en la figura de abajo) en el que se verifica lo siguiente:

- Tiene una parte no paralelizable (la fracción notada por f en la figura) que permanece constante aunque varíe el número de procesadores p disponibles. El resto se puede parallelizar con un grado de paralelismo ilimitado y, además, repartiéndolo por igual entre los p procesadores disponibles.
- El tiempo de ejecución secuencial permanece constante aunque varíe el número de procesadores disponibles p



Las etiquetas de la figura se describen a continuación:

- T_s constante que representa el tiempo de ejecución secuencial (es independiente de p)
- f constante que representa la fracción del tiempo de ejecución secuencial del código que supone la parte no paralelizable
- $(1-f)$ es la fracción del tiempo de ejecución secuencial del código que supone la ejecución de la parte paralelizable
- p variable que representa el número de procesadores disponibles

Deducción:

La ganancia en prestaciones para este modelo de código ideal sería (suponiendo 0 el tiempo de sobrecarga):

$$S(p) = \frac{T_s}{T_p(p)} = \frac{T_s}{f \times T_s + \frac{(1-f) \times T_s}{p}} = \frac{1}{f + \frac{1-f}{p}} = \frac{p}{fp + (1-f)} = \frac{p}{1 + f(p-1)}$$

En la práctica la ganancia en prestaciones será menor debido a que (1) el grado de paralelismo estará limitado, (2) puede ser difícil equilibrar la carga de trabajo y (3) la parallelización puede añadir un tiempo de sobrecarga (*overhead*) debido a la necesidad de comunicación/sincronización y a las operaciones extras que puede requerir la parallelización. Es decir:

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

7. Bibliografía

1. Gene M. Amdahl, 1967. “*Validity of the single processor approach to achieving large scale computing capabilities*”. In Proceedings of the April 18-20, 1967, spring joint computer conference (AFIPS '67 (Spring)). ACM, New York, NY, USA. <http://doi.acm.org/10.1145/1465482.1465560>
2. John L. Gustafson, 1988. “*Reevaluating Amdahl's law*”. Commun. ACM 31, 5 (May 1988), 532-533. <http://doi.acm.org/10.1145/42411.42415>
3. Julio Ortega Lopera, Mancia Anguita López y Alberto Prieto, 2005. “*Arquitectura de Computadores*”. Thomson.

CAPÍTULO 3. ARQUITECTURAS CON PARALELISMO A NIVEL DE THREAD (TLP)

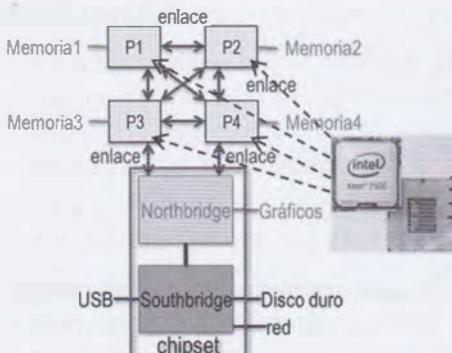
1. Introducción

Este capítulo se centra en arquitecturas que permiten ejecutar en paralelo o concurrentemente múltiples flujos de instrucciones o *thread* que comparten memoria, son arquitecturas con paralelismo a nivel de *thread* (*Thread-Level Parallelism*) con una única instancia del Sistema Operativo (SO). El SO se encarga de gestionar los flujos de instrucciones. Los modelos de programación paralela que siguen el paradigma de variables compartidas se implementan más fácilmente en este tipo de arquitecturas. Los modelos de programación de paso de mensajes, por su parte, están más próximas a la arquitecturas TLP con múltiples instancias del SO o multicamputadores. En arquitecturas TLP con una instancia del SO, debido a la compartición del sistema de memoria, aparecen conceptos propios no aplicables a multicamputadores, como son coherencia del sistema de memoria, consistencia del sistema de memoria y sincronización entre flujos de instrucciones. El sistema de memoria incluye en particular la memoria principal y todos los niveles de caché. Cuando se use arquitecturas TLP en este capítulo se estará refiriendo a arquitecturas con una instancia del SO. En este tema hay una sección dedicada a clasificar estas arquitecturas (Sección 2) y una sección dedicada a cada uno de los conceptos mencionados (Secciones 3, 4 y 5).

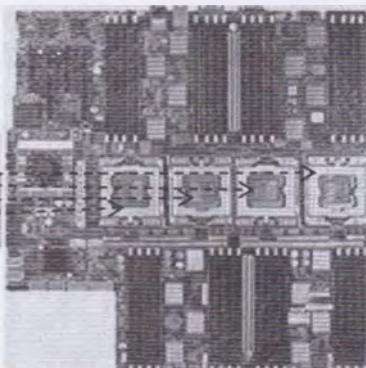
2. Arquitecturas TLP

Las arquitecturas TLP con una instancia del SO se pueden clasificar en (ver Figura 1):

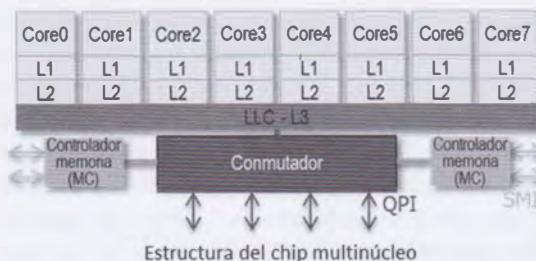
- Multiprocesadores: Ejecutan varios flujos de instrucciones en paralelo en un computador con varios núcleos/procesadores (cada flujo en un núcleo/procesador distinto). Se pueden encontrar multiprocesadores en un chip (multinúcleos), en una placa (Figura 1(a)), en un armario o formados por varios armarios.
- Multinúcleos (*multicores*): Ejecutan varios flujos de instrucciones en paralelo en un chip de procesamiento con múltiples núcleos, cada uno en un núcleo distinto. Un chip multinúcleo es un *multiprocesador en un chip* (Figura 1(b)). El primer multiprocesador en un chip comercializado fue el POWER4 de IBM. La denominación *multicores* se usa porque Intel denominó así a sus multiprocesadores en un chip, y denominó procesador a los chips o encapsulados de procesamiento y núcleos (*cores*) a los procesadores.



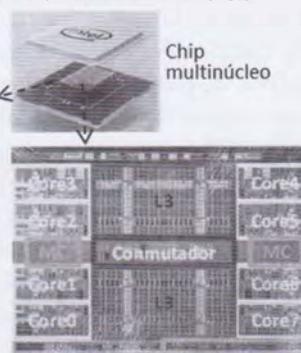
Estructura del multiprocesador NUMA en una placa



(a) Multiprocesador en una placa

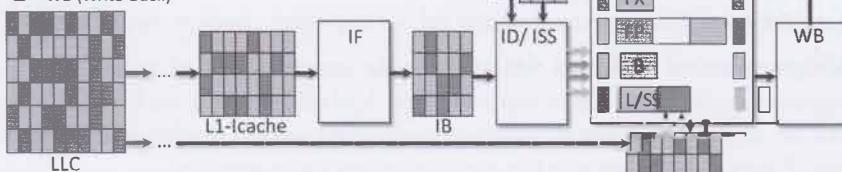


Estructura del chip multinúcleo



(b) Multinúcleo o Multiprocesador en un chip

- Etapas cache:
- IF (Instruction Fetch)
 - ID/ISS (Instruction Decoder/Issue)
 - Ex (Execution): FX (Fixed Point instr.), FP (Floating Point), L/S ((memory) Load/Store), B (Branch)
 - WB (Write-Back)



Memoria Cache: ■ LLC (Last Level Cache) ■ Icache (Instruction cache)

■ Dcache (Data cache)

Almacenamiento entre etapas: ■ IB (Instruction Buffer) ■ RF (Register File)

Estructura de un núcleo con multithread simultáneo (con dos threads o flujos de instrucciones, thread 0 y thread 1)

Figura 1. Arquitecturas TLP con una instancia del SO.

- Núcleos (cores) multithread: Núcleo de procesamiento (procesador) en el que se ha modificado su arquitectura ILP (Instruction Level Parallelism) para ejecutar flujos de instrucciones concurrentemente o en paralelo (Figura 1(c)).

(c) Núcleo con multithread simultáneo

En los primeros multiprocesadores (que aparecieron en los años 60) el acceso a memoria era uniforme, es decir, todos los procesadores tardaban lo mismo en acceder a todas las posiciones de memoria (UMA en Figura 2(a)). Para incrementar la escalabilidad, en los años 90, se distribuyeron físicamente los módulos de memoria compartida entre los procesadores (NUMA en Figura 2(b), ver también Figura 1(a)), permitiendo de esta forma configuraciones de cientos de procesadores manteniendo unas buenas prestaciones. En esta última configuración el acceso a memoria ya no es uniforme, los procesadores no tardan lo mismo en acceder a todas las posiciones de memoria, acceden más rápido a aquellas que se encuentran en los módulos que tienen más cercanos. Por ejemplo, P4 en la Figura 2(b) o en la Figura 1(a) tarda menos en acceder a las direcciones almacenadas en M4 que a aquellas almacenadas en M1, M2 o M3, mientras que P4 en Figura 2(a) tarda lo mismo en acceder a M1, M2, M3 o M4. En la Figura 1(b), todos los procesadores (núcleos) tardan lo mismo en acceder al último nivel de memoria caché (L3) del chip, es un UMA.

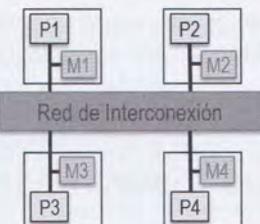
(a) UMA (*Uniform Memory Access*)(b) NUMA (*Non-Uniform Memory Access*)

Figura 2. Estructura de multiprocesadores UMA (a) y NUMA (b). M1, M2, M3 y M4 contienen cada uno un trozo del espacio de direcciones de memoria que puede direccionar todos los procesadores (P1, P2, P3 y P4). En (a) los nodos conectados a través de la red contienen procesadores o módulos de memoria y en (b) cada nodo contiene procesador y memoria.

3. Coherencia del sistema de memoria

La presencia de cachés en un computador conlleva que pueda haber varias copias de una misma dirección en el sistema de memoria de un computador; en particular, además de estar en memoria principal, puede haber una copia en la caché de un procesador (núcleo) y, en multiprocesadores, en la caché de varios procesadores. Si un procesador escribe en la copia de una dirección de memoria que tiene en su caché, se presenta una situación de incoherencia porque la copia de la dirección en esta caché no coincidiría con el contenido de la dirección en memoria principal, habría entonces dos copias de una dirección en el sistema de memoria con distinto contenido (Figura 3, Pk escribe en la copia de D que tiene en su caché). En este caso, habría una situación de incoherencia entre caché y memoria principal (memoria tiene D con 3 y la caché Ck con 4 en Figura 3). Si además, antes de que el procesador escriba, hay copia de esa dirección en otras cachés, también se

presentaría incoherencia entre cachés (Cj y Ck en la Figura 3 tienen distinto contenido en D). No se pueden admitir estas situaciones de incoherencia porque permitirían que dispositivos de E/S o procesadores pudieran acceder a valores no actualizados de una dirección en lugar de acceder a lo último que se ha escrito en la misma. Por ejemplo, debido a la incoherencia entre memoria principal y caché en la situación de la Figura 3, si un dispositivo de E/S o si el procesador Pi leen D, accederían al valor no actualizado de la memoria (obtendrían un valor de 3 en lugar de 4, que es lo último que se ha escrito en D) debido a la incoherencia entre caché y memoria, y si, por ejemplo, el procesador Pj accediera a D de nuevo, obtendría el valor no actualizado que hay en su caché debido a la incoherencia entre cachés (Cj y Ck). Se recuerda que las cachés y memoria principal se dividen en bloques de un tamaño igual a la unidad de transferencia entre caché y memoria. En la bibliografía se usa *marco de bloque* o *línea* de caché para denominar a cada una de estas divisiones en la caché, usando *bloque* para las subdivisiones en memoria principal. Cuando el contenido de una dirección no se encuentra en caché se copia en un marco de bloque de la misma todo el bloque donde se encuentra esta dirección. Dado que el tamaño de memoria principal es mayor que el tamaño de caché (hay más bloques en memoria que marcos de bloque en caché), se debe almacenar información que permita identificar el bloque de memoria que hay en un marco. Esta información se almacena en una tabla o directorio asociado a la caché. En este **directorío de caché** hay una entrada para cada marco de bloque que identifica el bloque de memoria con copia en ese marco e informa de su *estado* en la caché.

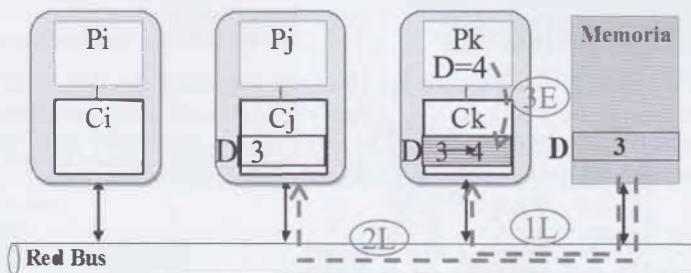


Figura 3. Incoherencia entre caché (Ck) y memoria principal, y entre cachés (Cj y Ck). Secuencia de accesos a memoria que han provocado estas incoherencias: 1L) primero Pk lee D de memoria (realmente se transfiere todo el bloque donde se encuentra la dirección D); 2L) en segundo lugar Pj lee D; 3E) en tercer lugar Pk escribe en D.

Las situaciones de incoherencia se deben abordar, bien no permitiendo que se produzcan o bien evitando que causen problemas en caso de permitirse. Actualmente las cachés implementan dos mecanismos de actualización de memoria principal:

- **Escritura directa (write-through).** Con escritura directa no se permiten situaciones de incoherencia entre caché y memoria principal porque, cada vez que se escribe en la copia de un bloque de memoria en una caché, se modifica también el bloque en memoria principal. En las aplicaciones ejecutadas en un computador, normalmente, si

se escribe en una dirección de memoria es muy probable que, tras un tiempo reducido, se escriba en direcciones próximas o en la misma dirección (*principios de localidad espacial y temporal*, respectivamente). Si es así sería más rentable escribir el bloque completo en memoria una vez realizadas las múltiples escrituras en la copia de caché.

- **Posescritura (write-back).** En una caché con posescritura se escribe en memoria principal un bloque modificado en una caché cuando se desaloja éste para hacer sitio a otro bloque de memoria al que se acaba de acceder. Así se pueden escribir a la vez en memoria principal múltiples modificaciones de un bloque, en lugar de ir las escribiendo una cada vez conforme se van generando. De esta forma también se evitan actualizaciones de memoria principal no necesarias (por no usarse inmediatamente por otros dispositivos o procesadores). Para que un controlador de caché sepa si la copia de un bloque se ha modificado o no en la caché y, por tanto, si se debe escribir en memoria principal cuando se desaloje, se añade un bit de estado en la entrada del directorio de caché de todos los marcos de bloque que informa de si la copia del bloque del marco es coherente o no con el bloque en memoria. El controlador de memoria activará (pondrá a 1) este bit de un marco cuando se modifique por primera vez la copia del bloque que contiene.

Lo usual es que las cachés usen posescritura. Si un computador usa cachés con posescritura se pueden producir situaciones de incoherencia entre caché y memoria principal. Para evitar problemas, se debe detectar cuándo un dispositivo o procesador accede a una posición de memoria modificada en la caché de otro dispositivo o procesador (el estado del bloque en caché informa si se ha modificado) y, en ese caso, se le debe suministrar la última actualización de la dirección (en realidad, de todo el bloque donde se encuentra esta dirección). Todo esto formaría parte del protocolo de mantenimiento de coherencia del sistema de memoria.

Aunque, como se acaba de comentar, se permiten situaciones de incoherencia entre cachés y memoria principal, no se admiten que haya incoherencia entre cachés. Para evitar estas situaciones de incoherencia entre cachés se deben cumplir dos condiciones:

- Condición 1 (**Propagación** de escrituras). Se debe garantizar que todo lo que se escribe en la copia de un bloque en una caché se propaga a las copias del bloque en otras cachés.
- Condición 2 (**Serialización** de escrituras). Se debe garantizar que las escrituras en una dirección se ven en el mismo orden por todos los procesadores; es decir, el sistema de memoria debe parecer que realiza en serie (una detrás de otra) las operaciones de escritura en la misma dirección. Debe dar la impresión de que son **atómicas**.

Se utilizan dos alternativas para propagar escrituras a otras cachés:

- **Escritura con actualización (write-update).** En este caso, cada vez que un procesador escribe en una dirección en su caché se escribe también en las copias de esa dirección en otras cachés. Si se cumplen los principios de localidad espacial y/o temporal, se puede generar un tráfico innecesario.

- **Escritura con invalidación** (*write-invalidate*). Con esta alternativa, antes de que un procesador modifique una dirección en su caché se invalidan las copias del bloque, que contiene la dirección, en otras cachés. Esta alternativa permite que se pueda seguir escribiendo, después de la primera escritura en el bloque, sin generar tráfico. Cuando eventualmente otro procesador acceda a este bloque, obtendrá el mismo de memoria (si está actualizada) o de una caché (actualizada). De esta forma se propaga la escritura. En este caso se necesita que el estado de un bloque en el directorio de caché pueda informar de si la copia del bloque en el marco se ha invalidado.

La propagación de las escrituras se puede realizar:

- Con una difusión de los paquetes de actualización o invalidación a todas las cachés (**difusión**).
- Con el envío de los paquetes de actualización o invalidación sólo a aquellas cachés con copia del bloque, que son las únicas que realmente necesitan recibirlas (**envío selectivo**).

La segunda alternativa requiere mantener una tabla o **directorío de memoria** que informe de las cachés que tienen copia de un bloque. Este directorio tendría una entrada para cada bloque con información de su estado en el sistema de memoria (en particular, en memoria principal) y con información de las cachés con copias del bloque.

En una estructura UMA se puede utilizar una red bus para las transferencias entre los procesadores (las cachés privadas de los procesadores) y la memoria que comparten (Figura 2(a), Figura 3). La red bus implementa la difusión de forma natural. Todos los paquetes enviados a través del bus se ven por todos los componentes conectados al bus (Condición 1 de coherencia). Además el orden en el que llegan las peticiones al bus establece un orden (serializa) los accesos, en particular, las escrituras (Condición 2 de coherencia). Todo esto hace que, en el caso de un multiprocesador UMA con red bus no sea necesario mantener información de las cachés con copias de los bloques y que incluso se pueda suprimir el directorio de memoria principal como se comenta más adelante.

3.1. Protocolos de mantenimiento de coherencia

Para diseñar o describir un protocolo de mantenimiento de coherencia se debe especificar:

- *Política de actualización* de memoria principal (escritura directa, posescritura).
- *Política de propagación* de escrituras de una caché a otras (escritura inmediata, escritura con invalidación)
- Comportamiento:
 - Los posibles *estados de un bloque en caché*. Las acciones (paquetes generados y cambios de estado) del controlador de caché ante eventos que reciba de un bloque dependerán del estado del bloque en caché.

- Los posibles *estados de un bloque en memoria principal* (o directorio). Las acciones del controlador de memoria principal ante eventos que reciba de un bloque dependerán de este estado del bloque. En realidad este estado también da información de la situación del bloque en cachés y, viceversa, el estado en caché también da información del estado en memoria principal.
- Los *paquetes que genera el controlador de caché* ante eventos (peticiones de lectura o escritura del procesador asociado a la caché, reemplazo del bloque o paquetes que llegan a través de la red) que reciba referentes a un bloque.
- Los *paquetes que genera el controlador de memoria principal* ante eventos (paquetes recibidos a través de la red) que reciba referentes a un bloque.
- *Relación de acciones con eventos y estados.*

En esta sección se van a describir cuatro protocolos de mantenimiento de coherencia, dos para multiprocesadores UMA con red bus (MSI y MESI) y dos para un multiprocesador NUMA en una placa (MSI con y sin difusión). Todos ellos usan posescritura, como política de actualización de memoria e invalidación, como política de propagación de escrituras a otras cachés. Las iniciales del nombre de los posibles estados de un bloque en caché forman parte del nombre del protocolo.

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

Los protocolos para buses se llaman de espionaje o sondeo porque todos los controladores, al estar conectados al bus, pueden espiar los accesos del resto sondeando el bus. MSI es el protocolo con menor número de estados que utiliza posescritura e invalidación. A continuación se completa la descripción del protocolo:

Estados de un bloque en caché

Son los tres estados que dan nombre al protocolo:

- **Modificado (M):** un bloque en este estado en caché es la única copia del bloque válida en todo el sistema de memoria. El controlador de caché debe responder con el bloque si le llega (i.e. ve en el bus) una petición con lectura del bloque, y deberá escribirlo en memoria si se reemplaza por otro bloque.
- **Compartido (C,S):** un bloque en este estado en caché es válido, también está válido en memoria y puede que haya copia válida en otras cachés. El controlador de caché invalida esta copia del bloque si le llega (i.e. ve en el bus) una petición de invalidación del bloque.
- **Inválido (I):** bien está físicamente en la caché pero en estado inválido (por propagación de invalidaciones desde otra caché) o bien no está físicamente. Si el procesador de la caché accede a una dirección de este bloque, se genera un fallo de caché y el controlador enviará un paquete de petición del bloque por la red (bus).

Estados de un bloque en memoria

En realidad se puede evitar almacenar esta información cuando se usa un bus. En caso de almacenarse, serían:

- **Válido:** el bloque está actualizado en la memoria principal y puede haber copia válida en una o varias cachés. El controlador de memoria responde con el bloque si ve en el bus una petición con lectura de un bloque que tiene en este estado.
- **Inválido:** el bloque no está actualizado en la memoria principal, hay una copia válida en una caché.

Paquetes que genera un controlador de caché

Son los paquetes que el controlador de caché puede enviar por el bus ante los eventos que se pueden producir en el procesador del nodo (lectura y escritura), en el controlador de caché (reemplazo de un bloque) o que llegan en forma de paquete desde otros nodos, ver Figura 4. La palabra nodo se usa aquí para denominar a componentes que se conectan entre sí a través de una red de interconexión. Un controlador de caché genera paquetes de petición (**Pt**) y de respuesta (**Rp**):

- Petición de lectura de un bloque (**PtLec(B)**): genera este paquete como consecuencia de una lectura con fallo de caché del procesador del nodo (evento **PrLec**), en este caso del bloque B. Como respuesta a esta petición recibirá un paquete de respuesta con el bloque (**RpBloque(B)**, Figura 4) de memoria o de la caché que lo tiene en estado Modificado, si la hubiera.
- Petición de acceso exclusivo a un bloque con lectura del bloque (**PtLecEx(B)**): lo genera como consecuencia de una escritura del procesador (evento **PrEsc**) en un bloque inválido en la caché. Como respuesta a esta petición se invalidarán las copias del bloque en otras cachés y recibirá un paquete de respuesta con el bloque (**RpBloque(B)**) de memoria o de la caché que lo tiene en estado Modificado, si lo hubiera.
- Petición de acceso exclusivo a un bloque (**PtEx(B)**): lo genera por escritura del procesador (evento **PrEsc**) en bloque en estado Compartido en la caché. Si el bloque estuviera válido pero en estado Modificado, no necesita pedir acceso exclusivo porque ya lo tiene (i.e. no hay copias del bloque en otras caches), por lo que no se generaría ningún paquete. Como respuesta a esta petición se invalidarán las copias del bloque en otras cachés. Hay implementaciones en las que no existe este paquete. En este caso se envía **PtLecEx(B)**, descartándose el paquete de respuesta con el bloque que genera la memoria principal.
- Petición de posescritura de un bloque (**PtPEsc(B)**): por el reemplazo de un bloque en estado Modificado.
- Respuesta con bloque (**RpBloque(B)**): lo genera cuando tiene en estado Modificado el bloque B solicitado por un paquete con petición de lectura de B (**PtLec(B)** o **PtLecEx(B)**) procedente del exterior (Figura 4). La memoria no tiene válido el bloque en este caso, por eso el controlador de caché debe responder con el bloque a la petición con lectura que observa en el bus.

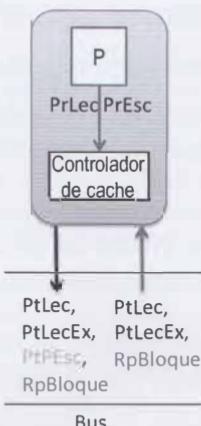


Figura 4. Paquetes generados por el controlador de caché de un nodo y eventos (paquetes) procedentes del exterior

Tabla 1. Protocolo MSI de espionaje. Acciones del controlador de caché de un nodo que provocan los eventos relacionados con un bloque B teniendo en cuenta el estado del bloque en la caché.

ESTADO ACTUAL de B	EVENTO	ACCIONES (del controlador de caché)	ESTADO SIGUIENTE de B
Modificado (M)	PrLec/ PrEsc(B)		Modificado
	PtLec(B)	- Genera paquete respuesta con B (RpBloque(B))	Compartido
	PtLecEx(B)	- Genera paquete respuesta con B (RpBloque(B)) - Invalida copia local	Inválido
	Reemplazo(B)	Genera paquete posescripción (PtPESc(B)), que escribe el bloque B en memoria	Inválido
Compartido (S)	PrLec(B)		Compartido
	PrEsc(B)	- Genera paquete PtLecEx(B) (ó PtEx(B), si lo hubiera), para invalidar las copias de B en otras cachés	Modificado
	PtLec(B)		Compartido
	PtLecEx(B)	Invalida copia local del bloque B	Inválido
Inválido (I)	PrLec(B)	- Genera paquete PtLec(B), que provoca la respuesta con el bloque B (RpBloque(B)) de memoria o de la caché con el bloque en estado Modificado, si la hubiera. - Recoge RpBloque(B).	Compartido
	PrEsc(B)	- Genera paquete PtLecEx(B), que provoca (1) la invalidación de las copias del bloque B en otras cachés y (2) la respuesta con el bloque (RpBloque(B)) de memoria o de la caché con el bloque en estado Modificado, si la hubiera. - Recoge RpBloque(B).	Modificado
	PtLec/ PtLecEx(B)		Inválido

Paquetes que genera el controlador de memoria

El controlador de memoria genera paquetes de respuesta (**Rp**):

- Respuesta con bloque (**RpBloque(B)**). Si se almacena el estado en memoria, el controlador genera este paquete cuando observa en el bus una petición de lectura de un bloque B (**PtLec(B)** o **PtLecEx(B)**) que tiene en estado Válido. Si no se almacena el estado de los bloques en memoria, siempre genera respuesta con el bloque cuando observa una petición de lectura. En este último caso, si un controlador de caché va a responder, éste inhibirá la respuesta de la memoria.

Tabla 2. Protocolo MESI de espionaje. Acciones del controlador de caché de un nodo que provocan los eventos relacionados con un bloque B teniendo en cuenta el estado del bloque en la caché.

ESTADO ACTUAL de B	EVENTO	ACCIONES (del controlador de caché)	ESTADO SIGUIENTE de B
Modificado (M)	PrLec/ PrEsc(B)		Modificado
	PtLec(B)	- Genera paquete respuesta con B (RpBloque(B))	Compartido
	PtLecEx(B)	- Genera paquete respuesta con B (RpBloque(B)) - Invalida copia local	Inválido
	Reemplazo(B)	- Genera paquete pionero/cifra (PtPEsc(B)), que escribe el bloque B en memoria	Inválido
Exclusivo (E)	PrLec		Exclusivo
	PrEsc		Modificado
	PtLec		Compartido
	PtLecEx	- Invalida copia local	Inválido
Compartido (S)	PrLec(B)		Compartido
	PrEsc(B)	- Genera paquete PtLecEx(B) (ó PtEx(B), si lo hubiera), para invalidar las copias de B en otras cachés	Modificado
	PtLec(B)		Compartido
	PtLecEx(B)	- Invalida copia local del bloque B	Inválido
Inválido (I)	PrLec(B) (hay cachés con copia de B)	- Genera paquete PtLec(B), que provoca la respuesta con el bloque B (RpBloque(B)) de memoria o de la caché con el bloque en estado Modificado, si la hubiera. - Recoge RpBloque(B).	Compartido
	PrLec(B) (no hay cachés con copia de B)	- Genera paquete PtLec(B), que provoca la respuesta con el bloque B (RpBloque(B)) de memoria. - Recoge RpBloque(B).	Exclusivo
	PrEsc(B)	- Genera paquete PtLecEx(B), que provoca (1) la invalidación de las copias del bloque B en otras cachés y (2) la respuesta con el bloque (RpBloque(B)) de memoria o de la caché con el bloque en estado Modificado, si la hubiera. - Recoge RpBloque(B).	Modificado
	PtLec/ PtLecEx(B)		Inválido

Relación de acciones con eventos recibidos y estados

En la Tabla 1 se pueden ver las acciones del controlador de caché de un nodo (paquetes generados y cambios de estado) que provocan los eventos relacionados con un bloque B teniendo en cuenta el estado del bloque en la caché (eventos del procesador o del controlador de caché del nodo, o recibidos del exterior a través de la red en forma de paquete, Figura 4).

3.1.2. Protocolo MESI (*Modified-Exclusive-Shared-Invalid*) de espionaje

Con el protocolo MSI siempre que se escribe en la copia de un bloque en una caché se genera un paquete de petición con acceso exclusivo al bloque, aunque no haya copias en otras cachés que se tengan que invalidar. Esto genera un tráfico innecesario, especialmente cuando se ejecutan aplicaciones secuenciales. Para evitar este tráfico, el protocolo MESI divide el estado S de MSI en dos estados: E (no hay copias en otras cachés) y S (hay al menos una copia en otra caché). Además del cambio mencionado en los estados en caché de un bloque, también hay cambios en las acciones generadas. En la Tabla 2 se muestran las acciones (se han destacado usando líneas más gruesas los cambios con respecto a la tabla para MSI).

3.1.3. Protocolos MSI (*Modified-Shared-Invalid*) para multiprocesadores NUMA

En esta sección se van a describir dos protocolos MSI con posescritura e invalidación para NUMA: uno que difunde las peticiones a todos los nodos y el otro que envía las peticiones sólo al nodo que tiene el bloque en el trozo de memoria principal más cercano al nodo (Figura 2(b)). Los protocolos son apropiados para un multiprocesador en una placa (Figura 1(a)). Al describir estos protocolos se distinguen tres tipos de nodos en la red NUMA:

- Nodo **solicitante** de un bloque (S). Es el nodo que genera una petición del bloque (PtLec, PtEx, PtLecEx o PtPEsc).
- Nodo **origen** de un bloque (O). En un multiprocesador NUMA la memoria está repartida físicamente entre los nodos, de forma que cada uno tiene en módulos de memoria más cercanos un trozo del espacio de direcciones total del multiprocesador (Figura 2(b)). Podríamos decir que esos módulos de memoria más cercanos se hospedan en el nodo. El nodo origen o *home* de un bloque es aquél que tiene el bloque en el trozo de memoria que hospeda.
- Nodo **propietario** de un bloque (P). Es un nodo que tiene copia del bloque en su caché.

3.1.3.1. Protocolo MSI para multiprocesadores NUMA sin difusión

Como no se usa difusión se necesita almacenar en el directorio de memoria principal, además del estado del bloque en memoria, información sobre las cachés con copia del bloque, para que las invalidaciones se puedan propagar sólo a los nodos con copia del

bloque. Se puede usar un vector de bits de presencia para almacenar esta información (Figura 5). Habría un bit para cada caché (cada nodo) conectada a la red NUMA. Un bit activo (a 1) significa que hay copia válida del bloque en esa caché. El directorio se divide en subdirectorios, uno por cada nodo de procesamiento del computador NUMA. Cada subdirectorio tiene una entrada para cada bloque del trozo de memoria principal que se hospeda en el nodo. El protocolo que se va a describir es el más sencillo que se puede implementar con directorio usando posescritura, invalidación y sin difusión. Todas las peticiones de un bloque se envían al nodo origen del bloque, que es quien tiene el subdirectorío con información sobre el bloque. El nodo origen propagará las invalidaciones a los nodos con copia de un bloque. Con esta propagación se cumplirá la Condición 1 de coherencia. El orden de llegada de las peticiones de un bloque al origen será el orden en el que los nodos van a ver los accesos a memoria de ese bloque. Todos van a ver el mismo orden en las escrituras, cumpliéndose entonces la Condición 2 necesaria para mantener coherencia entre cachés. A continuación se describe el protocolo.

Bloque	Estado Memoria	Bits de presencia para C0 C1 C2 C3			
•	•			•	
B-1	V	0	0	0	0
B	V	1	1	0	1
B+1	I	0	0	0	1
•	•			•	
•	•			•	
•	•			•	

Figura 5. Subdirectorío de memoria principal para un nodo de un computador NUMA con cuatro nodos con caché privada (cachés C0 ... C3). En el dibujo, el bloque B-1 está Válido en memoria compartida (V) y no tiene copia válida en cachés, el bloque B está Válido en memoria y hay copias válidas en la caché de los nodos 0,1 y 3. El bloque B+1 está Inválido (I) y la única copia válida está en la caché del nodo 3

Estados de un bloque en caché

Son los tres que dan nombre el protocolo que se comentaron en la Sección 3.1.1: Modificado (M), Compartido (C), Inválido (I).

Estados de un bloque en memoria

En el diseño más sencillo serían sólo dos los estados estables: Válido (en este caso puede haber copias válidas en una o en más cachés) e Inválido (hay una copia válida en una caché en estado Modificado).

Además de los estados estables puede haber estados “Pendiente” de un estado estable: Pendiente de válido o Pendiente de inválido. Un estado pendiente en un bloque indica que se está procesando un acceso a memoria del bloque. Hasta que no se pueda procesar una nueva petición sobre ese bloque su estado continuará en pendiente. A las peticiones

que se reciben de un bloque “pendiente” se responde con un paquete de reconocimiento negativo al solicitante para que intente de nuevo la petición más tarde.

Paquetes generados por controladores de caché o de memoria y otras acciones

En la Figura 6 se pueden ver las posibles transferencias a través de la red entre nodo Solicitante (S), Origen (O) y Propietario (P) en esta implementación sencilla. Como refleja esta figura, los accesos a memoria no son instantáneos, la latencia puede depender en el peor caso de cuatro transferencias en secuencia a través de la red. A continuación se dan más detalles de las circunstancias en las que se generan estos paquetes y de otras acciones relacionadas (se pueden leer también en secuencia todos los párrafos a) -1a), 2a), 3a) y 4a), ver Figura 6-, todos los b) y todos los c)):

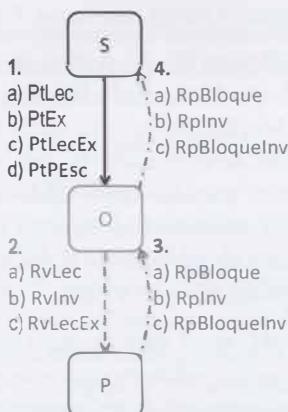


Figura 6. Paquetes generados por el controlador de caché en un protocolo de coherencia para NUMA sin difusión

- Paquete de petición, **Pt**, de

1. Nodo S (solicitante) a O (origen del bloque), cuando S y O no son el mismo nodo:

a) Lectura del bloque, **PtLec(B)**, en caso de producirse una lectura del procesador del nodo solicitante ($PrLec(B)$) con fallo de caché (i.e. no hay copia válida del bloque B en la caché privada del nodo). Cuando reciba la respuesta con el bloque $RpBloque(B)$ (4a)), introducirá el bloque en la caché en estado Compartido.

b) Petición de acceso exclusivo sin lectura, **PtEx(B)**, en caso de producirse una escritura del procesador del nodo solicitante ($PrEsc(B)$) en un bloque B que tiene en su caché en estado Compartido. En este caso no necesita pedir el bloque (su lectura) porque lo tiene válido. Cuando reciba la respuesta confirmando la invalidación en memoria y en otras cachés $RpInv(B)$ (4b)), modificará el bloque y cambiará su estado a Modificado.

- c) Lectura con acceso exclusivo, **PtLecEx(B)**, en caso de producirse una escritura del procesador del nodo solicitante ($PrEsc(B)$) en un bloque B del que no tiene copia válida en su caché. Cuando reciba la respuesta con el bloque confirmando su invalidación en memoria y en otras cachés $RpBloqueInv(B)$ (4c)), escribirá el bloque en memoria, lo modificará y pondrá su estado en el directorio de caché a Modificado.
 - d) Posescritura, **PtPEsc(B)**, en caso de que el controlador de caché reemplace un bloque B que se encuentra en estado Modificado en la caché. El bloque dejará de estar físicamente en la caché.
- Paquete de reenvío, **Rv**, de petición de
2. Nodo \circ (origen del bloque) a nodos con copia, P (propietarios):
- a) Lectura del bloque, **RvLec(B)**, en el caso de que \circ reciba $PtLec(B)$ de S (ver 1a)), tenga el bloque B en estado Inválido en memoria principal (estarán entonces en estado Modificado en una caché) y esté la única copia válida del bloque en la caché de otro nodo (si está en el nodo origen se resuelve localmente sin re-enviar paquete a otro nodo de la red). El nodo \circ pone el bloque en el directorio en estado Pendiente de válido hasta que se actualice el bloque en memoria una vez recibido de la única caché con copia válida (3a)). Entonces podrá responder al nodo S (4a)).
 - b) Invalidación, **RvInv(B)**, en el caso de que \circ reciba: (1) $PtEx(B)$ de S (ver 1b)) y el bloque B estuviera válido en cachés de otros nodos, ó (2) $PtLecEx(B)$ (1c)) y el bloque B estuviera válido en memoria y en la caché de otros nodos (consultando el directorio de memoria sabe qué cachés tienen copia). Antes de generar los paquetes de reenvío de invalidación, se pone el estado de B en el directorio a Pendiente de inválido para que no sean atendidas las peticiones que se reciban del bloque hasta que las copias del mismo en caché estén invalidadas. Cuando \circ reciba confirmación de las invalidaciones (3b)): (1) dejará activo sólo el bit de presencia de S en el directorio, (2) pondrá el estado de B en memoria a Inválido, (3) responderá a S confirmando su invalidación en memoria y en otras cachés, si recibió $PtLecEx(B)$ en lugar de $PtEx(B)$ incluirá en esta respuesta el propio bloque.
 - c) Lectura con invalidación, **RevLecEx(B)**, en el caso de que \circ reciba $PtLecEx(B)$ de S (ver 1c)), tenga el bloque B en estado Inválido en memoria y esté la única copia válida del bloque en la caché de otro nodo (si está en el origen se resuelve localmente sin reenvío a otro nodo). El nodo \circ deja el estado del bloque en Pendiente de inválido. Cuando \circ reciba la confirmación de la invalidación con el bloque (3c)): (1) dejará activo sólo el bit de presencia de S en el directorio, (2) pondrá el estado de B en memoria a Inválido y (3) responderá a S con el bloque confirmando su invalidación (4c)).

- Paquete de respuesta, **Rp**, de

3. Nodo P (propietarios) a O (origen del bloque):

- Respuesta con bloque, **RpBloque(B)**, en caso de que P reciba RvLec(B) (ver 2a)). Antes de responder, P cambia el estado del bloque B en su caché de Modificado a Compartido. Cuando O recibe el paquete: (1) escribe el bloque en memoria principal, (2) pasa el estado del bloque de Pendiente de válido a Válido y (3) responde con el bloque al nodo S (4a)).
- Respuesta confirmando invalidación, **RpInv(B)**, en el caso de que P hubiera recibido RvInv(B) (2b)). Antes de responder, P invalida la copia local del bloque B en su caché.
- Respuesta con bloque confirmando invalidación, **RpBloqueInv(B)**), en el caso de que P hubiera recibido RvLecEx(B) (2c)). Antes de responder, P invalida la copia local del bloque B en su caché.

4. Nodo O (origen del bloque) a S (solicitante), S y O no son el mismo nodo:

- Respuesta con bloque, **RpBloque(B)**, en caso de que O reciba del nodo S PtLec(B) (ver 1a)) y después de asegurarse: (1) de que tiene copia Válida del bloque en memoria y (2) de activar el bit de presencia de S. Si tiene el bloque en estado Inválido, antes de responder envía RvLec(B) a la única caché con copia válida (2a)), pone el estado a Pendiente de válido y espera a recibir la respuesta con el bloque (3a)) de la única caché propietaria. Una vez recibida la respuesta y antes de responder a S, el origen O: (1) actualiza la memoria, (2) activa el bit de presencia de S en el directorio y (3) pasa el estado a Válido en memoria.
- Respuesta confirmando invalidación, **RpInv(B)**, en caso de que O reciba de S PtEx(B) (1b)) y una vez que O: (1) se asegure que no hay copias en otras cachés (si las hay debe esperar a que se invaliden, ver 2b) y 3b)), (2) deje activo sólo el bit de presencia de S y (3) pase a Inválido la copia del bloque en memoria.
- Respuesta con bloque confirmando invalidación, **RpBloqueInv(B)**, en caso de que O reciba de S PtLecEx(B) (1c)) y una vez que O: (1) se asegure que no hay copias en otras cachés (si las hay debe esperar a que se invaliden, ver 2b) y 3b), 2c) y 3c)), (2) deje activo sólo el bit de presencia de S y (3) pase a Inválido la copia del bloque en memoria. En el caso de que, al recibir la petición de S, haya copias en cachés, espera hasta que recibe: la respuesta confirmación invalidación RpInv(B) a los reenvíos de invalidación RvInv(B) en caso de que el bloque esté Válido en memoria, ó la respuesta con bloque confirmando invalidación RpBloqueInv(B) en respuesta al reenvío de lectura e invalidación RvLecEx(B) en caso de que esté Inválido en memoria.

El nodo origen procesa las escrituras en una dirección secuencialmente (una detrás de otra) en el orden en el que llegan. Esto garantiza que todos los nodos vean este orden en las escrituras (Condición 2 de coherencia). La implementación también garantiza la Condición

2 de consistencia secuencial (Sección 4). El motivo es que el nodo origen no atiende a una lectura de un bloque que llega después de una escritura en el mismo bloque hasta que están invalidadas las copias del bloque en cachés (como consecuencia de la escritura), no hay posibilidad entonces de que un nodo pueda ver lo que otro ha escrito antes de que esa escritura la puedan ver el resto de nodos. Un computador con esta implementación podría garantizar también la Condición 1 de consistencia secuencial si un procesador no usa el valor de una lectura ni comienza una nueva escritura hasta haber recibido la respuesta del nodo origen de los accesos que le preceden en el código que ejecuta.

Se puede implementar protocolos con menor latencia permitiendo que un nodo propietario conteste directamente al nodo solicitante en lugar de hacerlo a través del nodo origen (consultar, por ejemplo, el protocolo basado en directorios descrito en J. Ortega, M. Anguita, 2005).

Relación de acciones con eventos recibidos y estados

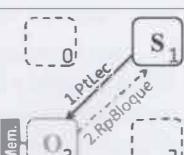
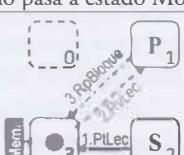
En este apartado se van a ver las acciones que se generan (paquetes generados entre nodos y cambios en el directorio y en caché) teniendo en cuenta el estado (en caché y en memoria) del bloque accedido. Se van a describir varias situaciones representativas usando un ejemplo. A partir de estos casos se pueden deducir, teniendo en cuenta los apartados anteriores, las acciones del resto de situaciones posibles. Se va a suponer que se tiene un multiprocesador en una placa como el de la Figura 1(a) con 4 nodos/procesadores (N0 a N3) y que se generan la siguiente secuencia de accesos a una dirección de memoria D, con nodo origen N3, que inicialmente no se encuentra en ninguna caché:

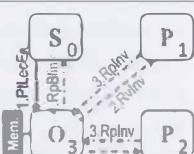
1. Lectura generada por el procesador del nodo 1
2. Escritura generada por el procesador del nodo 1
3. Lectura generada por el procesador del nodo 2
4. Lectura generada por el procesador del nodo 3
5. Escritura generada por el procesador del nodo 0
6. Escritura generada por el procesador del nodo 1

La entrada en el directorio de cada bloque tendrá 4 bits de presencia, uno por cada nodo. S, P y M notan nodo Solicitante, nodo Propietario y Memoria, respectivamente. Suponemos que BD es el bloque al que pertenece la dirección D. Como D no se encuentra inicialmente en ninguna caché, la entrada del directorio para ese bloque tendrá al inicio el bloque en estado Válido y todos los bits de presencia desactivos (a 0):

Bloque	Estado Memoria	Bits de presencia para			
		C0	C1	C2	C3
BD	V	0	0	0	0
•	•	•	•	•	•
•	•	•	•	•	•
•	•	•	•	•	•

En la siguiente tabla se resumen las acciones para la secuencia de eventos:

ESTADO INICIAL	EVENTO	ACCIONES	ESTADO SIGUIENTE
N0) Inválido S-N1) Inválido N2) Inválido N3) Inválido M-N3) Válido Entrada Directorio de memoria para el bloque: M C0 C1 C2 C3 V 0 0 0 0	P1 lee D	 <ol style="list-style-type: none"> N1 envía petición de lectura del bloque BD (PtLec(BD)) a N3 N3 recibe PtLec(BD). Como tiene el bloque BD en estado Válido: (1) pone el bit de N1 en el directorio a 1 y (2) envía paquete de respuesta con el bloque a N1 (RpBloque(BD)). N1 recibe RpBloque(BD) e introduce el bloque en caché en estado Compartido 	N0) Inválido S-N1) Compartido N2) Inválido N3) Inválido M-N3) Válido M C0 C1 C2 C3 V 1 0 0 0
N0) Inválido S-N1) Compartido N2) Inválido N3) Inválido M-N3) Válido M C0 C1 C2 C3 V 0 1 0 0	P1 escribe en D	 <ol style="list-style-type: none"> N1 envía petición de acceso exclusivo para BD (PtEx(BD)) a N3. N3, cuando recibe PtEx(BD): (1) pasa BD a estado Inválido y (2) envía paquete de respuesta a N1 confirmando invalidación (RpInv(BD)). N1, recibida la respuesta, modifica el bloque y lo pasa a estado Modificado. 	N0) Inválido S-N1) Modificado N2) Inválido N3) Inválido M-N3) Inválido M C0 C1 C2 C3 I 0 1 0 0
N0) Inválido N1) Modificado S-N2) Inválido N3) Inválido M-N3) Inválido M C0 C1 C2 C3 I 0 1 0 0	P3 lee D	 <ol style="list-style-type: none"> N2 envía PtLec(BD) a N3 porque no tiene BD. N3, como tiene BD en estado Inválido, (1) pone en la entrada del bloque en el directorio estado Pendiente de válido, y (2) reenvía (RvLec(BD)) la petición al 	N0) Inválido N1) Compartido S-N2) Compartido N3) Inválido M-N3) Válido M C0 C1 C2 C3 V 0 1 1 0

ESTADO INICIAL	EVENTO	ACCIONES	ESTADO SIGUIENTE
		<p>nodo N1 (que según el directorio tiene copia válida del bloque)</p> <p>3. N1, cuando recibe RvLec(BD): (1) pasa BD en su caché a Compartido, y (2) envía a N3 un paquete de respuesta con el bloque (RpBloque(BD)).</p> <p>4. N3, cuando recibe la respuesta de N1 (RpBloque(BD)): (1) activa el bit de N2 en el directorio, (2) escribe BD en memoria y pone el estado del bloque en el directorio a Válido, y (3) responde con el bloque a N2 (RpBloque(BD))</p> <p>5. N2, cuando recibe RpBloque(BD), introduce el bloque en su caché en estado Compartido</p>	
N0) Inválido N1) Compartido N2) Compartido S-N3) Inválido M-N3) Válido M C0 C1 C2 C3 [V] 0 1 1 0	P3 lee D	 <p>N3 lee BD de su propia memoria, lo introduce en su caché en estado Compartido y activa el bit de presencia de C3 en la entrada de BD del directorio.</p>	N0) Inválido N1) Compartido N2) Compartido S-N3) Compartido M-N3) Válido M C0 C1 C2 C3 [V] 0 1 1 1
S-N0) Inválido N1) Compartido N2) Compartido N3) Compartido M-N3) Válido M C0 C1 C2 C3 [V] 0 1 1 1	P0 escribe en D	 <p>1. N0 envía a N3 petición de acceso exclusivo a BD con lectura PtLecEx(BD).</p> <p>2. N3 recibe PtLecEx(BD) y, como tiene el bloque en estado Válido: (1) pone en el directorio el estado de BD en Pendiente de inválido y (2) envía los paquetes RvInv(BD) a los nodos que, según el directorio, tienen copia del bloque, e invalida también la copia de BD que tiene en su caché local.</p> <p>3. N1, N2, cuando reciben RvInv(BD): (1) invalidan su copia de BD y (2) responden</p>	S-N0) Modificado N1) Inválido N2) Inválido N3) Inválido M-N3) Inválido M C0 C1 C2 C3 [I] 1 0 0 0

ESTADO INICIAL	EVENTO	ACCIONES	ESTADO SIGUIENTE											
		<p>a N3 confirmando la invalidación RpInv(BD).</p> <p>4. N3, cuando recibe todas las RpInv(BD), incluido la confirmación de su caché local: (1) Deja activo sólo el bit de N0 en el directorio, (2) pone el estado de BD en el directorio a Inválido, y (3) envía respuesta con el bloque a N0 confirmando invalidación RpBloqueInv (espera a todas las invalidaciones para garantizar un orden en los accesos a BD, necesario para garantizar coherencia).</p> <p>5. N0 recibe RpBloqueInv(BD) de N3, escribe BD en su caché, modifica BD y lo pone en estado Modificado</p>												
N0) Modificado S-N1) Inválido N2) Inválido N3) Inválido M-N3) Inválido M C0 C1 C2 C3 <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>I</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table>	I	1	0	0	0	0	P1 escribe en D	 <p>1. N1 envía a N3 petición de lectura de BD con acceso exclusivo PtLecEx(BD)</p> <p>2. N3 recibe PtLecEx(BD) y, como tiene el bloque en estado Inválido: (1) pone en el directorio el estado de BD en Pendiente de inválido y (2) envía RvLecInv(BD) al nodo que, según el directorio, tienen la única copia del bloque, N0.</p> <p>3. N0, cuando reciben RvLecInv(BD): (1) invalidan su copia de BD y (2) responden a N3 con el bloque confirmando la invalidación RpBloqueInv(BD).</p> <p>4. N3, cuando recibe la RpBloqueInv(BD) de N0: (1) Deja activo sólo el bit de N1 en el directorio, (2) pone el estado de BD en el directorio a Inválido, y (3) envía respuesta con el bloque a N1 confirmando invalidación RpBloqueInv(BD).</p> <p>5. N1 recibe RpBloqueInv(BD) de N3, escribe BD en su caché, modifica BD y lo pone en estado Modificado</p>	N0) Inválido S-N1) Modificado N2) Inválido N3) Inválido M-N3) Inválido M C0 C1 C2 C3 <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>I</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> </table>	I	0	1	0	0
I	1	0	0	0	0									
I	0	1	0	0										

3.1.3.2. Protocolo MSI para multiprocesadores NUMA con difusión

Al usar difusión, se envían los paquetes de petición a todas las cachés tengan o no copia del bloque. Por tanto, no se requiere almacenar en el directorio de memoria principal información sobre las cachés con copia del bloque. El protocolo que se va a describir es el más sencillo que se puede implementar con posescritura e invalidación. Los estados de un bloque en caché y en memoria coinciden con los que tiene una implementación sin difusión.

Paquetes generados por controladores de caché o de memoria y otras acciones

En la Figura 7 se pueden ver las posibles transferencias entre nodos. Los paquetes de petición se difunden a todos los nodos, sean o no propietarios (P). Comparado con una alternativa sin difusión, esta implementación aumenta el número de paquetes que se envían a través de la red, pero disminuye la latencia de los accesos a memoria porque se eliminan los paquetes de reenvío de petición desde el nodo origen a nodos propietarios. Los nodos propietarios no necesitan el reenvío de paquetes de petición porque reciben las peticiones directamente del nodo solicitante. Usando difusión se requiere como máximo una secuencia de tres paquetes para completar un acceso a memoria (Figura 7) mientras que con el protocolo sin difusión se necesita un máximo de cuatro (Figura 6). A continuación se comentan los paquetes usados en este protocolo:

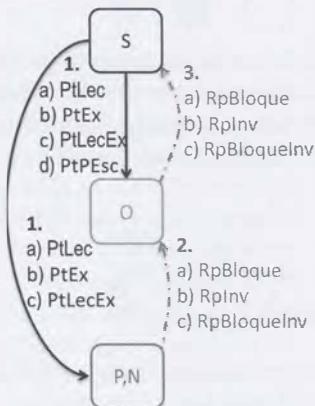


Figura 7. Paquetes generados por el controlador de caché en un protocolo de coherencia para NUMA con difusión

- Paquete de petición (**Pt**) de
 1. Nodo S (solicitante) a todos los nodos N, lo recibe el nodo O (origen del bloque), los nodos P (propietarios) y el resto de nodos. Son los mismos paquetes que envía el nodo S al nodo O en el protocolo sin difusión y se generan como consecuencia de los mismos eventos.
- Respuesta (**Rp**) de

2. Nodos N a O (origen del bloque):

- Respuesta con bloque de un nodo P, **RpBloque(B)**, en caso de que P reciba PtLec(B) (1a)) de un bloque B que tiene en estado Modificado (en memoria estará Inválido). P cambia el estado del bloque B en su caché a Compartido. Cuando O reciba el paquete: (1) escribirá el bloque en memoria principal, (2) pasará el estado del bloque a Válido y (3) responderá con el bloque al nodo S (3a)). El resto de nodos no hacen nada.
- Respuesta de N confirmando invalidación, **RpInv(B)**, en el caso de que el nodo reciba (1) PtEx(B) (1b)), o (2) PtLecEx(B) (1c)) y tenga el bloque en Compartido o Inválido. Los nodos con copia en estado Compartido primero invalidan la copia local del bloque B en su caché y entonces es cuando responde a O confirmando invalidación (3b)).
- Respuesta con bloque confirmando invalidación, **RpBloqueInv(B)**, en el caso de que un nodo reciba PtLecEx(B) (1c)) de un bloque que tiene en estado Modificado. Primero invalida la copia local del bloque B en su caché y entonces responde a O con el bloque confirmando invalidación (3c)).

3. Nodo O (origen del bloque) a S (solicitante):

- Respuesta con bloque, **RpBloque(B)**, en caso de que O reciba del nodo S una petición de lectura de un bloque que tiene en estado Válido o, en el caso de tener el bloque en estado Inválido, una vez que haya recibido la respuesta con el bloque (2a)) de la caché que tiene la única copia válida. En este último caso, el nodo O pone el estado del bloque en Pendiente de válido y, cuando recibe la respuesta, lo pasa a Válido.
- Respuesta confirmando invalidación, **RpInv(B)**, en caso de que O reciba una petición de acceso exclusivo a un bloque (1b)) de un nodo S y una vez que O invalide la copia del bloque en memoria. Invalidará la copia en memoria cuando haya recibido las respuestas confirmando invalidación de todos los nodos N (2b)) y de su caché local, mientras no reciba estas respuestas el estado del bloque en memoria será un estado Pendiente de inválido.
- Respuesta con bloque confirmando invalidación, **RpBloqueInv(B)**, en caso de que O reciba de S una petición de acceso exclusivo a un bloque con lectura del bloque (1c)) y una vez que ha recibido las confirmaciones de invalidación de todas las cachés (2b) y 2c)), mientras recibe estas respuestas estará en estado Pendiente de inválido. Si el estado en O del bloque es Inválido cuando recibe la petición de S, O recibirá el bloque actualizado, que tiene que enviar a S, en la respuesta de una de las cachés (2c)).

Relación de acciones con eventos recibidos y estados

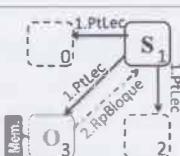
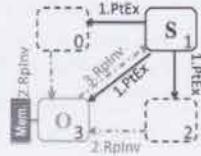
En este apartado se van a ver las acciones que se generan (paquetes generados entre nodos y cambios de estado en memoria y en caché) teniendo en cuenta el estado (en

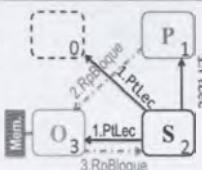
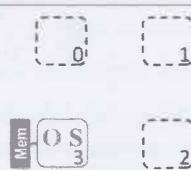
caché y en memoria) del bloque accedido. Se van a describir las mismas situaciones que la Sección 3.1.3.1. Se va a suponer, como en la sección mencionada, que se tiene un multiprocesador en una placa como el de la Figura 1(a) con 4 nodos/procesadores (N0 a N3) y que se generan la siguiente secuencia de accesos a una dirección de memoria D, con nodo origen N3, que inicialmente no se encuentra en ninguna caché:

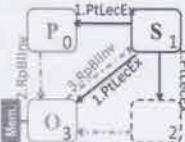
1. Lectura generada por el procesador del nodo 1
2. Escritura generada por el procesador del nodo 1
3. Lectura generada por el procesador del nodo 2
4. Lectura generada por el procesador del nodo 3
5. Escritura generada por el procesador del nodo 0
6. Escritura generada por el procesador del nodo 1

Como D no se encuentra inicialmente en ninguna caché, la entrada del directorio para ese bloque tendrá al inicio estado Válido.

En la siguiente tabla se resumen las acciones para la secuencia de accesos:

ESTADO INICIAL	EVENTO	ACCIONES	ESTADO SIGUIENTE
N0) Inválido	P1 lee D		N0) Inválido
S-N1) Inválido			S-N1) Compartido
N2) Inválido			N2) Inválido
N3) Inválido			N3) Inválido
M-N3) Válido			M-N3) Válido
		<ol style="list-style-type: none"> 1. N1 difunde petición de lectura del bloque BD (PtLec(BD)) a todos los nodos. 2. N3 recibe PtLec(BD). Como tiene el bloque BD en estado Válido envía paquete de respuesta con el bloque a N1 (RpBloque(BD)). El resto de nodos que reciben la petición, N0 y N2, no hacen nada. 3. N1 recibe RpBloque(BD) e introduce el bloque en caché en estado Compartido 	
N0) Inválido	P1 escribe en D		N0) Inválido
S-N1) Compartido			S-N1) Modificado
N2) Inválido			N2) Inválido
N3) Inválido			N3) Inválido
M-N3) Válido		<ol style="list-style-type: none"> 1. N1 difunde petición de acceso exclusivo para BD (PtEx(BD)) a todos los nodos. 2. N3, cuando recibe PtEx(BD) envía paquete de respuesta a N1 confirmando invalidación (RpInv(BD)) una vez que ha recibido la 	M-N3) Inválido

ESTADO INICIAL	EVENTO	ACCIONES	ESTADO SIGUIENTE
		<p>respuesta confirmando invalidación de todos los nodos (RpInv(BD)) y ha pasado BD a estado Inválido en memoria. Mientras no reciba todas las invalidaciones el bloque estará en la memoria en Pendiente de inválido. El resto de nodos que reciben la petición responden confirmando invalidación aunque no tenían copia del bloque.</p> <p>3. N1, recibida la respuesta de N3, modifica el bloque y lo pasa a estado Modificado.</p>	
N0) Inválido N1) Modificado S-N2) Inválido N3) Inválido M-N3) Inválido	P2 lee D	 <p>1. N2 envía PtLec(BD) a todos los nodos.</p> <p>2. N3, como tiene BD en estado Inválido debe esperar a recibir el bloque del nodo propietario antes de responder. El bloque estará en Pendiente de válido mientras dure la espera. N1, como tiene BD en estado modificado, responde con el bloque al nodo origen N3 y pasa el estado del bloque a Compartido en su caché. N0 descarta la petición.</p> <p>3. N3, cuando recibe la respuesta de N1 (RpBloque(BD)): (1) escribe BD en memoria, (2) pone el estado del bloque en el directorio a Válido y (3) responde con el bloque a N2 (RpBloque(BD))</p> <p>4. N2, cuando recibe RpBloque(BD), introduce el bloque en su caché en estado Compartido</p>	N0) Inválido N1) Compartido S-N2) Compartido N3) Inválido M-N3) Válido
N0) Inválido N1) Compartido N2) Compartido S-N3) Inválido M-N3) Válido	P3 lee D	 <p>N3 lee BD de su propia memoria, lo introduce en su caché en estado Compartido</p>	N0) Inválido N1) Compartido N2) Compartido S-N3) Compartido M-N3) Válido

ESTADO INICIAL	EVENTO	ACCIONES	ESTADO SIGUIENTE
S-N0) Inválido N1) Compartido N2) Compartido N3) Compartido M-N3) Válido	P0 escribe en D	 <p>1. N0 envía a todos los nodos petición de lectura de BD con acceso exclusivo PtLecEx(BD)</p> <p>2. N3 recibe PtLecEx(BD) y como tiene el bloque Válido en memoria, (1) prepara el bloque para enviarlo a N0, (2) pone en el directorio el estado de BD en Pendiente de inválido y (3) espera a recibir las respuestas confirmando invalidación del resto de nodos y de su caché local antes de responder a N0. N1 y N2 como tienen el bloque en caché en estado Compartido, invalidan su copia y responden al nodo N3 confirmando invalidación, RpInv(BD). N3 invalida la copia de su caché.</p> <p>3. N3, cuando recibe todas las RpInv(BD) y ha invalidado su copia local: (1) envía respuesta con el bloque a N0 confirmando invalidación RpBloqueInv(BD) (espera a todas las invalidaciones para garantizar un orden en los accesos a BD y así garantizar coherencia) y (2) pone el estado de BD en el directorio a Inválido.</p> <p>4. N0 recibe RpBloqueInv de N3, escribe BD en su caché, modifica BD y lo pone en estado Modificado.</p>	S-N0) Modificado N1) Inválido N2) Inválido N3) Inválido M-N3) Inválido
N0) Modificado S-N1) Inválido N2) Inválido N3) Inválido M-N3) Inválido	P1 escribe en D	 <p>1. N1 envía a todos los nodos petición de lectura de BD con acceso exclusivo PtLecEx(BD)</p> <p>2. N3 recibe PtLecEx(BD) y como tiene el bloque en estado Inválido en memoria: (1)</p>	N0) Inválido S-N1) Modificado N2) Inválido N3) Inválido M-N3) Inválido

ESTADO INICIAL	EVENTO	ACCIONES	ESTADO SIGUIENTE
		<p>pone en el directorio el estado de BD en Pendiente de inválido y (2) espera a recibir las respuestas confirmando invalidación del resto de nodos y de su caché local antes de responder a N1. N0, como tienen el bloque en caché en estado Modificado, invalida su copia y responde al nodo N3 con el bloque confirmando invalidación, RpBloqueInv(BD). N2 responde confirmando invalidación, RpInv(BD).</p> <p>3. N3, cuando recibe RpBloqueInv(BD) de N0 y RpInv(BD) de N2: (1) envía respuesta con el bloque a N0 confirmando invalidación RpBloqueInv(BD) y (2) pone el estado de BD en el directorio a Inválido.</p> <p>4. N1 recibe RpBloqueInv(BD) de N3, escribe BD en su caché, modifica BD y lo pone en estado Modificado</p>	

4. Consistencia del sistema de memoria

El modelo de consistencia de memoria de una herramienta de programación (modelo de consistencia del software, Figura 8) o de un computador (modelo de consistencia del hardware) especifica (las restricciones en) el orden en el que se realizan (o parece que se realizan) las operaciones de memoria (lectura, escritura), operaciones a las mismas o distintas direcciones y emitidas por el mismo o distinto flujo de instrucciones o procesador.

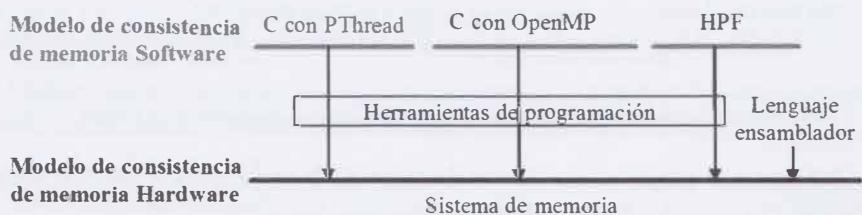


Figura 8. Modelos de consistencia de memoria

El concepto de consistencia del sistema de memoria no se debe confundir con el concepto de coherencia (Sección 3). La coherencia sólo abarca operaciones de memoria realizadas en una misma dirección de memoria, mientras que consistencia hace referencia a accesos a todas las direcciones de memoria. El hardware de las arquitecturas CC-NUMA y UMA mantiene coherencia y, respecto a la consistencia, no hay un modelo generalizado.

Las herramientas de programación que ofrecen un modelo de **consistencia secuencial** facilitan el trabajo al programador. El resultado que esperaría el programador por lógica obtener de la ejecución paralela de un programa es el que se obtiene si la consistencia es secuencial. Se deben cumplir dos condiciones para mantener consistencia secuencial [S. V. Adve y K. Gharachorloo, 1996]:

- Condición 1 (**Orden del programa**). Las operaciones de memoria que ejecuta cada flujo de instrucciones deben realizarse (o parecer que se realizan) en el orden descrito por el código que ejecuta el flujo de instrucciones.
- Condición 2 (**Orden global o atomicidad**). Todos los flujos de instrucciones deben ver el mismo orden en todos los accesos a memoria (el mismo orden serie global) como si la ejecución de las operaciones de memoria fuese **atómica**; es decir, como si se ejecutaran de forma instantánea. La ejecución de operaciones de memoria en el hardware actual es claramente noatómica debido a la presencia de cachés, pero podría parecerlo si todos los procesadores ven el mismo orden en todos los accesos a memoria. Esta condición garantiza que todas las operaciones de memoria parecen ejecutarse una detrás de otra en serie aunque en distintas ejecuciones puedan hacerlo en distinto orden. Si no se cumple esta segunda condición, y aunque se cumpliese la primera, un flujo podría ver lo que otro ha escrito antes de que esa escritura sea visible al resto de flujos, por lo que verían distinto orden en los accesos a memoria.

Cuestión 1. Conteste a las siguientes cuestiones teniendo en cuenta que los flujos de instrucciones se ejecutan en paralelo:

1. ¿Qué espera el programador que ocurra si se ejecutan los flujos de instrucciones F1 y F2? (K1 y K2 son variables compartidas, inicialmente K1=K2=0)

Accesos a memoria	F1	F2	
(1.1) W(K2)	K1=1;	(2.1) W(K2)	K2=1;
(1.2) R(K1)	if (K2=0) { Sección crítica F1 };	(2.2) R(K1)	if (K1=0) { Sección crítica F2 };

(Flujo.Orden acceso memoria)operación escritura W(variable) o lectura R(variable). Ej.: (1.1)W(k1)

2. ¿Qué espera el programador que se almacene en la variable local reg1 si llega a ejecutarse reg1=A? (A y B son variables compartidas, inicialmente A=B=0)

F1	F2	F3
A=1;	if (A=1) B=1;	if (B=1) reg1=A;

Accesos a memoria		
F1	F2	F3
(1.1) W(A)	(2.1) R(A)	(3.1) R(B)
	(2.2) W(B)	(3.2) R(A)

3. ¿Qué espera el programador que se almacene en la variable local copia? (A y K son variables compartidas, inicialmente K es 0)

Accesos a memoria	F1	Accesos a memoria	F2
(1.1) W(A)	A=1; K=1;	(2.1) R(K)	while (K=0) {}; (2.2) R(A)
(1.2) W(K)			copia=A;

Repuestas:

1. Espera que ninguno de los flujos de control o que sólo uno entre en su sección crítica.
2. Espera que `reg1` contenga 1.
3. Espera que `copia` contenga 1.

Justificaciones:

1. El programador espera que pueda ocurrir una de las siguientes posibilidades:
 - Que los dos flujos encuentren al llegar a la sentencia `if` que no se cumple la condición (porque se supone que los dos han ejecutado la operación previa al `if`). Ninguno entra en su sección crítica en este caso.
- Posibles órdenes esperados en los accesos a memoria en este caso (todos estos órdenes cumplen la primera condición de consistencia secuencial):
- ✓ (1.1) (2.1) (2.2) (1.2),
 - ✓ (1.1) (2.1) (1.2) (2.2),
 - ✓ (2.1) (1.1) (2.2) (1.2), o
 - ✓ (2.1) (1.1) (1.2) (2.2)
- Que uno de los flujos pueda llegar a su `if` antes que el otro. Entonces podría encontrar la condición verdadera (si el otro aún no ha ejecutado la operación previa a su `if`) y entrar en su sección crítica. Cuando el otro llegue a su `if` se espera que encuentre que no se cumple la condición porque se tiene la confianza en que el flujo que entró en la sección crítica ejecutó la operación que hay antes del `if`.

Posibles órdenes esperados en los accesos a memoria (todos estos órdenes cumplen la primera condición de consistencia secuencial):

- ✓ (1.1) (1.2) (2.1) (2.2), o
 - ✓ (2.1) (2.2) (1.1) (1.2)
2. Si F3 ejecuta `reg1=A` es porque ha encontrado que `B=1`. Si `B=1` es porque F2 ha encontrado que `A=1`. Luego se espera que `reg1` valga 1 tras la ejecución paralela de los tres trozos de código si se llega a ejecutar `reg1=A`.

Orden de ejecución que lleva a que se ejecute la instrucción (3.2) `reg1=A`:

- ✓ (1.1) (2.1) (2.2) (3.1) (3.2)

3. F2 espera en el while mientras k sea 0. Cuando encuentre que k es 1 saldrá del while y ejecutará $copia=A$. Se confía en que cuando F1 ejecute $k=1$ haya ejecutado la operación que hay antes en el código, $A=1$, que hace A igual a 1. Por eso se espera que $copia$ tenga 1 cuando termine la ejecución de los dos trozos de código.

Posibles órdenes esperados en los accesos a memoria (todos cumplen la primera condición de consistencia secuencial):

- ✓ (2.1) ... (2.1) (1.1) (2.1) ... (2.1) (1.2) (2.1) (2.2),
- ✓ (1.1) (2.1) ... (2.1) (1.2) (2.1) (2.2),
- ✓ (1.1) (1.2) (2.1) (2.2)

En la actualidad, ningún modelo de consistencia a nivel del hardware garantiza consistencia secuencial debido a la penalización en tiempo de ejecución que supondría no poder alterar el orden de los accesos a memoria independientes (i.e. accesos a memoria a distintas direcciones) del código que ejecuta un flujo. La penalización se agrava especialmente si no se permite que las lecturas adelanten a escrituras anteriores, debido al tiempo que supone completar una escritura y a que una operación no puede ejecutarse hasta no tener sus datos disponibles. Una escritura requiere siempre transferencias a través de la red, esté o no el dato a modificar en la caché del nodo de la red en el que se ejecuta la escritura (ver Sección 3). Estas transferencias incrementan su tiempo de ejecución. Los compiladores también reordenan accesos a memoria independientes.

Un modelo de consistencia relajado se caracteriza por:

- Los órdenes entre operaciones de memoria que no garantiza (en la Tabla 3 se pueden ver modelos de consistencia hardware de procesadores actuales):
 - Pueden relajar orden del programa (llamado también, en modelos de consistencia hardware, orden en memoria). Los modelos relajados pueden permitir que en el código ejecutado en un flujo (procesador) no se garantice el orden del programa de dos accesos a memoria *a distintas direcciones*. Difieren en cuanto a los órdenes que permiten relajar; pueden permitir alterar el orden entre: (1) una escritura y una lectura posterior en el código (relajaría entonces $W \rightarrow R$), (2) una escritura y una escritura posterior ($W \rightarrow W$), (3) una lectura y una lectura posterior ($R \rightarrow R$), y/o (4) una lectura y una escritura posterior ($R \rightarrow W$).
 - Pueden relajar orden global (atomicidad). Hay modelos que permiten que un flujo (procesador) pueda ver lo que otro ha escrito antes de que esta escritura sea visible al resto de flujos; es decir, antes de que se haya completado globalmente.
- Los mecanismos (instrucciones máquina en el caso de los modelos del hardware) que hay que añadir para garantizar un orden entre operaciones de memoria cuando resulta necesario para la correcta ejecución de los programas.

Resulta necesario garantizar un orden en los accesos a memoria cuando se necesita comunicar datos entre flujos (entre procesadores) a través de variables compartidas (memoria compartida). Por ejemplo, en el tercer código de la Cuestión 1, el flujo F1 pasa un dato a F2 a través de la variable compartida A. El programador ha añadido código de sincronización para que la comunicación se pueda realizar. En particular, ha añadido `while (k=0) {}` antes del acceso a la variable compartida A en F2 y `k=1` en F1 después del acceso a la variable compartida A. Pero el código de sincronización añadido no bastaría para garantizar la comunicación si, por ejemplo, el modelo de consistencia no garantiza el orden entre escrituras (W->W) o entre lecturas (R->R). Si no se garantiza el orden W->W, F2 puede salir del bucle antes de que F1 escriba un 1 en A porque F1 puede ejecutar la segunda operación de escritura (`k=1`) antes que la primera (`A=1`). En este caso habría que utilizar la instrucción que ofrezca el modelo de consistencia para garantizar un orden entre estas dos escrituras (ver Figura 13 en la Sección 5). Si no se garantiza el orden R->R, igualmente F2 puede salir del bucle antes de que F1 escriba un 1 en A, en este caso porque, en F2, la lectura de A se puede adelantar a la última lectura de k (la lectura de la última iteración del bucle, cuando la lectura de k va a devolver 1). Las instrucciones para garantizar un orden se usan, por tanto, en código de sincronización. Hay dos tipos de código de sincronización: adquisición y liberación. El código de **adquisición** (`while (k=0) {}` en el código 3 de la Cuestión 1) se usa antes del acceso a variables compartidas, un flujo de instrucciones ejecuta este código para *adquirir* el derecho a acceder a variables compartidas. El código de **liberación** (`k=1` en el código 3) se usa después de acceder a variables compartidas, un flujo de instrucciones ejecuta este código para *liberar* a uno de los flujos que está esperando para acceder a las variables compartidas.

Tabla 3. Modelos de consistencia de memoria relajados implementados en procesadores actuales. (sigue la estructura de la tabla de modelos relajados de [S. V. Adve y K. Gharachorloo, 1996])

Modelo	Orden del programa relajado			Orden global		Instrucciones para garantizar los órdenes relajados por el modelo
	W->R	W->W	R->RW	Lec. anticipada	propia de otro	
Sparc-TSO, x86-TSO	Si			Si		I-m-e (instrucciones lectura-modificación-escritura atómica)
Sparc-PSO	Si	Si		Si		I-m-e, STBAR (instrucción <i>STore BARrier</i>)
Sparc-RMO	Si	Si	Si	Si		MEMBAR (instrucción <i>MEMory BARrier</i>)
PowerPC	Si	Si	Si	Si	Si	SYNC, ISYNC (instrucciones <i>SYNChronization</i>)
Itanium	Si	Si	Si	Si		LD.ACQ, ST.REL, MF (ACQuisition LoD, RElease STore, Memory Fence), y cmpxchg8.acq y otras I-m-e
ARMv7	Si	Si	Si	Si	Si	DMB (Data Memory Barrier)
ARMv8	Si	Si	Si	Si	Si	LDA LDAR, STL STLR (LoD-Acquire, STore-reLease 32b 64b), LDAEX LDAXR, STLEX STLXR (LoD-Acquire eXclusive, Store-reLease eXclusive 32b 64b), DMB

Cada fila de la Tabla 3 describe un modelo de consistencia hardware distinto. Se diferencian en los órdenes (local y global) que no garantizan (órdenes relajados) y/o en las instrucciones máquina que ofrece el modelo para garantizar los órdenes que se relajan. En todos los modelos se relaja el orden **W->R** porque todos los procesadores (núcleos) usan un **buffer para las escrituras** en la etapa de ejecución del cauce segmentado. Desde este buffer se van llevando las escrituras a caché. Hasta que una escritura no deja el buffer no es visible al resto de procesadores. El buffer permite ocultar el tiempo que supone completar una escritura en memoria. Una lectura puede adelantar a las escrituras del buffer. Además, si un procesador ejecuta una lectura habiendo escrituras en su buffer a la misma dirección, se permite que se pueda obtener el valor de la lectura directamente de la escritura del buffer más reciente a la misma dirección, antes, por tanto, de que esta escritura se haya realizado (llevado a caché) y la puedan ver el resto de procesadores. Por este motivo aparece en la tabla que todos los modelos de consistencia permiten que un procesador pueda leer una escritura **propia** antes de que esa escritura sea visible al resto.

El modelo de consistencia TSO (*Total Store Ordering*) de los procesadores SPARC de Sun (ahora Oracle) y el de los procesadores de la familia x86 de Intel sólo relajan el orden del programa **W->R**. Para garantizar un orden se pueden usar las instrucciones de lectura-modificación-escritura atómica de una variable que incluye el repertorio de instrucciones. Estas instrucciones, que se introdujeron en los procesadores para implementar funciones de sincronización (Sección 5), permiten garantizar en estos casos el orden **W->R** porque llevan a caché todas las escrituras pendientes en el buffer de escrituras, incluida la escritura de la propia instrucción de lectura-modificación-escritura, antes de que se ejecuten las instrucciones que hay detrás en el código.

En el modelo de consistencia PSO de SPARC se relaja el orden **W->W** además de **W->R**. Para establecer un orden entre escrituras el modelo añadió la instrucción máquina **stbar**, que garantiza que las operaciones de escritura que se encuentran en el código antes de esta instrucción se han ejecutado antes de empezar la ejecución de las que hay después.

El modelo de consistencia del PowerPC de Apple/IBM/Motorola relaja el orden entre todas las instrucciones de acceso a memoria del programa que ejecuta un procesador (**W->R**, **W->W**, **R->R** y **R->W**) y además tampoco garantiza orden global porque permite que un procesador vea lo que otro ha escrito antes de que esta escritura sea visible al resto de procesadores del sistema. Esto último hace que los procesadores puedan ver distinto orden global en los accesos a memoria. En el modelo del PowerPC se usan dos instrucciones máquina para garantizar orden, **sync** e **isync**. Ambas garantizan un orden entre las instrucciones que hay delante y detrás de ellas en el código, incluido accesos a memoria. La instrucción **sync** permite además que una escritura anterior sea visible a todos los procesadores impidiendo que uno pueda ver lo que el procesador que ejecuta **sync** ha escrito en memoria antes de que lo vean el resto de procesadores (establece un orden global). La instrucción **sync** se usa en código de sincronización de liberación, en el código de adquisición bastaría con usar **isync**, que requiere menos

tiempo de ejecución que `sync`. Las instrucciones para garantizar orden entre operaciones de memoria del PowerPC permiten establecer el siguiente orden entre las operaciones de acceso a memoria de un flujo de instrucciones (SA y SL son código de sincronización de adquisición y liberación, respectivamente):

$$R,W \rightarrow SA \rightarrow R,W \quad \text{y} \quad R,W \rightarrow SL \rightarrow R,W \quad (1)$$

A los modelos de consistencia que no garantizan ningún orden entre operaciones de acceso a memoria en el código que ejecuta un flujo (no garantizan ningún orden del programa) y que ofrecen instrucciones máquina para establecer el orden reflejado en (1) se les llama **modelos de consistencia de ordenación débil**. El modelo de consistencia de memoria de PowerPC, el RMO de SPARC y el de ARMv7 son modelos de este tipo. La Figura 9 muestra el orden que se establece con un modelo de ordenación débil en un ejemplo en el que `nthread` flujos suman en paralelo los componentes de un vector `a []`.

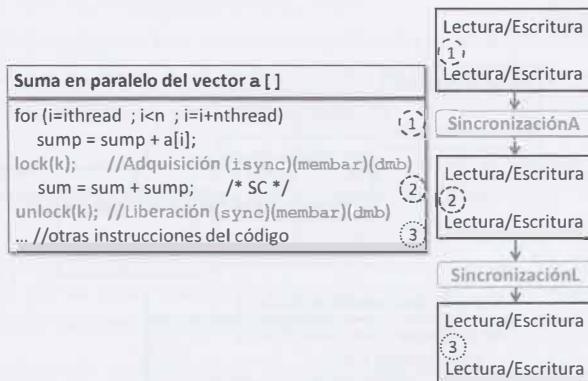


Figura 9. Orden que se establece con las instrucciones de ordenación en modelos de consistencia de ordenación débil. Ej.: procesador PowerPC de Apple/IBM/Motorola (`isync`, `sync`), RMO de SPARC de Sun (`membar`), ARMv7 (`dmb`)

El modelo de consistencia del procesador Itanium de Intel también relaja todos los órdenes del programa, `W->R`, `W->W`, `R->R` y `R->W`. En este modelo se tiene en cuenta que el código de sincronización de adquisición se usa antes de acceder a memoria compartida y que el de liberación se usa después de acceder a memoria compartida. Para ello añade instrucciones máquina de acceso a memoria de adquisición (se distinguen en el ensamblador del Itanium por el sufijo `.acq`) que garantizan un orden entre el acceso de esta instrucción y de instrucciones de memoria posteriores, e instrucciones máquina de acceso a memoria de liberación (sufijo `.rel`) que garantizan un orden entre accesos a memoria anteriores y el acceso a memoria de la propia instrucción con acceso de liberación. Estas instrucciones permiten implementar código de sincronización que garantizan los siguientes órdenes:

$$SA \rightarrow R,W \quad \text{y} \quad R,W \rightarrow SL \quad (2)$$

A los modelos de consistencia que no garantizan ningún orden entre operaciones de acceso a memoria en el código que ejecuta un flujo y que permiten cumplir (2) se le denomina en la bibliografía **modelo de consistencia de liberación**. Itanium y ARMv8 tienen modelos de este tipo. La Figura 10 muestra el orden que se establece con un modelo de consistencia de liberación en un ejemplo en el que nthread flujos de instrucciones suman en paralelo los componentes de un vector $a[]$. El modelo de consistencia de liberación permite aprovechar el paralelismo a nivel de instrucción en mayor grado que el modelo de ordenación débil como se puede deducir comparando la Figura 9 y la Figura 10. Como refleja la Figura 10, las instrucciones de acceso a memoria que están antes en el código que una primitiva de sincronización de adquisición ((1) en la figura) se pueden ejecutar en paralelo a las que hay después ((2)), mientras que con un modelo de ordenación débil no es posible (Figura 9). Igualmente, las instrucciones de memoria que hay antes de una primitiva de liberación ((2) en la figura) se pueden ejecutar en paralelo a las que hay después ((3)), lo que tampoco ocurre si el modelo es de ordenación débil.

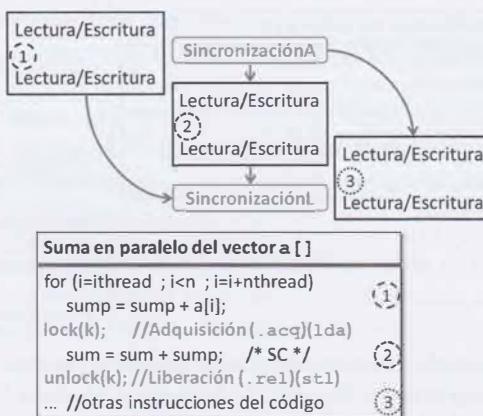


Figura 10. Orden que se establece con las instrucciones de ordenación en modelos de consistencia de liberación. Ej.: Procesador Itaninum de Intel (instrucciones con sufijos .acq, .rel), ARMv8 (prefijos lda y stl)

Cuestión 2. ¿Qué ocurre si los ejemplos de la Cuestión 1 se ejecuta en arquitecturas SPARC con el modelo de consistencia TSO (*Total Store Ordering*)?

Respuesta

TSO relaja el orden entre una escritura y una lectura posterior (W->R). El resto de órdenes están garantizados.

En el primer ejemplo hay accesos de lectura después de escritura en el código ejecutado por un flujo por lo que podrían obtenerse resultados imposibles con

consistencia secuencial. En este caso podría ocurrir que los dos flujos entren en su sección crítica porque cuando ejecutan la lectura (segunda operación de memoria del código que ejecutan los flujos, (1.2) y (2.2)) no se garantiza que se haya ejecutado la escritura anterior ((1.1) y (2.1)). Si esto es así los dos flujos de control pueden ejecutar primero la lectura y encontrar que se cumple la condición del `if`. Entonces son posibles estos otros órdenes globales que permiten que los dos flujos entren en la sección crítica:

- ✓ (1.2) (2.2) (1.1) (2.1),
- ✓ (1.2) (2.2) (2.1) (1.1),
- ✓ (2.2) (1.2) (1.1) (2.1),
- ✓ (2.2) (1.2) (2.1) (1.1),
- ✓ (1.2) (2.1) (2.2) (1.1),
- ✓ (2.2) (1.1) (1.2) (2.1)

En el resto de ejemplos no hay escrituras seguidas de lecturas. Por tanto, se obtendría el mismo resultado que con un modelo de consistencia secuencial, que son los resultados comentados en la Cuestión 1.

Cuestión 3. ¿Qué ocurre si los ejemplos de la Cuestión 1 se ejecuta en arquitecturas SPARC con el modelo de consistencia PSO (*Parcial Store Ordering*)?

Respuesta

PSO relaja el orden entre escritura y lectura y el orden entre escrituras ($W \rightarrow R$ y $W \rightarrow W$). El resto de órdenes están garantizados.

En el primer ejemplo podría ocurrir lo ya indicado en la Cuestión 2 puesto que hay escrituras seguidas de lecturas en el código ejecutado por un flujo y PSO relaja este orden.

En el tercer ejemplo hay en F1 una operación de escritura seguida por otra de escritura y PSO relaja $W \rightarrow W$. Si F1 escribe en K un 1 (acceso a memoria (1.2) antes de escribir en A un 1 (acceso a memoria (1.1)), F2 podría salir del bucle antes de que A esté a 1 y leer en A un 0 ((2.2)). Por tanto, la variable local copia podría tener un 0 en lugar de un 1 al terminar la ejecución paralela. Entonces, además de los órdenes globales de acceso posibles con consistencia secuencial para el ejemplo 3 (ver Cuestión 1), son posibles estos otros órdenes globales en los que copia acaba con un 0:

- ✓ (2.1) ... (2.1) (1.2) (2.1) (2.2) (1.1),
- ✓ (1.2) (2.1) (2.2) (1.1).

En el segundo ejemplo se obtendría el mismo resultado que con un modelo de consistencia secuencial.

Cuestión 4. ¿Qué ocurre si los ejemplos de la Cuestión 1 se ejecuta en arquitecturas PowerPC?

Respuesta

El modelo de consistencia de los procesadores PowerPC relaja todos los órdenes, tanto orden del programa ($W \rightarrow R$, $W \rightarrow W$, $R \rightarrow R$ y $R \rightarrow W$) como orden global (permite que un procesador pueda ver lo que otro ha escrito antes de que esta escritura sea visible al resto de procesadores).

En el primer ejemplo podría ocurrir lo ya indicado en la Cuestión 2 puesto que hay escrituras seguidas de lecturas en el código ejecutado por un flujo y el modelo de consistencia PowerPC relaja este orden.

En el tercer ejemplo podría ocurrir lo ya indicado en la Cuestión 3 porque el modelo de consistencia de PowerPC relaja $W \rightarrow W$. Al relajar también $R \rightarrow R$ son posibles otros órdenes en los accesos globales además de los indicados en la Cuestión 3 que permiten también que se lea en A un 0, por ejemplo:

- ✓ (2.1) ... (2.1) (2.2) (1.1) (1.2) (2.1),
- ✓ (2.2) (1.1) (1.2) (2.1)

En el segundo ejemplo, la relajación del orden global (atomicidad) de los procesadores PowerPC podría ocasionar un resultado distinto que el obtenido con un modelo de consistencia secuencial aunque las instrucciones se ejecutaran en el orden del programa. Debido a la relajación de la atomicidad (segunda condición de consistencia secuencial), no se garantiza que todos los procesadores vean el mismo orden en los accesos a memoria. Como se ha comentado en la Sección 3, las escrituras en una caché de un nodo en la red se propagan a las cachés de otros nodos. Puede ocurrir que la escritura en A de 1 de F1 (1.1) llegue antes a la caché del nodo que ejecuta F2 que a la del que ejecuta F3 y que incluso llegue a esta última después de la escritura en B de 1 de F2 (2.2) (ver Figura 11). En esas circunstancias F3 puede leer en B un 1 (3.1) y después leer en A un 0 (3.2) en lugar de 1 (por no haber llegado todavía a su caché la propagación de la escritura en A de 1 de F1). F2 y F3 verán entonces distinto orden en los accesos a memoria aunque se garantice orden local (Condición 1 de consistencia secuencial):

- Para F2: (1.1) (2.1) (2.2) (3.1) (3.2)
- Para F3: (2.1) (2.2) (3.1) (3.2) (1.1)

Accesos a memoria		
F1	F2	F3
A=1; if (A=1) B=1;	if (A=1) B=1; reg1=A;	$\xrightarrow{1} B=1$ reg1=A;
		3

Accesos a memoria		
F1	F2	F3
(1.1)W(A)	(2.1)R(A)	$\xrightarrow{1} R(B)$
	(2.2)W(B)	(3.2)R(A)

Figura 11. Posible propagación de escrituras en un multiprocesador que no cumple la Condición 2 de consistencia secuencial en un ejemplo con tres flujos de control. Los números dentro de circunferencias indican el orden de llegada de las escrituras

5. Sincronización

La comunicación, uno-a-uno o colectiva, entre flujos de instrucciones (procesadores) cuando estos comparten variables (memoria) se lleva a cabo a través de variables compartidas (memoria compartida). En la Figura 12 se puede ver un ejemplo de comunicación uno-a-uno en la que un flujo F1 envía valor a otro F2 a través de la variable compartida A. Para que la comunicación de datos entre flujos de instrucciones que se ejecutan de forma concurrente o en paralelo se lleve a cabo sin problemas es necesario añadir código que sincronice el flujo emisor y el receptor del dato para evitar que el receptor (consumidor) del dato acceda a la variable compartida antes de que el emisor (productor) lo haya calculado y colocado en dicha variable. El programador podría añadir al código de la Figura 12, por ejemplo, el código para sincronización mostrado en la Figura 13.

	Ejecución secuencial	Ejecución paralela en dos flujos	
Leng. alto nivel	<pre>... A=valor; ... copia=A; ...</pre>	<u>F1</u> <pre>...</pre>	<u>F2</u> <pre>copia=A; ...</pre>
Ensamblador	<pre>... mov A, rax ... mov rbx, A ...</pre>	<u>F1</u> <pre>...</pre>	<u>F2</u> <pre>mov A, rax ... mov rbx, A ...</pre>

Figura 12. Ejecución secuencial y paralela de un trozo de código. A es una variable compartida, valor y copia son privadas. Se muestra el ensamblador suponiendo procesadores de la familia x86.

Ejecución Paralela (inicialmente K=0)	
F1	F2
<pre>... A=valor; //Si es necesario, usar instrucción ... //... para mantener orden W->W K=1; ...</pre>	<pre>... while (K==0) { }; //Si es necesario, usar instrucción ... //... para mantener orden R->R copia=A; ...</pre>

Figura 13. Posible código de sincronización para una comunicación uno-a-uno (en gris). A y K son variables compartidas, valor y copia son variables privadas. Según el modelo de consistencia del procesador será o no necesario utilizar instrucciones que garanticen el orden entre escrituras en F1 y el orden entre lecturas en F2.

Cuestión 5. (a) ¿Se puede usar el código de sincronización de la Figura 13 en computadores con procesadores SPARC con modelo de consistencia TSO? (b) ¿y si tiene el modelo PSO? (c) ¿Qué tendría que añadir el programador al código de sincronización para que se pueda usar con TSO? (d) ¿y con PSO?

Respuesta

(a), (b), (c) Las preguntas (a) y (c) están contestadas en la Cuestión 2. La pregunta (b) está contestada en la Cuestión 3.

(d) Se tendría que añadir una instrucción `stbar` en el código de sincronización de liberación, antes de asignar 1 a `K`, para garantizar que no se escribe en `K` un 1 hasta que no se haya escrito en `A` el contenido de `copia`.

En la Figura 14 se muestra un ejemplo de código paralelo con comunicación colectiva, además la figura muestra la versión secuencial del código. Se trata de un código secuencial para la suma de los componentes de un vector `a[]`. En la versión paralela, primero cada uno de los `nthread` flujos de instrucciones calcula en una variable local `sump` la suma de un subconjunto de los componentes de `a[]`. Después cada uno acumula en la variable compartida `sum` el resultado parcial que ha calculado en `sump`. Para que la versión paralela imprima un resultado de suma correcto se debe añadir código de sincronización para asegurar que:

1. Los flujos de instrucciones acceden secuencialmente a modificar la variable compartida `sum`; es decir, que ejecuten uno detrás de otro en secuencia, no importa el orden, la sentencia `sum = sum + sump`, que incluye leer `sum (R(sum))`, modificar `sum` sumándole la variable local `sump` y escribir resultado en `sum (W(sum))`.
2. El flujo `ithread=0` imprime el contenido de la variable `sum` cuando todos los flujos hayan acumulado en esta variable la suma parcial que han calculado en su variable privada `sump`.

Código secuencial	Código paralelo con problemas de sincronización (inicialmente <code>sum=0</code>)	Accesos de un flujo F a variables compartidas
<pre>for (i=0; i<n; i++) { sum = sum + a[i]; } printf(sum);</pre>	<pre>for (i=ithread ; i<n; i=i+nthread) { sump = sump + a[i]; } sum = sum + sump; /*SC*/ if (ithread==0) printf(sum);</pre>	(F.2)R(sum) (F.3)W(sum) (F.3)R(sum)

Figura 14. Ejemplo de código paralelo con comunicación colectiva (muchos-a-uno o muchos-a-muchos) que tiene problemas de sincronización: suma de los componentes de un vector. La variable en negrita (`sum`) es compartida por todos los flujos, las variables en cursiva deben ser privadas (`i`, identificador del flujo `ithread` y `sump`) y el resto (`n`, `nthread`, `a[]`) podrían ser compartidas o privadas.

A las secciones de código con acceso a variables compartidas a las que se requiere acceder de forma secuencial (en exclusión mutua) por parte de los flujos de instrucciones se las suele denominar en la bibliografía **secciones críticas** (SC en la Figura 14).

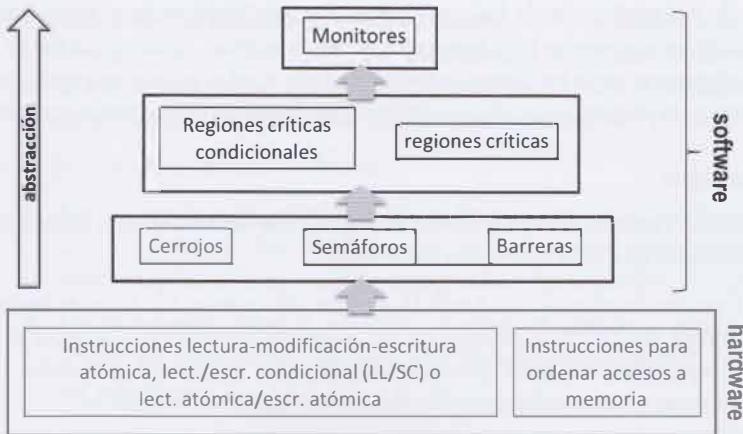


Figura 15. Primitivas para sincronización de flujos de instrucciones a nivel hardware y a nivel software. Las primitivas que se estudian en este capítulo están destacadas en gris.

Código secuencial	Código paralelo sin problemas de sincronización (inicialmente $sum=0$)	Accesos de un flujo F a variables compartidas
<pre>for (i=0; i<n; i++) { sum = sum + a[i]; } printf(sum);</pre>	<pre>for (i=ithread ; i<n; i=i+nthread) { sump = sump + a[i]; } lock(k); sum = sum + sump; /*SC*/ unlock(k); barrier(id_barrier, nthread); if (ithread==0) printf(sum);</pre>	(F.1)R/W(k) (F.2)R(sum) (F.3)W(sum) (F.4)W(k) (F.5)R/W(bar[id_barrier]) (F.6)R(sum)

Figura 16. Ejemplo de código paralelo con comunicación colectiva (muchos-a-uno o muchos-a-muchos) sin problemas de sincronización: suma de los componentes de un vector. Las variables en negrita son compartidas (sum, k, bar), las variables en cursiva deben ser privadas (i, ithread, sump) y el resto (n, nthread, a[]) podrían ser compartidas o privadas.

En la Figura 15 se puede ver una jerarquía de primitivas de sincronización. Se incluye funciones de sincronización software que ofrecen los sistemas operativos o las herramientas de programación e instrucciones máquina que ofrecen los procesadores actuales para implementar sincronización. La jerarquía refleja las primitivas de un nivel se pueden usar para implementar primitivas del nivel superior. En cualquier caso las instrucciones máquina de sincronización se usan para implementar funciones software de sincronización. En el código de la Figura 12 se podrían usar **cerrojos** en lugar del código usado en la Figura 13 para garantizar que la comunicación uno-a-uno se lleva a cabo sin problemas. En el código con comunicación colectiva de la Figura 14 se pueden usar **cerrojos** para el acceso secuencial a la sección crítica y una **barrera** delante del *if* para asegurar que todos los flujos han acumulado su resultado parcial en la variable compartida antes de que el flujo 0 la imprima (Figura 16). Con estas dos funciones

software de sincronización de bajo nivel bastaría para implementar sincronización en comunicaciones uno-a-uno y en comunicaciones colectivas. A continuación se describe cómo implementar estas funciones software de bajo nivel y las instrucciones que ofrece el hardware para implementar eficientemente funciones software para sincronización.

5.1. Cerrojos

Un cerrojo, también denominado en la bibliografía **mutex** (*mutual exclusion*), consta de dos funciones que se utilizan para sincronizar:

1. Función de adquisición **lock()** o cierre del cerrojo. Un flujo de instrucciones ejecuta un **lock()** cuando quiere **adquirir** el derecho a acceder a variables compartidas, lo hace cerrando/adquiriendo el cerrojo (requisito 1.1). Su implementación debe cumplir además los siguientes requisitos:
 - 1.2. Si varios flujos intentan la adquisición/cierre a la vez, sólo uno de ellos lo debe conseguir, el resto debe pasar a una etapa de **espera**.
 - 1.3. Todos los procesos que ejecuten **lock()** con el cerrojo cerrado/adquirido deben quedar en **espera**.
2. Función de liberación **unlock()** o apertura del cerrojo. Un flujo de instrucciones ejecuta un **unlock()** cuando ha terminado el acceso a las variables compartidas con el objetivo de **liberar** a *uno* de los flujos que está esperando el acceso a estas variables, el flujo liberado adquiere/cierra el cerrojo (requisito 2.1). Su implementación debe cumplir además el siguiente requisito:
 - 2.2. Si no hay flujos en **espera**, permitirá que el siguiente flujo que ejecute la función **lock()** adquiera/cierre el cerrojo sin espera.

Posibles implementaciones de cerrojos:

- **Cerrojo simple.** En la implementación más sencilla, se usa como cerrojo una variable compartida **k** que toma dos valores: 0 (abierto) y 1 (cerrado). El código de las funciones del cerrojo es el siguiente:

1. Función de liberación **unlock(k)** o apertura del cerrojo: abre el cerrojo escribiendo un 0

```
unlock(k) {
  k = 0 ;
} /* k compartida */
```

2. Función de adquisición **lock(k)** o cierre del cerrojo:

```
lock(k) {
  while (leer-asignar_1-escibir(k) == 1) {} ;
} /* k compartida */
```

- Lee el cerrojo y lo cierra escribiendo un 1 (requisito 1.1). Se requiere una operación `leer-assignar_1-escribir(k)` indivisible; es decir, entre lectura y escritura no se debe realizar ningún otro acceso a `k` para evitar que más de un flujo encuentre el cerrojo abierto a la vez. En caso contrario no se cumpliría el requisito 1.2.
- Si de la anterior lectura se obtiene:
 - Un 1 (cerrojo adquirido/cerrado), el flujo espera (en el bucle `while` del `lock()`) porque hay un flujo que tiene el cerrojo cerrado/adquirido y estará accediendo a las variables compartidas. Así se cumple el requisito 1.3.
 - Un 0 (cerrojo abierto), el flujo adquiere el cerrojo y puede salir del `lock(k)` (para pasar a acceder a las variables compartidas). Se cumplen los requisitos 2.1 y 2.2.
- **Cerrojo con etiqueta.** Un cerrojo con etiqueta establece un orden FIFO (*First Input First Output*, primero en entrar primero en salir) en la adquisición del cerrojo, garantiza que los flujos de instrucciones adquieran el cerrojo (entren en la sección crítica) en el orden en el que ejecutan la función de adquisición. El código de las funciones del cerrojo con etiqueta es el siguiente (las variables compartidas están en negrita, las variables que deben ser privadas están en cursiva):
 - 3. Función de liberación `unlock(contador.lib)`:

```
unlock(contador.lib) {
    contador.lib = (contador.lib + 1) mod max_flujos_instr;
} /* contador.lib compartida */
```

- 4. Función de adquisición `lock(contador.adq, contador.lib)`:

```
lock(contador.adq, contador.lib) {
    etiqueta = contadores.adq;
    contador.adq = (contador.adq + 1) mod max_flujos_instr;
    while (etiqueta <> contadores.lib) {};
} /* contador.adq, contador.lib compartidas */
```

- Adquiere el cerrojo el flujo cuya etiqueta coincide con el contador de liberación. Se requiere una operación `leer-incrementar-escribir(contadores.adq)` indivisible; es decir, entre lectura y escritura no se debe realizar ningún otro acceso a memoria para evitar que más de un flujo pueda leer el mismo valor de `contador.adq` (más de uno tendría entonces la misma etiqueta). En caso contrario no se cumpliría el requisito 1.2.
- Si la etiqueta de un flujo no coincide con el valor del contador de liberación, el flujo espera (en el bucle `while` del `lock()`) porque hay un flujo que tiene el cerrojo cerrado/adquirido (estará accediendo a las variables compartidas).

Dependiendo del modelo de consistencia de memoria se deberá o no añadir a estos códigos instrucciones para garantizar un orden entre los accesos a las variables

compartidas que usan las funciones del cerrojo (`k` en el cerrojo simple, `contador.adq` y `contador.lib` en el cerrojo con etiqueta) y los accesos a variables compartidas del código en el que se inserten estas funciones. Las instrucciones a añadir dependerán del modelo de consistencia de memoria particular de los procesadores de la arquitectura para la que se está implementando el código.

En los códigos implementados en esta sección para cerrojos, los flujos esperan en un bucle. Alternativamente, se puede hacer una implementación de cerrojos en los que se usen funciones del SO para suspender a los flujos que tengan que esperar en lugar de que queden esperando en un bucle ocupando un procesador (espera ocupada o *busy-waiting*). El programador tendrá que sopesar si le interesa más usar un cerrojo con esta espera ocupada o un cerrojo con bloqueo del SO teniendo en cuenta el retardo que introduce usar funciones del SO.

La Figura 16 usa un cerrojo simple para garantizar el acceso secuencial de los flujos de instrucciones a la sección crítica del ejemplo de la Figura 14.

5.2. Barreras

Una barrera es un punto en el código en el que los flujos de instrucciones que colaboran en la ejecución del código se esperan entre sí. Cuando todos han llegado a la barrera, salen de la misma y continúan ejecutando el código que hay después de la barrera. Los siguientes códigos para barrera cumplen estos requisitos (las variables compartidas están en negrita, las variables que deben ser privadas están en cursiva):

La variable `bar []` contiene una lista de barreras. Para cada barrera se necesita (1) un contador, `bar [] . cont`, para llevar la cuenta del número de flujos que han llegado ya a la barrera (han entrado en la función `barrera`), (2) una variable de espera, `bar [] . bandera`, para implementar la espera ocupada y (3) una variable cerrojo, `bar [] . cerrojo`, para implementar accesos secuenciales a variables compartidas. El primer código para barrera podría dejar los flujos bloqueados sin poder avanzar si estos reutilizan la misma barrera dos o más veces en el código. Si algún flujo se queda suspendido mientras está en el bucle esperando a que la bandera pase de 0 a 1 y despierta cuando se ha hecho la bandera 0 de nuevo (en la siguiente reutilización de la barrera), no ha visto que la bandera ha estado a 1 y, al despertar, seguirá esperando en el mismo bucle con la bandera a 0. Este flujo nunca llegará al siguiente uso de la barrera, por lo que el contador de la barrera en el siguiente uso nunca llegará a sumar el número de flujos esperados. En consecuencia, la bandera nunca volverá a ser 1 y el resto de flujos quedarán bloqueados en el bucle de espera del siguiente uso de la barrera. El segundo código de barrera elimina este problema cambiando el valor de la espera en el bucle de un uso a otro de la barrera, lo hace la primera sentencia de esta barrera complementando la bandera local de espera.

5.3. Instrucciones para implementar sincronización

Para implementar eficientemente código de sincronización los procesadores añaden:

Opción 1. Instrucciones atómicas de lectura-modificación-escritura de una variable (se garantiza que entre la lectura y la escritura no se realiza ningún otro acceso a memoria).

Opción 2. Instrucciones de lectura y escritura especiales (carga enlazada y almacenamiento condicional) que permiten emular una operación atómica de lectura-modificación-escritura. Para ser más concretos, se implementan de forma que permiten comprobar si entre la lectura y la escritura se ha modificado la variable por parte de algún otro flujo, en cuyo caso no habrá tenido éxito el intento de lectura-modificación-escritura atómica.

Las instrucciones atómicas de lectura-modificación-escritura (opción 1) que se encuentran en los procesadores actuales suelen ser una o varias de las que aparecen en la Tabla 4. Como ejemplo se muestran en la tabla las instrucciones incorporadas en los procesadores de la familia x86 de Intel. En los procesadores PowerPC y ARM (a partir de ARMv6) usan la opción 2: instrucciones `lwarx/stwcx`. en PowerPC (Figura 17) y, en ARM, `ldrex/strex` (Figura 18). En la Tabla 4 se puede ver también el código de una función `lock(k)` de cerrojo simple implementado usando cada uno de las instrucciones máquina atómicas. En estos códigos se tendrían que utilizar, en algunos modelos de consistencia, instrucciones para mantener orden entre accesos a memoria. Con las implementaciones de `lock(k)` de la tabla, los flujos de instrucciones que esperan adquirir el cerrojo escriben en la variable compartida `k` en cada iteración del bucle de espera. Estas escrituras provocan accesos a través de la red (del protocolo de mantenimiento de coherencia). Para reducir tiempo de ejecución (consumo de

potencia y uso de la red de acceso a memoria) se pueden eliminar escrituras usando una implementación de `lock(k)` con dos bucles anidados. En el bucle interno se lee el cerrojo (sin escribir) para detectar cuándo se abre (se libera). Una vez detectado que está abierto el flujo procede a intentar la adquisición del cerrojo ejecutando la instrucción de lectura-modificación-escritura atómica. En los ejemplos de la Figura 17 y Figura 19 se usa esta última alternativa. El modelo de consistencia de memoria en los ejemplos de la Figura 17 y la Figura 18(a) es de ordenación débil (Figura 9) y en los ejemplos de la Figura 18(b) y la Figura 19 es de liberación (Figura 10).

Tabla 4. Instrucciones máquina de lectura-modificación-escritura atómica. `Oper` en `Fetch&Oper()` es una operación conmutativa y asociativa (`Oper`=suma/add, multiplicación, and, or, ...).

Instr.	Test&Set(x)	Fetch&Oper(x, a)	Compare&Swap(x, a)
Descripción	<pre>Test&Set(x) { temp = x ; x = 1 ; return (temp) ; } /*x compartida*/</pre>	<pre>Fetch&Add(x, a) { temp = x ; x = x + a ; return (temp) ; } /*x compartida, a local*/</pre>	<pre>Compare&Swap(a, b, x) { if (a==x) { temp = x ; x = b; b = temp; } } /*x compartida, a y b locales*/</pre>
Ej.:procesadores x86	<pre>mov reg,1 xchg reg,mem</pre>	<code>lock xadd reg,mem</code>	<code>lock cmpxchg mem,reg</code>
cerrojo simple	<pre>lock(k) { while (Test&Set(k)==1) {}; }</pre>	<pre>lock(k) { while (Fetch&Or(k)==1) {}; }</pre>	<pre>lock(k) { b=1 do Compare&Swap(0, b, k); while (b==1); }</pre>

```

lock:          #lock(M[r3])
    li      r4,1
    #Para posteriormente cerrar el cerrojo asigna a r4 un 1
bucle:        lwarx r5,0,r3
    #carga y reserva: r5←M[r3]
    cmpwi r5,0
    #Si está cerrado (a 1) ...
    bne   bucle
    stwcxz r4,0,r3
    #pone a 1 el cerrojo (r4=1): M[r3] ← r4
    bne   bucle
    isync
    #Para asegurar que se ha adquirido el cerrojo antes de ...
    #realizar los accesos a memoria que hay después del lock

```

```

unlock:        #unlock(M[r3])
    sync
    li      r1,0
    stw   r1,0(r3)
    #Espera a que terminen los accesos anteriores
    #antes de abrir el cerrojo
    #Abre el cerrojo

```

Figura 17. Ejemplo de cerrojo simple para un procesador PowerPC (consistencia de ordenación débil) emulando `Test&Set()` con `lwarx/stwcz..` La implementación usa dos bucles anidados para evitar la escritura continua en la variable cerrojo por parte de todos los flujos de instrucciones en espera de adquirir el cerrojo.

```

lock:          #lock(M[r1])
    mov     r0, #1      #Para posteriormente cerrar el cerrojo asigna a r0 un 1
bucle:        ldrex   r5, [r1]      #Lee cerrojo
    cmp     r5, #0      #Comprueba si el cerrojo es 0 (abierto)
    streseq r5, r0, [r1] #Si el cerrojo está abierto intenta escribir (un 1) ...
    cmpeq   r5, #0      #Comprueba si ha tenido éxito la escritura
    bne     bucle      #Vuelve a intentarlo si no ha tenido éxito
    dbm          #Para asegurar que se ha adquirido el cerrojo antes de ...
                  #realizar los accesos a memoria que hay después del lock

```

```

unlock:        # unlock(M[r1])
    dmb
    mov     r0, #0      #Espera a que terminen los accesos anteriores ...
    str     r0, [r1]      #antes de abrir el cerrojo
                      #Abre el cerrojo

```

(a) ARMv7

```

lock:          #lock(M[r1])
    mov     r0, #1      #Para posteriormente cerrar el cerrojo asigna a r0 un 1
bucle:        ldaex   r5, [r1]      #Lee cerrojo con ordenación de adquisición
    cmp     r5, #0      #Comprueba si el cerrojo es 0 (abierto)
    streseq r5, r0, [r1] #Si el cerrojo está abierto intenta escribir (un 1) ...
    cmpeq   r5, #0      #Comprueba si ha tenido éxito la escritura
    bne     bucle      #Vuelve a intentarlo si no ha tenido éxito

```

```

unlock:        # unlock(M[r1])
    mov     r0, #0
    stl     r0, [r1]      #Abre el cerrojo usando almacenamiento con liberación

```

(b) ARMv8

Figura 18. Ejemplo de cerrojo simple para (a) un procesador ARMv7 (consistencia de ordenación débil) y (b) un procesador ARMv8 (consistencia de liberación) para AArch32. En ambos se emula `Test&Set()`, con `ldrex/strex` en ARMv7 y `ldaaex/strex` en ARMv8 [ARM 2015].

```

lock:          //lock(M[lock])  M[D]: acceso a dirección de memoria D
    mov ar.ccv = 0      // cmpxchg compara con ar.ccv, ...
                      // que es un registro de propósito específico
    mov r2 = 1          // cmpxchg utilizará r2 para poner el cerrojo a 1
spin:          // Se implementa espera ocupada
    ld8  r1 = [lock] ;  // Carga el valor actual del cerrojo en r1
    cmp.eq p1, p0 = r1, r2; // Si r1=r2 entonces cerrojo está a 1 y se hace p1=1
    (p1) br.cond.spnt spin ; // Si p1=1 se repite el ciclo; spnt indica que se ...
                      // usa una predicción estática para el salto de "no tomar"
    cmpxchg8.acq r1 = [lock], r2 ; // Intento de adquisición escribiendo 1 ...
                      // IF [lock]=ar.ccv THEN [lock]←r2; siempre r1←[lock]
    cmp.eq p1, p0 = r1, r2 // Si r1!=r2(r1=0) => cerrojo era 0 y se hace p1=0
    (p1) br.cond.spnt spin ; // Si p1=1 se ejecuta el salto

```



```

unlock:        //unlock(M[lock])
    st8.r1 [lock] = r0 ; // Libera asignando un 0, en Itanium r0 siempre es 0

```

Figura 19. Ejemplo de cerrojo simple para un procesador Itanium (consistencia de liberación) con `compare&swap`. La implementación usa dos bucles anidados para evitar la escritura continua en la variable cerrojo por parte de todos los flujos de instrucciones en espera de adquirir el cerrojo.

Una instrucción con predicado (p son registros predicado) se ejecuta si el predicado está a 1.

6. Problemas

Problema 1. En un multiprocesador SMP con 4 procesadores o nodos (N0-N3) basado en un bus y que implementa el protocolo MESI para mantener la coherencia se produce la siguiente secuencia de accesos a memoria a una dirección D que no se encuentra en ninguna caché:

1. Lectura generada por el procesador 1
2. Lectura generada por el procesador 2
3. Escritura generada por el procesador 1
4. Escritura generada por el procesador 2
5. Escritura generada por el procesador 3

Indicar los estados en las cachés del bloque en el que se encuentra D y las acciones que se producen en el sistema para cada uno de estos eventos.

Solución

Datos del ejercicio

Se accede a una dirección de memoria cuyo bloque k no se encuentra en ninguna caché, luego debe estar actualizado en memoria principal y el estado en las cachés se considera inválido.

Estado del bloque en las cachés y acciones generadas ante los eventos que se refiere a dicho bloque

Se va a utilizar una tabla para describir las acciones generadas para cada evento teniendo en cuenta el estado del bloque. En la tabla se usará una fila para cada uno de los 5 eventos.

Hay 4 nodos con caché (N0 a N3). Intervienen N1, N2 y N3. Se van a utilizar las siguientes siglas o acrónimos:

MP: Memoria Principal.

PtLec(k): paquete de petición de lectura del bloque k.

PtLecEx(k): paquete de petición de lectura del bloque k y de petición de acceso exclusivo al bloque k.

RpBloque(k): paquete de respuesta con el bloque k.

Se va a suponer que no existe en el sistema paquete de petición de acceso exclusivo a un bloque sin lectura (no existe PtEx). Por tanto, en los casos en los que se generaría este paquete, el controlador de caché genera un paquete de petición de acceso exclusivo con lectura (PtLecEx).

ESTADO INICIAL	EVENTO	ACCIÓN	ESTADO SIGUIENTE
N1) Inválido N2) Inválido N3) Inválido	1. P1 lee k	<p>1. N1 (el controlador de caché de N1) genera y deposita en el bus una petición de lectura del bloque k (PtLec(k)) porque no lo tiene en su caché válido.</p> <p>2. MP (el controlador de memoria de MP), al observar PtLec(k) en el bus, genera la respuesta con el bloque (RpBloque(k)).</p> <p>3. N1 recoge del bus la respuesta depositada por la memoria principal (RpBloque(k)), el bloque entra en la caché de N1 en estado Exclusivo ya que no hay copia del bloque en otra caché.</p>	N1) Exclusivo N2) Inválido N3) Inválido
N1) Exclusivo N2) Inválido N3) Inválido	2. P2 lee k	<p>1. N2 genera y deposita en el bus una PtLec(k) porque no tiene k en su caché en estado válido.</p> <p>2. N1 observa PtLec(k) en el bus y, como tiene el bloque en estado Exclusivo, lo pasa a Compartido (la copia que tiene ya no es la única válida en cachés). MP, al observar PtLec(k) en el bus, genera la respuesta con el bloque (RpBloque(k)).</p> <p>3. N2 recoge RpBloque(k) que ha depositado la memoria, el bloque entra en estado Compartido en la caché de N2 (porque hay copia del bloque en otra caché).</p>	N1) Compartido N2) Compartido N3) Inválido
N1) Compartido N2) Compartido N3) Inválido	3. P1 escribe en k	<p>1. N1 genera petición de lectura del bloque k con acceso exclusivo (PtLecEx(k)) (suponemos que no hay petición de acceso exclusivo sin lectura, no hay PtEx).</p> <p>2. N2 observa PtLecEx(k) y, como la petición incluye acceso exclusivo (Ex) a un bloque que tiene en su caché válido (Compartido), pasa su copia a estado Inválido. MP genera RpBloque(k) porque observa en el bus una petición de k con lectura (Lec). N1 no recoge RpBloque(k) depositada por la memoria porque tiene el bloque válido.</p> <p>3. N1 modifica la copia de k que tiene en su caché y lo pasa a estado Modificado.</p>	N1) Modificado N2) Inválido N3) Inválido

ESTADO INICIAL	EVENTO	ACCIÓN	ESTADO SIGUIENTE
N1) Modificado N2) Inválido N3) Inválido	4. P2 escribe en k	<ol style="list-style-type: none"> 1. N2 genera petición de lectura de k con acceso exclusivo (PtLecEx(k)) 2. N1 observa PtLecEx(k) y, como tiene el bloque en estado Modificado (es la única copia válida en todo el sistema), inhibe la respuesta de MP y genera respuesta con el bloque RpBloque (k), y además, como el paquete pide acceso exclusivo a k (Ex), invalida su copia. 3. N2 recoge RpBloque(k), introduce k en su caché, lo modifica y lo pone en estado Modificado 	N1) Inválido N2) Modificado N3) Inválido
N1) Inválido N2) Modificado N3) Inválido	5. P3 escribe en k	<ol style="list-style-type: none"> 1. N3 genera petición de lectura con acceso exclusivo de k PtLecEx(k) 2. N2 observa PtLecEx(k) y, como tiene el bloque en estado Modificado, inhibe la respuesta de MP y genera respuesta con el bloque RpBloque (k), y además, como el paquete pide acceso exclusivo a k (Ex), invalida su copia de k. 3. N3 recoge RpBloque(k), introduce el k en su caché, lo modifica y lo pone en estado Modificado 	N1) Inválido N2) Inválido N3) Modificado

Problema 2. Para un multiprocesador de memoria distribuida se quiere implementar un protocolo para mantenimiento de coherencia basado en directorios. Suponiendo que se necesitan dos bit de estado para un bloque en el directorio de memoria principal y que el tamaño de una línea de caché es de 64 bytes, calcular el porcentaje del tamaño de memoria principal que supone el tamaño del directorio de vector de bits de presencia para **(a)** un multiprocesador con 16 nodos con caché y **(b)** para un multiprocesador con 256 nodos.

Solución

Datos del ejercicio:

- ✓ Tamaño Línea de Caché (bloque de memoria): 64 Bytes = TLC
- ✓ 2 bit de estado por bloque en el directorio

(a) Porcentaje del tamaño de la memoria que supone el tamaño del directorio para un multiprocesador con 16 nodos con caché.

$$\text{Tamaño_directorio} = \text{Nº_de_bloques_memoria} \times \text{Espacio_por_bloque_en_directorio}$$

$$Nº_de_bloques_memoria = \frac{TM}{TLC} \quad (\text{TM: Tamaño Memoria principal})$$

Espacio por bloque en directorio = 18 bits (teniendo en cuenta que hay 16 nodos y 2 bits de estado)

$$\text{Tamaño directorio} = \frac{TM}{TLC} \times (16b + 2b)$$

$$\% \text{ de TM} = \frac{\frac{TM}{TLC} \times 18b}{TM} \times 100 = \frac{18b}{TLC} \times 100 = \frac{18b}{64B \times 8b/B} \times 100 \approx 3,5\%$$

El directorio de memoria no ocupa en este caso un porcentaje elevado del tamaño de la memoria principal. La implementación de protocolos de mantenimiento de coherencia con vector de bits de presencia es aceptable para multiprocesadores CC-NUMA con un número pequeño de nodos como los que pueda haber en implementaciones en una placa o en un chip.

(b) Porcentaje del tamaño de la memoria que supone el tamaño del directorio para un multiprocesador con 256 nodos con caché.

Espacio por bloque en directorio = 258 bits (teniendo en cuenta que hay 256 nodos y 2 bits de estado)

$$\text{Tamaño directorio} = \frac{TM}{TLC} \times (256b + 2b)$$

$$\% \text{ de TM} = \frac{\frac{TM}{TLC} \times 258b}{TM} \times 100 = \frac{258b}{TLC} \times 100 = \frac{258b}{64B \times 8b/B} \times 100 = \frac{258}{512} \times 100 \approx 50,4\%$$

El directorio de memoria ocuparía algo más de la mitad en este caso y superaría el tamaño de la memoria con 512 nodos. Para un número de nodos elevado no se puede utilizar un vector de bits de presencia para controlar las cachés con copia de un bloque debido al tamaño que supondría el directorio. Se pueden ver otras alternativas de implementación para grandes multiprocesadores en Julio Ortega, Mancia Anguita, 2015.

Problema 3. Se tiene un multiprocesador CC-NUMA con protocolo MSI basado en directorio distribuido (sin difusión) que usa vector de bits de presencia. Para un bloque B que se encuentra en memoria principal en estado Inválido, indicar:

(a) Cuál sería el contenido de la entrada del directorio para ese bloque en esta situación. Razone su respuesta.

- (b) Las transiciones de estados (en caché y en el directorio), la secuencia de paquetes generados por el protocolo de coherencia y el contenido al que pasa la entrada del directorio para ese bloque si un procesador que no tiene el bloque en su caché y no está en el nodo origen del bloque escribe en una dirección de dicho bloque. Razone su respuesta.

Solución

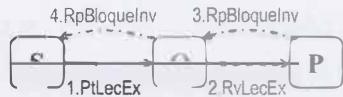
- (a) Si el bloque está inválido en memoria principal habrá una copia del bloque válida en un único caché. El contenido en la entrada del bloque del directorio en esa situación tendrá en Inválido el estado del bloque en memoria y, en su vector de bits de presencia, tendrá un único bit activo, el resto estarán a 0. El bit activo será el que corresponda a la caché que tiene la única copia válida del bloque en el sistema:

Estado	Vector de bits de presencia						
I	0	...	0	1	0	...	0

- (b) Si un procesador de un nodo que no tiene el bloque en su caché y no está en el nodo origen del bloque escribe en una dirección de dicho bloque:

1. Se produce un fallo de escritura que provoca la emisión desde el nodo del procesador que escribe, o nodo solicitante (S), al nodo origen del bloque (O) de un paquete de petición de lectura con acceso exclusivo al mismo PtLecEx(B).
2. El nodo origen (O) al recibir este paquete, como tiene el bloque Inválido, reenvía la petición al nodo propietario (P) del bloque RvLecEx(B). Obtiene el nodo propietario del directorio, el bit activo de la entrada del directorio para el bloque identifica al único nodo P. Antes de reenviar la petición pone el estado del bloque en memoria a Pendiente de inválido. El nodo O no atiende peticiones de un bloque en estado pendiente, así se garantiza coherencia.
3. El nodo propietario (P), al recibir la petición de lectura y acceso exclusivo a B, RvLecEx(B), invalida la copia que tiene del bloque (porque pide acceso exclusivo) y responde con el bloque (porque pide lectura) al origen confirmando acceso exclusivo. Por tanto, envía al origen un paquete de respuesta con el bloque confirmando la invalidación de la copia del bloque en su caché, RpBloqueInv(B).
4. El nodo O, cuando recibe la respuesta la envía al nodo solicitante RpBloqueInv(B). Antes del envío, deja activo en el vector de bits de presencia del bloque en el directorio sólo el bit del nodo S y pasa el estado del bloque a Inválido, porque lo va a modificar S.
5. El nodo S, cuando recibe el paquete con el bloque, RpBloqueInv (B), lo introduce en su caché, lo modifica y pasa su estado a Modificado en la entrada del bloque en el directorio de su caché (no confundir con el directorio de memoria principal).

Estado inicial	Evento	Estado final	Diagrama de paquetes
O-Memoria) Inválido		O-Memoria) Inválido	
S) Inválido		S) Modificado	
P) Modificado		P) Inválido	
Acceso remoto			

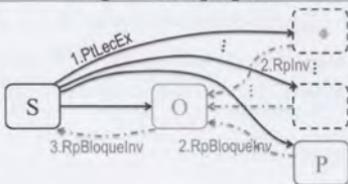


Problema 4. Contestar el apartado (b) del ejercicio anterior suponiendo que en el CC-NUMA se implementa un protocolo MSI con difusión.

Solución

Si un procesador de un nodo que no tiene el bloque en su caché y no está en el nodo origen del bloque escribe en una dirección de dicho bloque:

1. Se produce un fallo de escritura que provoca la difusión desde el nodo del procesador que escribe, o nodo solicitante (S), a todos los nodos N (tengan o no el bloque) de un paquete de petición de lectura con acceso exclusivo al bloque PtLecEx(B).
2. El único nodo propietario (P), al recibir la petición de lectura y acceso exclusivo a B, PtLecEx(B), invalida la copia que tiene del bloque (porque pide acceso exclusivo) y responde con el bloque (porque pide lectura) al origen confirmando acceso exclusivo, es decir, confirma la invalidación del bloque en su caché. Por tanto, envía al nodo origen (O) un paquete de respuesta con el bloque confirmando la invalidación de la copia del bloque en su caché, RpBloqueInv(B). El resto de nodos envía a O confirmación de invalidación RpInv(B). El nodo origen del bloque: (1) al ver que la petición solicita acceso exclusivo, espera recibir confirmación de invalidación de todos los cachés y (2) al ver que se pide lectura del bloque y que tiene el bloque solicitado en estado Inválido, sabe que recibirá el bloque de alguna caché. Mientras recibe las respuestas dejará el estado del bloque en Pendiente de inválido.
3. El nodo origen (O), cuando recibe todas las respuestas, envía RpBloqueInv(B) al nodo S. Antes del envío pasa el estado del bloque a Inválido, porque lo va a modificar el nodo S.
4. El nodo S, cuando recibe el paquete con el bloque, RpBloqueInv (B), lo introduce en su caché, lo modifica y pasa su estado a Modificado en la entrada del bloque en el directorio de su caché.

Estado inicial	Evento	Estado final	Diagrama de paquetes
O-Memoria) Inválido		O-Memoria) Inválido	
S) Inválido	Fallo de escritura	S) Modificado	
P) Modificado		P) Inválido	
Acceso remoto			

Problema 5. Se dispone de un multiprocesador CC-NUMA con 4 procesadores o nodos (N0-N3) y una memoria de 64 GBytes direccionada por bytes (16 GBytes por nodo). Para mantener la coherencia de caché, el multiprocesador implementa el protocolo MSI basado en directorios distribuidos (con vector de bits de presencia) y sin difusión. Cada procesador dispone de una caché de datos de último nivel de 8 MBytes con marcos de 32 bytes. En el multiprocesador se están ejecutando en paralelo cuatro *threads* (en N0, N1, N2 y N3) que acceden a dos vectores X[] e Y[] de 32 elementos cada uno, de 32 bits. Los vectores se encuentran almacenados a partir de una dirección de memoria múltiplo de 32: primero están almacenados los componentes de X[] y, justo a continuación, los elementos de Y[]. Conteste a las siguientes preguntas:

- (a) ¿Cuál es el tamaño del subdirectorio de memoria principal de un nodo? (considerar que se almacenan estados estables y estados pendientes de un estado estable)
- (b) ¿Cuántos bloques de memoria ocupan los vectores X[] e Y[]?
- (c) Suponiendo que inicialmente los bloques que contienen ambos vectores no están en ninguna caché, ¿cuál será entonces inicialmente el contenido de las entradas del directorio de memoria principal para cada uno de estos bloques?
- (d) Indicar los estados estables de los bloques en las cachés y los cambios en los contenidos del directorio de memoria principal ante la siguiente secuencia de eventos (considere que inicialmente los bloques que contienen ambos vectores no están en ninguna caché):
 1. Lectura generada por el procesador 0 a X[0]
 2. Escritura generada por el procesador 0 a Y[0]
 3. Lectura generada por el procesador 1 a X[1]
 4. Lectura generada por el procesador 2 a X[2]
 5. Escritura generada por el procesador 2 a Y[2]
 6. Escritura generada por el procesador 1 a Y[1]

NOTA: Suponga que bloques distintos se almacenan en la caché de cada procesador en marcos de bloque diferentes.

Solución

(a) Datos:

- ✓ Tamaño memoria de un nodo (TMN) = 16 GB
- ✓ Tamaño bloque memoria y tamaño de la línea de caché (TLC) = 32 B
- ✓ Nº de bits para almacenar el estado de un bloque en memoria según el protocolo MSI = 1 (para almacenar estados estables Válido o Inválido) + 1 (para indicar si está pendiente de un estado estable)

Tamaño subdirectorio de un nodo (TSN) = número de entradas en el directorio × número de bits de cada entrada = número de bloques de la memoria del nodo × (número de nodos + bits de estado) bits

$$TSN = \frac{TMN}{TLC} \times (4 + 2) b = \frac{16\text{ GB}}{32\text{ B}} \times 6 b = \frac{2^{34}\text{ B}}{2^5\text{ B}} \times 6 b = 2^{30} \times 3 b = \frac{2^{30} \times 3 b}{2^3 b/B} = 2^{27} \times 3 B = 2^7 \times 3 \times (2^{20}\text{ B}) = 128 \times 3\text{ MB} = 384\text{ MB}$$

(b) Datos:

- ✓ Los vectores $X[]$ e $Y[]$ tienen 32 elementos de 32 bits cada uno, es decir, 32 elementos de 4 Bytes cada uno.
- ✓ Tamaño del bloque memoria y tamaño línea de caché (TLC) = 32 B
- ✓ $X[]$ está almacenado en una dirección múltiplo de 32 e $Y[]$ se encuentra justo detrás
- ✓ La memoria se dirección a nivel de byte (una dirección es un byte)

Total bloques de $X[] + Y[]$:

$$\begin{aligned} \text{Total bloques} &= \frac{2 \text{ vectores} \times 32 \text{ comp./vector} \times 4 \text{ B/comp.}}{32 \text{ B/bloque}} \\ &= 8 \text{ (4 bloques cada uno de los vectores)} \end{aligned}$$

(c) Los bits de una entrada son 6 (como se ha indicado en (a)): 4 bits de presencia en caché (uno por cada nodo) + 2 bits de estado.

Para los 4 bloques el contenido del directorio será el siguiente: los 4 bits de presencia a 0, porque no hay copia válida en ninguna caché, el bit de estado estable a 1 (es decir a Válido) por estar válido en la memoria (ya que no hay ninguna caché que haya escrito en el bloque) y el bit de estado pendiente a 0 (por no estar pendiente de un estado estable).

(d) Sólo intervienen en la secuencia el primer bloque de $X[]$ (se notará por B0, contiene las 8 primeros componentes $X[0]X[1]X[2]X[3]X[4]X[5]X[6]X[7]$) y el primero de $Y[]$ (se notará por B4, contiene las 8 primeros componentes $Y[0]Y[1]Y[2]Y[3]Y[4]Y[5]Y[6]Y[7]$). El estado estable en caché puede ser Modificado (M), Compartido (S) o Inválido (I), y los estados estables en el directorio (memoria) Válido

(V) o Inválido (I). La entrada del directorio de un bloque contiene el estado en memoria del bloque (V o I), y los bits de presencia del bloque en los nodos N0, N1, N2 y N3 (que podrán estar a 0 o 1).

Situación	Contenido directorio entradas 1 ^{er} bloque de X [] (B0) y 1 ^{er} bloque de Y [] (B4)	E N0 N1 N2 N3	Estado en cachés	Evento										
Inicial	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>V</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table> x-B0 <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>V</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table> y-B4	V	0	0	0	0	V	0	0	0	0		Inválido todos los bloques en todas las cachés N0-3(B0):I ; N0-3(B4):I	1. N0 lee X [0]
V	0	0	0	0										
V	0	0	0	0										
Después de evento 1	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>V</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table> x-B0 <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>V</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table> y-B4	V	1	0	0	0	V	0	0	0	0		N0 (B0): S N1-3(B0): I N0-3 (B4): I	2. N0 esc. Y [0]
V	1	0	0	0										
V	0	0	0	0										
Después de evento 2	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>V</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table> x-B0 <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>I</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table> y-B4	V	1	0	0	0	I	1	0	0	0		N0 (B0): S N1-3(B0): I N0 (B4): M N1-3(B4): I	3. N1 lee X [1]
V	1	0	0	0										
I	1	0	0	0										
Después de evento 3	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>V</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table> x-B0 <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>I</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table> y-B4	V	1	1	0	0	I	1	0	0	0		N0-1 (B0): S N2-3(B0): I N0 (B4): M N1-3(B4): I	4. N2 lee X [2]
V	1	1	0	0										
I	1	0	0	0										
Después de evento 4	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>V</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> </table> x-B0 <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>I</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table> y-B4	V	1	1	1	0	I	1	0	0	0		N0-2 (B0): S N3(B0): I N0 (B4): M N1-3(B4): I	5. N2 esc. Y [2]
V	1	1	1	0										
I	1	0	0	0										
Después de evento 5	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>V</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> </table> x-B0 <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>I</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> </table> y-B4	V	1	1	1	0	I	0	0	1	0		N0-2 (B0): S N3(B0): I N2 (B4): M N0,1,3(B4): I	6. N1 esc. Y [1]
V	1	1	1	0										
I	0	0	1	0										
Después de evento 6	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>V</td><td>1</td><td>1</td><td>1</td><td>0</td></tr> </table> x-B0 <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>I</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> </table> y-B4	V	1	1	1	0	I	0	1	0	0		N0-2 (B0): S N3(B0): I N1 (B4): M N0,2,3(B4): I	
V	1	1	1	0										
I	0	1	0	0										

Notas sobre la notación usada en la tabla:

- ✓ N0(B0):S significa que el bloque 0 (B0) está en estado Compartido (S) en la caché del nodo 0 (N0).
- ✓ N1-3(B0):I significa que el bloque 0 (B0) está en estado Inválido (I) en las cachés de los nodos 1, 2 y 3.

Problema 6. Se va a ejecutar en paralelo el siguiente código (inicialmente x e y son 0):

F1	F2
x=1;	y=1;
x=2;	y=2;
print y ;	print x ;

Indicar los resultados que se pueden imprimir si (considerar que el compilador no altera el código):

- (a) Se ejecutan F1 y F2 en un multiprocesador con consistencia secuencial.
- (b) Se ejecutan F1 y F2 en un multiprocesador basado en un bus que garantiza todos los órdenes excepto el orden W->R. Esto es debido a que los procesadores tienen buffer de escritura, permitiendo el procesador que las lecturas en el código que ejecuta adelantan a las escrituras que tiene su buffer. Obsérvese que hay varios posibles resultados.

Solución

El compilador no altera ningún orden garantizado ya que se supone, según el enunciado, que no altera el código.

F1	F2
(1.1) x=1;	(2.1) y=1;
(1.2) x=2;	(2.2) y=2;
(1.3) print y ;	(2.3) print x ;

- (a) Si F1 es el primero que imprime puede imprimir 0, 1 o 2, pero F2 podrá imprimir sólo 2. Esto es así porque se mantiene orden secuencial (los accesos a memoria del código que ejecuta un procesador se ven en el orden en el que están en dicho código) y, por tanto, cuando F1 lee y (instrucción (1.3) en el código), ya ha asignado a x un 2 (1.2) porque esta escritura está antes en el código que la lectura de y.

De igual forma, si F2 es el primero que imprime podrá imprimir 0, 1 o 2, pero P1 sólo puede imprimir 2. Esto es así porque se mantiene orden secuencial y, por tanto, cuando P2 lee x ((2.3) en el código), ha escrito ya en y un 2 (2.2) porque esta escritura está antes en el código que la lectura de x.

Se puede obtener como resultado de la ejecución las combinaciones que hay en cada una de las siguientes líneas:

F1 F2

- 0 2 (en este caso F1 imprime 0 y F2 imprime 2)
- 1 2
- 2 2
- 2 0
- 2 1

(b) Si no se mantiene el orden W->R, además de los resultados anteriores, los dos procesos pueden imprimir:

F1 F2

1	1 (en este caso F1 imprime 1 y F2 imprime 1)
0	1
1	0
0	0

Se pueden imprimir también estas combinaciones porque no se asegura que cuando un procesador ejecute la lectura de la variable que imprime print ((1.3) y (2.3) en los códigos) haya ejecutado las instrucciones anteriores que escriben en x (F1 en los puntos (1.1) y (1.2)) o en y (F2 en los puntos (2.1) y (2.2)). Esto es así porque no se garantiza el orden W->R y, por tanto, una lectura puede adelantar a escrituras que estén antes en el código secuencial. F1 puede leer y (1.3) antes de escribir en x 2 (1.2) o incluso antes de escribir en x 1 (1.1), por lo que F2 podría imprimir 0, 1 o 2, como F1, cuando F1 imprime primero. Igualmente, F2 puede leer x (2.3) antes de escribir en y 2 (2.2) o antes de escribir en y 1 (2.1), por lo que F1 podría imprimir 0, 1 o 2 como F2 cuando éste imprime primero. Todas las combinaciones son posibles.

Problema 7. Se va a ejecutar en paralelo el siguiente código (inicialmente x e y son 0):

<u>F1</u>	<u>F2</u>
<code>x=30;</code>	<code>while (flag==0) {};</code>
<code>y=40;</code>	<code>r1=x;</code>
<code>flag=1;</code>	<code>r2=y;</code>

Indicar qué datos puede obtener F2 en r1 y r2 si (considere que el compilador no altera el orden de los accesos a memoria del código):

- (a) Se ejecutan F1 y F2 en un multiprocesador con consistencia secuencial.
- (b) Se ejecutan en un multiprocesador con consistencia de liberación. Razone su respuesta.

Solución

- (a) Si se implementa consistencia secuencial, en r1 se almacena 30 y en r2 40.

Razonamiento:

Esto es así porque se garantizan los órdenes de acceso a memoria W->W y R->R que aparecen en el código que ejecuta un flujo:

- 1) En F1, al garantizarse el orden W->W, la escritura de 1 en flag no se realiza hasta que no se han realizado las escrituras en x e y que preceden a la escritura de flag en el código de F1.

2) Hasta que F2 no encuentra en flag un 1 no almacena en $r1$ el contenido de x ni en $r2$ el contenido de y . Al mantenerse el orden $R \rightarrow R$, en F2 no se adelanta la lectura de x ni la lectura de y a la lectura de flag en la última iteración del bucle.

Por tanto, como se garantiza $W \rightarrow W$ y además se garantiza $R \rightarrow R$, si F2 ve en flag un 1 y, por tanto, sale del bucle, va a ver al leer en x 30 y en y 40.

(b) Si se implementa consistencia de liberación se pueden dar los siguientes resultados

<u>r1</u>	<u>r2</u>
0	0
0	40
30	0
30	40

Razonamiento:

1) Al no garantizarse el orden entre accesos de escritura ($W \rightarrow W$), F1 podría escribir en flag un 1 antes de escribir 30 en x y/o 40 en y (obsérvese que la escritura $y=40$ podría también adelantar a $x=30$). Por lo que F2 podría leer en flag un 1 y, por tanto, salir del bucle, antes de que en x y/o en y se pudiera ver los valores que escribe F1 en estas variables (x y/o y podrían ser 0).

2) Al no garantizarse el orden entre accesos de lectura ($R \rightarrow R$), si se permite ejecutar lecturas cuya ejecución depende de una condición de salto antes de verificar que se cumple la condición, en F2 se podría, en la última iteración del bucle, adelantar la lectura de x o la lectura de y , o ambas a la de flag. En este caso F2 podría entonces leer los valores de x e y que tienen en su caché antes de que F1 los modifique y modifique flag. En estas circunstancias podría obtenerse en $r1$ o en $r2$ o en ambos un 0.

Problema 8. Se quiere implementar un cerrojo simple en un multiprocesador SMP basado en procesadores de la línea x86 de Intel, en particular, procesadores Intel Core. **(a)** Teniendo en cuenta el modelo de consistencia de memoria que ofrece el hardware de este multiprocesador ¿podríamos implementar la función de liberación del cerrojo simple usando “`mov k, 0`”, siendo k la variable cerrojo? Razona su respuesta. **(b)** ¿Cómo se debería implementar la función de liberación de un cerrojo simple si se usan procesadores Itanium? Razona su respuesta.

Solución

(a) La función de liberación la utiliza un flujo después de acceder a las variables compartidas para permitir que otros flujos puedan acceder a estas variables. Los procesadores de la línea x86 sólo relajan $W \rightarrow R$. Teniendo esto en cuenta se podría implementar con la escritura “`mov k, 0`” la función de liberación puesto que en el multiprocesador no se permite que una escritura adelante a una lectura o escritura anterior y el acceso a variables compartidas está antes que el código de liberación; por tanto, la liberación

del cerrojo (escritura) no va a realizarse antes de haber terminado los accesos (lectura y escritura) a las variables compartidas.

(b) El procesador Itanium implementa un modelo de ordenación que relaja todos los órdenes. No obstante, proporciona instrucciones para garantizar órdenes apropiados cuando resulta necesario; en particular, proporcionan una escritura con liberación que garantiza que no se escribe hasta que no hayan terminado los accesos a memoria que preceden a la instrucción de liberación. Para implementar la función de liberación de un cerrojo simple bastaría usar una instrucción de escritura en memoria con liberación (st.rel)

Problema 9. Se ha ejecutado el siguiente código en un multiprocesador con un modelo de consistencia que no garantiza W->R y W->W (garantiza el resto de órdenes):

```

(1) sump = 0;
(2) for (i=ithread ; i<8 ; i=i+nthread) {
(3)     sump = sump + a[i];
}
(4) while (Fetch_&_Or(k,1)==1) {};
(5) sum = sum + sump;
(6) k=0;

```

Conteste a las siguientes preguntas (considere que el compilador no altera el código y que la instrucción atómica de lectura-modificación-escritura no permite que lecturas posteriores adelanten a la escritura de la instrucción):

- (a) Indicar qué se puede obtener en sum si se suma la lista $a=\{1,2,3,4,5,6,7,8\}$. Se supone que k y sum son variables compartidas que están inicialmente a 0 (el resto de variables son privadas), nthread = 3, ithread es el identificador del thread o flujo de instrucciones en el grupo (0,1,2) que está colaborando en la ejecución. Si hay varios posibles resultados, se tienen que dar todos ellos. Justifique su respuesta.
- (b) ¿Qué resultados se pueden obtener si lo único que no garantiza el modelo de consistencia es el orden W->R? Justificar respuesta.

Solución

(a) No hay problemas con el código de sincronización de adquisición porque `Fetch_&_Or(k,1)` garantiza que la variable compartida sum se lee una vez que la escritura de 1 que cierra el cerrojo se ha realizado. Esto impide que el flujo que va a adquirir (cerrar) el cerrojo lea sum antes de que el flujo que tiene el cerrojo escriba en sum.

Al no garantizarse el orden W->W, la liberación del cerrojo, es decir la escritura de 0 en k, puede adelantar a los accesos a la variable compartida y, en particular, a la escritura de sum. Si esto ocurre, más de un flujo podría leer el mismo valor de sum, acumular a ese valor su variable local sump y escribir el resultado. En la variable sum acaba acumulado entonces, tras la escritura de los flujos que han leído lo mismo de sum, sólo el contenido de sump del último que ha escrito.

Para obtener los posibles resultados, primero hay que obtener lo que calcula cada flujo en `sump`. Teniendo en cuenta que el bucle `for` asigna iteraciones consecutivas a distintos flujos (turno rotatorio) el thread 0 obtiene $1+4+7=12$, el 1 obtiene $2+5+8=15$ y el 2 suma $3+6=9$.

Lectura de <code>sum</code>	Resultado	Comentario
Todos ven distinto valor	$12+15+9=36$	Si no hay problemas debido a que la escritura de liberación adelante a los accesos a <code>sum</code> anteriores
Todos leen 0 de <code>sum</code>	12	Si los tres flujos de control llegan a leer el valor 0 inicial de <code>sum</code> y el último que escribe en <code>sum</code> tras actualizar su valor es el thread 0
	15	Si los tres flujos de control llegan a leer el valor 0 inicial de <code>sum</code> y el último que escribe en <code>sum</code> es el thread 1
	9	Si los tres flujos de control llegan a leer el valor 0 inicial de <code>sum</code> y el último que escribe en <code>sum</code> es el thread 2
Si dos leen el mismo valor en <code>sum</code> , y el otro un valor distinto	$12+15=27$ ó $12+9=21$	<ul style="list-style-type: none"> - Si el flujo de control 0 ha logrado acumular primero sin problemas y el 1 y el 2 leen el valor que tiene <code>sum</code> tras la acumulación de 0. Si esto ocurre entonces 1 y 2 acceden al mismo valor, 12, le acumulan cada uno lo que han calculado (en <code>sump</code>) y escriben el resultado de la acumulación en <code>sum</code>. El resultado será 27 si el último que escribe es 1 y 21 si el último que escribe es 2. - Si los flujos 1 y 2 leen 0 los dos y acumulan su resultado parcial. Si de los dos escribe el último 1, entonces 0 sumará 12 a 15 obteniéndose 27. Si escribe el último 2, entonces 0 sumará 12 a 9, obteniéndose 21.
	$15+12=27$ ó $15+9=24$	<ul style="list-style-type: none"> - Si el flujo de control 1 ha logrado acumular primero sin problemas y el 0 y el 2 acceden al valor que tiene <code>sum</code> justo tras la acumulación de 1. Si esto ocurre entonces 0 y 2 acceden al mismo valor, 15, le acumulan cada uno su <code>sump</code> y escriben el resultado de la acumulación en <code>sum</code>. El resultado será 27 si el último que escribe es 0 y 24 si el último que escribe es 2. - Si los flujos 0 y 2 leen 0 los dos y acumulan su resultado parcial. Si de los dos escribe el último 0, entonces 1 sumará 15 a 12 obteniéndose 27. Si escribe el último 2, entonces 1 sumará 15 a 9, obteniéndose 24.
	$9+12=21$ ó $9+15=24$	<ul style="list-style-type: none"> - Si el flujo de control 2 ha logrado acumular primero sin problemas y el 0 y el 1 acceden al valor que tiene <code>sum</code> justo tras la acumulación de 2. Si esto ocurre entonces 0 y 2 acceden al mismo valor, 9, le acumulan cada uno su <code>sump</code> y escriben el resultado de la acumulación en <code>sum</code>. El resultado será 21 si el último que escribe es 0 y 24 si el último que escribe es 1. - Si los flujos 0 y 1 leen 0 los dos y acumulan su resultado parcial. Si de los dos escribe el último 0, entonces 2 sumará 9 a 12 obteniéndose 21. Si escribe el último 1, entonces 2 sumará 9 a 15, obteniéndose 24.

(b) En este caso la liberación del cerrojo no puede adelantar nunca a los accesos a la variable compartida `sum` porque la liberación es una escritura y no se admiten que las escrituras adelanten a accesos anteriores en el código. Por tanto, se hace el acceso a en exclusión mutua (secuencial) por parte de los tres flujos. Cualquier orden es posible. El único resultado posible sería la suma de todos los componentes de la lista porque se acumula correctamente en `sum` la suma parcial calculada en `sump` por todos los flujos de control: $12+15+9=36$.

Problema 10. ¿Qué ocurre si en el segundo código para implementar barreras visto en este Capítulo si se elimina la variable local, `cont_local`, sustituyéndola en los puntos del código donde aparece por el contador compartido asociado a la barrera `bar[id].cont`?

Solución

```

Barrera(id, num_flujos)
{
(1)    band_local= !(band_local)
(2)    lock(k);
(3)    ++bar[id].cont; //Sec. Crítica
(4)    unlock(k);
(5)    if (bar[id].cont == num_flujos) {
(6)        bar[id].cont=0;
(7)        bar[id].band=band_local;
    }
(9)    else while (bar[id].band != band_local) {};
}

```

Pueden surgir problemas pudiendo incluso no funcionar bien como barrera.

Si se usa el contador global en el `if` (línea de código (5)) que comprueba si contador es ya igual al número de flujos `num_flujos` que se sincronizan con la barrera, un flujo puede encontrar cuando llegue al `if` que el contador es igual a `num_flujos` sin ser el último que ha incrementado el contador. Esto puede provocar el siguiente problema:

Además del flujo que ha hecho el contador igual a `num_flujos`, otros que lo han incrementado antes pueden encontrar la condición del `if` verdadera y escribir, por tanto, en contador global y en la bandera global ((6) y (7)). Todas estas escrituras provocan accesos a través de la red para transferir el bloque de memoria que contiene la información sobre la barrera cuando una transferencia por cada variable a modificar sería suficiente. Estas transferencias adicionales simplemente devalúan prestaciones, nada más. El problema grave aparece si la barrera se vuelve a reutilizar por los mismos flujos y el SO suspende a uno de los flujos que encuentran contador igual a `num_flujos` antes de escribir un 0 en el contador global (6). Puede ocurrir entonces que, mientras este flujo está bloqueado, el resto de flujos salgan de la barrera y vuelvan a reutilizarla. Conforme los flujos van ejecutando de nuevo la barrera, incrementan el

contador global (3) y se quedan esperando a que éste llegue a ser igual a num_flujos (9). Pero nunca llegará a alcanzar este valor porque cuando el flujo suspendido (antes de (a6)) vuelve a ejecutarse pondrá a 0 el contador global (ejecutará (6)) perdiéndose la cuenta del número de flujos que están ya esperando en la barrera.

Problema 11. Simplifique el segundo código para barreras visto en clase suponiendo que se ha implementado para una arquitectura que dispone de instrucciones `Fetch&Add()`.

Solución

A continuación se pueden ver los cambios realizados:

```
barrera(id, num_flujos) {
    bandera_local = !(bandera_local) //se complementa bandera local
    cont_local = fecht&Add(bar[id].cont,1); //Se incrementa contador
    if (cont_local == num_flujos-1) { //Si han llegado todos los flujos ...
        bar[id].cont = 0; //... poner contador de flujos a 0 y ...
        bar[id].bandera = bandera_local; //... cambiar bandera para liberar ...
        //... flujos en espera
    }
    else while (bar[id].bandera != bandera_local) {}; //Espera si no ...
    //... han llegado todos los flujos
}
```

Se decrementa uno a num_flujos en el `if` porque `Fetch&Add()` devuelve el valor antes de la modificación, no después de la modificación. Por tanto, cuando llega el último flujo a la barrera devuelve `num_flujos-1`.

Problema 12. Se quiere paralelizar el siguiente bucle de forma que la asignación de iteraciones a los procesadores disponibles se realice en tiempo de ejecución (dinámicamente):

```
for (i=0; i<100; i++) {
    Código que usa i
}
```

Considerar que: (1) las iteraciones del bucle son independientes, (2) el único orden no garantizado por el sistema de memoria es W->R, (3) las instrucciones atómicas garantizan que escriben antes de que se puedan realizar lecturas posteriores y (4) el compilador no altera el código.

Teniendo esto en cuenta:

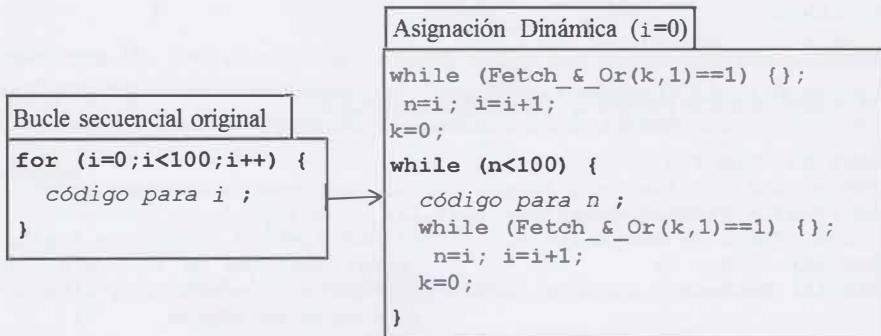
- (a) Paralelizar el bucle para su ejecución en un multiprocesador que implementa la primitiva `Fetch&Or()` para garantizar exclusión mutua.

- (b) Paralelizar el bucle en un multiprocesador que además tiene la primitiva Fetch&Add () .

Solución

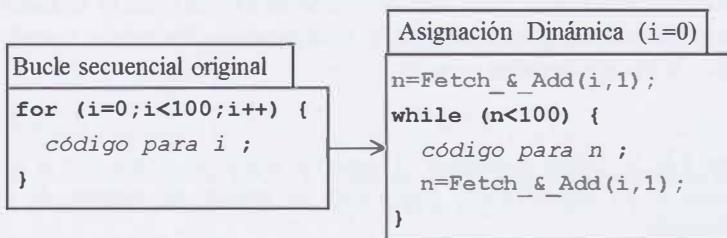
Se debe tener en cuenta que el único orden que no garantiza el hardware es el orden W->R.

- (a) Usando Fetch&Or ():



Se supone que el compilador no cambia de sitio $k=0$.

- (b) Usando Fetch&Add:



Problema 12. Un programador está usando el siguiente código para barreras (bar es un vector compartido, k es una variable compartida, el resto son variables locales, Fetch&Or (k, 1) () realiza su escritura antes que se ejecuten las lecturas posteriores):

```

Barrera(id, num_flujos)
{
    band_local= !(band_local)
    while (Fetch&Or(k,1)==1) {};
        cont_local = ++bar[id].cont;
    k=0;
    if (cont_local == num_flujos) {

```

```

        bar[id].cont=0;
        bar[id].band=band_local;
    }

    else while (bar[id].band != band_local) {};
}

```

Contestar razonadamente a las siguientes cuestiones (considere que el compilador no altera el código):

- ¿Funciona bien este código como barrera en un multiprocesador en el que lo único que no garantiza su modelo de consistencia es el orden W->R? ¿Por qué?
- Funciona bien este código como barrera en un multiprocesador con modelo de consistencia de memoria de ordenación débil? ¿Por qué?

Solución

```

Barrera(id, num_flujos)
{
    (1)   band_local= !(band_local)
    (2)   while (fetch_&_or(k,1)==1) {};
    (3)   cont_local = ++bar[id].cont; //SC
    (4)   k=0;
    (5)   if (cont_local == num_flujos)
    (6)       bar[id].cont=0;
    (7)       bar[id].band=band_local;
    (8)   }
    (9)   else while (bar[id].band != band_local)
    {};
}

```

Habrá problemas si dos o más flujos pueden estar al mismo tiempo ejecutando accesos a variables compartidas de la sección crítica; es decir, en este caso, accediendo a la variable compartida `bar[id].cont`. Esto podría ocurrir si:

- La apertura del cerrojo o escritura en `k` de 0 adelanta a los accesos anteriores a la variable compartida `bar[id].cont`, porque un flujo podría encontrar el cerrojo abierto estando otro flujo aún en la sección crítica (línea de código (3)), o
- El acceso a la variable compartida (lectura) adelanta a la operación atómica de escritura y lectura `Fetch&Or()`, porque el flujo que la adelanta accedería a la variable compartida (`bar[id].cont`) pudiendo estar otro flujo accediendo a ella. El enunciado del ejercicio dice que esto no puede ocurrir.

- Sí, funciona bien como barrera. La escritura en `k` de 0 (4) no puede adelantar a los accesos anteriores a la variable compartida `bar[id].cont` (3), puesto que la arquitectura no admite que una escritura pueda adelantar a accesos a memoria anteriores en el orden del programa. Por tanto, un flujo asigna un 0 a `k` cuando ya ha escrito en la variable compartida.

(b) No, no funciona bien como barrera. La escritura en k de 0 (4) puede adelantar a los accesos anteriores a la variable compartida $\text{bar}[\text{id}].\text{cont}$ (3), puesto que la arquitectura permite que una escritura pueda adelantar a lecturas o escrituras anteriores en el orden del programa. Por tanto, un flujo puede asignar un 0 a k cuando aún no ha escrito en la variable compartida y , por tanto, otro podría entrar en la sección crítica por encontrarse el cerrojo abierto y podría leer el mismo valor de $\text{bar}[\text{id}].\text{cont}$ que el flujo que le abrió el cerrojo. Si esto ocurre, este contador nunca llegaría a num_flujos y todos los procesos se quedarían esperando en el bucle (9).

Problema 14. Se quiere implementar un programa que calcule en paralelo la siguiente expresión en un multiprocesador en el que sólo se relaja el orden W->R y en el que sólo se dispone de primitiva de sincronización `Test&Set()`:

$$d = \frac{1}{N} \sum_{i=1}^N x_i^2 - \bar{x}^2, \text{ donde } \bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

Un programador ha implementado el código de abajo. El código lo ejecutan `nthread` flujos en paralelo. Respecto a las variables, `ithread` es una variable local que nota el identificador del flujo dentro del grupo de `nthread`; `i`, `med1` y `varil` son variables locales; `i`, `med`, `vari`, el vector `x` y `N` son variables compartidas; inicialmente `med`, `vari`, `med1` y `varil` son 0.

```

(1)  ...
(2)  for (i=ithread; i<N; i=i+nthread) {
(3)      med1=med1+x[i];
(4)      varil=varil+x[i]*x[i];
(5)  }
(6)  med = med + med1/N; vari = vari + varil/N;
(7)  vari= vari - med*med;
(8)  if (ithread==0) printf("varianza = %f", vari); //imprime
...

```

Contestar a las siguientes cuestiones (considere que el compilador no altera el código):

- Se ha ejecutado este código usando varios flujos y se ha visto que, aunque `N` y el vector `x` no varían, no siempre se imprime lo mismo. ¿Por qué ocurre esto?
- Añadir lo mínimo necesario para solucionar el problema teniendo en cuenta que sólo se dispone para implementar sincronización de `Test&Set()`. Indique qué variables son ahora compartidas y cuáles locales.
- Escribir el programa suponiendo que el multiprocesador además tiene instrucciones de sincronización `Fetch&Add()`. Indicar qué variables son compartidas y cuáles locales.

- (d) Escribir el programa ahora suponiendo que el multiprocesador sólo tiene instrucciones de sincronización `Compare&Swap()`. Indicar qué variables son compartidas y cuáles locales.

NOTA: En todos los apartados se puede añadir o quitar variables si se estima conveniente y se pide la implementación con mejores prestaciones

Solución

- (a) Hay varios tipos de errores en el código:

1. No se accede en exclusión mutua a las variables compartidas `med` y `vari` por parte de los diferentes flujos (línea de código (5)). Esto permite que puedan intentar varios flujos a la vez acumular el resultado parcial que han calculado (carrera). Por ejemplo, varias pueden leer el mismo valor de `med`, acumular a ese valor su variable local `med1` y escribir el resultado en `med`. El resultado acumulado en `med` será entonces el valor que han leído todos los flujos más el contenido de `med1` de sólo una de ellos, en particular, del último que ha escrito en `med`. Por lo que no se acumularía lo calculado por todos ellos.
2. Las operaciones de la línea (6) leen y modifican la variable compartida `vari`. Tal y como está el código, esa operación la realizan todos los flujos. Esto puede llevar a restar varias veces a `vari` el resultado de elevar al cuadrado `med`. Para evitar este problema se puede introducir la línea (6) dentro del `if` que acompaña al `printf` para que el flujo 0 sea el único que modifique esta variable compartida.
3. El flujo 0 obtiene el valor definitivo de `vari` a partir de `med` y `vari` (6) e imprime (7) sin esperar a que todos los flujos acumulen en las variables compartidas `med` y `vari` los resultados parciales que han obtenido en el bucle en las variables locales `med1` y `vari1`. Esto supone que, cuando imprime, pueden haber intentado acumular su resultado parcial el flujo 0 y una combinación del resto de flujos en un número de 0 a `nthread-1`. Por este motivo, aunque se accediera en exclusión mutua a las variables compartidas, se podrían imprimir distintos resultados en distintas ejecuciones dependiendo de cuántos flujos han realizado la acumulación antes de que el flujo 0 imprima el resultado.

- (b) Se accederá en exclusión mutua a `med` y `vari` (punto 1 de (a)) implementando un cerrojo simple con una variable compartida `k1`, se tiene que añadir una barrera antes de que imprima el flujo 0 (punto 3) y se introduce (6) dentro del `if` (punto 2). La barrera se debe implementar también usando un cerrojo simple (`k2`). El código resultante sería el siguiente:

```

(1)  for (i=ithread;i<N;i=i+nthread) {
(2)    medl=medl+x[i];
(3)    varil=varil+x[i]*x[i];
(4)  }
(5)  medl=medl/N; varil=varil/N;

      while (Test&Set(k1)) {}; //lock(k1) Punto 1
(5)  med = med + medl; vari = vari + varil;
      k1=0; //unlock(k1) Punto 1

      bandera_local= !(bandera_local) //Inicio Punto 3
      while (Test&Set(k2)) {}; //lock(k2)
      bar[id].cont+=1; cont_local = bar[id].cont;
      k2=0; //unlock(k2)
      if (cont_local ==num_flujos) {
          bar[id].cont=0;
          bar[id].bandera= bandera_local;
      }
      else while (bar[id].bandera!= bandera_local) {}; //Punt. 3

(6)  if (ithread==0) { vari= vari - med*med; //Punto 2
(7)                  printf("varianza = %f", vari);}

bandera_local, cont_local, ithread, i, medl y varil son variables
locales del flujo; k1, k2, bar[], med, vari, x[] y N son variables compartidas;
inicialmente k1, k2, med, vari, medl y varil son 0.

```

(c) Con Fetch&Add() no es necesario usar variables compartidas extras como cerrojos para el acceso en exclusión mutua:

```

(1)  for (i=ithread;i<N;i=i+nthread) {
(2)    medl=medl+x[i];
(3)    varil=varil+x[i]*x[i];
(4)  }
(5)  Fetch&Add(med,medl/N); Fetch&Add(vari,varl/N); //Punto 1

      bandera_local= !(bandera_local) //Inicio Punto 3
      cont_local = Fetch&Add(bar[id].cont,1);
      if (cont_local==num_flujos-1) {
          bar[id].cont=0;
          bar[id].bandera= bandera_local; }
      else while (bar[id].bandera!= bandera_local) {}; //Punt. 3

(6)  if (ithread==0) { vari= vari - med*med; //Punto 2
(7)                  printf("varianza = %f", vari);}

bandera_local, cont_local, ithread, i, medl y varil son variables
locales; bar[], med, vari, x[] y N son variables compartidas; inicialmente med,
vari, medl y varil son 0.

```

(d) Con Compare&Swap() no es necesario usar variables compartidas extras como cerrojos para el acceso en exclusión mutua:

```

(1)  for (i=ithread;i<N;i=i+nthread) {
(2)      medl=medl+x[i];
(3)      varil=varil+x[i]*x[i];
(4)  }
(5)  medl=medl/N; varil=varil/N;
    do                                //Inicio Punto 1
(5)      a = med;
(5)      b = a + medl;    //a y b son variables locales
      Compare&Swap(a,b,med);
      while (a!=b);
      do
(5)          a = vari;
(5)          b = a + varil;
          Compare&Swap(a,b,vari);
      while (a!=b);                  //Punto 1

bandera_local= !(bandera_local)          //Inicio Punto 3
do
    cont_local = bar[id].cont;
    b = cont_local + 1;
    Compare&Swap(cont_local,b,bar[id].cont);
    while (cont_local!=b);
    if (cont_local==num_flujos-1)  {
        bar[id].cont=0;
        bar[id].bandera= bandera_local; }
    else while (bar[id].bandera!= bandera_local) {};//Punt. 3

(6)  if (ithread==0) { vari= vari - med*med;      //Punto 2
(7)                  printf("varianza = %f", vari);}

a, b, bandera_local, cont_local, ithread, i, medl y varil son
variables locales; bar[], med, vari, x[] y N son variables compartidas; inicialmente
med, vari, medl y varil son 0.

```



Problema 15. Se ha extraído la siguiente implementación de cerrojo (*spin-lock*) para x86 del *kernel* de Linux (<http://lxr.free-electrons.com/source/arch/x86/include/asm/spinlock.h>):

```

typedef struct {
    unsigned int slock;
} raw_spinlock_t;

/*
/*Para un número de procesadores menor que 256=2^8
#ifndef (NR_CPUS < 256)
static __always_inline void __ticket_spin_lock(raw_spinlock_t *lock)
{
    short inc = 0x0100;

```

```

-   asm volatile (
-       "lock xaddw %w0, %1\n" /*w: se queda con los 16 bits menos
significativos*/
-       "1:                 \t" /*b: se queda con el byte menos
significativo*/
-       "cmpb %h0, %b0     \n\t" /*h: coge el byte que sigue al menos
significativo*/
-       "je 2f              \n\t" /*f: forward */
-       "rep ; nop          \n\t" /*retardo, es equivalente a pause*/
-       "movb %1, %b0       \n\t"
-       /* don't need lfence here, because loads are in-order */
-       "jmp 1b              \n" /*b: backward */
-       "2: "
-       : "+Q" (inc), "+m" (lock->slock) /*%0 es inc, %1 es lock->slock */
-   /*Q asigna cualquier registro al que se pueda acceder con rh: a, b, c y d;
ej. ah, bh ... */
-   :
-   : "memory", "cc");
- }

static __always_inline void __ticket_spin_unlock(raw_spinlock_t *lock)
{
    asm volatile( "incb %0" /*%0 es lock->slock */
    : "+m" (lock->slock)
    :
    : "memory", "cc");
}

```

Conteste a las siguientes preguntas:

- (a) Utiliza una implementación de cerrojo con etiquetas. ¿Cuál es el contador de adquisición y cuál es el contador de liberación?
- (b) Describir qué hace xaddw %w0, %1 ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?
- (c) Describir qué hace cmpb %h0, %b0 ¿opera con el contador de adquisición, con el de liberación o con los dos? ¿qué operaciones hace con ellos?
- (d) ¿Por qué se usa el prefijo lock delante de la instrucción xaddw?

NOTA: Se pueden ver detalles de la interfaz entre C/C++ y ensamblador en el manual de gcc (<http://gcc.gnu.org/onlinedocs/>)

Solución

- (a) lock->slock contiene el contador de liberación en los bits de 0 a 7 (lock->slock[7...0]) y el de adquisición en los bits de 8 a 15 (lock->slock[15...8]).
- (b) xaddw %w0, %1 almacena los 16 bits (sufijo w) menos significativos de lock->slock (%01) en los 16 bits menos significativos del registro al que se ha asigna inc (%0) y asigna a lock->slock (contador de adquisición y contador de liberación) el resultado de sumarlo con inc. Como consecuencia: (1) incrementa en uno el contador de adquisición (lock->slock[15...8]) dado que inc tiene un 1 en el bit 8 (inc

contiene 0x0100) y (2) almacena en `inc[15...8]` (%h0) el valor de este contador antes de la modificación y en `inc[7...0]` (%b0) el valor del contador de liberación (`lock->slock[7...0]`).

(c) `cmpb %h0, %b0` compara el valor actual del contador de liberación `inc[7...0]` (%b0) y el de adquisición `inc[15...8]` (%h0); es decir, resta ambos contadores modificando sólo el registro de estado. En las instrucciones posteriores se usa el resultado de la comparación (los bits de estado resultantes de la comparación). Si son iguales ambos contadores (bit z del registro de estado a 1), abandona la función `lock()` del cerrojo, y si son distintos actualiza el valor del contador de liberación cargando lo que hay en `lock->slock[7...0]` en `inc[7...0]` (%b0)

(d) Se requiere el prefijo `lock` para que la lectura y escritura en memoria que realiza la instrucción `xaddw` se hagan de forma atómica. Si `xaddw` no fuese atómica dos flujos de control podrían leer el mismo valor del contador de adquisición y, como consecuencia, más de un flujo podría entrar a la vez en una sección crítica.

7. Cuestiones

Cuestión 1. Diferencias entre núcleos con multithread temporal y núcleos con multithread simultánea.

Respuesta

Un núcleo con multithread simultánea es un núcleo superescalar que puede emitir a unidades funcionales instrucciones de múltiples threads o flujos de instrucciones distintos por lo que puede ejecutar operaciones de distintos threads en paralelo. Mientras que un núcleo con multithread temporal es un núcleo segmentado, superescalar o VLIW que ejecuta varios threads concurrentemente, pero no en paralelo (sólo emite instrucciones de un thread a unidades funcionales), lo que hace es multiplexar en el tiempo el uso de las etapas del cauce por parte de los threads. Para la multiplexación tiene hardware que se encarga de commutar entre threads.

Cuestión 2. Diferencias entre núcleos con multithread temporal de grano fino y núcleos con multithread temporal de grano grueso.

Respuesta

En un multithread temporal de grano fino (*FGMT- Fine Grain MultiThreading*) el hardware puede commutar de thread cada ciclo de reloj (con una penalización de 0 ciclos) (1) por turno rotatorio o (2) por un evento (dependencia funcional, fallo en caché de nivel 1, operación con CPI de varios ciclos, salto no predecible) combinado con

alguna técnica de planificación (p. ej. pasar a ejecutar el thread que lleva menos tiempo sin ejecutarse).

En un multithread temporal de grano grueso (CGMT- *Coarse Grain MultiThreading*) se conmuta de thread tras varios ciclos de reloj (con una penalización que puede ser distinta de 0) (1) trascurrido un nº de ciclos prestablecido o (2) por un evento estático (ejecución de una instrucción añadida al repertorio o ya incorporada, en este último caso podrían ser instrucciones de carga/almacenamiento o de salto) o dinámico (como el fallo de la caché de último nivel o una interrupción).

Cuestión 3. Se tiene un multiprocesador con protocolo MESI de espionaje. Si un controlador de caché observa en el bus un paquete de petición de lectura exclusiva de un bloque que tiene en estado S, debe (indicar cuál sería la respuesta correcta y razonar por qué es la respuesta correcta):

- Generar un paquete de respuesta con el bloque y pasar el bloque a estado I.
- Pasar el bloque a estado I.
- Generar un paquete de respuesta con el bloque y pasar el bloque a estado E.
- No tiene que hacer nada.

Respuesta

- Pasar el bloque a estado I.

Al estar su copia del bloque en estado Compartido no necesita entregar copia del bloque, porque lo tiene válido la memoria y será ella quien generará un paquete de respuesta con el bloque. Al ser la petición de acceso exclusivo al bloque, significa que quien ha enviado el paquete va a escribir en el bloque, por tanto, la copia que tiene el nodo ya no será válida, de ahí que pase a estado Inválido.

Cuestión 4. Se tiene un multiprocesador con protocolo MESI de espionaje. Si un nodo observa en el bus un paquete de petición de lectura exclusiva de un bloque que tiene en estado M, ¿qué debe hacer? Razonar respuesta.

Respuesta

- Generar un paquete de respuesta con el bloque, porque el paquete de petición incluye lectura del bloque y la memoria no tiene copia válida, él tiene la única copia válida en todo el sistema.
- Pasar el bloque a estado I, porque si se solicita un acceso exclusivo al bloque es porque quien ha enviado el paquete va a escribir en su caché en la copia que va a recibir, por tanto, la copia que está en estado M ya no estará actualizada, será inválida.

Cuestión 5. Suponga un multiprocesador con el protocolo MESI de espionaje. Si el procesador de un nodo escribe en un bloque que tiene en su caché en estado I, debe (indique cuál sería la respuesta correcta y razoné por qué es la respuesta correcta):

- Generar paquete de petición de acceso Exclusivo al bloque y pasar el bloque a M en su caché.
- Generar paquete de petición de acceso Exclusivo al bloque y pasar el bloque a estado E en su caché.
- Generar paquete de petición de acceso Exclusivo al bloque con lectura y pasar el bloque a estado E en su caché.
- Generar paquete de petición de acceso Exclusivo al bloque con lectura y pasar el bloque a M en su caché.

Respuesta

d) Generar paquete de petición de acceso Exclusivo al bloque con lectura y pasar el bloque a M.

Razonamiento:

- Paquete con lectura: como la caché del nodo no tiene copia válida del bloque (debido al estado Inválido), el controlador de caché del nodo tiene que generar un paquete que incluya lectura.
- Paquete con acceso exclusivo: como se va a escribir en la copia que se reciba del bloque, el controlador de caché del nodo tiene que pedir acceso exclusivo para que las copias del bloque en otras cachés sean invalidadas. Esto es necesario porque las siguientes lecturas del bloque que se realicen en otros nodos deberán acceder a la última modificación del mismo en lugar de acceder a copias no actualizadas del bloque que estén en sus cachés.
- Pasar a estado Modificado: como el nodo va a escribir en el bloque cuando lo reciba, será la única copia válida del bloque en el sistema y, por tanto, deberá estar en estado Modificado para que el controlador de caché sepa que debe responder con el bloque si se recibe alguna petición que incluya lectura del bloque y que debe escribirlo en memoria si se reemplaza el bloque en la caché.

Cuestión 6. ¿Cuál de los siguientes modelos de consistencia permite mejores tiempos de ejecución? Justificar respuesta.

- modelo de ordenación débil
- modelo implementado en los procesadores de la línea x86
- modelo de consistencia secuencial
- modelo de consistencia de liberación

Respuesta

d) modelo de consistencia de liberación

Tanto el modelo de ordenación débil como el de consistencia de liberación relajan todos los órdenes entre operaciones de acceso a memoria ($W \rightarrow R$, $W \rightarrow W$, $R \rightarrow W, R$) por lo que permitirán mejores tiempos de ejecución que el resto porque ningún acceso tiene que esperar a otro.

Pero el de liberación ofrece mejores prestaciones que el de ordenación débil porque:

1. El modelo de ordenación débil ofrece instrucciones máquina que permiten que, si S es una operación de sincronización, se garanticen los siguientes órdenes (W y L son operaciones de escritura y lectura, respectivamente):

- $S \rightarrow WR$ (hasta que no termina la operación de sincronización no puede empezar ningún acceso a memoria posterior)
- $WR \rightarrow S$ (hasta que no terminen los accesos a memoria que hay antes de una operación de sincronización no puede empezar la operación de sincronización)

Con estos órdenes los accesos a memoria que hay detrás de una sincronización no pueden empezar hasta que no hayan acabado todos los accesos que hay delante de la sincronización, independientemente de si se trata de una sincronización de adquisición o liberación. Por tanto, se garantiza el siguiente orden entre operaciones de acceso a memoria y operaciones de sincronización de adquisición SA y de liberación SL :

$WR \rightarrow SA \rightarrow WR \rightarrow SL \rightarrow WR$ (ver Figura 9)

Teniendo esto en cuenta se deduce que no hay ningún tipo de paralelismo entre operaciones de acceso a memoria antes y después de operaciones de sincronización.

2. El modelo de liberación ofrece instrucciones máquina menos restrictivas que permiten garantizar los siguientes órdenes entre accesos a memoria, lectura L y escritura W , y operaciones de sincronización de adquisición SA y liberación SL :

- $SA \rightarrow WR$ (hasta que no termina la operación de sincronización de adquisición no puede empezar ningún acceso a memoria posterior)
- $WR \rightarrow SL$ (hasta que no terminen los accesos a memoria que hay antes de una operación de sincronización de liberación no puede empezar la operación de sincronización de liberación)

Con estos órdenes los accesos a memoria que hay detrás de una sincronización de liberación pueden empezar aunque no hayan terminado los que hay delante, y los que hay delante de una operación de adquisición pueden terminar después de las que hay detrás (ver Figura 10). O sea, las operaciones de acceso a memoria que hay antes de la operación de adquisición se pueden realizar en paralelo a las operaciones que hay detrás de la adquisición y antes de la liberación, y las operaciones de acceso a memoria que hay

detrás de la operación de liberación se pueden realizar en paralelo a las que hay entre la operación de adquisición y la de liberación.



Cuestión 7. Indicar qué función no se corresponde con la serie (justificar respuesta):

- a) lock()
- b) Fetch&Or()
- c) Compare&Swap()
- d) Test&Set()

Respuesta

Hay una función software de cerros, lock(), y tres instrucciones máquina utilizadas para mejorar prestaciones en la sincronización de flujos de instrucciones: Fetch&Or(), Compare&Swap() y Test&Set().

Luego lock() no se corresponde con la serie.



8. Bibliografía

- S.V. Adve, K. Gharachorloo, 1996. “*Shared Memory Consistency Models: A Tutorial*”, IEEE Computer, vol. 19, nº 12, diciembre.
- ARM, 2015. “*ARM Architecture Reference Manual. ARMv8*.”
- Julio Ortega Lopera, Mancia Anguita López y Alberto Prieto, 2005. “*Arquitectura de Computadores*”. Thomson.

CAPÍTULO 4. MICROARQUITECTURAS CON PARALELISMO ENTRE INSTRUCCIONES (ILP)

1. Introducción

La tecnología de integración electrónica ha proporcionado circuitos integrados de mayor tamaño incluyendo transistores de menores dimensiones y más rápidos. La forma de aprovechar estas mejoras tecnológicas para aumentar la velocidad de los microprocesadores fundamentalmente ha venido de la mano de dos estrategias, el aumento de la frecuencia de reloj y el diseño de microarquitecturas capaces de completar cada vez más instrucciones por ciclo. La expresión de la velocidad de procesamiento de instrucciones que se obtiene a partir de la del tiempo de CPU, T_{CPU} , pone de manifiesto esta situación:

$$V_{CPU} = \frac{NI}{T_{CPU}} = \frac{NI}{NI \times CPI \times T_{ciclo}} = \frac{1}{CPI \times T_{ciclo}} = IPC \times F \quad (1)$$

donde, como se ha visto en el Capítulo 1, NI es el número de instrucciones, CPI el número de ciclos por instrucción, IPC el número de instrucciones terminadas por ciclo ($IPC=1/CPI$), T_{ciclo} el tiempo de ciclo de reloj, y F la frecuencia de reloj ($F=1/T_{ciclo}$). Así, el incremento en la frecuencia de reloj (F) y en el número de instrucciones finalizadas por ciclo (IPC) han proporcionado incrementos exponenciales en la velocidad de los microprocesadores según lo establecido en la ley de Moore hasta que, en los primeros años del siglo XXI, las dimensiones de los transistores y el número de transistores incluidos en los circuitos dieron lugar a una cierta interrelación entre las mejoras en frecuencia de reloj y en IPC , al tiempo que la disipación de potencia podría alcanzar valores inaceptablemente altos. Entonces se empezó a plantear la consecución de mejoras de prestaciones en los microprocesadores a través de la integración de varios procesadores (también denominados núcleos de procesamiento, o *cores*), y empiezan a aparecer los microprocesadores multinúcleo o *multicore*.

En este Capítulo se analizan los conceptos que permiten diseñar procesadores (o núcleos, o *cores*) capaces de terminar varias instrucciones por ciclo gracias al aprovechamiento del *paralelismo entre instrucciones* (ILP). Las microarquitecturas de estos procesadores se basan en la segmentación de cauce (o *pipelining*), según la que el procesador se diseña a partir de una serie de etapas que procesan de forma independiente las distintas fases por las que pasa una instrucción. Por ejemplo, existe una etapa encargada de captar instrucciones (IF, de *Instruction Fetch*), otra etapa de decodificación (ID, de *Instruction Decode*) que toma las instrucciones captadas por la etapa IF, las decodifica y capta sus operandos del banco del registros del procesador y las pasa a la siguiente etapa de ejecución (EX, de *Execution*), y así sucesivamente. En una microarquitectura de procesador segmentado, las etapas definen un *cauce* donde cada etapa trabaja independientemente a partir de las instrucciones que le proporciona la etapa anterior y genera resultados para la etapa posterior. Por lo tanto, se

está aprovechando el paralelismo de instrucciones (ILP) dado que varias instrucciones se procesan al mismo tiempo, aunque se encuentran en distintas fases de su procesamiento. Si cada etapa puede procesar una fase de la instrucción y cada etapa tarda un ciclo en realizar su trabajo, pasado el tiempo de latencia inicial necesario para que la primera instrucción pase por todas las etapas, el procesador podrá terminar una instrucción por ciclo, en lugar de requerir varios ciclos por instrucción (tantos ciclos como etapas existan en el procesador). No obstante, hay que tener en cuenta que pueden existir *dependencias de datos* entre instrucciones, instrucciones de salto condicional (*dependencias de control*), y *colisiones* entre instrucciones que necesitan el mismo recurso en un momento determinado. Estas dependencias y colisiones provocan esperas de las instrucciones en las etapas que, al no producir resultados en esos ciclos, reducen el rendimiento del cauce.

Como se ha dicho antes, las mejoras tecnológicas han permitido diseñar microarquitecturas, con cauces más complejos, en los que las etapas pueden procesar más de una instrucción por ciclo. En este Capítulo nos centraremos en las dos alternativas fundamentales de diseño de procesadores capaces de completar más de una instrucción por ciclo: los procesadores superescalares, y los procesadores VLIW (de *Very Long Instruction Word*). En los procesadores superescalares es fundamentalmente el hardware el que se encarga de reordenar las instrucciones, renombrar sus operandos, predecir los saltos, etc. para alcanzar un uso eficiente del cauce y poder conseguir rendimientos lo más próximos al valor del IPC máximo que proporciona la microarquitectura. En cambio, en los procesadores VLIW es el compilador el que debe ordenar las instrucciones para evitar dependencias y utilizar eficientemente el cauce. Dado que en el momento de la compilación no se dispone de toda la información de las condiciones en las que se procesan las instrucciones, se han propuesto algunas estrategias para mejorar el aprovechamiento de los cauces en arquitecturas VLIW, muchas de ellas están relacionadas con el denominado procesamiento especulativo.

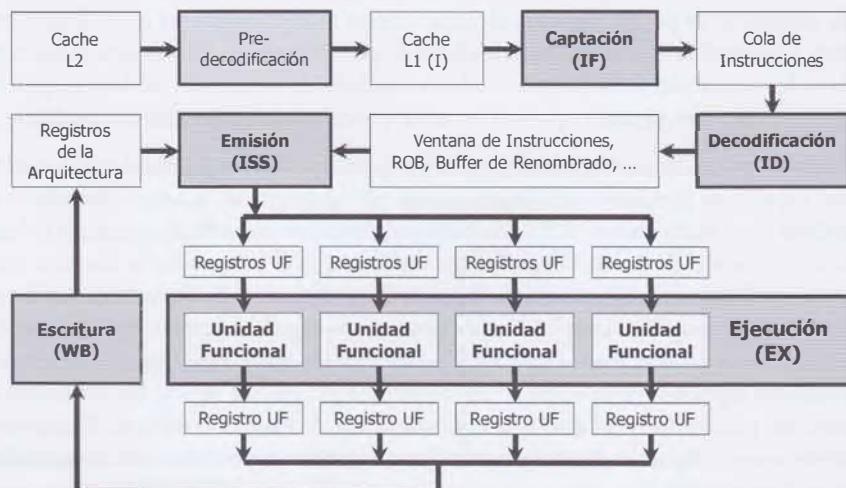


Figura 1. Esquema de un cauce superescalar

En el esquema anterior (Figura 1) se presentan los detalles fundamentales del aprovechamiento del paralelismo entre instrucciones a través de procesamiento segmentado que realizan los procesadores superescalares y los VLIW.

2. Procesadores Superescalares

Un procesador superescalar es un procesador segmentado que puede finalizar más de una instrucción por ciclo y que posee recursos hardware para extraer el paralelismo entre instrucciones dinámicamente. La Figura 1 proporciona un esquema con las principales etapas de un procesador superescalar, junto con los recursos más relevantes que se utilizan para gestionar las dependencias entre instrucciones, renombrando los operandos y reordenando la ejecución de las operaciones que codifican, para finalizar un número de instrucciones por ciclo lo más próximo al IPC máximo que puede alcanzar el procesador. Todas las etapas de un procesador superescalar pueden procesar varias instrucciones por ciclo. Para aprovechar al máximo las posibilidades de procesamiento de instrucciones en paralelo que proporcionan las distintas etapas, el procesador incluye una serie de elementos como ventanas de instrucciones o estaciones de buffers, buffers de renombramiento, buffers de reordenamiento, etc. que se describen en las secciones siguientes. En primer lugar se describirá el procesamiento de las instrucciones a través del cauce para considerar posteriormente la forma de gestionar las instrucciones de salto.

2.1 Procesamiento de instrucciones en un cauce superescalar

Para explicar la actividad de las distintas etapas del cauce superescalar utilizaremos la secuencia de instrucciones de la Tabla 1 como ejemplo. En la última columna de la Tabla 1 se indican los ciclos que necesita la unidad funcional correspondiente para ejecutar la operación (en la etapa EX de la Figura 1).

Tabla 1. Esquema de un cauce superescalar

Instrucción		Significado	Ciclos
1	add r3, r1, r2	r3 := r1+r2	1
2	mul r3, r3, r4	r3 := r3 * r4	3
3	sw (r5), r3	M(r5) := r3	1
4	lw r5, (r6)	r5 := M(r6)	2
5	add r6, r5, r4	r6 := r5+r4	1
6	add r5, r3, r4	r5 := r3+r4	1

En la Figura 1 se ha incluido una etapa de *predecodificación* que existe en algunos procesadores. Esta etapa añade algunos bits a las instrucciones cuando pasan desde la memoria principal o desde niveles de caché inferiores (L2, L3, etc.) a la memoria caché de instrucciones en el primer nivel (L1). Con ello se consigue acelerar el proceso de decodificación posterior, o incluso que algunas instrucciones complejas (como las de salto) puedan empezar a procesarse ya desde su captación. Después, la etapa de captación de instrucciones (IF) toma varias instrucciones de la memoria caché de instrucciones y

las pasa a la cola de instrucciones. Las instrucciones se captan según el orden en que se encuentran en la memoria (que es el orden en el que están en el código), y se introducen en ese mismo orden en la cola de instrucciones, desde donde se transfieren a la unidad de decodificación (ID) para decodificarse en el mismo orden. Desde ahí pasan a una estructura denominada ventana de instrucciones o estación de reserva. En la Figura 2 se muestran los ciclos que tardan en captarse y decodificarse las instrucciones de la Tabla 1 y los contenidos de la cola de instrucciones en esos ciclos. Se está suponiendo que la etapa IF es capaz de captar tres instrucciones por ciclo y la etapa ID puede decodificar dos instrucciones por ciclo.

Instrucción		1	2	3	4	..
1	add r3, r1, r2	IF	ID			..
2	mult r3, r3, r4	IF	ID			..
3	sw (r5), r3	IF		ID		..
4	lw r5, (r6)		IF	ID		..
5	add r6, r5, r4		IF		ID	..
6	add 5, r3, r4		IF		ID	..

Cola de instrucciones

1	add	r3, r1, r2	sw	(r5), r3	add	r6, r5, r4
2	mult	r3, r3, r4	lw	r5, (r6)	add	r6, r5, r4
3	sw	(r5), r3	add	r6, r5, r4		
4			add	r5, r3, r4		
5						
Tras ciclo 1			Tras ciclo 2		Tras ciclo 3	

Figura 2. Captación y decodificación de instrucciones y evolución de la cola de instrucciones

Una vez decodificadas, las instrucciones pasan a la *ventana de instrucciones* desde donde se emiten a las unidades funcionales que las ejecutan, cuando tienen sus operandos disponibles y la unidad funcional está libre. Un ejemplo de ventana de instrucciones se muestra en la Figura 3 las instrucciones se introducen ordenadamente en las líneas libres de la ventana de instrucciones. Cada línea tiene un campo para indicar el código de operación (que permitirá emitir la instrucción a la unidad funcional donde se pueda ejecutar), otro campo para indicar el lugar donde se almacenará el resultado de la operación (el registro destino), y dos campos por operando. El campo “Operando” almacena el contenido del operando cuando el campo OK (campo de operando válido) está a 1, o el lugar desde donde se accederá al operando cuando esté disponible si el campo OK está a 0 (campo de operando no válido). Así, las dos primeras líneas de la ventana de la Figura 3 almacenan las instrucciones 1 y 2 de la Tabla 1 que han sido decodificadas en el ciclo (2). Los operandos de la instrucción 1 se han captado del banco de registros dado que los registros r1 y r2 no están siendo utilizados por instrucciones anteriores. Por eso, en los campos Operando 1 y Operando 2 se encuentra el contenido de dichos registros, [r1] y [r2], y los campos OK1 y OK2 están a 1. El resultado de la instrucción 1 se escribirá en el registro r3. La instrucción 2 tiene el operando 2 disponible, por eso en el campo Operando 2 se tiene el contenido, [r4], y el campo

OK2 está a 1. Sin embargo, como el otro operando es el registro r_3 , no estará disponible hasta que se ejecute la instrucción add. Por esta razón, en el campo Operando 1 se encuentra el código que identifica el registro r_3 , y el campo OK1 está a 0.

#	Cod. Op.	Reg. Dest.	Oper. 1	OK1	Oper.2	OK2
1	add	r_3	$[r_1]$	1	$[r_2]$	1
2	mult	r_3	r_3	0	$[r_4]$	1
3						
...

Figura 3. Ejemplo de ventana de instrucciones o estación de reserva

El estado de la ventana de instrucciones de la Figura 3 corresponde al final del ciclo (2), una vez la etapa ID ha finalizado la decodificación de las instrucciones 1 y 2. En el ciclo siguiente se puede iniciar la ejecución de la instrucción 1 (add) dado que tiene sus operandos disponibles y el sumador está libre. Es decir, en el ciclo (3) se ha emitido la instrucción 1 que se ejecutará en dicho ciclo (3). Como la instrucción de suma consume un ciclo, al final del ciclo (3) se tendrá el resultado de $[r_1] + [r_2]$, y por lo tanto, el contenido de Operando 1 que faltaba en la instrucción 2. Así, al final del ciclo (3), el estado de la ventana de instrucciones se muestra en la Figura 4. La instrucción mult podrá empezar a ejecutarse en el siguiente ciclo dado que tiene sus dos operandos disponibles y la unidad de multiplicación está libre. También se han introducido en la ventana las instrucciones 3 y 4 que se han decodificado durante el ciclo (3) en la unidad ID. En el campo de registro destino de la instrucción sw no se indica ningún registro, dado que el destino es la memoria. En la instrucción lw solo se utilizan los campos correspondientes a un operando.

#	Cod. Op.	Reg. Dest.	Oper. 1	OK1	Oper.2	OK2
1	mult	r_3	$[r_3]$	1	$[r_4]$	1
2	sw		$[r_5]$	1	r_3	0
3	lw	r_5	$[r_6]$	1
4						
...

Figura 4. Ventana de instrucciones tras el ciclo (3) del ejemplo

El estado de la ventana tras el ciclo (4) se muestra en la Figura 5. La instrucción mult se está ejecutando y se habrán introducido las instrucciones 5 y 6 que han sido decodificadas precisamente en el ciclo (4).

#	Cod. Op.	Reg. Dest.	Oper. 1	OK1	Oper.2	OK2
1	sw		$[r_5]$	1	r_3	0
2	lw	r_5	$[r_6]$	1	---	---
3	add	r_6	r_5	0	$[r_4]$	1
4	add	r_5	r_3	0	$[r_4]$	1
...

Figura 5. Ventana de instrucciones tras el ciclo (4) del ejemplo

La evolución de la ventana de instrucciones que se pone de manifiesto en las Figuras 4 y 5 corresponde a una *emisión ordenada* de instrucciones. Es decir, las instrucciones se empiezan ejecutar en el mismo orden en que se encuentran en el programa. De esta manera, la instrucción de carga de memoria (lw) no se puede empezar a ejecutar a pesar de que tiene sus operandos (en este caso un único operando) válidos y la unidad de acceso a memoria está libre. Si la instrucción lw se pudiera empezar a ejecutar a pesar de que la instrucción sw está bloqueada, el procesador implementaría *emisión desordenada*.

En cualquier caso, entre las instrucciones de acceso a memoria, sw y lw, hay que tener en cuenta un detalle importante derivado del hecho de que se trata de instrucciones de acceso a memoria. La instrucción sw escribe en la dirección [r5] y la instrucción lw lee de la dirección [r6]. Si $[r5]=[r6]$, la instrucción lw debería esperar a que se complete la instrucción sw dado que habría una dependencia de tipo RAW. En este caso el problema se podría evitar dado que se dispone de los contenidos de las direcciones en el momento de emitir la instrucción lw antes que sw, y se podría evitar la emisión de lw si fueran iguales. Si, aunque no se conozca si las direcciones son iguales o no, se permite que la instrucción lw adelante a la sw, se habla de *adelantamiento especulativo*, dado que se está suponiendo que es poco probable que puedan coincidir las direcciones de memoria. Por supuesto, el procesador debe disponer de recursos que le permitan deshacer el efecto de la instrucción lw adelantada si finalmente las direcciones de memoria fueran iguales. Más adelante describiremos algunos ejemplos más del procesamiento especulativo que implementan los procesadores superescalares y los VLIW.

Las microarquitecturas de los procesadores superescalares han tendido a distribuir la ventana de instrucciones a través de las denominadas *estaciones de reserva*. De esta forma se dispone de espacio suficiente para almacenar las instrucciones mientras esperan a que se pueda iniciar su ejecución, pero distribuyendo este almacenamiento en estructuras más pequeñas (menos líneas para almacenar instrucciones cada una), con menos complejidad hardware y con un acceso más rápido. Así, en lugar de disponer de una única ventana de instrucciones donde, tal y como se ha visto, se introducen las instrucciones tras ser decodificadas y desde donde se emiten a las distintas unidades funcionales, existen varias estaciones de reserva. En cada una de ellas solo se introducen las instrucciones que se ejecutarían en algunas de las unidades funcionales del procesador. Por ejemplo, puede haber una estación de reserva por cada unidad funcional, y tras la decodificación, cada instrucción se introduciría en la estación de reserva asignada a la unidad funcional donde se ejecutaría la operación codificada por la instrucción. También pueden existir estaciones de reserva a las que se asignan varias unidades funcionales, donde se introducirían las instrucciones que se ejecutarían en ellas. Por ejemplo, es relativamente frecuente que existan dos estaciones de reserva en el procesador, una que permite acceder a las unidades funcionales que operan con datos en coma flotante y otra para los que operan con datos enteros. Además de otras dos estaciones de reserva para las instrucciones de acceso a memoria, una para las instrucciones de carga de datos desde memoria y otra para las escrituras. Los campos de las líneas de las estaciones de reserva son similares a los de la ventana de instrucciones de la Figura 3, aunque en el caso de

que se tenga una estación de reserva por unidad funcional, no haría falta el campo de código de operación (Cod.Op. en la Figura 3). En realidad, la ventana de instrucciones corresponde al caso de disponer de una única *estación de reserva centralizada*. Cuando el procesador dispone de varias estaciones de reserva, la etapa de decodificación, ID, también se debe encargar de seleccionar la estación de reserva que almacenará la instrucción hasta que pueda comenzar su ejecución. Por esta razón, si el procesador incluye varias estaciones de reserva en realidad tendrá un etapa de decodificación y emisión, ID/ISS, donde ISS proviene del término inglés *issue* que suele utilizarse para “emisión”, y el inicio de la ejecución de las instrucciones desde la estación de reserva correspondiente se denominará “envío” (traducción del término inglés *dispatch*). En el caso de que se tenga una ventana de instrucciones, o estación de reserva centralizada, seguiremos hablando de la etapa para la decodificación (ID), y de la etapa de emisión (ISS) que inicia la ejecución de la instrucción en la unidad funcional que le corresponda.

Instrucción		1	2	3	4	5	6	7	8	9	10	..
1	add r3, r1, r2	IF	ID	EX								..
2	mult r3, r3, r4	IF	ID		EX	EX	EX					..
3	sw (r5), r3	IF		ID				EX				..
4	lw r5, (r6)		IF	ID					EX	EX		..
5	add r6, r5, r4		IF		ID						EX	..
6	add r5, r3, r4		IF		ID						EX	..

(a)

Instrucción		1	2	3	4	5	6	7	..
1	add r3, r1, r2	IF	ID	EX					..
2	mult r3, r3, r4	IF	ID		EX	EX	EX		..
3	sw (r5), r3	IF		ID				EX	..
4	lw r5, (r6)		IF	ID	EX	EX			..
5	add r6, r5, r4		IF		ID		EX		..
6	add r5, r3, r4		IF		ID			EX	..

(b)

Figura 6. Traza de procesamiento hasta la ejecución de las instrucciones de la Tabla 1: (a) emisión ordenada; (b) emisión desordenada

En la Figura 6 se muestra la traza de procesamiento de la secuencia de instrucciones de la Tabla 1 hasta que se han emitido y ejecutado todas. Se ha supuesto que el procesador dispone de tantas unidades funcionales como sea necesario para que no haya colisiones (hay dos sumadores), aunque solo hay una unidad de acceso a memoria para cargar o almacenar datos. La Figura 6(a) corresponde a la emisión ordenada (se puede ver que la etapa de ejecución, EX, de una instrucción empieza como muy pronto en el mismo ciclo que la instrucción anterior) y la Figura 6(b) a la emisión desordenada (hay instrucciones que empiezan a ejecutarse en ciclos anteriores al momento en que empieza la ejecución de instrucciones que las preceden en el código). Como se puede observar en la Figura 6, la emisión desordenada consigue que la ejecución de las instrucciones termine antes. No obstante, para conseguir que la ejecución de las instrucciones se produzca como muestra

la Figura 6(b) el procesador debe implementar algunas estrategias que consideramos a continuación.

Por un lado, entre las instrucciones existen dependencias de datos. Así, podemos tener dependencias de datos de tipo RAW (Read-After-Write), WAR (Write-After-Read) y WAW (Write-After-Write). En las dependencias RAW, una instrucción necesita un operando (Read) después (After) de que sea generado (Write) por otra instrucción previa, tal y como ocurre entre las instrucciones 2 y 1, 3 y 2, y 5 y 4 de la Tabla 1. Para solucionar este tipo de riesgo, ya hemos visto más arriba que se bloquean las instrucciones en la ventana de instrucciones hasta que no tengan sus operandos disponibles.

En cuanto a las dependencias WAR y WAW, realmente no son verdaderas dependencias de datos, ya que aparecen debido al uso repetido que ha hecho el compilador del mismo registro como destino de diferentes instrucciones. Se pueden evitar usando diferentes registros para almacenar los resultados. Por ejemplo, entre las instrucciones 4 y 3, y entre las instrucciones 6 y 5 existen dependencias WAR causadas en ambos casos por el uso de $r5$. Así, la instrucción 4 escribirá en el registro $r5$ que está siendo utilizado por la instrucción 3 para la dirección en donde va a almacenar el dato correspondiente. Lo mismo ocurre entre las instrucciones 6 y 5, también debido a que la instrucción 6 va a escribir el resultado en $r5$, que es utilizado por la instrucción 5 para uno de sus operandos. Por otra parte, también existe una dependencia WAW entre las instrucciones 2 y 1 a causa del uso repetido del registro de destino $r3$. Si la instrucción 2 no escribe su resultado en $r3$ después de que lo haga la instrucción 1, las siguientes instrucciones no utilizarán el valor correcto de $r3$. Sin embargo, para evitar estos dos tipos de riesgos WAR y WAW bastaría con que las instrucciones 2, 4 y 6 escribieran sus resultados en otros registros, por ejemplo en $r7$, $r8$ y $r9$. Esta técnica se denomina *renombramiento* de registros.

El renombramiento de registros puede ser aplicado por el propio compilador al asignar los registros de la arquitectura, pero también puede implementarse en hardware. Esto es lo usual en los procesadores superescalares, donde se incluyen estructuras de buffers con una serie de campos específicos. Un ejemplo de este tipo de *buffers de renombramiento* se muestra en la Figura 7, donde cada una de sus líneas tiene unos campos con una función asociada. Así, el campo *entrada válida* indica, cuando está a 1, que esa línea se está utilizando para hacer un renombrado de registro; el campo *registro de destino* indica el registro que se ha renombrado en esa línea; el campo *valor* es el que corresponde al espacio reservado para el renombrado (la nueva ubicación para el dato asociada al registro de destino usado en la instrucción); el campo *valor válido* indica si el contenido de lo almacenado en el campo *valor* es válido; y finalmente, el campo *último* indica si esa línea contiene el último renombramiento que se ha hecho para el registro. El *buffer de renombramiento* se consulta al captar los operandos para pasarllos a la ventana de instrucciones (o a la estación de reserva correspondiente en su caso). Antes de comprobar si los operandos están en el banco de registros se comprueba si en el buffer de renombramiento se ha hecho ningún renombramiento del registro que se busca. En el caso de que no haya renombramientos del registro, el operando se toma del banco de registros.

#	Entrada Válida	Registro de Destino	Valor	Valor Válido	Último
1	1	r3	351	1	0
2	1	r3	-	0	1
3	1	r5	-	0	1
4					

Figura 7. Buffer de renombramiento de acceso asociativo tras el ciclo (3)

La Figura 7 muestra el estado del buffer de renombramiento tras el tercer ciclo del procesamiento de las instrucciones de la Tabla 1 teniendo en cuenta que, al comenzar el procesamiento de las instrucciones, el buffer se encuentra vacío. Las dos primeras líneas muestran los renombramientos del registro r3 que se hicieron tras el ciclo 2 (una vez decodificadas esas instrucciones). La instrucción 1, add r3, r1, r2, da lugar a un renombramiento para el registro r3 en la línea 1, accede al banco de registro para captar los registros r1 y r2 y podría empezar a ejecutarse. Al final del ciclo (3) habrá terminado la ejecución y el resultado se escribe en el campo de *Valor* de la línea 1 del buffer (se ha supuesto que el resultado es 351) y el campo se *Valor Válido* se pone a 1. La instrucción 2, mul r3, r3, r4, también se termina de decodificar en el ciclo (2) y da lugar a otro renombramiento para r3, ahora en la línea 2. Como es el último renombramiento que se ha hecho al registro r3, el campo Último de la línea 2 está a 1, y el de la línea 1 está a cero. La instrucción tiene como operandos al registro r4, cuyo valor se captaría del banco de registros, y el registro r3. En este caso se comprobaría que se ha hecho un renombramiento (en la línea 1) para ese registro y que el valor no está disponible (el campo de *Valor Válido* se pondrá a 1 al final del ciclo 1, cuando termine la ejecución de la suma). Por esta razón, la instrucción 2 quedaría en la ventana de instrucciones. En el campo de operando correspondiente a r4 tendría el valor de éste, [r4], captado del banco de registros y el campo de OK correspondiente estará a 1. Sin embargo en el otro campo de operando habrá una marca que apunta a la línea 1 del buffer de renombramiento, y el campo de OK correspondiente estaría a 0. Al terminar la ejecución de la instrucción 1, al mismo tiempo que se escribe el resultado en la línea 1 del buffer de renombramiento, también se escribirá en los campos de operando de la ventana de instrucciones que tengan su campo de OK a cero y la marca correspondiente a la línea 1 del buffer de renombramiento en el campo de operando. Así, la instrucción 2 puede empezar a ejecutarse después del ciclo (3).

Al final del ciclo (3) se habrán decodificado las instrucciones 3 y 4. La instrucción 3 no da lugar a ningún renombramiento dado que escribe en memoria. El registro r5 no ha sido renombrado y se capta del banco de registros. En cambio, el registro r3 ha sido renombrado por última vez en la línea 2, y en el campo de operando de la ventana de instrucciones correspondiente se pondrá una marca que apunta a la línea 2 y se pone el campo de OK correspondiente a 0. Hasta que no se produzca el resultado de la multiplicación al final del ciclo (6) no se tendrá ese valor esperado.

Finalmente, la instrucción 4 accede al registro r6 en el banco de registros dado que no se ha renombrado, y da lugar a un renombramiento del registro r5. Si la emisión es desordenada podría emitirse. El buffer de renombramiento que se ha descrito se denomina

buffer de acceso asociativo porque para determinar si hay renombrado de un registro y encontrar la línea en la que se ha hecho, se accede asociativamente utilizando como campo de búsqueda el campo de *Registro Destino*. En el caso de que haya varias líneas con el mismo registro destino, se utiliza la línea en la que el campo de *Último* esté a 1.

Para completar la descripción del procesamiento de las instrucciones en un cauce superescalar faltan los aspectos relacionados con el momento en que la instrucción, una vez ha terminado de ejecutarse, actualiza sus resultados en el banco de registros del procesador y, por lo tanto, termina de procesarse. Independientemente de que las instrucciones en un procesador superescalar finalicen su ejecución (en la etapa EX de la Figura 1) de forma desordenada (con un orden diferente al que tienen en el código), existen dos alternativas para la finalización del procesamiento de las instrucciones que, por otra parte, definen el denominado *modelo de consistencia del procesador*: modelo con *finalización desordenada* o con *finalización ordenada*. En la primera alternativa, lo mismo que las instrucciones se terminan de ejecutar desordenadamente, y por lo tanto generan sus resultados también desordenadamente, también escriben dichos resultados en el banco de registros y finalizan su procesamiento desordenadamente. La segunda alternativa, sin embargo, actualiza los resultados de las instrucciones y su procesamiento en el mismo orden en que las instrucciones aparecen en el código, independientemente de que los resultados se hayan generado en otro orden. La finalización ordenada es la más frecuente en los microprocesadores más recientes dado que se ha comprobado que en el aprovechamiento del paralelismo influye más el hecho de que la ejecución de las instrucciones se haga desordenadamente, en el momento en que los operandos y los recursos que necesitan estén disponibles, que el orden en que las instrucciones escriban sus resultados en el banco de registros y abandonen el cauce. Por otra parte, la finalización ordenada se puede implementar gracias a una estructura relativamente sencilla y versátil (aunque bastante exigente en cuanto a recursos hardware) que es el denominado *buffer de reordenamiento* (ROB). Así, además de para la finalización ordenada, el ROB también puede utilizarse para implementar el renombramiento de registros (el ROB serviría como buffer de renombramiento) e incluso, como veremos, se aprovecha en el procesamiento de las instrucciones de salto.

#	Cod. Oper.	Registro Destino	Valor	OK	Unidad	Pred	Flush	Estado
1								
2								
3	add	r3	351	1		0	0	f
4	mult	r3	--	0	mult	0	0	x
5	sw	--	--	0		0	0	i
6	lw	r5	--	0	carga (load)	0	0	x
7	add	r6	--	0		0	0	i
8	add	r5	--	0		0	0	i
9								
10								

Figura 8. Buffer de reordenamiento tras el ciclo (4) en la traza de emisión desordenada (Figura 6(b))

La Figura 8 muestra una posible configuración para un ROB. Las líneas que lo constituyen se configuran como una cola circular (en la Figura 8 la línea 1 va después de la número 10 y viceversa) con dos punteros. Un puntero señala a la línea del ROB en la que se introducirá la próxima instrucción que llegue desde la unidad de decodificación (en la Figura 8 apuntaría a la línea 9 del ROB), y el otro indica la instrucción del ROB a la que le corresponde escribir su resultado en el banco de registros y abandonar el procesador (apuntaría a la línea 3 del ROB en la Figura 8). Las instrucciones se introducen ordenadamente en el ROB desde la etapa de decodificación (ID o ID/ISS) y también abandonan ordenadamente el ROB (*se retiran* del ROB) al escribir sus resultados en los registros. En el ROB se podrán introducir tantas instrucciones por ciclo como instrucciones se decodifiquen por ciclo en el procesador. El número de instrucciones que puedan escribir sus resultados en el banco de registros y retirarse es una característica del procesador y da una idea de la velocidad pico que puede tener el procesador: el número de instrucciones que pueden retirarse por ciclo multiplicado por la frecuencia de reloj nos daría el número máximo de instrucciones que podrían procesarse por unidad de tiempo.

El ROB de la Figura 8 tiene 10 líneas que configuran un buffer circular. Las instrucciones se han ido introduciendo ordenadamente en el buffer tras los ciclos en los que se van decodificando y están almacenadas en posiciones correlativas en el ROB a partir de una determinada línea (a partir de la línea 3 en el ejemplo de la Figura 8). Las instrucciones se van retirando ordenadamente desde la línea 3 y las nuevas instrucciones que se vayan introduciendo lo hacen a partir de la línea 9. Cuando se haya introducido una instrucción en la línea 10, las siguientes se utilizan a partir de la línea 1. Si se llena el ROB se detiene la decodificación de instrucciones hasta que queden líneas libres al retirarse instrucciones. Los campos de las líneas del ROB que se muestran en la Figura 8 tienen el siguiente significado, de izquierda a derecha: número de línea del ROB (no hace falta que esté implementado como tal en la línea); código de la operación a ejecutar para cada instrucción (*Cod. Oper.*); registro del banco de registros del procesador donde se debe escribir el resultado de la instrucción cuando se retire (*Registro Destino*); campo donde se almacenará temporalmente el resultado de la operación hasta que la instrucción se retire (*Valor*); campo que indica si el valor almacenado para el dato en la línea del ROB es válido (*OK*); unidad funcional que se utiliza para ejecutar la operación que codifica la instrucción (*Unidad*); campo para indicar si la instrucción se introdujo en el cauce como resultado de una predicción (*pred*); y campo para indicar que la instrucción se retira sin almacenar el resultado en el registro (*flush*). También se ha incluido un campo de estado que indica si la instrucción ha finalizado su ejecución (*f*), se encuentra ejecutándose (*x*), o se encuentra en la estación de reserva esperando a ser enviada para su ejecución (*i*). Para que una instrucción pueda retirarse, tanto dicha instrucción como todas las que estén en líneas anteriores del ROB tienen que tener a 1 el campo valor válido, *OK* (el campo de estado será igual a *f*) o el campo *flush* igual a 1. Los campos *pred* y *flush* se utilizan en la gestión de las instrucciones de salto condicional, como veremos

en la Sección 4.1.2, y también en la gestión de las interrupciones. En la descripción que se proporciona a continuación no hay que tenerlos en cuenta.

El estado del ROB corresponde al final del ciclo (4) de la traza correspondiente a la emisión desordenada de las instrucciones de la Tabla 1 (Figura 6(b)). Al finalizar ese ciclo (4) la instrucción add que está en la línea 3 del ROB habrá escrito el resultado en el campo de *Valor* de esa línea del ROB y se podrá retirar en el siguiente ciclo. La instrucción de multiplicación está en la línea 4 del ROB y ha empezado a ejecutarse en el multiplicador. La instrucción de almacenamiento está en la línea 5 del ROB y espera en la ventana de instrucciones (o estación de reserva centralizada) el resultado de la multiplicación. La instrucción de carga de memoria que se encuentra en la línea 6 del ROB, tal y como habíamos supuesto en la traza de la Figura 6(b), puede adelantar a la instrucción de almacenamiento y se está ejecutando (la unidad de carga de memoria está realizando el acceso a memoria para captar el dato. Las instrucciones de suma que están en las líneas 7 y 8 del ROB están esperando en la ventana de instrucciones a tener sus operandos y poder ser emitidas para su ejecución. Las instrucciones que están esperando en la ventana de instrucciones (o en su caso en las estaciones de reserva) no tienen información (en la línea del ROB en la que se encuentran) acerca de la unidad que van a utilizar para ejecutarse. El campo de *flush* es igual a 0 en todas las instrucciones, como explicaremos al hablar de la gestión de las instrucciones de salto. En la Figura 9 se muestra el estado de la ventana de instrucciones al final del ciclo (4), es decir, en el mismo ciclo al que corresponde el estado del ROB de la Figura 8. Como se puede ver en la Figura 9, en los campos de *Reg.Dest* de las instrucciones de suma que están en las líneas 2 y 3 se encuentran los números de línea del ROB donde están esas instrucciones y en cuyo campo de *Valor* se almacenarán temporalmente los resultados, respectivamente las líneas 7 y 8 del ROB. En los campos de operando de la ventana de instrucciones cuyos bits de *OK* están a 0 (esperan que se genere el valor del operando correspondiente) se encuentran los números de línea del ROB donde están las instrucciones que producirán esos resultados. Así, las instrucciones *sw* (línea 1 de la ventana de instrucciones) y *add* (línea 3 de la ventana) necesitan el operando *r3* producido por la instrucción de multiplicación que está en la línea 4 del ROB. La instrucción *add* (línea 2 de la ventana) necesita el operando *r5* producido por la instrucción de acceso a memoria que está en la línea 6 del ROB. Cuando se generen esos resultados, al mismo tiempo que se almacenan en el campo *Valor* del ROB, también se almacenarán en los campos que almacenan el índice de la línea del ROB correspondiente. Como se puede ver, el ROB permite implementar el renombrado de registros, el campo *Valor* es precisamente el almacenamiento alternativo (el renombramiento) del registro que se indica en el campo *Registro Destino* de esa línea del ROB. No hace falta el campo de *Último* que se indica en la Figura 7 dado que las instrucciones se introducen ordenadamente en el ROB y el último renombrado de un registro se produce en la línea del ROB donde se encuentre la instrucción más reciente que utilice ese registro.

#	Cod. Op.	Reg. Dest.	Oper. 1	OK1	Oper.2	OK2
1	sw	--	[r5]	1	4	0
2	add	7	6	0	[r4]	1
3	add	8	4	0	[r4]	1
...

Figura 9. Ventana de instrucciones tras el ciclo (4) en la traza de emisión desordenada (Figura 6(b))

En la Figura 10 se muestra la traza correspondiente a la emisión desordenada para la secuencia de instrucciones de la Tabla 1. Incluye las etapas que se hemos notado como ROB y WB que corresponden, respectivamente, a la escritura, en la correspondiente línea del ROB, del resultado generado al ejecutarse la instrucción, y a la retirada de la instrucción del ROB con la escritura del contenido del campo *Valor* en el registro de destino correspondiente del banco de registros de la arquitectura. Las instrucciones de escritura en memoria acceden a ésta en la etapa de ejecución (EX en la Figura 1), no necesitan etapa de escritura de resultados en el ROB, y en la etapa WB tampoco generan escrituras en el banco de registros. En la Figura 10 se ha supuesto que se pueden escribir en el ROB dos resultados por ciclo como máximo y que se pueden retirar como máximo dos instrucciones que escriban resultados en el banco de registros. Es decir, las instrucciones de almacenamiento no cuentan, y por eso en el ciclo 8 se retiran tres instrucciones (una es de almacenamiento). Como se puede ver en la Figura 10, la finalización es ordenada: si se ordenan los índices de los ciclos en los que se retiran las instrucciones del código según la posición de las instrucciones en el código se obtiene una serie creciente. En la Figura 10 las instrucciones se retiran en los ciclos 5, 8, 8, 8, 9, 9 (sin embargo terminan de ejecutarse, desordenadamente, en los ciclos 3, 6, 7, 4, 5, 7).

Instrucción			1	2	3	4	5	6	7	8	9
1	add	r3, r1, r2	IF	ID	EX	ROB	WB				
2	mult	r3, r3, r4	IF	ID		EX	EX	EX	ROB	WB	
3	sw	r5), r3	IF		ID				EX	WB	
4	lw	r5, (r6)		IF	ID	EX	EX	ROB		WB	
5	add	r6, r5, r4		IF		ID		EX	ROB		WB
6	add	r5, r3, r4		IF		ID			EX	ROB	WB

Figura 10 Traza de procesamiento de las instrucciones de la Tabla 1 con emisión desordenada

2.2. Procesamiento de las instrucciones de salto

Los procesadores superescalares son procesadores segmentados. Por esta razón, las instrucciones de salto y las interrupciones tienen un efecto muy pernicioso en sus prestaciones puesto que pueden ocasionar cambios en la secuencia de instrucciones que hay que procesar en el cauce: no solo puede que se hayan malgastado ciclos procesando instrucciones que no tenían que haber entrado en el procesador, sino que además hay que corregir los efectos que esas instrucciones hayan podido tener en el estado del

procesador. Además, en un procesador superescalar, el número de instrucciones que se captan y decodifican en cada ciclo puede ser considerable (dos o más por ciclo). Por lo tanto: (1) la probabilidad de que haya que procesar instrucciones de salto en un ciclo aumenta, y (2) el número de instrucciones que se pueden introducir incorrectamente en el procesador también es elevado.

En cuanto a las instrucciones de salto, el principal problema lo plantean las instrucciones de *salto condicional*. En el caso de las instrucciones de *salto incondicional*, mediante una etapa de *predecodificación* se podría marcar convenientemente esas instrucciones para que, en la misma etapa IF de captación, se puedan empezar a procesar (calcular la dirección de destino del salto) y se capte la instrucción de destino de salto ya en el ciclo siguiente. En las instrucciones de salto condicional se podría, igualmente, adelantar el cálculo de la dirección de destino del salto, pero hasta que el procesador no determine el valor de la condición no sabe si se va a producir el salto o no. La técnica usual para el procesamiento de las instrucciones de salto condicional es la *predicción de saltos*, que se basa en determinar la alternativa (saltar/no saltar) más probable, y continuar el procesamiento, tras la instrucción de salto, con la secuencia de instrucciones que corresponde a dicha opción más probable. Cuando la condición de salto se evalúa, se comprueba si la predicción que se había hecho era correcta o no. En el caso de que no sea correcta habrá que retomar el procesamiento de instrucciones a partir de la primera instrucción de la alternativa que no se tomó y deshacer el efecto de las instrucciones que se hayan introducido en el cauce incorrectamente. En este caso, por tanto, existe una cierta penalización. Obviamente, la eficacia de este procedimiento se basa en la capacidad para predecir correctamente el resultado de la instrucción de salto antes de que ésta se termine de ejecutar (para lo que hace falta que se haya evaluado la condición de la que depende).

Existen procedimientos de predicción estáticos en los que el procesador actúa a partir de características de la propia instrucción de salto. Así, según el procedimiento, se tiene en cuenta la dirección del salto (si es hacia direcciones anteriores o posteriores a la dirección de la instrucción de salto), la condición de salto utilizada (saltar si igual a cero, si mayor que cero, etc.), o incluso se permite que el compilador ponga a uno o a cero un bit determinado del código máquina de la instrucción de salto para elegir si se predice saltar o no saltar (por ejemplo en base al comportamiento de saltos/no saltos observado en simulaciones de trazas del código). En todos estos casos, la predicción de saltos se especifica a través del código que se va a ejecutar y las predicciones no tienen en cuenta el comportamiento real del código al ejecutarse con las entradas correspondientes (la *dinámica* de ejecución del código).

Las técnicas de predicción dinámica tienen en cuenta la historia del comportamiento pasado de cada instrucción de salto condicional para determinar el comportamiento más probable cada vez que dicha instrucción entre en el cauce. Para ello se utiliza una estructura, denominada usualmente BTB (del inglés *Branch Target Buffer*). En las líneas del BTB se introduce información de las instrucciones de salto del programa en ejecución que ya se han procesado con anterioridad en dicha ejecución. Así, en cada

línea, junto con la dirección de la instrucción de salto, también se almacenan una serie de bits que codifican la historia la ejecución del salto en las últimas iteraciones y, según las implementaciones, también se suelen incluir campos para la dirección de destino de salto si éste debe producirse, o incluso la propia instrucción de destino de salto, para acelerar así el acceso a dicha instrucción en el caso de que la predicción sea saltar.

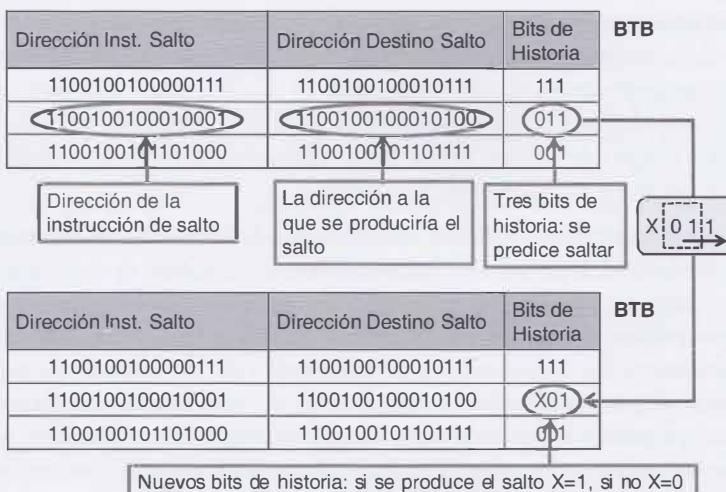


Figura 11. Esquema del funcionamiento de un predictor de salto con tres bits de historia

Cuando una instrucción llega al cauce se comprueba si hay información de ella en el BTB buscando (asociativamente) si su dirección se encuentra en el campo correspondiente del BTB. Si es la primera vez que se procesa la instrucción, no estará incluida en el BTB, por lo que (1) se hace una predicción de salto prefijada inicialmente (por ejemplo una predicción estática teniendo en cuenta si el salto es hacia atrás o hacia adelante, o simplemente una predicción por defecto utilizando un valor inicial establecido para los bits de historia), (2) se le asigna una línea del BTB y (3) el campo de bits de historia tomará los valores correspondientes al resultado, acertado o no, de la predicción. En el caso de que la instrucción de salto que entra en el cauce ya esté incluida en el BTB, se realiza la predicción según los bits de historia, e igualmente se actualizarán dependiendo de si se ha acertado o no en la predicción. La Figura 11 describe un procedimiento de predicción dinámica de salto basado en tres bits de historia. Estos tres bits proporcionan información del comportamiento (de salto/no salto) de la instrucción en las tres últimas veces en que se ha procesado: uno significa que se produjo el salto y cero que no se produjo. Así, 011 significa que la última vez que se procesó la instrucción de salto condicional no se produjo salto, pero las dos veces anteriores sí se dio lugar a salto. Como en los bits de historia 011 hay mayoría de veces que indican que se produjo salto, la predicción que producen será la de *Tomar el Salto*. El nuevo valor que toman los bits de historia será X01 donde X es 0 si finalmente no se produjo el salto, ó 1 si se produjo el salto. Los bits 01 en X01

provienen de desplazar a la derecha los bits de historia que había (011). Así el valor 0 que antes se refería a la última vez que se procesó la instrucción de salto ahora informará de lo que ocurrió la penúltima vez, y el valor 1 que indicaba lo que ocurrió en la penúltima vez, ahora se refiere a la antepenúltima vez. Obviamente el bit 1 menos significativo en 011 se pierde dado que solo se tiene en cuenta el comportamiento de salto/no salto de la antepenúltima vez que se procesó la instrucción de salto. Dada la importancia de disponer de un esquema de predicción con elevados porcentajes de acierto, ha habido un trabajo de investigación bastante activo en este ámbito y se han propuesto una gran cantidad de procedimientos de predicción que, en muchos casos necesitan cantidades de memoria considerables para los bits de historia. En (J. Ortega, M. Anguita, 2005) se puede encontrar más información acerca de otros procedimientos de predicción de saltos que se utilizarán en los problemas de este capítulo.

La recuperación tras una predicción incorrecta implica continuar el procesamiento a partir de la dirección de la primera instrucción de la secuencia alternativa de instrucciones, que no se ejecutó debido a la predicción que finalmente resultó incorrecta, y deshacer el efecto que pudieran haber tenido las instrucciones que se introdujeron en el cauce de forma incorrecta. La determinación de la dirección de la instrucción a partir de la que continuar el procesamiento se puede acelerar si, en el caso de que la predicción fuese “saltar”, se guarda la dirección de la instrucción siguiente a la de salto, o si, en el caso de que la predicción fuese “no saltar”, se calcula la dirección de destino de salto al mismo tiempo que se están procesando las instrucciones predichas, y aunque no sea necesario disponer de dicha dirección, y guardarla. En cuanto a deshacer el efecto de las instrucciones incorrectamente procesadas, el ROB permite llevar a cabo esta tarea con relativa facilidad. Así, como se muestra en la Figura 8, el ROB puede incluir en cada línea un campo de predicción, *pred*, que permite marcar las instrucciones que se han introducido *especulativamente* en el cauce tras la predicción de un salto (incluso se puede indicar el tipo de predicción, T o NT que se haya hecho). Además, existe otro campo de un bit frecuentemente denominado bit de vaciado o de *flush*. Este bit de *flush* se activa (por ejemplo, se pone a 1) en todas las instrucciones que se introdujeron especulativamente tras la instrucción de salto, en el momento en que se resuelve la condición de salto y, al procesar la instrucción de salto, se determina que la predicción era incorrecta. Todas las instrucciones marcadas con el campo *pred* (todas las que se habían introducido especulativamente en el cauce) se retiran del ROB cuando sea su turno y no realizan ninguna modificación en el procesador si el bit de *flush* está activado.

3. Procesadores VLIW

Un VLIW es un procesador segmentado que puede terminar más de una operación por ciclo en el que el compilador es el principal responsable de agrupar operaciones que puedan procesarse en paralelo para definir instrucciones que, de esta forma, se codifican a través de las denominadas palabras de instrucción larga (LIW, de *Long Instruction Word*)

o muy larga (VLIW, de *Very Long Instruction Word*). Mientras que en un procesador superescalar, cada una de las operaciones de una instrucción VLIW se codificaría mediante una única instrucción y las dependencias entre ellas se pueden comprobar en el propio cauce. En un procesador VLIW, gracias al trabajo del compilador, las operaciones empaquetadas en una instrucción VLIW son independientes y se emiten a mismo tiempo a las unidades funcionales sin más comprobación. Así, mientras que en un procesador superescalar, la planificación de instrucciones es *dinámica*, en un procesador VLIW es *estática*.

Esto contribuye a simplificar la microarquitectura del procesador VLIW con respecto al superescalar ya que, al ceder al compilador la responsabilidad de la planificación de las operaciones, los procesadores VLIW pueden implementarse con microarquitecturas más sencillas que las de los procesadores superescalares dado que no se necesitarían estructuras como estaciones de reserva, buffers de renombramiento o buffers de reorden. Por esta razón los procesadores VLIW se contemplaron como una alternativa a las microarquitecturas superescalares para conseguir aumentar el número de operaciones que se pueden completar por unidad de tiempo, con un menor consumo energético que el que se tendría con las microarquitecturas, cada vez más complejas, de los procesadores superescalares. En los procesadores VLIW, el incremento en transistores que proporcionan las mejoras en la tecnología electrónica se aprovecharía para aumentar el número de unidades funcionales que pueden trabajar en paralelo. No obstante, el compilador debe ser capaz de encontrar, en los programas, un número suficientemente elevado de operaciones que puedan ejecutarse en paralelo para aprovechar las unidades funcionales disponibles. Para ello, necesita aplicar ciertas estrategias, algunas de las cuales requieren incluir ciertos recursos hardware en la microarquitectura del procesador para aliviar la limitación esencial que surge de disponer, en el momento de la compilación, de menos información sobre los detalles de la ejecución. Además, frente a un procesador superescalar, un procesador VLIW tiene la desventaja de una menor portabilidad de los códigos entre procesadores que, aunque miembros de una misma familia, presenten ciertas diferencias en la microarquitectura, ya que el compilador debe tener en cuenta las características concretas del cauce para la que se esté generando código (el número de unidades funcionales, sus tiempos de latencia, etc.).

La Figura 12 muestra un esquema del cauce de un procesador VLIW. La unidad de captación IF tomaría una o varias instrucciones VLIW y las pasaría a la cola de instrucciones. Dado que una instrucción VLIW codifica varias operaciones que dan lugar a varias instrucciones de los códigos escalares propios de los procesadores superescalares, aunque el procesador VLIW capte una sola instrucción VLIW por ciclo, esto es equivalente al acceso a varias instrucciones escalares en un procesador superescalar. Desde la cola de instrucciones, se toman las instrucciones VLIW y pasan a la unidad de decodificación, ID, donde se determinan las operaciones que se emiten a cada uno de los *slots*, o *huecos de emisión*. En la Figura 12 se ha supuesto que hay cuatro *huecos*, y por lo tanto las instrucciones VLIW codifican cuatro operaciones. Cada una de esas operaciones debe

poder ejecutarse en alguna de las unidades funcionales que están asignadas a cada *hueco*. En la Figura 12, a cada *hueco* se ha asignado una única unidad funcional.

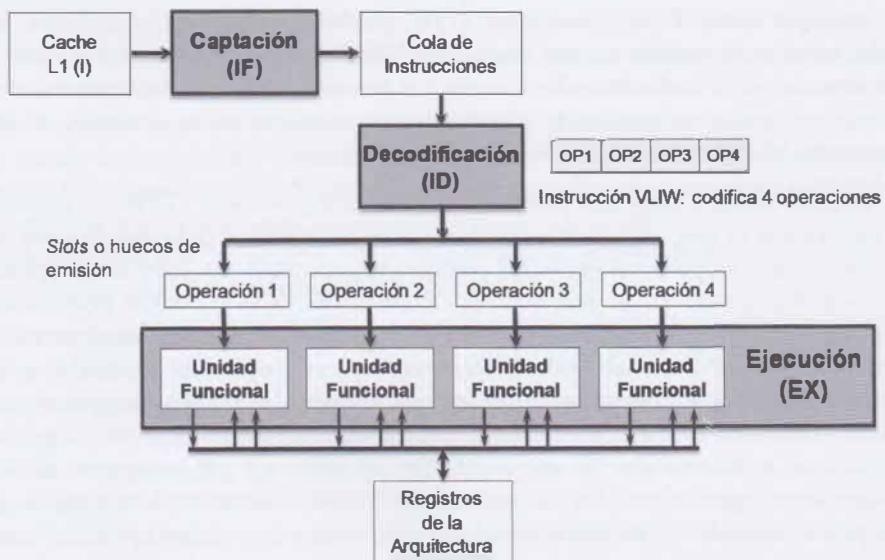


Figura 12 .Esquema de un cauce de procesador VLIW

Por lo tanto, para aprovechar eficientemente los recursos del procesador VLIW, el compilador debe encontrar tantas operaciones independientes como *huecos* de emisión existan (operaciones codificadas en las instrucciones VLIW) y, además, cada una de esas operaciones debe poderse ejecutar en las unidades funcionales a las que puede accederse desde cada *hueco*. Por otra parte, si no se encuentra una operación independiente para un hueco determinado, debe insertarse una operación de tipo *nop* (no operar), dando lugar a códigos de programa VLIW poco densos que, si no se implementa alguna estrategia adecuada, necesitan más memoria para almacenarse que los equivalentes para un procesador superescalar. En las siguientes secciones se describen tres aproximaciones complementarias para incrementar el paralelismo que el compilador extrae de los códigos, las *técnicas software* para el aumento del tamaño de los denominados *bloques básicos*, las *instrucciones con predicado*, y el *procesamiento especulativo*.

3.1. Técnicas software para ampliar los bloques básicos

Los riesgos de control asociados a instrucciones de salto condicional dificultan la planificación del código por parte del compilador dado que, en tiempo de compilación, no se podría conocer con total certeza qué comportamiento tendrán los saltos del programa. Por lo tanto, puede resultar difícil encontrar operaciones independientes para completar las instrucciones VLIW. Cuantos menos saltos aparezcan en el código, más sencilla será la planificación.

Así, si se denomina *bloque básico* al conjunto de operaciones situadas entre dos instrucciones de salto, el compilador debe encontrar la planificación más adecuada para las operaciones de cada uno de los bloques básicos del programa. A continuación se describen algunas técnicas software para aumentar el número de instrucciones independientes de los bloques básicos y facilitar con ello la planificación local de instrucciones que debe hacer el compilador. Estas técnicas son el desenrollado de bucles y la segmentación software (*software pipelining*). Para explicarlas partiremos del código de la Figura 13 que corresponde a la suma de un número, a , a los n elementos de un vector, x , de números en coma flotante de 64 bits (8 bytes) que se encuentran en memoria, en posiciones consecutivas. Las primeras tres instrucciones del código de la Figura 13 inicializan los registros $r1$, $f0$, $r3$, que sirven, respectivamente, para indicar el número de iteraciones que faltan, guardar en $f0$ el elemento a , y apuntar a los elementos del vector. Después empieza el bucle, que se repite n veces. El cuerpo del bucle constituye un bloque básico en este caso dado que no contiene instrucciones de salto.

```

inicio:    lw      r1, n    ; carga en r1 el número de iteraciones
           ld      f0, a    ; carga en f0 el valor de a
           add    r3,r0,r0  ; hace r3 igual a 0 (índice del vector)
bucle:    ld      f2,x(r3) ; f2 se carga con el elemento x[i]
           add    f4,f2,f0  ; f4 = x[i] + s
           sd      x(r3),f4  ; guardar en x[i] el resultado(f4)
           subi   r1,r1,#1  ; decrementar el número de elementos
           addi   r3,r3,#8  ; Apuntar al siguiente elemento del vector
           bnez   r1,bucle ; Si quedan elementos se salta

```

Figura 13. Ejemplo de código (instrucciones escalares): suma de número a los elementos de un vector

El compilador debe utilizar las operaciones que codifican las instrucciones escalares del código, para construir las correspondientes instrucciones VLIW. Para ello necesita encontrar las operaciones que pueden ejecutarse en paralelo y tener en cuenta las características del procesador VLIW: cuántos *huecos* o *slots* tienen las instrucciones VLIW, qué tipo de operaciones puede poner en cada hueco, y cuál es el retardo de cada operación. En las descripciones que se van a realizar a continuación se considera un procesador VLIW con tres *huecos* de emisión, uno permite el acceso a una ALU donde se ejecutan las operaciones lógicas y aritméticas con enteros y los saltos, otro para las de coma flotante, y otro para los accesos a memoria. La Figura 14 muestra una planificación de las operaciones del código de la Figura 13 teniendo en cuenta la configuración del procesador VLIW y las dependencias entre las instrucciones. Se deben respetar las dependencias RAW entre las operaciones correspondientes a la instrucción $add f4, f2, f0$ con las operaciones de acceso a memoria $ld f0, a$ y $ld f2, x(r3)$; entre $bnez r1, bucle$ y $subi r1, r1, #1$; entre $addi r3, r3, #8$ y $add r3, r0, r0$; y entre $sd x(r3), f4$ y $add f4, f2, f0$. Las operaciones correspondientes a las instrucciones $sd x(r3), f4$ y $addi r3, r3, #8$, se pueden emitir al mismo tiempo (y pueden colocarse en *slots* distintos de la misma instrucción VLIW) debido a que la escritura en el registro $r3$ por parte de la

instrucción `addi r3, r3, #8` se hace después de que las dos operaciones hayan accedido al valor que tenía `r3` antes y no se viola la dependencia WAR dentro de la instrucción. En el código de la Figura 14 quedan bastantes *slots* que deben dejarse sin operaciones (se cargan con códigos `nop` de “no operar”).

Etiqueta	<i>slot 1: Op. Mem.</i>	<i>slot 2: Op. ALU</i>	<i>slot 3: Op. FP</i>
inicio	ld f0,a	add r3,r0,r0	nop
	lw r1,n	nop	nop
bucle	ld f2,x(r3)	nop	nop
	nop	subi r1,r1,#1	add f4,f2,f0
	nop	bnez r1,bucle	nop
	sd x(r3),f4	addi r3,r3,#8	nop

Figura 14. Código VLIW con tres slots para el ejemplo de la Figura 13

Para estimar el tiempo que tarda en ejecutarse el código de la Figura 14 hay que considerar que estamos ante un procesador segmentado, y habrá que tener en cuenta los retardos de las unidades funcionales utilizadas, y los tiempos en los que deba detenerse el cauce si una operación de las codificadas en la instrucción VLIW necesita el resultado de alguna operación codificada en instrucciones anteriores que todavía no lo hayan generado. Una posible estimación del tiempo de ejecución de las instrucciones del código de la Figura 14 se proporciona en la Figura 15. Para hacer esta estimación se ha considerado que las operaciones en coma flotante y los accesos a memoria (suponiendo el dato está en caché) tienen un retardo de dos ciclos. También se ha considerado que hay una única unidad de acceso a memoria a la que se accede a través del *slot 1*. En cuanto a la instrucción de salto, se considera que también necesita dos ciclos, por lo que la instrucción inmediatamente posterior se introducirá en el cauce y se ejecutará tanto si se debe producir el salto como si no. Por esta razón es posible ubicar tras la instrucción de salto, operaciones que se deben completar independientemente de si se produce, o no se produce el salto (es lo que se denomina *salto retardado* en un procesador segmentado).

ciclo	Etiqueta	<i>slot 1: Op. Mem.</i>	<i>slot 2: Op. ALU</i>	<i>slot 3: Op. FP</i>
1	inicio	ld f0,a	add r3,r0,r0	nop
2				
3		lw r1,n	nop	nop
4				
5	bucle	ld f2,x(r3)	nop	nop
6				
7		nop	subi r1,r1,#1	add f4,f2,f0
8		nop	bnez r1,bucle	nop
9		sd x(r3),f4	addi r3,r3,#8	nop

Figura 15. Estimación del tiempo de la ejecución del código de la Figura 14

Teniendo en cuenta la Figura 15, el código tardaría en ejecutarse $4 + 5n$ ciclos, puesto que el bucle debe completar n iteraciones: una por componente del vector. En cuanto al

uso de la memoria, debido a los *nop* que hay que introducir, solo en 9 de los 18 slots se codifica información útil para completar el programa. Se aprovecha únicamente un 50% de espacio que ocupa el programa.

En el código de la Figura 13 hay un primer bloque constituido por las primeras tres instrucciones, y un segundo con el cuerpo del bucle, incluyendo seis instrucciones. Las tres últimas se dedican a gestionar el acceso a los elementos del vector y el final del bucle, y se puede considerar que las tres instrucciones esenciales para el cómputo a realizar con el vector son las tres primeras instrucciones del bucle: una instrucción para acceder a memoria y cargar el dato del vector, otra para hacer la suma, y otra para escribir el resultado. Por tanto existe riesgo RAW entre la instrucción de carga `ld f2, x(r3)` y la de suma `add f4, f2, f0`, y entre la de suma y almacenamiento `sd x(r3), f4`.

El *desenrollado de bucles* transforma el bucle de partida en un nuevo bucle cuyas iteraciones incluyen las operaciones de varias de las iteraciones del bucle de partida no desenrollado. Se tienen así bloques básicos con más instrucciones y, al mismo tiempo, disminuye el número de instrucciones de salto a ejecutar para controlar el bucle dado que el número de iteraciones es menor. Al tener bloques básicos con más instrucciones, el compilador dispone de más operaciones sin dependencias entre ellas para ubicarlas en los *slots* de las instrucciones VLIW. Por ejemplo, si suponemos que el valor de *n* es múltiplo de cinco, podremos desenrollar cinco veces el bucle del código de la Figura 13, y tendríamos el código de la Figura 16, donde se pueden apreciar muchas instrucciones de carga de memoria y almacenamiento en memoria independientes entre sí, y también instrucciones de suma independientes. Al disponerse de un bloque básico más grande hay más operaciones independientes para distribuir entre los *slots* de las instrucciones VLIW.

inicio:	lw	r1, n	; carga en r1 las iteraciones
	ld	f0, a	; carga en f0 el valor de a
	add	r3, r0, r0	; r3 = 0 (índice del vector)
bucle:	ld	f2, x(r3)	; f2 = x[i]
	add	f4, f2, f0	; f4 = x[i] + s
	sd	x(r3), f4	; x[i] = f4
	ld	f6, x+8(r3)	; f6 = x[i+1]
	add	f8, f6, f0	; f8 = x[i+1] + s
	sd	x+8(r3), f8	; f8 = x[i+1]
	ld	f10, x+16(r3)	; f10 = x[i+2]
	add	f12, f10, f0	; f12 = x[i+2] + s
	sd	x+16(r3), f12	; x[i+2] = f12
	ld	f14, x+24(r3)	; f14 = x[i+3]
	add	f16, f14, f0	; f16 = x[i+3] + s
	sd	x+24(r3), f16	; x[i+3] = f16
	ld	f18, x+32(r3)	; f18 = x[i+4]
	add	f20, f18, f0	; f20 = x[i+4] + s
	sd	x+32(r3), f20	; x[i+4] = f20
	subi	r1, r1, #5	; quedan 5 iteraciones menos
	addi	r3, r3, #40	; Apuntar al siguiente elemento
	bnez	r1, bucle	; Si quedan elementos saltar a bucle

Figura 16. Código de la Figura 4.13 con el bucle desenrollado cinco veces

Etiqueta	slot 1: Op. Mem.		slot 2: Op. ALU		slot 3: Op. FP	
inicio	ld	f0, s	add	r3, r0, r0	nop	
	lw	r1, n	nop		nop	
bucle	ld	f2, x(r3)	nop		nop	
	ld	f6, x+8(r3)	nop		add	f4, f2, f0
	ld	f10, x+16(r3)	nop		add	f8, f6, f0
	ld	f14, x+24(r3)	nop		add	f12, f10, f0
	ld	f18, x+32(r3)	nop		add	f16, f14, f0
	sd	f4, x(r3)	nop		add	f20, f18, f0
	sd	f8, x+8(r3)	nop		nop	
	sd	f12, x+16(r3)	subi	r1, r1, #5	nop	
	sd	f16, x+24(r3)	bnez	r1, bucle	nop	
	sd	f20, x+32(r3)	addi	r3, r3, #40	nop	

Figura 17. Código VLIW con tres slots tras el desenrollado de la Figura 16

Como se puede ver en la Figura 17, se reduce el tanto por ciento de memoria que no codifica operaciones (se pasa de un 50% a un 41%), y el tiempo de ejecución de este código VLIW se puede estimar en $4 + 15(n/5)$. Es decir, se ganan 2 ciclos por iteración.

Ciclo	Etiqueta	slot 1: Op. Mem.		slot 2: Op. ALU		slot 3: Op. FP	
1	inicio	ld	f0, s	add	r3, r0, r0	nop	
2							
3		lw	r1, n	nop		nop	
4							
5	bucle	ld	f2, x(r3)	nop		nop	
6							
7		ld	f6, x+8(r3)	nop		add	f4, f2, f0
8							
9		ld	f10, x+16(r3)	nop		add	f8, f6, f0
10							
11		ld	f14, x+24(r3)	nop		add	f12, f10, f0
12							
13		ld	f18, x+32(r3)	nop		add	f16, f14, f0
14							
15		sd	f4, x(r3)	nop		add	f20, f18, f0
16		sd	f8, x+8(r3)	nop		nop	
17		sd	f12, x+16(r3)	subi	r1, r1, #5	nop	
18		sd	f16, x+24(r3)	bnez	r1, bucle	nop	
19		sd	f20, x+32(r3)	addi	r3, r3, #40	nop	

Figura 18. Estimación del tiempo para el código desenrollado de la Figura 17

El problema del desenrollado de bucles es que el tamaño de la memoria ocupado por el programa es mayor (a pesar de que se pueda mejorar el uso de la misma: si se comparan las Figuras 14 y 17 se puede ver que el programa desenrollado ocupa el doble de memoria que el no desenrollado. Existe una técnica que permite transformar los

bucles aumentando el número de operaciones independientes que contienen los bloques básicos sin que se produzca un incremento sustancial en el tamaño del cuerpo del bucle. Se trata de la *segmentación software* (*software pipelining* en inglés).

La segmentación software se basa en que en el cuerpo de un bucle suelen existir instrucciones que cargan los datos, otras que hacen operaciones sobre los mismos, y por último están las instrucciones que almacenan los resultados. Entre las instrucciones de una misma iteración suelen darse dependencias de tipo RAW porque las instrucciones que operan con los datos necesitan que las que los cargan se hayan ejecutado y las que almacenan los resultados necesitan que las que hacen los cálculos terminen de ejecutarse. Sin embargo, si se construyen bucles en cuyas iteraciones se utilicen instrucciones que cargan los datos que se utilizan en la iteración siguiente, se hacen los cálculos con los datos que se cargaron en la iteración anterior, y se almacenan los resultados de los cálculos de la iteración anterior, las instrucciones del cuerpo del bucle no tienen dependencias de datos entre ellas.

Iteración i	Iteración i+1	Iteración i+2
ld f2, x(r3)		
add f4, f2, f0	ld f2, x+8(r3)	
sd x(r3), f4	add f4, f2, f0	ld f2, x+16(r3)
	sd x+8(r3), f4	add f4, f2, f0
		sd x+16(r3), f4

Figura 19. Fundamento de la segmentación software

Esta idea se ilustra en la Figura 19 para el bucle de la Figura 13. En cada iteración del bucle transformado se almacena el resultado de la iteración i-esima, se realiza la operación de la iteración i+1, y se carga el dato para la iteración i+2. Por supuesto, antes de iniciarse las iteraciones habrá que captar los datos de la primera y segunda iteraciones del bucle inicial, y la operación de la primera iteración del bucle inicial. Estas instrucciones constituyen el denominado *prólogo* al bucle con segmentación software. Además, después del bucle transformado habrá que almacenar los datos de las dos últimas iteraciones del bucle inicial, y realizar la operación de la última iteración del bucle inicial.

Etiqueta	slot 1: Op. Mem.	slot 2: Op. ALU	slot 3: Op. FP
	ld f0, s	add r3, r0, r0	nop
	lw r1, n	nop	nop
prologo	ld f2, x(r3)	nop	nop
	ld f2, x+8(r3)	nop	add f4, f2, f0
bucle	sd x(r3), f4	subi r1, r1, #1	add f4, f2, f0
	ld f2, x+16(r3)	bnez r1, bucle	nop
•	nop	addi r3, r3, #8	nop
epilogo	sd x+8(r3), f4	nop	add f4, f2, f0
	sd x+16(r3), f4	nop	nop

Figura 20. Código VLIW para el programa de la Figura 13 con segmentación software

Ciclo	Etiqueta	slot 1: Op. Mem.		slot 2: Op. ALU		slot 3: Op. FP	
1		ld	f0, s	add	r3, r0, r0	nop	
2							
3		lw	r1, n	nop		nop	
4							
5	prologo	ld	f2, x(r3)	nop		nop	
6							
7		ld	f2, x+8(r3)	nop		add	f4, f2, f0
8							
9	bucle	sd	x(r3), f4	subi	r1, r1, #1	add	f4, f2, f0
10		ld	f2, x+16(r3)	bnez	r1, bucle	nop	
11		nop		addi	r3, r3, #8	nop	
12	epilogo	sd	x+8(r3), f4	nop		add	f4, f2, f0
13							
14		sd	x+16(r3), f4	nop		nop	

Figura 21. Estimación del tiempo para el código desenrollado de la Figura 20

La Figura 20 muestra el código VLIW que se tendría para el programa de la Figura 13 cuyo bucle ha sido trasformado mediante segmentación software. En este caso, el tanto por ciento de memoria que no codifica operaciones es el 40.7% y el código necesita menos memoria que en el caso del desenrollado. En la Figura 21 se muestra una estimación para el tiempo de ejecución. Como se puede ver este tiempo será $11 + 3n$. Es decir, a partir de $n=3$ iteraciones, con la segmentación software se tiene un tiempo menor que en el caso inicial. Tanto la Figura 18 como la Figura 21 ponen de manifiesto que, en este ejemplo, acelerar los accesos a memoria o disponer de varias unidades de carga, podría reducir los tiempos de los programas.

3.2. Instrucciones con predicado

Otra forma de conseguir bloques básicos más grandes para disponer de más operaciones paralelas que distribuir en los *slots*, es reducir el número de instrucciones de salto condicional. Las *instrucciones con predicado*, también denominadas instrucciones de *ejecución condicional o vigilada*, contribuyen a eliminar muchos de los saltos condicionales y sustituyen dependencias de control por dependencias de tipo RAW en los códigos. En general reducir el número de saltos condicionales es conveniente en cualquier microarquitectura segmentada y también los repertorios de instrucciones escalares utilizados en los procesadores superescalares han ido incorporando este tipo de instrucciones.

Esta técnica se implementa a través de un nuevo tipo de operando que recibe el nombre de *predicado* que puede tomar dos valores correspondientes a las opciones de verdadero o falso. Existen instrucciones que permiten evaluar condiciones y dar valores a estos operandos, y se puede asignar un predicado a una instrucción (se habla de *predicar la instrucción*) que hace que la instrucción se ejecute o no según el valor del predicado. Existen repertorios de instrucciones en los que *cualquier instrucción se puede predicar* (es decir, su ejecución se puede hacer condicional). En otros repertorios solo se pueden predicar instrucciones de un tipo determinado.

En los repertorios donde se utilizan predicados debe haber instrucciones que permiten asignar valores a los predicados, y alguna forma de poder asignar un predicado a una instrucción (que según el valor del predicado se ejecuta o no). En algunos procesadores incluso existen registros especiales para almacenar los predicados (en realidad solo hace falta un bit por predicado, con lo que en un único registro de 64 bits habría espacio para 64 registros de predicado). Un formato usual para asignar un predicado, p , a una instrucción, INSTRUC, podría ser (p) INSTRUC, y significa que la instrucción sólo se ejecutaría si p fuese *verdadero* (ó $p=1$). Para asignar valores a los predicados se puede utilizar un formato del tipo $p1, p2$ cmp.cnd rx, ry donde cnd es la condición que se comprueba entre los registros rx y ry (lt, le, gt, ge, eq, ne,...). Si la condición es verdadera $p1=1$ y $p2=0$, y si es falsa, $p1=0$ y $p2=1$. También suelen existir instrucciones, con un formato similar al anterior, que solo asignan valor a un predicado. Así, la instrucción p cmp.cnd rx, ry hace $p=1$ si se verifica la condición de comparación entre los registros, y $p=0$ si no se cumple.

En la Figura 22 se muestra un ejemplo de cómo las instrucciones con predicado permiten evitar instrucciones de salto para la sentencia

```
if (r1==r2) r3=0 else if (r1>0) r3=1;
```

donde $r1$, $r2$ y $r3$ son registros del procesador y el registro $r0=0$. La primera instrucción de la Figura 22 se ejecuta siempre, y se utiliza para asegurar que el valor del registro $p3$ es igual a 0 (la condición de la instrucción cmp.ne siempre es falsa). La segunda instrucción también se ejecuta siempre, y asigna valores a los predicados $p1$ y $p2$. Si se cumple la condición $p1=1$ ($p2=0$) se ejecutará la instrucción tercera, y en caso contrario, $p2=1$ ($p1=0$) se ejecutará la instrucción cuarta. Si se ejecuta la instrucción cuarta se hace $p3=1$ si $r1>0$, ó $p3=0$ en caso contrario. Si $p3$ se hace igual a 1, se ejecutará la instrucción quinta, que hace $r3=1$. Obsérvese que como el valor de $p3$ se asigna en una instrucción que depende de otra condición (el valor de $p2$), si el predicado $p2$ no se hubiese hecho igual a 1, no podríamos estar seguros del valor que tiene $p3$, y por eso es por lo que hay que inicializarlo (en la primera instrucción). Esto no ocurre con los predicados $p1$ y $p2$ a los que se asigna valores en la segunda instrucción, que se ejecuta siempre.

	p3	cmp.ne	r0, r0	; hace $p3=0$
(p1)	p1, p2	cmp.eq	r1, r2	; Si $r1=r2$ entonces $p1=1$ y $p2=0$
(p2)		add	r3, r0, r0	; $r3=0$ si $p1=1$
(p3)	p3	cmp.gt	r1, r0	; Si $r1>0$ se hace $p3=1$
		addi	r3, r0, #1	; Si $r1>0$ $r3=1$

Figura 22. Codificación de una sentencia “if-then-else-if” mediante instrucciones con predicado

Al evitar el uso de instrucciones de salto, utilizando las instrucciones con predicado se puede conseguir aumentar el tamaño de los bloques básicos y facilitar la tarea del compilador. Como se ha indicado más arriba, al utilizar las instrucciones con predicado los riesgos de control se transforman en riesgos RAW. Así, entre las instrucciones segunda y tercera existe un riesgo RAW debido a $p1$, y entre las instrucciones segunda y cuarta

existe un riesgo RAW debido a p2. También entre la segunda y la cuarta y entre la cuarta y la quinta hay riesgos RAW debidos a p3.

3.3. Procesamiento especulativo

Las instrucciones con predicado o de ejecución condicional permiten que el procesador pueda aplicar técnicas de procesamiento especulativo para aprovechar la información de que disponga acerca de la mayor o menor probabilidad de que se verifique una condición y, por lo tanto haya que ejecutar una u otra alternativa tras una instrucción de salto. Ya se ha visto un ejemplo de procesamiento especulativo en el caso del procesamiento de instrucciones de salto condicional por parte de los procesadores superescalares. En el caso de los procesadores VLIW, la especulación debe hacerse en el momento de la compilación. Es difícil tener en cuenta en ese momento toda la casuística que puede darse durante la ejecución del programa. Por esta razón, para mejorar el rendimiento del compilador en la especulación que lleva a cabo, los procesadores VLIW de propósito general suelen incorporar ciertos recursos hardware.

#	slot OP1		slot OP2	
1	lw	r1, dato		
2			add	r3, r1, r5
3	begz	r10, loop	add	r6, r3, r7
4	lw	r8, 0(r10)		
5	lw	r9, 0(r8)		
6	loop:	sw (r9), r6	

Figura 23. Código VLIW con dos slots

En la Figura 23 se muestra un código VLIW con instrucciones de dos slots. La operación del primer slot de la instrucción 3 es un salto condicional que se producirá si r10=0. Se puede interpretar que ese salto condicional tiene como objetivo evitar que en la instrucción 4 se produzca un acceso a memoria (lw r8, 0(r10)) a la dirección 0. Esta dirección de memoria suele corresponder a un área protegida a la que no se puede acceder desde el modo de usuario y generaría una excepción. El compilador puede estimar que es poco probable que el valor de ese registro r10 sea igual a cero y puede adelantar la operación de acceso a memoria del primer slot de la instrucción 4 mediante una instrucción de ejecución condicional. Si dispone de instrucciones con predicado, se puede adelantar el acceso a memoria, generándose un código como el de la Figura 24, en el que se ha ahorrado una instrucción y se adelanta el inicio del acceso a memoria, que suele tener una latencia considerablemente mayor que otras operaciones.

#	slot OP1		slot OP2	
1	lw	r1, dato	p	cmp.ne r0, r0
2	(p) lw	r8, 0(r10)	add	r3, r1, r5
3	begz	r10, loop	add	r6, r3, r7
4	lw	r9, 0(r8)		
5	loop:	sw (r9), r6		

Figura 24. Código VLIW con dos slots y predicados

No obstante, los códigos de las Figuras 23 y 24 pueden no tener el mismo comportamiento frente a posibles excepciones. Por ejemplo, en el código de la Figura 23, si $r10$ es cero, es decir, una dirección no permitida para el acceso a memoria, no se generaría las cargas y no habría excepciones. En el caso de la Figura 24, lo que ocurría depende de la forma en que se hayan implementado las instrucciones con predicado. Si se produce el acceso a memoria y posteriormente se carga, o no, $r8$ con el resultado de ese acceso después de evaluar $r10$, se producirá la excepción de todas formas dado que el acceso a memoria se ha generado. Una forma de evitar este problema es incluir en el repertorio una operación de carga de memoria especulativa del tipo `lw.s rd,desplaz(rs), compensa`, para indicar que la operación de carga se ha anticipado. Además, todos los registros del procesador disponen de un bit de marca para gestionar las excepciones como se indica a continuación: (1) si la carga adelantada da lugar a una excepción, se marca el registro de destino pero no se atiende la excepción; (2) si una operación utiliza un operando marcado provocará que se marque el registro resultado de esa operación; y (3) la excepción se atenderá si se intenta almacenar un resultado envenenado, en cuyo caso, tras atender la excepción, se ejecutará un código de reparación de los resultados incorrectos que se encuentra a partir de la dirección indicada en compensa. Existen otras alternativas para asegurar un comportamiento coherente frente a las excepciones cuando hay movimientos de código especulativos (J. Ortega, M. Anguita, 2005). Entre ellas están los bits de veneno, el uso de centinelas, etc. Algunas de esas técnicas se basan en el uso de estructuras similares a los ROB.

Un orden adecuado de los accesos a memoria puede contribuir considerablemente a conseguir unas buenas prestaciones. En un procesador VLIW, el compilador puede ordenar las instrucciones de acceso a memoria como mejor convenga si en el momento de la compilación se puede determinar que la dirección en la que escribe una instrucción de almacenamiento es distinta a la dirección donde va a leer otra instrucción y, por lo tanto, no hay dependencias RAW. El problema surge cuando no se conocen las direcciones a las que se va a acceder. En este caso, si el compilador adelanta una instrucción de carga (LOAD) a una de escritura (STORE) el adelantamiento es *especulativo*. En un procesador superescalar, se suelen incluir recursos hardware que permiten implementar este adelantamiento especulativo. Resultaría bastante beneficioso que un procesador VLIW también incluyera este tipo de recursos. Una posible estrategia para implementar el adelantamiento especulativo de LOADs a STOREs sería la siguiente: (1) se incluye en el repertorio una instrucción que comprueba la coincidencia de direcciones; (2) el compilador ubicaría esta instrucción en la posición original del LOAD que se ha adelantado y realizaría una labor de *centinela*; (3) cuando se ejecuta el LOAD que se ha adelantado especulativamente, el hardware guarda la dirección a la que se ha realizado el acceso; (4) si los sucesivos STORE no acceden a esa dirección, la especulación ha sido correcta; (5) si la especulación no es correcta porque algún STORE escribe en la dirección de la que el LOAD lee, el LOAD se vuelve a ejecutar cuando llega a la instrucción centinela y si, además, se han ejecutado instrucciones que dependen del dato cargado por el LOAD

habrá que repetir todas esas instrucciones (se incluirán en un código de compensación que se ejecutará junto con la instrucción centinela). En la Figura 25 se muestra un ejemplo del funcionamiento de ésta. La Figura 25(a) incluye las operaciones iniciales, las resultantes tras el adelantamiento, y el código VLIW suponiendo un procesador con dos *slots*. Dado que no se puede asegurar en el momento de la compilación que los contenidos de $r2$ y $r5$ sean distintos, se trata de un adelantamiento especulativo. Al adelantar la carga de memoria, $1w\ r4, (r5)$, se adelanta el acceso al dato que se necesitará después, ejecutándose además en paralelo con la multiplicación. La Figura 25(b) ilustra el uso de una instrucción de carga de memoria especulativa, $1w_e\ r4, (r5)$, y de la instrucción centinela, $1w_c$. Si los registros $r2$ y $r5$ finalmente coinciden, habrá que repetir la carga de memoria y la suma (que se habrá realizado con un valor incorrecto en $r4$). Estas dos instrucciones se introducen en el código de corrección que se ubica en la dirección $etiq$ de la instrucción centinela.

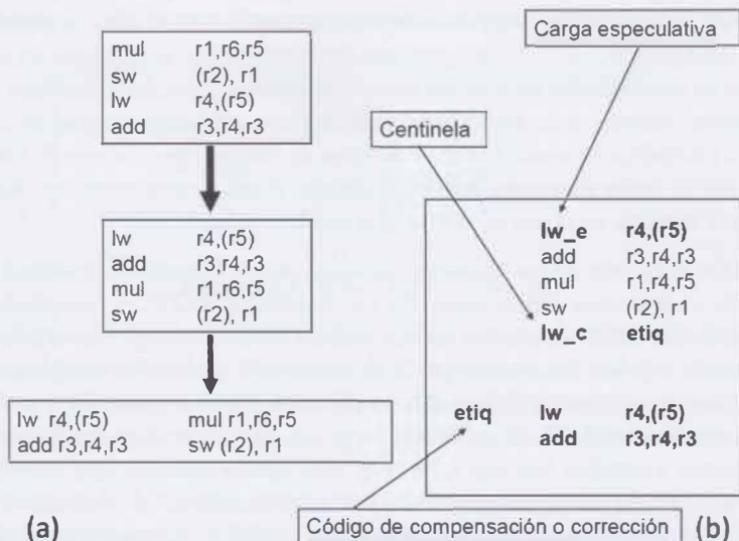


Figura 25. Ejemplo de adelantamiento de LOAD a STORE especulativo

Estos ejemplos relacionados con el procesamiento especulativo, ponen de manifiesto que el aprovechamiento eficiente de las arquitecturas VLIW en procesadores de propósito general precisa de la inclusión de ciertas estructuras hardware que faciliten el trabajo del compilador. Se reduciría así la diferencia en la complejidad de la microarquitecturas de los VLIW de propósito general con respecto a los procesadores superescalares.

4. Problemas

Problema 1. El fragmento de código que se proporciona a continuación:

```

1      lw      r1,0x1ac          ; r1←(0x1ac)
2      lw      r2,0xc1f          ; r2←(0xc1f)
3      add    r3,r0,r0          ; r3←r0+r0 (r3←0)
4      mul    r4,r2,r1          ; r4←r2+r1
5      add    r3,r3,r4          ; r3←r3+r4
6      add    r5,r0,0x1ac        ; r5←r0+0x1ac
7      add    r6,r0,0xc1f        ; r6←r0+0xc1f
8      sub    r5,r5,#4          ; r5←r5+4
9      sub    r6,r6,#4          ; r6←r6+4
10     sw     (r5),r3          ; (r5)←r3
11     sw     (r6),r4          ; (r6)←r4

```

se ejecuta en un procesador que puede captar y emitir cuatro instrucciones por ciclo. Indique el orden en que se emitirán las instrucciones para cada uno de los siguientes casos: (a) Una ventana de instrucciones centralizada con emisión ordenada; y (b) Una ventana de instrucciones centralizada con emisión desordenada.

NOTA: considere que hay una unidad funcional para la carga (que consume 2 ciclos), otra para el almacenamiento (1 ciclo), tres para la suma/resta(1 ciclo), y una para la multiplicación (4 ciclos).

Solución

(a) *Emisión ordenada.* En la Tabla 2 se muestran los ciclos en los que cada instrucción pasa por cada una de las etapas del cauce de la Figura 26.

Tabla 2. Ventana centralizada con emisión Ordenada

Instrucciones		1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	lw r1,0x1ac	IF		ID	EX	EX									
2	lw r2,0xc1f	IF		ID			EX	EX							
3	add r3,r0,r0	IF		ID			EX								
4	mul r4,r2,r1	IF		ID				EX	EX	EX	EX				
5	add r3,r3,r4	IF		ID								EX			
6	add r5,r0,0x1ac	IF		ID								EX			
7	add r6,r0,0xc1f	IF		ID								EX			
8	sub r5,r5,#4	IF		ID									EX		
9	sub r6,r6,#4				IF	ID							EX		
10	sw (r5),r3					IF	ID							EX	
11	sw (r6),r4						IF	ID							EX

En el ciclo (1) se captan las primeras cuatro instrucciones, 1-4, que pasan a la cola de instrucciones. Como no nos indican que dicha cola tenga un tamaño determinado

podemos suponer que tiene espacio suficiente para almacenar todas las instrucciones de la secuencia que se considera. Por tanto, en el ciclo (2) se captarán las cuatro instrucciones siguientes, 5-8, y en el ciclo (3) las últimas instrucciones de la secuencia, 9-11.

Igualmente, como no nos indican que haya limitación de tamaño en la ventana de instrucciones centralizada, también podremos suponer que tiene tamaño suficiente para contener las instrucciones de la secuencia considerada. Por ello, en el ciclo (2) se decodificarán las cuatro primeras instrucciones, en el ciclo (3) las cuatro siguientes, y en el ciclo (4) las tres últimas instrucciones.

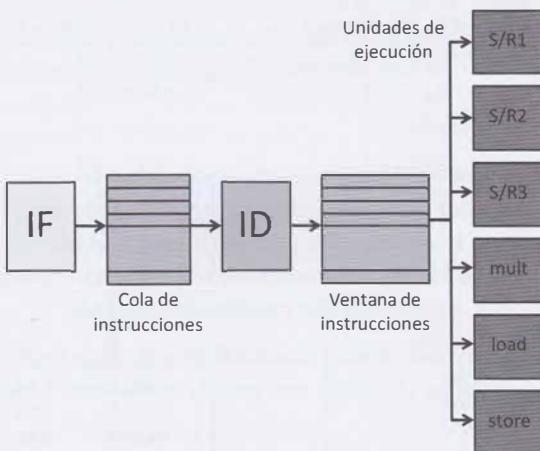


Figura 26. Esquema de etapas del procesador utilizadas en el problema

En cuanto a la ventana de instrucciones, el enunciado del problema no incluye ninguna indicación sobre si es alineada (hasta que no se emiten las instrucciones que se introducen tras su decodificación en un mismo ciclo, no se pueden emitir nuevas instrucciones) o no alineada (se puede emitir cualquier instrucción incluida en la ventana de instrucciones siempre que reúna las condiciones correspondientes de independencia, disponibilidad, etc.). Consideraremos que se trata de una ventana no alineada, que por otra parte es el caso más común.

Por tanto, en el ciclo (3) la instrucción 1 ya ha sido decodificada y puede emitirse desde la ventana de instrucciones. Aunque se pueden emitir cuatro instrucciones por ciclo, la instrucción 2 no puede emitirse hasta que no acabe de ejecutarse la instrucción 1, dado que solo hay una unidad de carga de datos de memoria: existe una dependencia estructural entre las instrucciones 1 y 2. Así, la instrucción 2 podría empezar a ejecutarse en el ciclo 5 y terminaría su ejecución en el ciclo 6. La instrucción 3 no tiene dependencias con la instrucción 2 y también podría emitirse junto con la instrucción 2. Hay que tener en cuenta que, puesto que la emisión es ordenada, aunque la instrucción 3 no tiene dependencias con la instrucción 2 y podría emitirse antes que ella, dado que la emisión es ordenada, debe esperar hasta que se emita la instrucción 2, que va delante en el código.

La instrucción 4 debe esperar a que termine la ejecución de la instrucción 2, puesto que uno de sus operandos es r2, que precisamente es el registro que carga la instrucción de acceso a memoria 2. Por lo tanto, se emitirá en el ciclo (7).

Como uno de los operandos de la instrucción 5 es r4, que es calculado precisamente por la instrucción 4, no puede emitirse hasta que no termine la ejecución de la instrucción 4 (que precisamente calcula ese valor). Por lo tanto, hasta el ciclo (11) no podría emitirse, al mismo tiempo que las instrucciones 6 y 7 que le siguen, que no dependen de la instrucción 5, y son independientes entre si. Al disponerse de tres unidades de suma/resta, se pueden emitir cada una de esas instrucciones a una unidad diferente y no habría problema de dependencia estructural. Las instrucciones 6 y 7, a pesar de no depender de la instrucción 4 no podrían emitirse antes que la instrucción 5 porque la emisión es ordenada.

Las instrucciones 8 y 9 dependen de las instrucciones 6 y 7, porque necesitan r5 y r6, respectivamente. Por lo tanto no pueden emitirse hasta que no termine la ejecución de las instrucciones 6 y 7, en el ciclo (12). Como hay tres unidades de suma/resta se pueden emitir las dos en el mismo ciclo.

Las instrucciones 10 y 11 también dependen de las instrucciones 8 y 9 y tendrían que esperar hasta el ciclo (13). No obstante, dado que solo hay una unidad de almacenamiento en memoria, primero se emite la instrucción 10 en el ciclo (13) y después la instrucción 11, en el ciclo (14).

(b) Emisión desordenada. En la Tabla 3 se muestran los ciclos en los que las instrucciones van ejecutando cada una de sus etapas.

Tabla 3. Ventana centralizada con emisión Desordenada

Instrucciones		1	2	3	4	5	6	7	8	9	10	11	12
1	lw r1,0x1ac	IF		ID	EX	EX							
2	lw r2,0xc1f	IF		ID			EX	EX					
3	add r3,r0,r0	IF		ID	EX								
4	mul r4,r2,r1	IF		ID					EX	EX	EX	EX	
5	add r3,r3,r4	IF		ID									EX
6	add r5,r0,0x1ac	IF		ID	EX								
7	add r6,r0,0xc1f	IF		ID	EX								
8	sub r5,r5,#4	IF		ID		EX							
9	sub r6,r6,#4				IF	ID	EX						
10	sw (r5),r3				IF	ID							EX
11	sw (r6),r4				IF	ID							EX

En cuanto a la captación y a la decodificación, la situación es la misma que en el caso de emisión desordenada. Del mismo modo, para emitirse la instrucción 2 debe esperar a la instrucción 1 como en el caso de la emisión ordenada dado que lo que determinaba esta espera era que solo había una unidad de carga de memoria. Sin embargo, la instrucción

3, que no depende ni de la instrucción 1 ni de la 2, puede emitirse al mismo tiempo que la instrucción 1.

La instrucción 4 debe esperar al ciclo (7), igual que en el caso de la emisión ordenada dado que necesita tener el valor de r2. Igualmente, la instrucción 5 tiene que esperar al ciclo (11), dado que necesita el valor de r4, calculado por la instrucción 4, como uno de sus operandos.

Sin embargo, las instrucciones 6 y 7 pueden emitirse en el ciclo (4), una vez han sido decodificadas en el ciclo (3). En efecto, no dependen de operandos que deben ser calculados en instrucciones previas, existen unidades de suma/resta suficientes, y se pueden emitir cuatro instrucciones por ciclo (en el ciclo (4) se emitirían tres instrucciones).

Las instrucciones 8 y 9 deben esperar a que acabe la ejecución de la 6 y la 7, respectivamente por lo que podrían emitirse en el ciclo (5). Igual que en el caso anterior, hay unidades funcionales suficientes y no se emitirían más de cuatro instrucciones por ciclo (se emitirían tres instrucciones, la 2, la 8, y la 9).

En cuanto a las instrucciones de almacenamiento, la instrucción 10 debe esperar al ciclo (12) para poderse emitir, dado que depende de r3, y ese valor solo estará disponible cuando acabe la ejecución de la instrucción 5. Sin embargo, la instrucción 11 podría emitirse en el ciclo (11) dado que el operando r4 del que depende está disponible al terminar la instrucción 4 en el ciclo anterior. En este caso, se ha evitado la colisión de las instrucciones para acceder a la única unidad de almacenamiento gracias a la emisión desordenada.

Como se puede observar, con la emisión desordenada la ejecución de las instrucciones se completa dos ciclos antes que con emisión ordenada.

Problema 2. Considerando el mismo fragmento de código utilizado en el Problema 1, y que el procesador dispone de las mismas unidades funcionales de suma/resta, multiplicación, carga, y almacenamiento de datos, con los mismos ciclos de retardo. Indique el tiempo que tardan en ejecutarse todas las instrucciones, si el procesador dispone de una estación de reserva, con capacidad para tres instrucciones, para cada una de sus unidades funcionales: (a) Suponiendo que el envío desde la estación de reserva es ordenado; y (b) Suponiendo que el envío desde la estación de reserva es desordenado.

Solución

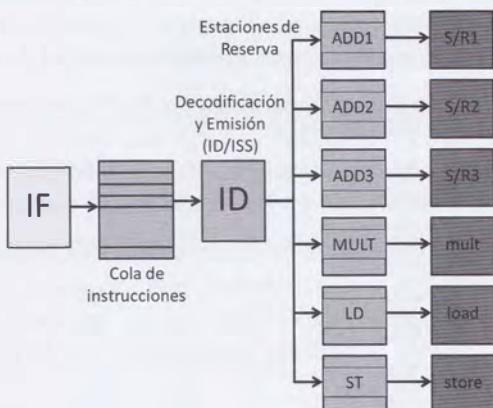
(a) *Envío ordenado.* La Tabla 4 muestra la evolución temporal de las instrucciones, a través de las distintas etapas, hasta que termina su ejecución (Figura 27).

Dado que no nos indican que exista límite en el número de instrucciones que se pueden almacenar en la cola de instrucciones, suponemos que ésta tiene espacio para al menos las 11 instrucciones que se analizan y, por lo tanto, las instrucciones 1-4 se captan en el ciclo (1), las 5-8 en el ciclo (2), y las 9-11 en el ciclo (3).

Tabla 4. Estaciones de reserva con envío ordenado

Instrucciones		Est.res.	1	2	3	4	5	6	7	8	9	10	11	12	13
1	lw r1,0x1ac	LD	IF	ID	EX	EX									
2	lw r2,0xc1f	LD	IF	ID			EX	EX							
3	add r3,r0,r0	ADD1	IF	ID	EX										
4	mul r4,r2,r1	MULT	IF	ID					EX	EX	EX	EX			
5	add r3,r3,r4	ADD1		IF	ID								EX		
6	add r5,r0,0x1ac	ADD2		IF	ID	EX									
7	add r6,r0,0xc1f	ADD3		IF	ID	EX									
8	sub r5,r5,#4	ADD2		IF	ID		EX								
9	sub r6,r6,#4	ADD3			IF	ID	EX								
10	sw (r5),r3	ST			IF	ID							EX		
11	sw (r6),r4	ST			IF	ID								EX	

Al final del ciclo 2, las instrucciones 1-4 se habrán decodificado y emitido (aunque para en la gráfica se sigue manteniendo la notación ID para identificar ésta etapa, en realidad se realiza la decodificación y emisión a la estación de reserva correspondiente, ID/ISS).

**Figura 27.** Esquema de etapas del procesador utilizadas en el problema

En la estación de reserva de la unidad de carga de datos (LD) estarán las instrucciones 1 y 2, en la estación de reserva de una de las unidades de suma/resta (ADD1) estará la instrucción 3, y en la estación de reserva de la unidad de multiplicación (MULT) estará la instrucción 4. Desde cada una de esas estaciones de reserva se realizará el envío de la instrucción a la unidad correspondiente.

En el ciclo (3) se enviará la instrucción 1 a la unidad de carga para que empiece a ejecutarse. La instrucción 2 se mantendrá en la estación de reserva hasta que la instrucción 1 termine y la unidad de carga quede libre. La instrucción 3 también puede emitirse a la unidad de suma 1 dado que no depende de las instrucciones anteriores. La instrucción 4, sin embargo, debe esperar en la estación de reserva de la multiplicación hasta que termine de ejecutarse la instrucción 2.

Al final del ciclo (3), las instrucciones 5-8 se habrán decodificado y han pasado a las estaciones de reserva correspondientes. La forma de emitir instrucciones a las tres estaciones de reserva, ADD1, ADD2, ADD3, de las distintas unidades de suma depende del algoritmo que implemente la etapa de decodificación/emisión (ID en la Tabla 4). Aquí se ha considerado que se van asignando instrucciones a las estaciones menos cargadas, y si hay varias estaciones con la misma carga se asigna a aquella que tenga instrucciones de las que dependa la instrucción a emitir (ordenadamente según el índice de la estación de reserva, si existieran varias en esa situación o si las estaciones de reserva están vacías). Según esto, cuando se van a emitir las cuatro instrucciones, las tres estaciones de reserva están vacías y la instrucción 5 se asigna a ADD1, la 6 a ADD2 y la 7 a ADD3. La instrucción 8 se asignaría a la estación de reserva ADD2 porque las tres instrucciones tienen una instrucción y precisamente la instrucción 8 depende de la asignada a ADD2 (tiene como operando el resultado que produce la instrucción 6).

En el ciclo (4) podrán enviarse las instrucciones 6 y 7 a los correspondientes sumadores/restadores. Al final del ciclo (4) se habrán decodificado las instrucciones 9-11. Dado que en las estaciones de reserva ADD1 y ADD2 quedan una instrucción en cada una (respectivamente la 5 y la 8), la instrucción 9 pasaría a la estación de reserva ADD3 y las instrucciones 10 y 11 pasan a la estación de reserva de la unidad de almacenamiento ST.

En el ciclo (5) se pueden enviar las instrucciones 8 y 9 a sus respectivos sumadores/restadores y la instrucción 2 a la unidad de carga que estará ya libre. Al final de este ciclo (5) tenemos la instrucción 4 en la estación de reserva MULT, la 5 en la estación de reserva ADD1, y las instrucciones 10 y 11 en la estación de reserva ST.

En el ciclo (7) empieza la ejecución de la instrucción 4 de multiplicación, en el (11) la instrucción 5 de suma (una vez ha acabado la multiplicación), y en los ciclos (12) y (13) respectivamente, las instrucciones 10 (una vez ha acabado la instrucción 5, de la que depende) y la 11 (una vez ha acabado la instrucción 10 y ha dejado libre la unidad de almacenamiento).

Por lo tanto, en este caso se necesitan 13 ciclos para ejecutar todas las instrucciones. Si lo comparamos con el problema anterior, donde se utiliza ventana centralizada, se tarda un ciclo más que en el caso de la emisión desordenada, y un ciclo menos que la emisión ordenada. En este caso, se puede ver que, aunque dentro de cada estación de reserva el envío es ordenado, las instrucciones sí pueden ejecutarse desordenadamente debido a que se pueden enviar desde varias estaciones de reserva que no están sincronizadas entre sí.

(b) Envío desordenado. La Tabla 5 muestra la situación correspondiente al envío desordenado desde cada estación de reserva. Solo en la estación de reserva ST se produce una situación en la que hay más de una instrucción esperando, pudiendo enviarse la instrucción más reciente y la más antigua no. Ése es el único cambio que se observa en la Tabla 5.

Tabla 5. Estaciones de reserva con envío desordenado

Instrucciones			Est.res.	1	2	3	4	5	6	7	8	9	10	11	12
1	lw	r1, 0x1ac	LD	IF	ID	EX	EX								
2	lw	r2, 0xc1f	LD	IF	ID			EX	EX						
3	add	r3, r0, r0	ADD1	IF	ID	EX									
4	mul	r4, r2, r1	MULT	IF	ID					EX	EX	EX	EX		
5	add	r3, r3, r4	ADD1	IF	ID									EX	
6	add	r5, r0, 0x1ac	ADD2	IF	ID	EX									
7	add	r6, r0, 0xc1f	ADD3	IF	ID	EX									
8	sub	r5, r5, #4	ADD2	IF	ID		EX								
9	sub	r6, r6, #4	ADD3		IF	ID	EX								
10	sw	(r5), r3	ST		IF	ID								EX	
11	sw	(r6), r4	ST		IF	ID									EX

Como se puede ver, se necesitan 12 ciclos para terminar la ejecución de todas las instrucciones, como en el caso de la emisión desordenada con ventana de instrucciones. De hecho, las Tablas 3 y 5 coinciden en cuanto a la distribución de las instrucciones en etapas y ciclos.

Problema 3. Considere que el fragmento de código siguiente, correspondiente a una arquitectura LOAD/STORE:

```

1      lw      r1, 0x10a          ; r1←M(0x10a)
2      lw      r2, 0xc1f          ; r2←M(0xc1f)
3      add    r3, r3, r4          ; r3←r3+r4
4      mul    r4, r2, r1          ; r4←r2*r1
5      add    r5, r0, 0x1ac        ; r5←r0+0x1ac
6      add    r6, r0, 0xc1f        ; r6←r0+0xc1f
7      sub    r5, r5, #4          ; r5←r5+4
8      sub    r6, r6, #4          ; r6←r6+4
9      sw     0(r5), r3          ; m(r5)←r3
10     sw     0(r6), r4          ; m(r5)←r4

```

se ejecuta en un procesador superescalar que es capaz de captar 4 instrucciones/ciclo; de decodificar 2 instrucciones/ciclo; de emitir (utilizando una ventana de instrucciones con emisión no alineada) 2 instrucciones/ciclo; de escribir hasta 2 resultados/ciclo en los bancos de registros correspondientes (registros de reorden, o registros de la arquitectura según el caso); y de completar (retirando instrucciones del ROB) hasta 2 instrucciones/ciclo.

- Indique el número de ciclos que tardaría en ejecutarse el programa suponiendo: (a) Emisión ordenada; y (b) Emisión desordenada. (c) Si el procesador funciona a 1 GHz, indique cuál es su velocidad pico y determine el valor del número de ciclos por instrucción promedio (CPI) para cada tipo de emisión.

NOTA: Consideré que tiene una unidad funcional de carga (con 2 ciclos de retardo); una de almacenamiento (con 1 ciclo de retardo); dos unidades de suma/resta (con 1 ciclo de retardo); y una de multiplicación (con 3 ciclos de retardo); y que no hay limitaciones para el número de líneas de la cola de instrucciones, ventana de instrucciones, buffer de reorden, puertos de lectura/escritura etc.

Solución

Para responder a cada uno de los apartados se construyen las Tablas 6 y 7 que describen las etapas por las que van pasando cada una de las instrucciones.

(a) En este caso la emisión es ordenada y, por lo tanto, el comienzo de la ejecución de una instrucción que es posterior a otra en el código siempre se producirá en un ciclo posterior (o como muy pronto en el mismo ciclo) al comienzo de la ejecución de la primera. El final de la ejecución, en cambio, podría no respetar ese orden.

Tabla 6. Ventana centralizada con emisión Ordenada

	Instrucciones	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	lw r1,0x1ac	IF	ID	EX	EX	ROB	WB								
2	lw r2,0xc1f	IF	ID			EX	EX	ROB	WB						
3	add r3,r3,r4	IF		ID		EX	ROB		WB						
4	mul r4,r2,r1	IF		ID				EX	EX	EX	ROB	WB			
5	add r5,r0,0x1ac	IF		ID				EX	ROB			WB			
6	add r6,r0,0xc1f	IF		ID				EX	ROB				WB		
7	sub r5,r5,#4	IF			ID			EX	ROB				WB		
8	sub r6,r6,#4	IF			ID					EX	ROB			WB	
9	sw 0(r5),r3			IF			ID			EX				WB	
10	sw 0(r6),r4			IF			ID				EX				WB

En la Tabla 6 se observa que las instrucciones se captan (IF) de cuatro en cuatro en ciclos consecutivos dado que nos dicen que no hay limitaciones en el tamaño de la cola de instrucciones. También se captan de dos en dos en ciclos consecutivos, al no haber limitaciones en la ventana de instrucciones.

A partir del ciclo (3) se puede emitir la instrucción 1, pero la instrucción 2 debe esperar que termine la ejecución de la 1 dado que las dos utilizan la unidad de carga de datos de memoria y sólo se dispone de una de esas unidades. La instrucción 3 no depende de las anteriores pero no se puede emitir antes que la instrucción 2. Las instrucciones 2 y 3 se pueden emitir en el mismo ciclo dado que utilizan unidades funcionales que están disponibles y se pueden emitir hasta dos instrucciones por ciclo. La instrucción 4 depende de la instrucción 2, y no puede emitirse hasta que termine la ejecución de ésta. En el ciclo (7) se pueden emitir las instrucciones 4 y 5 dado que no existe dependencia entre ellas, las unidades de cálculo que necesitan están disponibles y se pueden emitir hasta dos instrucciones por ciclo. Igualmente, las instrucciones 6 y 7 pueden emitirse en el ciclo (8). La instrucción 7 depende de la 5, pero en ese ciclo (8) ya ha terminado su ejecución.

En el ciclo (9) se pueden emitir las instrucciones 8 y 9. Las instrucciones 3 y 5 de las que depende la instrucción 9 ya han producido sus resultados y las unidades que necesitan están disponibles. Finalmente, la instrucción 10 se emite en el ciclo (10). Esta instrucción 10 nunca se podría emitir al mismo tiempo que la instrucción 9 dado que depende de la instrucción 8 y además, solo se dispone de una unidad de almacenamiento en memoria. No obstante, aquí tampoco se podría emitir al mismo tiempo que la 9 aunque hubiera varias unidades de almacenamiento y no tuviera dependencias, dado que en el ciclo (9) ya se emite el número máximo de dos instrucciones.

En la Tabla 6 también se pueden ver los ciclos en los que se producen las escrituras de los resultados en el ROB (dado que no se pueden escribir más de dos por ciclo). No obstante, en ningún caso se producen colisiones por esta causa. Las instrucciones de almacenamiento de registros en memoria, al no tener que escribir resultados en el banco de registros, no tienen que almacenarlos temporalmente en el ROB. Por eso estas instrucciones “no pasan” por la etapa ROB.

Las instrucciones se retiran de dos en dos, de forma ordenada. Como se puede ver, excepto en las instrucciones 1, 2, y 4, en todas las instrucciones deben esperarse algunos ciclos desde que se termina de escribir el resultado en el buffer de reorden (etapa ROB) y se retira la instrucción (etapa WB).

(b) La Tabla 7 describe la evolución de las instrucciones en los distintos ciclos y etapas para la emisión desordenada.

Tabla 7. Ventana centralizada con emisión Desordenada

Instrucciones		1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	lw r1,0x1ac	IF ID		EX	EX	ROB	WB								
2	lw r2,0xc1f	IF ID				EX	EX	ROB	WB						
3	add r3,r3,r4	IF		ID	EX	ROB			WB						
4	mul r4,r2,r1	IF		ID				EX	EX	EX	ROB	WB			
5	add r5,r0,0x1ac	IF		ID	EX	ROB					WB				
6	add r6,r0,0xc1f	IF		ID		EX	ROB					WB			
7	sub r5,r5,#4	IF			ID	EX		ROB				WB			
8	sub r6,r6,#4	IF			ID		EX	ROB					WB		
9	sw 0(r5),r3			IF			ID		EX					WB	
10	sw 0(r6),r4			IF			ID			EX					WB

A pesar de que se permite la emisión desordenada, en este caso se observa que no se ha ganado ningún ciclo con respecto al caso de la emisión ordenada. Ahora puede adelantarse la emisión de la instrucción 3 a la de la instrucción 2 y la emisión de las instrucciones 5 6 y 7 a la de la instrucción 4. No obstante, la colisión entre las instrucciones 1 y 2 por el acceso a la unidad de carga, y la dependencia de la instrucción 4 con respecto a la de la instrucción 2, hacen que esta instrucción 4 finalice (se retire del ROB) en el mismo ciclo (11) que en el caso de emisión ordenada. Como en el caso de la emisión ordenada se retiraban, al máximo ritmo posible (dos instrucciones por ciclo),

las instrucciones que quedaban después de la instrucción 4, en el caso de la emisión desordenada no se ganará nada. Lo que ocurre es que, entre que se termina la etapa ROB o la de ejecución en las instrucciones de almacenamiento, las instrucciones esperan más ciclos hasta que se pueden retirar.

En este caso, a pesar de que se pueden emitir dos instrucciones por ciclo, las instrucciones de almacenamiento, 9 y 10, no se pueden emitir en el mismo ciclo dado que hay colisión entre ellas al no existir más que una unidad de almacenamiento. Otro efecto a reseñar es que al terminar la ejecución de la instrucción 7 hay una colisión en la escritura en el ROB (solo se pueden escribir dos datos por ciclo en el ROB según indica el enunciado). Esto hace que la unidad de suma/resta donde se ejecuta la instrucción 7 se mantiene ocupada. Sin embargo, esto no causa más problemas dado que hay otra unidad de suma/resta libre donde se puede ejecutar la instrucción 8. El retraso en la emisión de la instrucción 8 se debe a que no se pueden emitir más de dos instrucciones por ciclo (las instrucciones 6 y 7).

(c) Teniendo en cuenta que el procesador puede finalizar dos instrucciones por ciclo y que utiliza un reloj de 1 GHz, la velocidad pico del procesador se puede calcular a partir de:

$$GIPS = \frac{2(inst/ciclo) \times 10^9 (ciclos/segundo)}{10^9} = 2$$

Es decir puede alcanzar hasta 2 GIPS. En cuanto al cálculo del CPI medio, como en los dos casos de emisión ordenada y desordenada considerada se ha tardado el mismo tiempo (14 ciclos), en los dos casos se tendrá el mismo valor. Para calcularlo se utiliza la expresión del tiempo de CPU que, dado que el tiempo de ciclo es de 1 nseg. (la inversa de 1 GHz, es decir 10^{-9} seg.), es igual a 14 nseg. Así:

$$T_{CPU} = NI \times CPI \times T_{ciclo} = 10 \text{ inst} \times CPI \times 10^{-9} (\text{seg/ciclo}) = 14 \times 10^{-9} \text{ seg}$$

y, despejando se tiene que $CPI=1.4$ ciclos/instrucción. Es decir, se tarda más de un ciclo por instrucción por lo que, a pesar de que en algunos ciclos se terminan más de una instrucción, por término medio, no se pone de manifiesto el carácter superescalar del procesador (CPI debería de ser menor que 1).

Problema 4. Para la secuencia de instrucciones del problema 3, incremente hasta un máximo de cuatro, el número de instrucciones que se pueden emitir y finalizar por ciclo, el número de escrituras por ciclo, y el número de unidades de carga de datos de memoria, para reducir el tiempo de procesamiento de la secuencia al mínimo posible (a) en el caso de la emisión ordenada, y (b) en el caso de la emisión desordenada. Determine el valor de la velocidad pico y del CPI promedio en estos casos considerando el mismo ciclo de reloj que en el problema 3.

Solución

Para resolver el problema se modifican las Tablas 6 y 7 considerando que no existen limitaciones (hasta un valor máximo de cuatro) en el número de instrucciones que se pueden decodificar, emitir o retirar. Tampoco existen limitaciones (hasta un valor máximo de cuatro) en el número de escrituras en el buffer de reorden y se tienen dos unidades de carga en lugar de una (en este problema podemos saber que no hace falta incrementar más el número de estas unidades ya que solo hay dos instrucciones de carga de memoria). Así, se obtendrán las Tablas 8 y 9 para la emisión ordenada y desordenada, respectivamente.

(a) En la Tabla 8 se observa que, dado que hay dos unidades de carga de memoria se pueden emitir las instrucciones 1 y 2 al mismo tiempo. Esto permite adelantar la emisión de la instrucción de multiplicación que, a su vez permite adelantar la emisión de las instrucciones que la siguen. La instrucción 10 no se puede emitir en el ciclo (7), junto con la instrucción 9 porque está esperando el dato que proporciona la instrucción 4, no porque haya una única unidad de almacenamiento en memoria.

Como se puede ver, se produce una reducción de cuatro ciclos respecto al caso de la emisión ordenada del problema 3. En realidad no habría que emitir más de tres instrucciones por ciclo, ni escribir en el ROB más de dos resultados por ciclo. No obstante, si en lugar de finalizar cuatro instrucciones por ciclo se pudiesen finalizar siete instrucciones por ciclo, se reduciría un ciclo más. No obstante, terminar un número de instrucciones tan elevado tendría un coste hardware considerable, y posiblemente se aprovecharía en pocas ocasiones.

Tabla 8 Ventana centralizada con emisión ordenada

Instrucciones		1	2	3	4	5	6	7	8	9	10
1	lw r1,0x1ac	IF	ID	EX	EX	ROB	WB				
2	lw r2,0xc1f	IF	ID	EX	EX	ROB	WB				
3	add r3,r3,r4	IF	ID	EX	ROB		WB				
4	mul r4,r2,r1	IF	ID			EX	EX	EX	ROB	WB	
5	add r5,r0,0x1ac	IF	ID			EX	ROB			WB	
6	add r6,r0,0xc1f	IF	ID			EX	ROB			WB	
7	sub r5,r5,#4	IF	ID				EX	ROB		WB	
8	sub r6,r6,#4	IF	ID				EX	ROB			WB
9	sw 0(r5),r3			IF	ID			EX			WB
10	sw 0(r6),r4			IF	ID				EX		WB

(b) En la Tabla 9 se puede observar el efecto de la emisión desordenada. Ahora se puede adelantar la emisión de las instrucciones 5, 6, 7, 8 y 9 con respecto a lo que ocurre en la Tabla 8. Aunque también se está suponiendo que se pueden escribir cuatro resultados por ciclo en el ROB (si se supusiera que se pueden escribir sólo dos resultados por ciclo en el ROB se retrasaría la escritura de las instrucciones 5 y 6 y también se retrasaría la emisión de las instrucciones 7 y 8, pero esto no afectaría al número de ciclos final).

Tabla 9. Ventana centralizada con emisión desordenada

Instrucciones		1	2	3	4	5	6	7	8	9	10
1	lw r1,0x1ac	IF	ID	EX	EX	ROB	WB				
2	lw r2,0xc1f	IF	ID	EX	EX	ROB	WB				
3	add r3,r3,r4	IF	ID	EX	ROB		WB				
4	mul r4,r2,r1	IF	ID			EX	EX	EX	ROB	WB	
5	add r5,r0,0x1ac	IF	ID	EX	ROB					WB	
6	add r6,r0,0xc1f	IF	ID	EX	ROB					WB	
7	sub r5,r5,#4	IF	ID		EX	ROB				WB	
8	sub r6,r6,#4	IF	ID		EX	ROB					WB
9	sw 0(r5),r3			IF	ID		EX				WB
10	sw 0(r6),r4			IF	ID			EX			WB

Como se puede ver, tampoco aquí se observan tiempos diferentes entre la emisión ordenada y la desordenada. Esto es debido a que, por la dependencia entre las instrucciones 2, 4 y 10, el tiempo mínimo que tardan en completarse estas tres instrucciones es 9 ciclos, que es precisamente el tiempo que se obtiene ya en el caso de la emisión ordenada (salvo el ciclo que debe esperarse porque solo pueden retirarse cuatro instrucciones, problema que también se presenta en la emisión desordenada).

Tabla 10. Ventana centralizada con emisión ordenada y multiplicación de 1 ciclo

Instrucciones		1	2	3	4	5	6	7	8	9
1	lw r1,0x1ac	IF	ID	EX	EX	ROB	WB			
2	lw r2,0xc1f	IF	ID	EX	EX	ROB	WB			
3	add r3,r3,r4	IF	ID	EX	ROB		WB			
4	mul r4,r2,r1	IF	ID			EX	ROB	WB		
5	add r5,r0,0x1ac		IF	ID		EX	ROB	WB		
6	add r6,r0,0xc1f		IF	ID		EX	ROB	WB		
7	sub r5,r5,#4		IF	ID			EX	ROB	WB	
8	sub r6,r6,#4		IF	ID			EX	ROB	WB	
9	lw 0(r5),r3			IF	ID			EX	WB	
10	lw 0(r6),r4			IF	ID				EX	WB

Tabla 11. Ventana centralizada con emisión desordenada y multiplicación de 1 ciclo

Instrucciones		1	2	3	4	5	6	7	8
1	lw r1,0x1ac	IF	ID	EX	EX	ROB	WB		
2	lw r2,0xc1f	IF	ID	EX	EX	ROB	WB		
3	add r3,r3,r4	IF	ID	EX	ROB		WB		
4	mul r4,r2,r1	IF	ID			EX	ROB	WB	
5	add r5,r0,0x1ac		IF	ID	EX	ROB		WB	
6	add r6,r0,0xc1f		IF	ID	EX	ROB		WB	
7	sub r5,r5,#4		IF	ID		EX	ROB	WB	
8	sub r6,r6,#4		IF	ID		EX	ROB		WB
9	sw 0(r5),r3			IF	ID		EX		WB
10	sw 0(r6),r4			IF	ID			EX	WB

Para conseguir diferencias entre la emisión ordenada y desordenada tendríamos que reducir el tiempo de la multiplicación a un ciclo. Tal y como se observa en las Tablas 10 y 11.

Como se puede observar, se gana un ciclo. Si hubiera dos unidades de almacenamiento en memoria otra vez se igualarían los tiempos, ahora por la limitación a cuatro instrucciones que pueden retirarse como máximo.

En cuanto a la última pregunta del problema, dado que el procesador puede finalizar cuatro instrucciones por ciclo y que utiliza un reloj de 1 GHz, la velocidad pico del procesador se puede calcular a partir de:

$$GIPS = \frac{4(inst/ciclo) \times 10^9 (ciclos/segundo)}{10^9} = 4$$

Así, ahora se podrían alcanzar hasta 4 GIPS dado que se ha doblado el número de instrucciones que pueden finalizarse por ciclo respecto al problema 3.

Respecto al cálculo del *CPI* medio, como en los dos casos de emisión ordenada y desordenada considerada se ha tardado de nuevo el mismo tiempo (10 ciclos), en los dos casos se tendrá el mismo valor. Así, utilizando la expresión del tiempo de CPU:

$$T_{CPU} = NI \times CPI \times T_{ciclo} = 10 \text{ inst} \times CPI \times 10^{-9} (\text{seg/ciclo}) = 10 \times 10^{-9} \text{ seg}$$

y, despejando, se tiene que *CPI*=1 ciclos/instrucción. Es decir, se tarda un ciclo por instrucción que correspondería al caso de un procesador segmentado a pleno rendimiento. Como sabemos que durante los primeros cinco ciclos no se termina ninguna instrucción (es lo que se denomina tiempo de latencia de inicio en un procesador segmentado), el procesador sí pone de manifiesto su carácter superescalar dado que debe ser capaz de terminar más de una instrucción por ciclo una vez termina la primera instrucción transcurrido el tiempo de latencia de inicio (a partir del ciclo 6).

Problema 5. Suponga que las cuatro instrucciones siguientes

multd	f4, f1, f2	; f4=f1*f2 (ciclo 2)
addd	f2, f4, f1	; f2=f4+f1 (ciclo 2)
subd	f4, f4, f1	; f4=f4-f1 (ciclo 3)
addd	f5, f2, f3	; f5=f2+f3 (ciclo 3)

se han decodificado en los ciclos indicados entre paréntesis, introduciéndose en una estación de reserva común para todas las unidades funcionales de coma flotante. Teniendo en cuenta que el procesador superescalar dispone de un ROB para implementar la finalización ordenada, y que la emisión es desordenada. Indique (a) cómo evolucionaría el ROB para esas instrucciones; (b) en qué momento empieza y termina la ejecución de

las instrucciones; y (c) cuáles son los valores que quedan en los registros de la arquitectura al terminar, si inicialmente $f1=3.0$, $f2=2.0$, y $f3=1.0$.

NOTA: la multiplicación tarda 4 ciclos, y la suma y la resta 2 ciclos; hay tantas unidades funcionales como sea necesario para que no haya riesgos estructurales; y se pueden enviar, retirar, etc. dos instrucciones por ciclo como máximo)

Solución

Para resolver el problema se parte de la Tabla 12, donde se indican los ciclos en los que (1) las instrucciones se terminan de decodificar (ID) y han pasado a la estación de reserva, (2) comienza y termina la ejecución de la operación correspondiente a la instrucción (EX), (3) el resultado de operación se almacena en el ROB (ROB), y (4) el momento en que después de retirar la instrucción del ROB, los resultados se han almacenado en el banco de registros de la arquitectura (WB).

Las instrucciones 2 y 3 deben esperar que termine la ejecución de la instrucción 1 dado que dependen de $f4$. La instrucción 4 debe esperar que termine la ejecución de la instrucción 2.

Tabla 12. Ciclos y etapas del procesamiento de la secuencia de instrucciones del problema

Instrucciones		2	3	4	5	6	7	8	9	10	11	12
1	multd	$f4, f1, f2$	ID	EX	EX	EX	EX	ROB	WB			
2	addd	$f2, f4, f1$	ID					EX	EX	ROB	WB	
3	subd	$f4, f4, f1$		ID				EX	EX	ROB	WB	
4	addd	$f5, f2, f3$		ID						EX	EX	ROB
												WB

(a) El ROB empieza a llenarse al final del ciclo (2), después de haberse decodificado las dos primeras instrucciones.

Final de (2)

#	Instrucción	Reg.Dest.	Valor	OK	Unidad	Flush
0	1	$f4$	-	0	-	0
1	2	$f2$	-	0	-	0

Al finalizar el tercer ciclo se introducen en el ROB las dos instrucciones restantes, y también habrá empezado la multiplicación.

Final de (3)

#	Instrucción	Reg.Dest.	Valor	OK	Unidad	Flush
0	1	$f4$	-	0	mult	0
1	2	$f2$	-	0	-	0
2	3	$f4$	-	0	-	0
3	4	$f5$	-	0	-	0

Hasta el ciclo (7) no ocurre nada en el ROB (en relación con las instrucciones que indica el problema). Como se muestra a continuación, se habrá terminado la multiplicación en el ciclo (6), y el resultado se almacena durante el ciclo (7) en el campo

de valor del registro 0 del ROB, el bit de OK pasará a 1. También se ha iniciado durante el ciclo (7) la ejecución de las instrucciones 2, y 3. Por tanto, al final del ciclo (7) el estado del ROB será el siguiente:

Final de (7)

#	Instrucción	Reg.Dest.	Valor	OK	Unidad	Flush
0	1	f4	6.0	1	Mult	0
1	2	f2	-	0	Add	0
2	3	f4	-	0	Sub	0
3	4	f5	-	0	-	0

En el ciclo (8) se siguen ejecutando las instrucciones 2 y 3 (terminan al final de dicho ciclo) y se retira la primer instrucción del ROB. Por lo tanto, el estado del ROB es:

Final de (8)

#	Instrucción	Reg.Dest.	Valor	OK	Unidad	Flush
1	2	f2	-	0	Add	0
2	3	f4	-	0	Sub	0
3	4	f5	-	0	-	0

Al final del ciclo (9), los resultados de las instrucciones 2 y 3 estarán almacenado en los campos de *Valor* de las líneas 2 y 3 del ROB, cuyos campos de *OK* estarán al valor 1 (dato válido), y habrá empezado la ejecución de la instrucción 4:

Final de (9)

#	Instrucción	Reg.Dest.	Valor	OK	Unidad	Flush
1	2	f2	9.0	1	Add	0
2	3	f4	3.0	1	Sub	0
3	4	f5	-	0	Add	0

Al final del ciclo (10) se habrán retirado las instrucciones 2 y 3 y se escribirán sus resultados en los registros f2 y f4. La instrucción 4 sigue ejecutándose hasta el final del ciclo (10):

Final de 10

#	Instrucción	Reg.Dest.	Valor	OK	Unidad	Flush
3	4	f5	-	0	Add	0

Al finalizar el ciclo (11), el resultado de la instrucción 4 se habrá almacenado en el registro 3 del ROB.

Final de (11)

#	Instrucción	Reg.Dest.	Valor	OK	Unidad	Flush
3	4	f5	10.0	1	Add	0

En el ciclo (12), se retirará la instrucción y se escribirá el resultado en f5.

(b) Teniendo en cuenta la Tabla 12 se observa que la ejecución de las instrucciones empieza en el ciclo 3 y acaba en el ciclo 10. La Tabla 12 nos permite ver también

en que momento empieza y termina la ejecución de cada una de las instrucciones. El procesamiento de las instrucciones termina en el ciclo (12), una vez se ha retirado del ROB la instrucción 4 y se escriben los resultados en f5.

(c) A partir de la evolución del ROB (el orden en que se han retirado las instrucciones y se almacenan los resultados en el banco de registros) al retirar cada instrucción se tiene:

Instrucción 1: f1=3.0; f2=2.0; f3=1.0; **f4=3.0*2.0=6.0**

Instrucción 2: f1=3.0; **f2=6.0+3.0=9.0**; f3=1.0; f4=6.0

Instrucción 3: f1=3.0; f2=9.0; f3=1.0; **f4=6.0-3.0=3.0**

Instrucción 4: f1=3.0; f2=9.0; f3=1.0; f4=3.0; **f5=9.0+1.0=10.0**

Por lo tanto, al final los registros quedan con los valores: f1=3.0; f2=9.0; f3=1.0; f4=3.0; f5=10.0

Problema 6. Un procesador superescalar de 64 bits es capaz de captar (IF), decodificar (ID), emitir (ISS), y finalizar (WB), tres instrucciones por ciclo. Dispone de una ventana de instrucciones centralizada y un buffer de reordenamiento (ROB) para realizar el renombramiento de registros y la finalización ordenada de instrucciones. Además, el procesador permite los adelantamientos de *stores* por *loads* no especulativos, pero no permite los adelantamientos especulativos.

Indique, para la secuencia de instrucciones siguiente:

(1)	lf	f1,0(r1)	;	f1=[r1+0]
(2)	lf	f2,8(r1)	;	f2=[r1+8]
(3)	addf	f3,f2,f1	;	f3=f2+f1
(4)	sf	16(r1),f3	;	[r1+16]=f3
(5)	lf	f4,24(r1)	;	f4=[r1+24]
(6)	mulf	f5,f4,f1	;	f5=f4*f1
(7)	sf	8(r1),f5	;	[r1+8]=f5
(8)	addi	r2,r1,#32	;	r2=r1+32
(9)	lf	f6,0(r2)	;	f6=[r2+0]
(10)	addf	f7,f6,f3	;	f7=f6+f3
(11)	sf	8(r2),f7	;	[r2+8]=f7

(a) ¿En cuántos ciclos se completa el procesamiento de las instrucciones suponiendo emisión ordenada?

(b) ¿En cuántos ciclos se completa el procesamiento si la emisión es desordenada?

(c) ¿Y si es desordenada y se permite adelantamiento especulativo de stores por loads?

NOTA: El procesador dispone de una unidad de carga con un retardo de 2 ciclos, una de almacenamiento con retardo de 1 ciclo, dos ALUs con 1 ciclo de retardo, un multiplicador de coma flotante con 3 ciclos de retardo y dos sumadores de coma flotante

con 2 ciclos de retardo. Hay espacio suficiente para las instrucciones de la secuencia indicada en la cola de instrucciones, la ventana de instrucciones, y el ROB.

Solución

(a) La Tabla 13 muestra el procesamiento de la secuencia de instrucciones, a través de las distintas etapas del procesador, en el caso de emisión ordenada y adelantamiento no especulativo.

Como se puede ver en la Tabla 13, entre las instrucciones 1 y 2 hay una dependencia estructural dado que ambas necesitan la unidad de carga de memoria y solo hay una. Por eso la instrucción 2 debe esperar al ciclo (5), una vez que la instrucción 1 ha terminado su ejecución en el ciclo (4). La instrucción 3 depende de las instrucciones 1 y 2, y tiene que esperar a que termine la ejecución de la instrucción 2 para empezar la ejecución. La instrucción 4 también depende de la instrucción 3.

Tabla 13. Procesamiento de las instrucciones de la secuencia con emisión ordenada y adelantamiento no especulativo

Instrucciones		1	2	3	4	5	6	7	8	9	10
1	lf f1,0(r1)	IF	ID	EX	EX	ROB	WB				
2	lf f2,8(r1)	IF	ID			EX	EX	ROB	WB		
3	addf f3,f2,f1	IF	ID					EX	EX	ROB	
4	sf 16(r1),f3		IF	ID						EX	WB
5	lf f4,24(r1)		IF	ID						EX	EX
6	multf f5,f4,f1		IF	ID							
7	sf 8(r1),f5			IF	ID						
8	addi r2,r1,#32			IF	ID						
9	lf f6,0(r2)			IF	ID						
10	addf f7,f6,f3				IF	ID					
11	sf 8(r2),f7				IF	ID					

Tabla 13. Procesamiento de las instrucciones de la secuencia con emisión ordenada y adelantamiento no especulativo (**continuación**)

Instrucciones		11	12	13	14	15	16	17	18	19	20
1	lf f1,0(r1)										
2	lf f2,8(r1)										
3	addf f3,f2,f1										
4	sf 16(r1),f3										
5	lf f4,24(r1)	ROB	WB								
6	multf f5,f4,f1	EX	EX	EX	ROB	WB					
7	sf 8(r1),f5				EX	WB					
8	addi r2,r1,#32				EX	ROB	WB				
9	lf f6,0(r2)					EX	EX	ROB	WB		
10	addf f7,f6,f3						EX	EX	ROB	WB	
11	sf 8(r2),f7								EX	WB	

La instrucción 5 puede emitirse al mismo tiempo que la instrucción 4. La instrucción 4 escribe el resultado de la instrucción 3 en memoria y la instrucción 5 lee desde memoria, pero las direcciones de memoria a la que acceden son diferentes (es decir

$r1+16$ y $r1+24$) y no hay problema de dependencia de datos RAW. La instrucción 6 depende de la 5 y la 7 de la 6, por lo que deben empezar su ejecución una después de que acabe la otra. La instrucción 8 no depende de las anteriores pero como la emisión es ordenada se emitirá al mismo tiempo que la instrucción 7.

Finalmente, la instrucción 9 depende de la 8, la instrucción 10 de la 9, y la 11 de la 10. Por lo tanto, cada instrucción solo puede emitirse y empezar a ejecutarse cuando acaba la instrucción anterior.

Como muestra la Tabla 13 la secuencia de instrucciones tarda 20 ciclos en ejecutarse. Si tenemos en cuenta que un procesador superescalar es un procesador segmentado, podemos estimar el número de instrucciones que terminan de ejecutarse por ciclo a partir de la expresión:

$$T_{CPU}(\text{ciclos}) = T_{latencia}(\text{ciclos}) + (N - 1) \times CPI$$

y sustituyendo, se tiene:

$$20 = T_{latencia}(\text{ciclos}) + (N - 1) \times CPI = 6 + 10 \times CPI$$

y al despejar:

$$CPI = \frac{20 - 6}{10} = 1.4$$

Por lo tanto, dado que se necesita, por término medio, más de un ciclo por instrucción, el procesador superescalar funciona muy por debajo de su velocidad pico, que le permitiría finalizar tres instrucciones por ciclo, y por lo tanto tener un $CPI=0.33$. Funciona, incluso con un rendimiento peor que un procesador segmentado no superescalar.

(b) La Tabla 14 muestra el procesamiento de las instrucciones con emisión desordenada. Las instrucciones 1 a 4 se procesan de igual forma que en la Tabla 13 debido a las dependencias RAW y estructurales que existen entre ellas.

Tabla 14. Procesamiento de las instrucciones de la secuencia con emisión desordenada y adelantamiento no especulativo

Instrucciones	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1 $lf\ f1, 0(r1)$	IF	ID	EX	EX	ROB	WB								
2 $lf\ f2, 8(r1)$	IF	ID			EX	EX	ROB	WB						
3 $lddf\ f3, f2, f1$	IF	ID					EX	EX	ROB					
4 $lf\ 16(r1), f3$	IF	ID							EX	WB				
5 $lf\ f4, 24(r1)$	IF	ID					EX	EX	ROB	WB				
6 $mult\ f5, f4, f1$	IF	ID							EX	EX	EX	ROB	WB	
7 $sf\ 8(r1), f5$		IF	ID									EX	WB	
8 $addi\ r2, r1, \#32$		IF	ID	EX	ROB								WB	
9 $af\ f6, 0(r2)$		IF	ID						EX	EX	ROB			WB
10 $addf\ f7, f6, f3$			IF	ID						EX	EX	ROB	WB	
11 $sf\ 8(r2), f7$			IF	ID								EX	WB	

La instrucción 5 podría adelantar a la instrucción 4. Se trata de una instrucción de almacenamiento, pero las direcciones a las que acceden son diferentes ($r1+16$ y $r1+24$) y no habrá dependencia RAW. Se podría adelantar más ciclos, pero la unidad de carga de memoria se encuentra ocupada ejecutando la instrucción 1 y la instrucción 2.

Adelantar la carga de memoria correspondiente a la instrucción 5 permite adelantar las instrucciones 6 y 7, entre las que hay dependencias (la 6 depende de la 5 y la 7 de la 6).

El comienzo de la ejecución de la instrucción 8 puede adelantarse al ciclo 5 dado que tiene sus datos disponibles, la unidad funcional que necesita está libre (una de las ALUs) y tenemos emisión desordenada. Esto permite adelantar la instrucción 9 de carga de memoria que podría ejecutarse antes que la instrucción 7, de almacenamiento, que la precede. No se trataría de un adelantamiento especulativo dado que la instrucción 7 tendría calculada la dirección de acceso a partir del ciclo (4), y la instrucción 9 a partir del ciclo (6), una vez termina la ejecución de la instrucción 8. Por tanto se conocerían las direcciones y serían direcciones diferentes (una es $r1+8$ y otra es $r2=r1+32$).

Las instrucciones 9, 10 y 11 también tienen que emitirse una tras otra debido a las dependencias entre ellas. La finalización (WB) de la instrucción 9 debe retrasarse un ciclo dado que no se pueden retirar más de tres instrucciones por ciclo.

En cuanto al valor de *CPI* promedio que se consigue, se tiene que:

$$14 = T_{latencia}(ciclos) + (N - 1) \times CPI = 6 + 10 \times CPI$$

y, despejando:

$$CPI = \frac{14 - 6}{10} = 0.8$$

Aquí sí se pone de manifiesto un comportamiento superescalar, ya que por término medio se tiene menos de un ciclo por instrucción. Es decir, se termina más de una instrucción por ciclo: $1 / 0.8 = 1.25$. Aunque sigue siendo menor que el máximo de tres instrucciones finalizadas por ciclo.

(c) No hay cambio respecto al caso en el que solo se permite adelantamiento no especulativo de *stores* por *loads* que se ha analizado en el caso (b) anterior. Esto se debe a que, en el análisis que se ha realizado no se tiene ningún adelantamiento no especulativo.

Problema 7. Un procesador superescalares capaz de captar, decodificar y retirar hasta dos instrucciones por ciclo, con una única ventana de instrucciones con emisión desordenada y un buffer de reordenamiento (ROB) para el renombrado y la finalización ordenada.

El procesador utiliza un predictor de saltos dinámico de dos bits que se consulta al captar las instrucciones de salto condicional. Si la dirección de memoria de la instrucción

de salto se encuentra en un BTB (Branch Target Buffer) que tiene el procesador, y la predicción es saltar, se cambia el valor del PC por la dirección de destino del salto (que también estará almacenada en el BTB) para que se empiece a captar instrucciones desde ahí en el siguiente ciclo.

Cuando la instrucción de salto entra en el buffer de reordenamiento (ROB) tras la decodificación, se marca con un bit `pred=1` si el predictor decidió tomar el salto o `pred=0` en caso contrario. Posteriormente, cuando se resuelve la condición del salto en la etapa de ejecución, se comprueba si la predicción se realizó con éxito, y en caso contrario, se marcan con `flush=1` todas aquellas instrucciones que se han introducido en el cauce de forma especulativa para que no actualicen los registros al ser retiradas, y se fija PC para que apunte a la instrucción siguiente a la de salto (la dirección de la instrucción de salto está guardada en el BTB) y se capten instrucciones a partir de ahí en el siguiente ciclo.

Como la primera vez que se capta una instrucción de salto condicional no se encontrará información de ella en el BTB, no se podrá predecir su comportamiento hasta la etapa de decodificación, en la que se emplea un predictor estático que predice “saltar” en caso de instrucciones de salto condicional a direcciones más bajas que la dirección de la instrucción (saltos hacia atrás) y “no saltar” para las instrucciones de salto condicional a direcciones mayores que la dirección de la instrucción (saltos hacia delante), y crea una entrada en el BTB en la que se guarda la dirección de la instrucción de salto, la de destino del salto y los dos bits de historia (11 si finalmente se produce el salto ó 00 si no se produce) .

Para el código siguiente:

```

1           lw      r1, it      ; r1=(it)
3           lf      f1, a       ; f1=(a)
4           lf      f2, b       ; f2=(b)
5   bucle: addf  f2, f2, f1   ; f2=f2+f1
6           subi  r1, r1, #1   ; r1=r1-1
7           bnez  r1, bucle  ; si r1!=0 no salta
8           sw      b, f2      ; (b)=f2
9           trap   #0          ; fin del programa

```

- (a) Realice una traza de la ejecución del programa en el procesador suponiendo que el contenido de `it` es igual a 2. ¿Cuántos ciclos tarda en ejecutarse el programa?
- (b) Estime la penalización (en ciclos) que se produce en el procesamiento del salto tanto si acierta como si falla el predictor del procesador.
- (c) ¿Cuántos ciclos de penalización en total se sufrirían si el contenido de `it` fuera igual a 50?

NOTA: Suponga que hay tantas unidades de ejecución y de acceso a memoria como sea necesario para que no haya colisiones por riesgos estructurales, y una latencia de 2 ciclos

para las instrucciones de carga de memoria y las operaciones de suma en coma flotante, y de 1 ciclo para las instrucciones aritméticas con enteros y los almacenamientos. Las instrucciones que van a continuación de una instrucción de salto y se captan en el mismo ciclo que dicha instrucción no continúan su procesamiento si se predice “saltar”.

Solución

(a) La traza de ejecución del programa para las dos iteraciones correspondientes al valor almacenado inicialmente en la dirección `it` se muestra en la Tabla 15.

Dado que se nos indica que hay unidades de acceso a memoria y de procesamiento suficientes para no ocasionar colisiones por riesgos estructurales, las instrucciones 1 y 2 que se captan en el ciclo (1) se pueden ejecutar simultáneamente y la instrucción 3 que se capta en el ciclo (2) puede empezar a ejecutarse en el ciclo (4) en otra unidad de carga de datos diferente. Se utilizan, por tanto, tres unidades de carga de datos de memoria.

Tabla 15. Traza de procesamiento del código del problema para dos iteraciones, $it=2$

Instrucciones		1	2	3	4	5	6	7	8	9	10	11	12	13
1	lw r1, it			IF	ID	EX	EX	ROB	WB					
2	lf f1, a			IF	ID	EX	EX	ROB	WB					
3	lf f2, b				IF	ID	EX	EX	ROB	WB				
4	addf f2, f2, f1				IF	ID			EX	EX	ROB	WB		
5	subi r1, r1, #1					IF	ID	EX	ROB			WB		
6	bnez r1, bucle					IF	ID		EX	ROB			WB	
7	sf b, f2						IF	---						
8	trap #0							IF	---					
9	addf f2, f2, f1							IF	ID		EX	EX	ROB	WB
10	subi r1, r1, #1								IF	ID	EX	ROB		WB
11	bnez r1, bucle									IF	ID	EX		WB
12	sf b, r2									IF	---			
13	addf f2, f2, f1									IF	ID	flush	flush	WB*
14	subi r1, r1, #1										IF	ID	flush	WB*
15	bnez r1, bucle										IF	---		
16	sf b, r2										IF	---		
17	sf b, f2											IF	ID	EX
18	trap #0												IF	ID
Penalizaciones				P			P	P						

La instrucción 4 se capta en el ciclo (2) junto con la instrucción 3 pero debe esperar a que termine de ejecutarse la instrucción 3 para empezar su ejecución, dado que hay dependencia de datos entre las instrucciones 3 y 4 debido al uso del registro `f2`.

En cualquier caso, la instrucción 5 se puede empezar a ejecutar antes que la instrucción 4 dado que la emisión es desordenada. La instrucción de salto condicional (instrucción 6) se capta junto con la instrucción 5 y debería esperar un ciclo, hasta el ciclo (6), para empezar a ejecutarse, dado que depende de `r1`.

De todas formas, la predicción correspondiente a la instrucción 6 se realizaría cuando esté en la etapa ID dado que es la primera vez que se capta y no hay información de ella en el BTB. Como se trata de una instrucción de salto hacia atrás la predicción es “saltar”, que coincidiría con lo que se determinaría en la etapa de ejecución EX, en el ciclo (6), puesto que $r1$ es distinto de cero ($r1$ era igual a 2 inicialmente, pero la instrucción 5 ha hecho $r1=1$). Dado que la predicción es correcta, las instrucciones 9 y 10 que se captaron en el ciclo (5), tras la predicción realizada en el ciclo (4) por la instrucción 6, seguirán su procesamiento. Las instrucciones 7 y 8 que se captaron en el mismo ciclo (4), cuando se estaba haciendo la predicción, simplemente se desechan y no pasarán a decodificarse, ni al ROB. Se considera que la instrucción de salto no tiene etapa de escritura en el buffer de reordenamiento (ROB).

El comienzo de la ejecución de la instrucción 9 debe esperar un ciclo dado que tiene una dependencia RAW con la instrucción 4 por el uso de $f2$. No obstante, como la emisión es desordenada, la instrucción 10 puede emitirse para empezar su ejecución antes que la instrucción 9, en el ciclo (7).

La instrucción de salto se vuelve a captar en el ciclo (6) junto con la instrucción que le sigue de escritura en memoria (instrucciones 11 y 12). Ahora se tiene información en el BTB para la instrucción de salto, y por lo tanto ya se puede hacerla predicción en su etapa de captación. Dado que en la ejecución anterior de esta instrucción de salto condicional se produjo el salto, los dos bits de historia se cargaron con 11, y la predicción es “saltar”. Por lo tanto, en el ciclo (7) se captarían las instrucciones 13 y 14. La instrucción 12, que se captó con la propia instrucción de salto 11, se desecha y no pasa ni a la etapa de decodificación ni se introduce en el ROB.

Sin embargo, cuando la instrucción de salto 11 llega a su etapa de ejecución, en el ciclo (8), ya la instrucción 10 se ha ejecutado en el ciclo (7) y ha hecho $r1=0$. Por lo tanto no se verifica la condición de salto y la predicción ha sido errónea. A partir de la información almacenada en el BTB se determinará la dirección a partir de la que tiene que proseguir la ejecución: la siguiente a la dirección de la instrucción de salto (la instrucción de almacenamiento en memoria). Esta instrucción se captaría en el ciclo (9).

Las instrucciones que se han captado en los ciclos (7) y (8) se han introducido erróneamente en el cauce. Las instrucciones 13 y 14 se habrán introducido en el ROB y se marcan como flush para que no tengan efecto al retirarse del ROB. Las instrucciones 15 y 16, que solo se habían captado, se pueden desechar.

Se ha considerado que las instrucciones que están marcadas como flush realmente no cuentan como instrucciones retiradas dado que no producen ningún efecto (son las etapas marcadas con WB*). Simplemente se produce un salto en el puntero correspondiente del ROB. Así, en el ciclo (12) realmente se retiran dos instrucciones, la 11 y la 17.

(b) Las penalizaciones para las predicciones tienen que tener en cuenta que el tipo de predicción depende de que sea la primera vez que se capta la instrucción (predicción estática) o de que estén definidos los bits de historia en las siguientes iteraciones.

Para la predicción estática:

- $P_{est_OK} = 1$ ciclos, dado que la predicción se hace en la etapa ID
- $P_{est_fallo} = 3$ ciclos, dado que hasta el final de la etapa EX de la instrucción de salto no se ha evaluado la condición para determinar si la predicción ha sido incorrecta.

Para la predicción dinámica de dos bits:

- $P_{din_OK} = 0$ ciclos, dado que la predicción se hace en la etapa IF
- $P_{din_fallo} = 2$ ciclos, dado que hasta el final de la etapa EX de la instrucción de salto no se ha evaluado la condición para determinar si la predicción ha sido incorrecta. Como en este caso se ha captado la instrucción que actualiza r1 en el ciclo (5), anterior al ciclo en el que se capta la instrucción de salto condicional (ciclo (6)), la etapa EX de la instrucción de salto no tiene que esperar un ciclo como ocurría en el fallo con la predicción estática.

La traza de la Tabla 15 corresponde a la ejecución del programa para $it=2$. Como se puede ver, se produce una predicción estática correcta, y una predicción dinámica incorrecta, por lo tanto la penalización total es $P = 1 + 2 = 3$ ciclos, tal y como está indicado en la Tabla 15.

(c) Para obtener la penalización en el caso de 50 iteraciones podemos tener en cuenta la siguiente expresión:

$$P(it) = P_{est_OK} + (it - 2) \times P_{din_OK} + P_{din_fallo} = 1 + (50 - 2) \times 0 + 2 = 3$$

Dado que cuando la predicción dinámica es correcta no hay penalización, solo hay penalización en la predicción estática correcta correspondiente a la primera iteración y en la predicción dinámica incorrecta de la última iteración. Así, independientemente del número de iteraciones, se tienen 3 ciclos de penalización.



Problema 8. En un programa, una instrucción de salto condicional (a una dirección de salto posterior) tiene el siguiente comportamiento en una ejecución de dicho programa:

NSSS NNSS NSSN SSSN

donde S indica que se produce el salto y N que no. Indique la penalización que se introduce si se utiliza: (a) Predicción fija (siempre se considera que se va a producir el salto); (b) Predicción estática (si el desplazamiento es negativo se toma y si es positivo no); (c) Predicción dinámica con dos bits, inicialmente en el estado (11); y (d) Predicción dinámica con tres bits, inicialmente en el estado (111).

NOTA: La penalización por saltos incorrectamente predichos es de 4 ciclos y para los saltos correctamente predichos es 0 ciclos.

Solución

Para conocer la penalización en cada una de las alternativas se tiene en cuenta la predicción que hace cada uno de los esquemas que se considera, y se compara con lo que finalmente ocurre (se produce o no se produce el salto).

(a) Dado que en el esquema de predicción fija considerado siempre se predice que se produce el salto, habrá penalización en todos los casos es los que no se produce (es decir, aparece N en la secuencia que se analiza). Como en NSSS NNSS NSSN SSSN hay seis N, la penalización es 6 penalizaciones \times 4 ciclos/penalización = 24 ciclos.

(b) El esquema predice *saltar* si el salto condicional se produce a una dirección anterior (desplazamiento negativo) y *no saltar* si se produce a una posición posterior (desplazamiento positivo). Como en este caso la instrucción saltaría a una dirección posterior, se produciría una penalización si se produce el salto dado que la predicción sería *no saltar*. Como en NSSS NNSS NSSN SSSN hay diez S, la penalización es 10 penalizaciones \times 4 ciclos/penalización = 40 ciclos.

(c) En este procedimiento de salto, al ser dinámico, hay que tener en cuenta el estado de los dos bits de historia que utiliza el procedimiento de predicción. La Tabla 16 indica los cambios de bits de historia, las predicciones y las penalizaciones que se producen.

Tabla 16. Evolución de bits de historia y predicciones para la secuencia

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Se Produce	N	S	S	S	N	N	S	S	N	S	S	N	S	S	S	N
Bits Historia	11	10	11	11	11	10	01	10	11	10	11	11	11	10	11	11
Se Predice	S	S	S	S	S	S	N	S	S	S	S	S	S	S	S	S
Penalización	X				X	X	X		X		X					X

La Figura 28 se muestra el diagrama de estados que utiliza el procedimiento para actualizar los bits de historia y realizar las predicciones. Inicialmente, los bits de historia se encuentran en el estado 11 y predicen *saltar*. Como se produce *no saltar* hay penalización y se produce la transición al estado 10 en el que la predicción sigue siendo *saltar*.

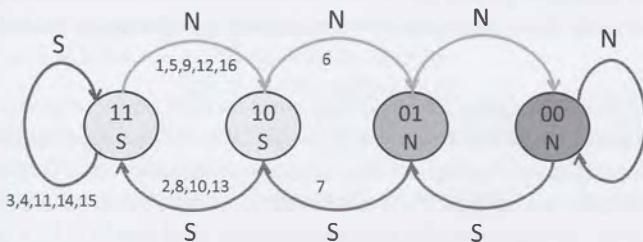


Figura 28. Diagrama de estados para la predicción dinámica de dos bits y transiciones para la secuencia

En el diagrama de la Figura 28, esta transición se indica marcando el correspondiente arco del grafo con “1” (es la primera ejecución de la instrucción de salto). Así, se irán marcando las transiciones para las siguientes ejecuciones de la instrucción de salto (2,...,16). El estado de los bits de historia tras la ejecución número 16 de la instrucción de salto es 10. El número de penalizaciones es siete, y por lo tanto la penalización total es 7 penalizaciones \times 4 ciclos/penalización = 28 ciclos.

(d) Cuando se utiliza la predicción dinámica con tres bits, la predicción que se hace viene determinada por el dígito (0 o 1) que aparece más veces en los bits de historia. En nuestro caso consideraremos que si hay mayoría de unos se predice *salir* y si hay mayoría de ceros se predice *no salir*. Tras la predicción, se desplazan los bits de historia hacia la derecha, y se pierde el bit menos significativo, en tanto que como bit más significativo se introduce un uno si se produce un salto o un cero si no se produce un salto. En la Tabla 17 se muestra este comportamiento con la secuencia que indica el problema.

Como se puede observar en la Tabla 17, aquí casualmente las penalizaciones se producen en los mismos lugares para la predicción con tres y con dos bits de historia. Se tienen, por tanto siete penalizaciones, que dan lugar a un total de 28 ciclos de penalización.

Tabla 17. Evolución de bits de los tres bits de historia y predicciones para la secuencia dada

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Se Produce	N	S	S	S	N	N	S	S	N	S	S	N	S	S	S	N
Bits Historia	111	011	101	110	111	011	001	101	110	011	101	110	011	101	110	111
Se Predice	S	S	S	S	S	S	N	S	S	S	S	S	S	S	S	S
Penalización	X				X	X	X		X			X				X

Problema 9. Para el bucle siguiente:

```

for (i=1; i<=5; i++)
{
    if (c=5) goto etiqueta;
    r=s+t;
    c=c+1;
    if (c>5) then goto etiqueta;
    r=r+t;
}
etiqueta:.....

```

un compilador ha generado el código ensamblador que se muestra en la Figura 29.

```

1      addi r1,r0,#5 ; r1=5 (iteraciones)
2      lw r2,datoc ; r2=c [datoc]=c
3      add r4,r1,r0 ; r4=r1+r0 (r4=r1=5)
4      bucle: sub r3,r1,r2 ; r3=5-c
5          beqz r3,etiqueta ; si r3=0(c=5) ir a etiqueta,1
6          addf f3,f2,f1 ; f3=f2+f1 (r=s+t)
7          addi r2,r2,#1 ; r2=r2+1 (c=c+1)
8          sub r3,r2,r1 ; r3=c-5
9          bgtz r3,etiqueta ; si r3>0(c>5) ir a etiqueta,2
10         addf f3,f3,f1 ; f3=f3+f1 (r=r+t)
11         subi r4,r4,#1 ; r4=r4-1 (c=c+1)
12         bnez r4,etiqueta ; si r4!=0 ir a bucle,3
13 etiqueta:

```

Figura 29. Código generado por el compilador para el programa del problema

Indique cuál es la penalización efectiva debida a los saltos, en función del valor inicial de c (número entero), considerando que el procesador utiliza: (a) Predicción estática (si el desplazamiento es negativo se predice saltar y si es positivo se predice no saltar); (b) Predicción dinámica con un bit (1 = predice Saltar; 0 = No Saltar; Se inicializa el bit de historia a 1); (c) Predicción dinámica con dos bits (11=predice Saltar; 10= Saltar; 01= No Saltar; 00= No Saltar; Se inicializan los bits de historia a 11). (d) ¿Cuál de los tres esquemas es más eficaz por término medio si hay un 50% de probabilidades de que c sea menor o igual a 0, un 25% de que sea mayor que 5; y un 25% de que sea cualquier número entre 1 y 5, siendo todos equiprobables?

NOTA: La penalización por saltos incorrectamente predichos es de 4 ciclos y para los saltos correctamente predichos es 0 ciclos.

Solución

Para resolver el problema se empieza por determinar los valores de las penalizaciones según el valor de la variable c , en cada uno de los procedimientos de penalización considerados y para cada una de las instrucciones de salto condicional que hay en el código.

En concreto, se tienen tres instrucciones de salto, la 5, 9, y 12. Las instrucciones 5 y 9 son saltos condicionales hacia adelante (hacia direcciones de memoria mayores) y la instrucción 12 es una instrucción de salto condicional hacia atrás (hacia direcciones de memoria menores) que controla el bucle. En lo que sigue, denominaremos instrucciones de salto condicional 1, 2, y 3, respectivamente, a las instrucciones de salto 5, 9, y 12. Si la instrucción de salto condicional i ($i=1,2,3$) da lugar a un salto, lo notaremos como Si, y si no da lugar a salto, Ni.

Para determinar el perfil de salto/no salto de cada una de las instrucciones se tiene en cuenta que no habrá más de 5 iteraciones, dado que el registro $r4$ que se utiliza para controlar el bucle se hace $r4=5$ en la instrucción 3. En la Tabla 18 se muestra la secuencia de Saltos/No saltos para las distintas instrucciones, según los valores de c , indicando entre paréntesis los valores de c y $r4$ (control del bucle) en cada iteración, en el momento en que la instrucción de salto condicional correspondiente se va a ejecutar.

Tabla 18. Secuencias de Saltos/No Saltos según el valor de c y las iteraciones del bucle

Iteración i	1		2		3		4		5	
$c \leq 0$	$(c \leq 0)$	N1	$(c \leq 1)$	N1	$(c \leq 2)$	N1	$(c \leq 3)$	N1	$(c \leq 4)$	N1
	$(c \leq 1)$	N2	$(c \leq 2)$	N2	$(c \leq 3)$	N2	$(c \leq 4)$	N2	$(c \leq 5)$	N2
	$(r4=4)$	S3	$(r4=3)$	S3	$(r4=2)$	S3	$(r4=1)$	S3	$(r4=0)$	N3
$c = 1$	$(c = 1)$	N1	$(c = 2)$	N1	$(c = 3)$	N1	$(c = 4)$	N1		
	$(c = 2)$	N2	$(c = 3)$	N2	$(c = 4)$	N2	$(c = 5)$	N2	$(c = 5)$	S1
	$(r4=4)$	S3	$(r4=3)$	S3	$(r4=2)$	S3	$(r4=1)$	S3		
$c = 2$	$(c = 2)$	N1	$(c = 3)$	N1	$(c = 4)$	N1				
	$(c = 3)$	N2	$(c = 4)$	N2	$(c = 5)$	N2	$(c = 5)$	S1		
	$(r4=4)$	S3	$(r4=3)$	S3	$(r4=2)$	S3				
$c = 3$	$(c = 3)$	N1	$(c = 4)$	N1						
	$(c = 4)$	N2	$(c = 5)$	N2	$(c = 5)$	S1				
	$(r4=4)$	S3	$(r4=3)$	S3						
$c = 4$	$(c = 4)$	N1								
$c = 5$	$(c = 5)$	N2	$(c = 5)$	S1						
$c > 5$	$(c > 5)$	S2								
	$(c > 6)$									

Se puede resumir la secuencia de saltos/no saltos de la Tabla 18 como se indica a continuación:

Si $c \leq 0$, la secuencia es $(N1N2S3)^4N1N2N3$, que es equivalente a:

$$N1N2S3N1N2S3N1N2S3N1N2S3N1N2N3$$

es decir, la secuencia para la instrucción de salto 1 es $N1N1N1N1 = (N1)^5$, para la instrucción de salto 2 es $N2N2N2N2 = (N2)^5$, y para la instrucción de salto 3 es $N3N3N3N3 = (N3)^5$

Si $1 \leq c \leq 5$, la secuencia es $(N1N2S3)^{5-c}S1$

Si $c > 5$, la secuencia de $N1S2$

Una vez determinadas las secuencias de saltos/no saltos que ocasionan las instrucciones en función del parámetro c podemos determinar la penalización para cada tipo de predictor, en cada caso.

(a) Para la predicción estática, las instrucciones de salto 1 y 2 saltan hacia adelante. Por lo tanto, para ellas la predicción es no saltar, y no se produce penalización si no se produce salto (N). En cambio, la instrucción 3 salta hacia atrás, y por tanto, no hay penalización si se produce el salto (S). Por tanto las penalizaciones son:

- Si $c \leq 0$, como la secuencia es $(N1N2S3)^4N1N2N3$, solo hay penalización debido al $N3$ último, es decir $P=4$.
- Si $1 \leq c \leq 5$, la secuencia es $(N1N2S3)^5-cS1$, solo hay penalización debido al $S1$ último, es decir $P=4$.
- Si $c > 5$, la secuencia es $N1S2$, y solo hay penalización debido al $S2$ último. Es decir $P=4$.

(b) En este caso la predicción es dinámica con un bit de historia, que se inicializa a 1. El diagrama de estados para este tipo de predicción se muestra en la Figura 30(a).

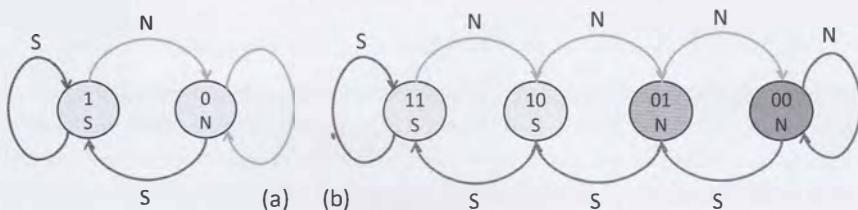


Figura 30. Diagrama de estados para la predicción dinámica con (a) un bit y (b) dos bits de historia

En este caso, hay que tener en cuenta la secuencia de instrucciones de salto de cada tipo, por separado:

- Si $c \leq 0$, la secuencia para la instrucción 1 es $(N1)^5$, se producirá una penalización para la primera ejecución ($N1$) dado que el bit de historia se inicializa a 1, y por lo tanto la predicción es saltar (según la Figura 30(a)), se produce la transición al estado 0, donde la predicción es no saltar y no se producirán más penalizaciones, por tanto, $P=4$ para $(N1)^5$. Para la instrucción 2, $(N2)^5$, y se tendrá lo mismo que en el caso anterior, $P=4$. En la instrucción 3, se tiene $(S3)^5N3$, y la penalización se produce en el último $N3$ ya que para los $S3$ el estado se mantendrá en el valor 1 al que estaba inicializado. Es decir $P=4$ para la instrucción 3. Por lo tanto, la penalización total será igual a $P=3 \times 4=12$ ciclos.
- Si $1 \leq c < 5$, dado que secuencia es $(N1N2S3)^5-cS1$, se tiene $(N1)^5-cS1$ para la instrucción 1, y habrá dos penalizaciones (el primer $N1$ y el último $S1$) salvo en el caso de que $c=5$. Para la instrucción de salto 2 la secuencia de saltos/no saltos es $(N2)^5-c$ y solo habrá penalización en la primera ejecución (el primer $N2$). Finalmente, la instrucción de salto 3 da lugar a la secuencia $(S3)^5-c$ y no genera penalización dado que el bit de historia está inicializado a 1, empieza prediciendo saltar y se mantiene ahí. El total de penalizaciones para estos valores de c es $P=3 \times 4=12$.
- Si $c=5$, la secuencia es $S1$, y no habrá penalización porque solo se ejecuta la instrucción de salto 1 y ésta da lugar a salto.
- Si $c > 5$, la secuencia para la instrucción 1 es $N1$ y habrá una penalización en este caso, dado que el bit de historia se inicializa a 1, y la predicción es saltar. En el caso

de la instrucción 2, como se produce un salto (S2), no hay penalización. Por lo tanto, en este caso solo hay una penalización, y $P=4$.

(c) En este caso se razona utilizando las secuencias para cada instrucción, igual que se ha hecho en el caso (b), pero se tiene en cuenta el diagrama de estados de la Figura 30(b) para un predictor dinámico con dos bits de historia.

- Si $c \leq 0$, la secuencia para la instrucción 1 es $(N1)^5$, se producirá una penalización para la primera ejecución ($N1$) dado que el bit de historia se inicializa a 11 y la predicción es saltar (según la Figura 30(a)), se produce la transición al estado 10, donde la predicción también es saltar y se produce otra penalización. Después se produce la transición al estado 01 donde la predicción es no saltar y no se producirán más penalizaciones, por lo que $P=8$ para $(N1)^5$. Para la instrucción 2, $(N2)^5$, y se tendrá lo mismo que en el caso anterior: $P=8$. En la instrucción 3, se tiene $(S3)^5N3$, y la penalización se produce en el último $N3$ ya que para los $S3$ el estado se mantendrá en el valor 11 al que estaba inicializado. Es decir $P=4$ para la instrucción 3. Por lo tanto, la penalización total será igual a $P=5 \times 4=20$ ciclos.
- Si $1 \leq c \leq 5$, se tiene $(N1)^{5-c}S1$ para la instrucción 1. Si $c \leq 3$, tendremos más de dos $N1$ seguidos, y habrá dos penalizaciones debido al primer y segundo $N1$, y otra más debido al último $S1$. Para $c=4$ se tendrá $N1S1$ y solo habrá penalización para el primer $N1$. Para $c=5$, se tiene $S1$ y no habrá penalización. Para la instrucción de salto 2 la secuencia de saltos/no saltos es $(N2)^{5-c}$. Si $c \leq 3$, tendremos más de dos $N2$ seguidos y habrá dos penalizaciones. Si $c=4$ habrá un $N2$, y por lo tanto una penalización. Finalmente, la instrucción de salto 3 da lugar a la secuencia $(S3)^{5-c}$ y no genera penalización dado que el bit de historia está inicializado a 1, empieza prediciendo saltar y se mantiene ahí. El total de penalizaciones para estos valores de c son los siguientes: si $c \leq 3$, $P=(3+2+0) \times 4=20$ ciclos; si $c=4$, $P=(1+1) \times 4=8$; y si $c=5$, $P=0$
- Si $c > 5$, la secuencia para la instrucción 1 es $N1y$ habrá una penalización en este caso dado que el bit de historia se inicializa a 1 y la predicción es saltar. En el caso de la instrucción 2, como se produce un salto (S2) no hay penalización. Por lo tanto, en este caso solo hay una penalización, y $P=4$.

En la Tabla 19 se resumen las penalizaciones para cada valor de c y los distintos tipos de procedimiento de predicción considerados.

Tabla 19. Penalizaciones según procedimiento de predicción y valor de c

	Estática	Dinámica (1 bit)	Dinámica (2 bits)
$c \leq 0$	4	12	20
$c=1$	4	12	20
$c=2$	4	12	20
$c=3$	4	12	20
$c=4$	4	12	8
$c=5$	4	0	0
$c > 5$	4	4	4

A partir de la Tabla 19 podemos determinar el mejor esquema según las probabilidades que tengamos de que c tome uno u otro valor. Tenemos que hay el 50% de probabilidades de que $c \leq 0$, el 25% de que $c > 5$, y el 25% de que $1 \leq c \leq 5$ siendo, en este caso, las probabilidades iguales para todos los valores de c , y por lo tanto iguales al 5%. Por lo tanto, las expresiones para la penalización media en cada esquema de predicción serán:

Caso (a). Dado que para cualquier valor de c la penalización de la predicción estática es 4, tenemos que $P_{estatica} = 4 \text{ ciclos}$.

Caso (b). La penalización media para la predicción dinámica con un bit de historia P_{Din1} se obtiene de la expresión:

$$\begin{aligned} P_{Din1} &= (p(c \leq 0) + p(c = 1) + p(c = 2) + p(c = 3) + p(c = 4)) \times 12 + \\ &\quad + p(c = 5) \times 0 + p(c > 5) * 4 = \\ &= (0.5 + 0.05 + 0.05 + 0.05) \times 12 + 0.25 \times 4 = 9.4 \text{ ciclos} \end{aligned}$$

Caso (c). En el caso de predicción dinámica con dos bits de historia, la penalización media P_{Din2} es:

$$\begin{aligned} P_{Din2} &= (p(c \leq 0) + p(c = 1) + p(c = 2) + p(c = 3)) \times 20 + (8 \times p(c = 4)) + \\ &\quad + p(c = 5) \times 0 + p(c > 5) * 4 = \\ &= (0.5 + 0.05 + 0.05 + 0.05) \times 20 + 0.05 \times 8 + 0.25 \times 4 = 14.4 \text{ ciclos} \end{aligned}$$

En este caso concreto, la mejor opción es la predicción estática, seguida por la dinámica de un bit de historia. La peor es la dinámica con dos bits de historia. Obviamente, si las probabilidades de que c tome los distintos valores cambian, el mejor esquema puede cambiar. Por ejemplo, si las probabilidades de que c tome los valores 4 ó 5 fuesen muy elevadas en relación con las demás, la predicción dinámica con dos bits de historia sería más competitiva. También sería interesante analizar qué cambios se producirían si en lugar de inicializar los bits de historia a 1 y a 11, se inicializan a 0 y a 00.



Problema 10. Indique cómo se podrían utilizar instrucciones con predicado para mejorar el paralelismo entre instrucciones (ILP) de la sentencia:

if (A=0) then A=A-1; else A=A+1;

NOTA: Puede utilizar las instrucciones con predicado que considere, para obtener el comportamiento más eficiente para este trozo de código.

Solución

Una posible implementación de la sentencia indicada en el programa mediante instrucciones máquina típicas sería:

```

lw      R1,0(R3)      ; Cargar A (cuya dirección está en R3)
bnez   R1,dir1        ; Saltar a dir1 si R1(=A) no es cero
subi   R1,R1,1         ; R1 = R1 - 1
j      dir2            ; Saltar a dir2
dir1: addi  R1,R1,1    ; R1 = R1 + 1
dir2: sw    0(R3),R1   ; Almacenar R1 en A

```

Mediante la utilización de predicados se podría transformar el código como se indica a continuación:

```

lw      R1,0(R3)      ; Cargar A (dirección en R3)
P1,P2 cmp   R1,0        ; Si R1=0, P1=1(P2=0), y si no P2=1(P1=0)
(P1)    subi  R1,R1,1    ; R1 = R1-1 si P1=1
(P2)    addi  R1,R1,1    ; R1 = R1+1 si P2=1
        sw    0(R3),R1   ; Almacenar R1 en A

```

De esta forma se evitan instrucciones de salto, pero existen dependencias de tipo WAR y WAW entre las instrucciones de suma y resta (las dos instrucciones leen R1 y escriben en R1). Estas dependencias pueden no ser resueltas por el hardware y ocasionar que se pierdan algunos ciclos. La situación se puede mejorar con el siguiente código:

```

lw      R1,0(R3)      ; Cargar A (dirección en R3)
P1,P2 cmp   R1,0        ; Si R1=0, P1=1(P2=0), y si no P2=1(P1=0)
(P1)    subi  R2,R1,1    ; R2 = R1-1 si P1=1
(P2)    addi  R4,R1,1    ; R4 = R1+1 si P2=1
(P1)    sw    0(R3),R2   ; Almacenar R2 en A
(P2)    sw    0(R3),R4   ; Almacenar R4 en A

```

Así, utilizando los predicados también para las instrucciones de almacenamiento se pueden renombrar los operandos. Al desaparecer dependencias explícitas, el compilador puede reorganizar el código con más libertad y aprovechar mejor las posibilidades que ofrece la máquina.

Problema 11. Suponga un procesador en el que a todas las instrucciones se le puede anteponer un predicado, p , con dos valores posibles, 0 y 1 (o “true” y “false”):

(p) instrucción

Así, si la instrucción tiene predicado sólo se ejecutaría si el valor del predicado p es 1 (o “true”). Dicho valor habrá sido establecido por una instrucción de comparación con el formato

[p1,p2] cmp.cnd rx, ry

donde cnd es la condición que se comprueba entre rx y ry (lt, le, gt, ge, eq, n̄e). Si la condición es verdadera, $p1=1$ [y $p2=0$], y si es falsa, $p1=0$ [y $p2=1$].

(a) Escriba la secuencia de instrucciones máquina que implementarían el siguiente código sin utilizar ninguna instrucción de salto:

```

if ((x≥y) and (y≥0)) then x=x+y;
else if ((y<0) and (x<y)) then x = 2*y; •

```

- (b) Optimice el código anterior para un procesador VLIW con dos slots de emisión, en el que las instrucciones de comparación sólo pueden colocarse en el primero de ellos.

NOTA: los datos x e y son de 32 bits y están en posiciones consecutivas de memoria y los predicados no se pueden suponer inicializados a ningún valor; y las latencias de las operaciones son de 1 ciclo para las sumas, restas y comparaciones, de tres ciclos para las multiplicaciones y de cuatro ciclos para las cargas de memoria).

Solución

- (a) Para asignar correctamente los predicados y evitar instrucciones de salto, es conveniente considerar el correspondiente diagrama de flujo del código. La Figura 31 proporciona dicho diagrama, para cuya elaboración se ha tenido en cuenta que para cargar los predicados se utilizan instrucciones de comparación de dos operandos y se han indicado los predicados que habría que utilizar con el valor que deberían tomar para habilitar la ejecución del camino correspondiente del diagrama de flujo.

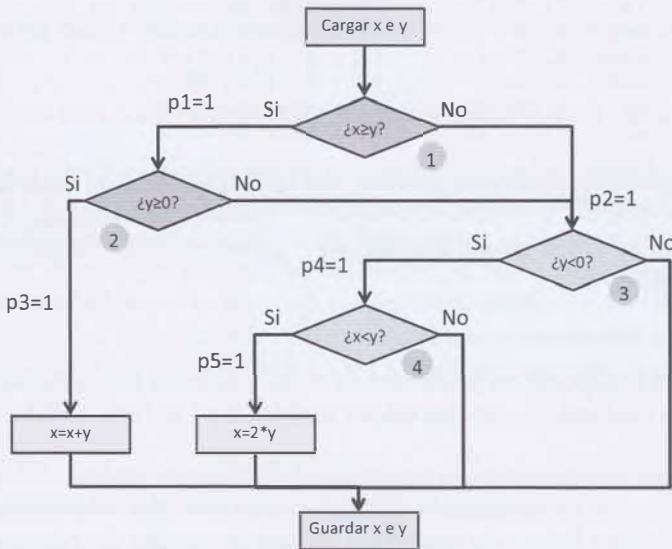


Figura 31. Diagrama de flujo para el código del programa

Las comparaciones marcadas con 1 y 2 en la Figura 31 corresponden a la condición del "if" del código y las marcadas con 3 y 4 a la alternativa del "else". Para que se ejecute $x=x+y$ es necesario que cumplan 1 y 2. Se puede utilizar un predicado p_1 que sólo si se cumplen 1 y 2 termina siendo igual a 1. Si cualquiera de las dos comparaciones no se cumple, se pasaría a ejecutar la opción del "else". Se puede utilizar un predicado, p_2 , que termina siendo igual a 1 si no se cumplen 1 ó 2.

El código a ejecutar sería el siguiente:

```

1           lw      r1,x      ; r1 = x
2           lw      r2,x      ; r2 = y
3       p3   cmp.ne r0,r0      ; Inicializamos p3=0
4       p4   cmp.ne r0,r0      ; Inicializamos p4=0
5       p5   cmp.ne r0,r0      ; Inicializamos p5=0
6   p1, p2 cmp.ge r1,r2      ; ¿x >= y?
7 (p1) p3, p2 cmp.ge r2,r0      ; ¿y >= 0?
8 (p2) p4   cmp.lt r2,r0      ; ¿y < 0?
9 (p4) p5   cmp.lt r1,r2      ; ¿x < y?
10 (p3)      add    r3,r1,r2 ; r3 = r1+r2
11 (p5)      slli   r3,r1,#1 ; r3 = 2*r1
12           sw      x,r3      ; x = r3

```

En dicho código, las instrucciones 3, 4 y 5 aseguran que los predicados p3, p4, y p5 toman el valor 0. Si no se hiciera esto, no podríamos saber con certeza el valor que toman esos predicados para las instrucciones 9, 10, y 11 dado no se puede asegurar que las instrucciones 7, 8, y 9 (las instrucciones en los que se asigna valor a p3, p4, y p5) se ejecuten (la ejecución depende de los valores de los valores de sus predicados).

Tras las instrucciones 6 y 7, p3 será igual a 1 si las condiciones de las instrucciones 6 y 7 se verifican las dos, y p2=1 si alguna de las dos condiciones de comparación no se cumple. Si p3 es igual a 1 se ejecutaría la instrucción 10, y si p2=1 se ejecutaría la instrucción 8. Si la condición se verifica para la comparación, se haría p4=1 y se ejecutaría 9, y si se verifica la condición tendríamos p5=1. Por tanto, solo si se verifican las dos condiciones tendríamos que p5=1, y entonces se podría ejecutar la instrucción 11. La instrucción 12 se ejecutaría siempre. En realidad podríamos ahorrarnos su ejecución si se introduce únicamente para los casos en los que p3 o p5 son iguales a 1. Teniendo en cuenta la ubicación de las instrucciones que se muestra en la Tabla 20 lo único que ocurriría es que habría dos instrucciones de almacenamiento, una en cada *slot*, que dependerían de los correspondientes predicados, p3, y p5. El interés de utilizarlas es que al ahorrarse accesos a memoria innecesarios no se ocuparía más ancho de banda de acceso a memoria que el imprescindible.

Tabla 20. Código VLIW para el diagrama de la Figura 31

slot 1	slot 2
lw r2,x	lw r1,x
cmp.ne r0,r0	nop
cmp.ne r0,r0	nop
p5 cmp.ne r0,r0	nop
p1, p2 cmp.ge r1,r2	nop
(p1) p3, p2 cmp.ge r2,r0	nop
(p2) p4 cmp.lt r2,r0	(p3) add r3,r1,r2
(p4) p5 cmp.lt r1,r2	nop
nop	(p5) slli r3,r1,#1
nop	sw x,r3

Tal y como están distribuidas las operaciones en la Tabla 20, cuando se necesitan los datos $r1$ y $r2$ ya han pasado los cuatro ciclos de retardo de la instrucción de carga de memoria. Obviamente, la operación $y=2*x$ no se implementa como una multiplicación, sino como un desplazamiento a la izquierda.

Es posible reducir el número de predicados en este problema. De hecho, el mismo predicado $p2$ se podría utilizar de forma similar a $p1$ para la alternativa del "else". Así, solo si las comparaciones 3 y 4 se cumplen, $p2$ acabaría siendo igual a 1 y se ejecuta $x=2*x$.

El código a ejecutar sería el siguiente:

```

        lw      r1,x      ; r1 = x
        lw      r2,x      ; r2 = y
(p1)  p1, p2  cmp.ge  r1,r2  ; ¿x >= y?
(p1)  p1, p2  cmp.ge  r2,r0  ; ¿y >= 0?
(p2)  p2    cmp.lt   r2,r0  ; ¿y < 0?
(p2)  p2    cmp.lt   r1,r2  ; ¿x < y?
(p1)  add    r3,r1,r2  ; r3 = r1+r2
(p2)  slli   r3,r1,#1  ; r3 = 2*r1
        sw      x,r3      ; x = r3

```

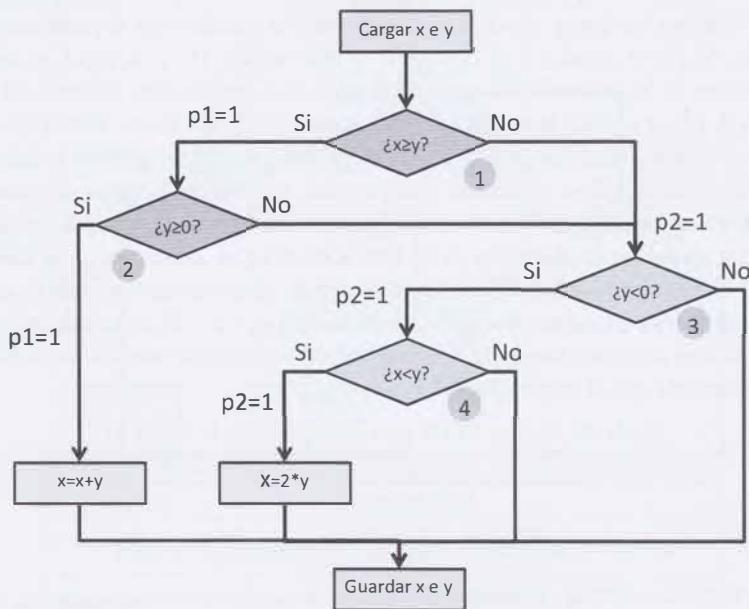


Figura 32. Diagrama de flujo para el código del programa con la nueva asignación de predicados.

La Tabla 21 muestra el correspondiente código VLIW para esa alternativa.

Tabla 21. Código VLIW para el diagrama de la Figura 32

slot 1		slot 2	
1w	r2, x	1w	r1, x
p1, p2	cmp.ge	r1, r2	nop
(p1)	p1, p2	cmp.ge	r2, r0
(p2)	p2	cmp.ltr	r2, r0
(p2)	p2	cmp.ltr	r1, r2
		nop	(p1) add r3, r1, r2
		nop	(p2) slli r3, r1, #1
		nop	sw x, r3

En cualquier caso, a pesar de que se necesitan tres instrucciones VLIW menos, el retardo de la carga de memoria hará que se haya que esperar tres ciclos más para enviar la primera instrucción de comparación, que necesita los datos x e y. El único beneficio es la reducción en el número de predicados (es necesario que los predicados se lean a principio del ciclo de ejecución y se escriban al final).

Problema 12. En un procesador de 32 bits con arquitectura LOAD/STORE (los accesos a memoria se hacen mediante instrucciones de carga y almacenamiento de registros y las operaciones se hacen con datos en registros), todas las instrucciones pueden predicarse. Para establecer los valores de los predicados se utilizan instrucciones de comparación (cmp) con el formato

(p) p1, p2 cmp.cnd x, y

donde cnd es la condición que se comprueba entre x e y (lt, le, gt, ge, eq, ne). Si la condición es verdadera p1=1 y p2=0, y si es falsa, p1=0 y p2=1. Una instrucción sólo se ejecuta si el predicable p=1 (ese valor se habrá establecido en otra instrucción de comparación previa). Para el siguiente código de alto nivel, escriba un código máquina para el procesador descrito que no tenga ninguna instrucción de salto:

```
for (i=0; i<2; i++)
    if (a[i]==0) then c[i]=a[i]+b[i];
    else c[i]=a[i]+1;
```

NOTA: los predicados no se encuentran inicializados a ningún valor.

Solución

El código que indica el problema contiene un bucle con dos iteraciones, por tanto podemos evitar la instrucción de control del bucle desenrollándolo. En este caso sólo que habría que repetir el cuerpo del bucle dos veces. Obviamente, esta solución solo se podrá adoptar si el número de instrucciones del cuerpo del bucle no es muy elevado (el tamaño del código crecería de forma considerablemente). Para evitar las instrucciones de salto dentro del bucle utilizaremos predicados.

```

1           lw      r1,0(a)    ; se carga a[1] en r1
2           p1,p2  cmp.eq   r1,0    ; ;a[1]=? p1 y p2 se cargan
3 (p1)      1w      r2,(b)    ; se carga b[1] en r2 si p1=1
4 (p1)      add     r3,r1,r2  ; r3=r1+r2 si p1=1
5 (p2)      add     r3,r1,#1  ; r3=r1+1 si p2=1
6           sw      0(c),r3   ; r3 se almacena en c[1]
7           1w      r1,4(a)   ; se carga a[2] en r1
8           p1,p2  cmp.eq   r1,0    ; ;a[1]=? p1 y p2 se cargan
9 (p1)      1w      r2,4(b)   ; se carga b[2] en r2 si p1=1
10 (p1)     add     r3,r1,r2  ; r3=r1+r2 si p1=1
11 (p2)     add     r3,r1,#1  ; r3=r1+1 si p2=1
12           sw      4(c),r3   ; r3 se almacena en c[2]

```

Las instrucciones 1-6 corresponden a la primera iteración y las 7-12 a la segunda. Dado que los datos son de 32 bits y se direccionan bytes, el desplazamiento que debe utilizarse con las direcciones a partir de las que empiezan los arrays (es decir a, b, y c) para acceder a los componentes utilizados en la segunda iteración es igual a 4, tal y como se observa en las instrucciones 7, 9, y 12.

Si el procesador no implementa renombrado de registros en hardware, se pueden reducir las dependencias entre las instrucciones sustituyendo los registros r1, r2 y r3 de las instrucciones 7 a 12, por r4, r5, y r6, respectivamente. Igualmente se puede hacer con los registros de predicado, y los p1 y p2 de las instrucciones 9, 10 y 11, se pueden sustituir por p3 y p4, respectivamente. Si se dispone de un procesador VLIW con dos *slots* y se puede emitir cualquier tipo de instrucción a cada *slot*, el código que se conseguiría es bastante compacto:

slot 1		slot 2	
lw	r1,0(a)	lw	r4,4(a)
p1,p2	cmp.eq	r1,0	p3,p4
(p1)	1w	r2,(b)	(p3)
(p2)	add	r3,r1,r2	(p3)
(p2)	add	r3,r1,#1	(p4)
	sw	0(c),r3	sw
			4(c),r6

Problema 13. Un procesador VLIW emite dos operaciones por ciclo (es decir, tiene dos campos o “slots” en cada instrucción VLIW) y todas las operaciones pueden predicarse. Los valores de los predicados se utilizan instrucciones de comparación (cmp) con el formato

$$(p) \quad p1[,p2] \quad \text{cmp.cnd} \quad x, y$$

donde cnd es la condición que se comprueba entre x e y (lt, le, gt, ge, eq, ne). Si la condición es verdadera $p1=1$ [y $p2=0$], y si es falsa, $p1=0$ [y $p2=1$]. Una instrucción sólo se ejecuta si el predicado p=1 (ese valor se habrá establecido en otra instrucción de comparación previa). ¿Cuál sería el código VLIW de la siguiente sentencia sin que haya operaciones de salto en las iteraciones del bucle, y considerando que las operaciones de comparación solo pueden aparecer en el primer *slot*, las de salto solo en el segundo, y el resto de operaciones en cualquiera de los dos *slots*?

```

for(i=0;i<n;i++)
    if (a[i]<0) then b[i]=a[i]+2;
    else b[i]=2*a[i];

```

NOTA: los predicados no se encuentran inicializados a ningún valor; tras las operaciones de carga de memoria se debe esperar un ciclo hasta que el dato esté disponible; y se implementa salto retardado de forma que la instrucción VLIW que se capte después de la instrucción de salto siempre se ejecuta.

Solución

Para empezar, se escribirá un código con instrucciones escalares que corresponderán a las operaciones que se codifican en los *slots* de las instrucciones VLIW. Un posible código con instrucciones típicas de un repertorio LOAD/STORE que se pueden predicar como se indica en el enunciado puede ser el siguiente:

```

1      inic:          addi   r1,r0,#n      ; r1 ← n (r0=0)
2                  add    r2,r0,r0      ; r2 ← 0 (para recorrer a y b)
3      bucle:         lw     r3,a(r2)      ; r3 apunta a a[i] (inic. a[0])
4      p1,p2         cmp.lt r3,r0      ; p1=1 (p2=0) si r3<0
5      (p1)          addi   r4,r3,#2      ; r4=a[i]+2
6      (p2)          add    r4,r3,r3      ; r4=2*a[i]
7                  sw    b(r2),r4      ; b[i] almacena r4
8                  addi   r2,r2,#4      ; apuntar al siguiente a[] y b[]
9                  subi   r1,r1,#1      ; una iteración menos
10     p3            cmp.gt r1,r0      ; p3=1 mientras r1>0
11     (p3)          j     bucle       ; salta si r1>0 (quedan iter.)

```

Las instrucciones con predicado que se han utilizado en el programa anterior permiten eliminar las instrucciones de salto condicional en el cuerpo del bucle, e incluso evitar el uso de una instrucción de salto condicional para controlar el final de las iteraciones: se utiliza una instrucción de salto incondicional con predicado. No obstante, habría que ver si la implementación hardware hace que esto sea más eficiente (dos instrucciones) que un salto condicional.

A continuación, la Tabla 22 muestra la ubicación de las operaciones en los dos *slots* de las instrucciones VLIW teniendo en cuenta las dependencias entre las instrucciones y que las instrucciones de comparación solo pueden ubicarse en el primer *slot* y las de salto en el segundo.

Tabla 22. Código VLIW para la secuencia de instrucciones escalares del problema

Etiqueta	slot1		slot2	
inic:	addi	r1,r0,#n	add	r2,r0,r0
bucle:	lw	r3,a(r2)	nop	
	p1,p2	cmp.lt r3,r0	nop	
	(p1)	addi r4,r3,#2	(p2)	add r4,r3,r3
		sw b(r2),r4	subi r1,r1,#1	
	p3	cmp.gt r1,r0	addi r2,r2,#4	
		nop	(p3)	j bucle
		nop		nop

Como se puede observar en el código, se utilizan siete instrucciones VLIW en las que no se “llenan” tres *slots* (más los dos *slots* tras el salto retardado). Se ha tenido en cuenta las dependencias RAW entre las instrucciones 2 y 3, 3 y 4, 4 y 5, 4 y 6, 5 y 7, 6 y 7, 9 y 10, y 10 y 11.

Entre las instrucciones 7 y 8 hay una dependencia WAR. Aunque se podrían emitir al mismo tiempo (se pueden ubicar en el mismo *slot*) si se tiene en cuenta que las lecturas del registro r2 se harían en una etapa anterior (decodificación y acceso a operandos) a la escritura en r2 que realiza la instrucción 8, aquí se ha preferido ponerlas en instrucciones VLIW distintas dado que se puede adelantar la instrucción 9 que es independiente de la 7 y la 8, y emitir la instrucción 8 junto con la instrucción 10, de comparación.

También se han puesto las dos operaciones de comparación en el primer *slot* y la de salto en el segundo. Además, teniendo en cuenta que el procesador implementa salto retardado ejecutando siempre la instrucción que sigue a la instrucción de salto se ha introducido una instrucción VLIW con dos *nop*: si hubieran operaciones y no tuvieran que ejecutarse porque se produce el salto, se podrían producir errores en los resultados.

Al ejecutar este código VLIW hay que tener en cuenta los ciclos de latencia entre las instrucciones. Por lo tanto, los ciclos en los que se producen las emisiones se muestran en la Tabla 23.

Así, después de emitir la instrucción en la que se accede a memoria en el ciclo 2 hay que esperar un ciclo para que el dato que se pasa a r3 esté disponible, y después de la instrucción que contiene el salto incondicional y se emite en el ciclo 8, dejamos, como se ha dicho antes, un no-operar (*nop*) ya que el salto se implementa como salto retardado de un hueco.

Tabla 23. Ciclos en las que se emiten las instrucciones VLIW de la Tabla 22

ciclo	Etiq.	<i>slot1</i>			<i>slot2</i>		
1	inic:		addi	r1, r0, #n	add	r2, r0, r0	
2	bucle:		lw	r3, a(r2)	nop		
3							
4		p1, p2	cmp.lt	r3, r0	nop		
5	(p1)		addi	r4, r3, #2	(p2)	add	r4, r3, r3
6			sw	b(r2), r4		subi	r1, r1, #1
7		p3	cmp.gt	r1, r0		addi	r2, r2, #4
8			nop		(p3)	j	bucle
9			nop			nop	

Se puede intentar reducir el tiempo que tarda en producirse la emisión de las instrucciones reorganizando el código, de forma que se ocupen los huecos que aparecen debido a los retardos (por supuesto, manteniendo las dependencias y las restricciones de ubicación de las operaciones en los distintos *slots*). Una alternativa se muestra en la Tabla 24.

Tabla 24. Ciclos de emisión de instrucciones VLIW con operaciones reordenadas

ciclo	Etiq.	slot1			slot2		
1	inic:		addi	r1, r0, #n		add	r2, r0, r0
2	bucle:		lw	r3, a(r2)		subi	r1, r1, #1
3		p3	cmp.gt	r1, r0		addi	r2, r2, #4
4		p1, p2	cmp.lt	r3, r0		nop	
5	(p1)		addi	r4, r3, #2	(p2)	add	r4, r3, r3
6			nop		(p3)	j	bucle
7			sw	(b) r2, r4		nop	

Como se puede ver en la Tabla 24 se ha conseguido introducir operaciones para hacer algo útil en el ciclo de espera entre la instrucción de acceso a memoria y la que necesita el dato. Además, después de la instrucción de salto, se ha introducido la operación de almacenamiento en memoria, que siempre se tiene que ejecutar (se produzca o no se produzca el salto).

De esta forma, se han ahorrado dos ciclos por iteración (en el caso de la Tabla 24 se tardarían 6 ciclos en lugar de los 8 ciclos que se tienen en la Tabla 24). Estos 2x n ciclos que se ahorran pueden suponer una reducción de tiempo considerable si el valor de n es elevado.

Problema 14. ¿Cómo transformaría la secuencia de instrucciones VLIW que se proporciona a continuación de forma que sólo se utilicen instrucciones de movimiento de datos condicionales y sin que haya ninguna instrucción de salto condicional.

Inst.	slot1		slot 2	
1	lw	r1, x(r2)	add	r10, r11, r12
2	nop		add	r13, r10, r14
3	beqz	r3, direc	nop	
4	lw	r4, 0(r3)	nop	
5	lw	r5, 0(r4)	nop	
direc:				

Solución

Para resolver el problema se van a considerar sólo las instrucciones que se incluyen en el campo de operación 1, ya que las del segundo slot son independientes de éstas y no afectarán a los cambios que vamos a realizar en el código. Por lo tanto, nos centraremos en eliminar los saltos condicionales de la siguiente secuencia de instrucciones:

```

1    lw      r1, x(r2)
2    nop
3    beqz  r3, direc
4    lw      r4, 0(r3)
5    lw      r5, 0(r4)
direc:

```

En esta secuencia, la instrucción de salto 3 se introduce para que no se ejecute la instrucción de carga 4 si $r3 = 0$. Por tanto, se puede suponer que la instrucción de salto tiene la función de evitar una violación del acceso a memoria. En la Figura 33 se ilustra el uso que se da a los registros $r3$ y $r4$ como punteros, en las instrucciones 4 y 5 de la secuencia anterior.

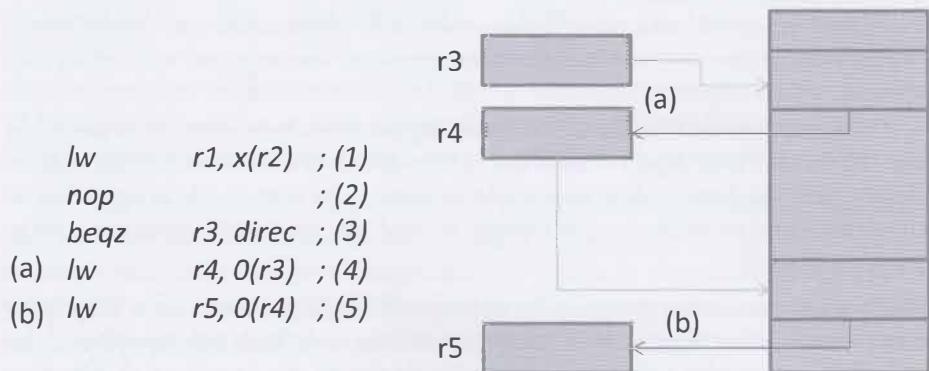


Figura 33. Uso de los registros $r3$, $r4$, y $r5$ en la secuencia inicial de instrucciones del *slot 1*

Así, si se adelantara la instrucción (4) para que estuviera delante de la instrucción de salto, la excepción que se produciría si $r3$ fuera igual a 0 haría que el programa terminase. Para que este cambio pueda realizarse es necesario que $r3$ sea distinto de cero siempre. En este caso, si existen registros disponibles, es posible utilizar instrucciones de movimiento condicional para evitar que se produzca la carga en caso de que se vaya a producir la excepción. El código sería:

```

addi    r6,r0,#1000 ; Se fija r6 a una dirección segura
lw      r1,x(r2)
mov    r7,r4      ; Se guarda r4 en r7
cmovnz r6,r3,r3  ; r3 a r6 si r3≠0
lw      r4,0(r6)  ; Carga especulativa
cmovz  r4,r7,r3  ; Si r3=0, r4 recupera su valor
beqz  r3,direc
lw      r5,0(r4)  ; Si r3≠0, hay que cargar r5

```

donde $r6$ y $r7$ son registros auxiliares. En $r6$ se carga primero una dirección segura (se sabe que no genera excepción porque contiene una dirección de memoria del espacio de usuario), y en $r7$ se introduce el valor previo de $r4$ para poder recuperarlo si la carga especulativa no debía realizarse. Como se puede comprobar, la especulación tiene un coste en instrucciones cuyo efecto final en el tiempo de ejecución depende de la probabilidad de que la especulación sea correcta o no. En la Figura 34 se proporciona una descripción gráfica de los efectos de las primeras transformaciones realizadas en la secuencia de operaciones del *slot 1*.

```

(0) addi    r6, r0, #1000 ; Fijamos r6 a una dirección segura
    lw      r1, x(r2)
(1) mov     r7, r4      ; Guardamos el contenido original de r4 en r7
(2) cmovnz r6, r3, r3  ; Movemos r3 a r6 si r3 es distinto de cero
(3) lw      r4, 0(r6)   ; Carga especulativa
(4) cmovz  r4, r7, r3  ; Si r3 es 0, hay que hacer que r4 recupere su valor
    beqz  r3, direc
    lw      r5, 0(r4)   ; Si r3 no es cero, hay que cargar r5

```

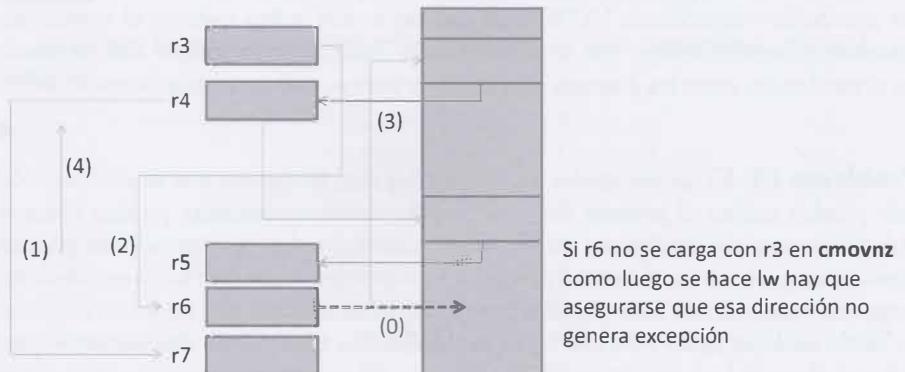


Figura 34. Primera transformación de las instrucciones del slot 1

También es posible evitar la instrucción de salto si se utilizan cargas especulativas (con la misma estrategia de la Figura 34) para las dos instrucciones de carga protegidas por el salto en el código inicial. En este caso el código sería el siguiente:

```

addi    r6, r0, #1000; Fijamos r6 a una dirección segura
lw      r1, x(r2)
mov     r7, r4      ; Se guarda el r4 inicial en r7
mov     r8, r5      ; Se guarda r5 en r8
cmovnz r6, r3, r3  ; r3 a r6 si r3 es distinto de cero
lw      r4, 0(r6)   ; Carga especulativa
lw      r5, 0(r4)   ; Carga también especulativa
cmovz  r4, r7, r3  ; Si r3=0, r4 recupera su valor
cmovz  r5, r8, r3  ; Si r3=0, r5 recupera su valor

```

Hay que tener en cuenta que la dirección `direc` a la que se produce el salto viene a continuación de este trozo de código. Si no fuese así, no se podría aprovechar tan eficientemente el procesamiento especulativo. Así, en general, cuando existen saltos a distintas direcciones y no existe un punto de confluencia de esos caminos no suele ser posible obtener mejores prestaciones mediante cambios especulativos que eliminan la instrucción de salto.

Tabla 25 Código VLIW final sin instrucciones de salto

Inst.		slot 1		slot 2
1	addi	r6, r0, #1000	add	r10, r11, r12
2	aw	r1, x(r2)	add	r13, r10, r14
3	cmovnz	r6, r3, r3	mov	r7, r4
4	lw	r4, 0(r3)	mov	r8, r5
5	lw	r5, 0(r4)	nop	
6	cmovz	r4, r7, r3	cmovz	r5, r8, r3

Finalmente, si suponemos que las operaciones pueden ubicarse en cualquiera de los *slots* de las instrucciones VLIW dado que no se nos indica nada en el enunciado, podemos generar el código que se muestra en la Tabla 25, en el que se han respetado las dependencias entre las distintas instrucciones y no se utilizan instrucciones de salto.

Problema 15. En un procesador VLIW con dos *slots*, las operaciones de comparación solo pueden utilizar el primero de ellos, pero las demás operaciones pueden ubicarse indistintamente en cualquiera de los *slots*. Además, todas las operaciones pueden predicarse, igual que en el caso del procesador del problema anterior. Las latencias de las cargas de memoria son muy elevadas (cinco ciclos) en relación con las de las unidades de suma/resta (un ciclo) e incluso las de multiplicación (dos ciclos). Precisamente, para paliar el efecto de las latencias elevadas en el acceso a memoria, se incluye una instrucción de carga de memoria especulativa `lw.srd,desplaz(rs)`, compensa que permite indicar la operación de carga del dato de forma anticipada, incluso adelantando a instrucciones de salto condicional. Además, todos los registros del procesador disponen de un bit de marca para gestionar las excepciones como se indica a continuación:

- (1) Si la carga adelantada da lugar a una excepción, se marca el registro de destino (como registro envenenado) pero no se atiende la excepción.
- (2) Si una operación utiliza un operando marcado provocará que también se marque el registro resultado de esa operación como registro envenenado.
- (3) La excepción se atenderá si se intenta almacenar un resultado envenenado. En dicho caso, tras atender la excepción se ejecutará el código de reparación almacenado a partir de la dirección compensa.

Optimice el código

add	r3, r2, r1
add	r4, r5, r3
beqz	r4, cero
lw	r7, dato
add	r8, r4, r7
mult	r2, r1, r8
cero:	sub r9, r1, r4
	mult r1, r9, r2
	sw resul, r1

de manera que no aparezca ninguna instrucción de salto, se tengan en cuenta las latencias de las operaciones, las ubicaciones permitidas de las operaciones en los slots, y se adelante la instrucción de carga especulativamente todo lo que sea posible para tratar de ocultar su latencia.

¿A partir de qué probabilidad de que se produzca una excepción es beneficioso utilizar esta estrategia especulativa?

Solución

En primer lugar, podemos evitar la instrucción de salto condicional utilizando un predicado:

	add	r3, r2, r1
	add	r4, r5, r3
p1	cmp.ne	r4, r0
(p1)	lw	r7, dato
(p1)	add	r8, r4, r7
(p1)	mult	r2, r1, r8
	sub	r9, r1, r4
	mult	r1, r9, r2
	sw	resul, r1

y ubicamos las operaciones en los dos slots de que disponen las instrucciones VLIW, tal y como se muestra en la Tabla 26. Para situar las operaciones se han tenido en cuenta sus dependencias y el hecho de que las operaciones de comparación tienen que ubicarse en el primer slot. Por otra parte, hay que introducir siete slots que quedan sin operación (los nop que hay que introducir).

Tabla 26. Código VLIW para la secuencia de instrucciones escalares del problema

Inst.	slot 1		slot 2
1	add	r3, r2, r1	nop
2	add	r4, r5, r3	nop
3	p1	cmp.ne	r4, r0
4	(p1)	lw	r7, dato
5	(p1)	add	r8, r4, r7
6	(p1)	mult	r2, r1, r8
7		nop	mult r1, r9, r2
8		sw	resul, r1
			nop

El número de ciclos que tardan en emitirse las instrucciones de la Tabla 26 se obtienen a partir de la Tabla 27. Hacen falta 14 ciclos.

Como se puede ver en la Tabla 27, la carga introduce varios ciclos de espera que ralentizan la emisión de las instrucciones. Por eso sería interesante adelantar esta instrucción lo más posible. Dado que existen dependencias RAW de la instrucción 1 con la 2, de la 2 con la 3, y de la 3 con la propia instrucción de carga (está vigilada por p1), sólo se podría adelantar la carga especulativamente para evitar el efecto de las

excepciones que puedan ocurrir en el acceso a memoria (por fallo de página, etc.), ya que si $r4$ toma el valor 0 al calcularse en la instrucción 2, la carga no debería haberse realizado y por tanto, no debería haber ocurrido ninguna excepción. La Tabla 28 muestra el nuevo código VLIW.

Tabla 27. Ciclos en los que se emiten las instrucciones del código VLIW de la Tabla 26

Ciclo	slot 1			slot 2
1		add	$r3, r2, r1$	nop
2		add	$r4, r5, r3$	nop
3	$p1$	cmp.ne	$r4, r0$	sub $r9, r1, r4$
4	($p1$)	lw	$r7, \text{dato}$	nop
5				
6				
7				
8				
9	($p1$)	add	$r8, r4, r7$	nop
10	($p1$)	mult	$r2, r1, r8$	nop
11				
12		nop		mult $r1, r9, r2$
13				
14		sw	$\text{resul}, r1$	nop

Tabla 28. Código VLIW con carga especulativa

Inst.	slot 1			slot 2
1		add	$r3, r2, r1$	lw.s $r7, \text{dato, compensa}$
2		add	$r4, r5, r3$	nop
3	$p1$	cmp.ne	$r4, r0$	sub $r9, r1, r4$
5	($p1$)	add	$r8, r4, r7$	nop
6	($p1$)	mult	$r2, r1, r8$	nop
7		nop		mult $r1, r9, r2$
8		sw	$\text{resul}, r1$	nop

El número de ciclos que se tardaría en emitir las instrucciones VLIW se muestra en la Tabla 29, donde se han tenido en cuenta los ciclos de espera de la carga y de la multiplicación entre instrucciones dependientes. Como se puede ver, se consigue la emisión de todas las instrucciones en once ciclos, tres ciclos menos que en el caso que no utilizaba la carga especulativa.

No obstante, hay que tener en cuenta que si se produce una excepción y la carga especulativa falla pero $r4 \neq 0$ y, por lo tanto $p1=1$, el resultado de $r7$ no sería correcto. Con el procedimiento descrito en el problema, la excepción que se produciría al ejecutarse $lw.s$ no se atendería y se marca el registro $r7$. Esto hace que también se marquen los registros $r8, r2$, y $r1$, y cuando se ejecute el acceso a memoria en la instrucción 8 de la Tabla 28, al intentar almacenar en memoria el registro marcado $r1$, se atenderá la excepción y se ejecutará el código de compensación que permite obtener el resultado correcto en los registros. Es decir:

compensa:	lw	r7, dato
	add	r8, r4, r7
	mult	r2, r1, r8
	mult	r1, r9, r2
	sw	resul, r1

Tabla 29. Ciclos en emitirse el código VLIW de la Tabla 28

Ciclo	slot 1		slot 2
1	add	r3, r2, r1	lw, s r7, dato, compensa
2	add	r4, r5, r3	nop
3	p1	cmp, ne	sub r9, r1, r4
4			
5			
6	(p1)	add	r8, r4, r7
7	(p1)	mult	r2, r1, r8
8		nop	
9			mult r1, r9, r2
10			
11	sw	resul, r1	nop

Los ciclos necesarios para emitir estas instrucciones (teniendo en cuenta la dependencia entre operaciones) se proporcionan en la Tabla 30.

Por tanto, el tiempo necesario para completar la emisión del código cuando hay excepción en el acceso a memoria especulativo sería igual a la suma del tiempo para el código optimizado (11 ciclos) más la penalización correspondiente al tiempo del código de compensación (11 ciclos). Es decir 22 ciclos.

Tabla 30. Ciclos en emitirse el código VLIW de compensación

Ciclo	slot 1		slot 2
1	lw	r7, dato	nop
2			
3			
4			
5			
6	add	r8, r4, r7	nop
7	mult	r2, r1, r8	nop
8			
9	mult	r1, r9, r2	nop
10			
11	sw	resul, r1	nop

Por tanto, el tiempo necesario para completar la emisión del código cuando hay excepción en el acceso a memoria especulativo sería igual a la suma del tiempo para el código optimizado (11 ciclos) más la penalización correspondiente al tiempo del código de compensación (11 ciclos). Es decir 22 ciclos.

Para que sea mejor utilizar la alternativa especulativa que la inicial (14 ciclos) debería ocurrir que:

$$14 \geq p \times 22 + (1 - p) \times 11$$

donde p es la probabilidad de que se produzca una excepción. Así, despejando

$$p \leq \frac{3}{11} = 0.2727$$

Por tanto, para que resulte beneficiosa la alternativa, debe ocurrir que en más del 72.7% de los casos ($1-p=0.727$) no se produzcan excepciones.

5. Bibliografía

- Julio Ortega Lopera, Mancia Anguita López, Alberto Prieto Espinosa, 2005, "Arquitectura de Computadores". Thomson.
- Julio Ortega Lopera, Jesús López Peñalver, 2008, "Problemas de Ingeniería de Computadores. Cien problemas resueltos de Procesadores Paralelos". Editorial Copicentro.