2º curso / 2º cuatr.

Grado en Ing. Informática

Arquitectura de Computadores: Exámenes y Controles

Examen Final AC 04/09/2012 resuelto

Material elaborado por los profesores responsables de la asignatura:
Mancia Anguita, Julio Ortega

1 Enunciado Examen del 04/09/2012

Cuestión 1.(0.5 puntos) Defina e indique la utilidad de las siguientes características o componentes que puede incluir un procesador e indique cuáles se implementan en superescalares, cuáles en VLIW y cuáles en ambos.

- buffer de renombramiento
- buffer de reordenamiento
- ventana de instrucciones
- estación de reserva
- predicación de instrucciones
- predicción de saltos

Cuestión 2.(0.5 puntos). Enuncie la ley de Amdahl en el contexto de procesamiento paralelo y deduzca la expresión que la caracteriza en este contexto. Defina claramente el punto de partida que utilice en la deducción y todas las etiquetas o variables que utilice.

Ejercicio 1. (2 puntos) Se dispone de un multiprocesador CC-NUMA con red estática y 4 nodos (numerados 0, 1, 2 y 3) con un modelo de consistencia que garantiza todos los órdenes menos W→R. Se ha implementado para este multiprocesador el siguiente código para obtener el número de componentes positivos de un vector de enteros:

```
numpos_local = 0;
for (i=ithread  ; i< Nmax  ; i=i+nthread) {
        if (a[i]>0) numpos_local=numpos_local+1;
}
while (test_&_set(k)) {};
numpos = numpos + numpos_local;
if (ithread==0) printf("Valores positivos en a[]:%d\n",numpos);
```

Donde `k`, `numpos`, `Nmax` y `a[0:Nmax-1]` son variables compartidas (`k` y `numpos` están inicializadas a 0), el resto son locales: `nthread` es igual a 4 e `ithread` es el identificador del thread que ejecuta el código (`ithread`=0,1,2,3).

Conteste a las siguientes cuestiones:

(a) ¿Imprime el código el resultado esperado? Razone su respuesta. Añada lo necesario para que imprima el resultado esperado.

(b) Suponga que el CC-NUMA no tiene `test_&_set()`, pero tiene `fetch_&_or()`. Haga en el código las modificaciones necesarias para que imprima el resultado esperado usando esta segunda instrucción.

(c) Suponga ahora que el CC-NUMA sólo dispone de `compare_&_swap()`. Haga en el código las modificaciones necesarias para que imprima el resultado esperado (se valorarán las prestaciones del código resultante).

(d) Suponga ahora que el CC-NUMA dispone de `fetch_&_add()`. Haga en el código las modificaciones necesarias para que imprima el resultado esperado usando esta instrucción (se valorarán las prestaciones del código resultante).

(e) Razone si el código que ha escrito en el apartado (a) obtiene el resultado esperado en caso de que el procesador no garantice tampoco W->W. ¿Qué debe proporcionar el modelo de consistencia y dónde se usaría para que se pueda obtenerse el resultado esperado?

Ejercicio 2. (2 puntos) Suponga que en un CC-NUMA de red estática de 4 nodos (N0-N3) se implementa un protocolo MSI basado en directorios sin difusión con dos estados en el directorio (válido e inválido). Cada nodo tiene 16 GBytes de memoria principal y una línea de cache supone 64 Bytes. Considere que el directorio utiliza vector de bits completo. (a) Calcule el tamaño del directorio de uno nodo en bytes. (b) Indique cual sería el contenido del directorio, las transiciones de estados (en cache y en el directorio) y la secuencia de paquetes generados por el protocolo de coherencia en los siguientes accesos sobre una dirección D que se encuentra en la memoria del nodo 3 (inicialmente D no está en ninguna cache, el orden de los accesos es el indicado por los números que les preceden):

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

Ejercicio 3. (1 puntos) 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 50% de las instrucciones son operaciones con la ALU (1 CPI), el 20% LOADs (4 CPI), el 10% STOREs (1 CPI) y el 20% BRANCHs (4 CPI).

Además, un 25% de las operaciones con la ALU utilizan operandos en registros, que no se vuelven a utilizar. ¿Se mejora o se empeoran las prestaciones si, para sustituir ese 25% 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 4? ¿En qué tanto por ciento empeora o mejora el tiempo de ejecución del conjunto de programas.

2 Solución Examen del 04/09/2012

Cuestión 1.(0.5 puntos) Defina e indique la utilidad de las siguientes características o componentes que puede incluir un procesador e indique cuáles se implementan en superescalares, cuáles en VLIW y cuáles en ambos.

- buffer de renombramiento
- buffer de reordenamiento
- ventana de instrucciones
- estación de reserva
- predicación de instrucciones
- predicción de saltos

Solución

- Definición: Un buffer de renombramiento es un recurso presente en los procesadores superescalares que permite asignar almacenamiento temporal a los datos asignados a registros del banco de registros de la arquitectura. La asignación a registros de la arquitectura la realiza el compilador o el programador en ensamblador. Utilidad: Mediante la asignación de registros temporales a los registros de la arquitectura se implementa el renombramiento de estos últimos con el fin de eliminar riesgos (dependencias) WAW y WAR. Dónde se implementa: superescalares.

- Definición: Un buffer de reordenamiento es un recurso presente en los procesadores superescalares que permite implementar la finalización ordenada de las instrucciones después de su ejecución. Utilidad: Esto permite implementar consistencia del procesador fuerte en la ejecución de instrucciones con registros y consistencia de memoria fuerte en la ejecución de instrucciones con acceso a memoria. Para implementar consistencia del procesador fuerte las instrucciones con registros terminan, es decir, escriben los resultados en los registros de la arquitectura, en el orden en que se encuentran en el código en memoria (código generado por el compilador o escrito por el programador). Para implementar consistencia de memoria fuerte los accesos a memoria deben terminar exactamente en el orden en que se encuentran en el código que hay en memoria. Dónde se implementa: superescalares.

- Definición: Una ventana de instrucciones es un recurso con almacenamiento a la que pasan las instrucciones tras ser decodificadas para ser emitidas desde ahí a la unidad de datos donde se ejecutarán cuando dicha unidad esté libre y los operandos que se necesitan para realizar la correspondiente operación estén disponibles. La emisión desde esa ventana de instrucciones puede ser ordenada o desordenada. Utilidad: Se utiliza para detectar las instrucciones que pueden pasar a ejecutarse por tener los operandos disponibles evitando, de esta forma, problemas por riesgos RAW. Dónde se implementa: superescalares.

- Definición: Las estaciones de reserva son recursos con almacenamiento presentes en los procesadores superescalares entre los que se distribuye la ventana de instrucciones. Una estación de reserva puede estar asociada a una o a varias unidades funcionales. Las instrucciones decodificadas se envían a una u otra estación de reserva según la unidad funcional (o el tipo de unidad funcional) donde se va a ejecutar la instrucción. Utilidad: Se utiliza para detectar las instrucciones que pueden pasar a ejecutarse por tener los operandos disponibles evitando, de esta forma, problemas por riesgos RAW. Dónde se implementa: superescalares.

- Definición: La predicación de instrucciones es un recurso que se incluye para eliminar ciertas instrucciones de salto condicional de los códigos. Utilidad: contribuye a reducir el número de riesgos de control (saltos) en la ejecución de programas en procesadores segmentados, estos riesgos provocan penalizaciones en ciclos de reloj porque provoca que haya ciclos de reloj en los que no se

ejecutan instrucciones debido a que no se sabe si la instrucción que se debe ejecutar a continuación es la que sigue a la instrucción de salto (en los casos en los que no se salta) en el código o la que se encuentra en la dirección de salto (en los casos en los que se salta). Dónde se implementa: en superescalares y en VLIW

- Definición: La predicción de saltos es un recurso que predice si el salto de una instrucción de salto condicional se va a tomar o no antes de que el procesador tenga evaluada la condición de salto y, por tanto, antes de saber si se va a saltar o no. Cuando se ejecuta una instrucción de salto, en lugar de esperar a que se resuelva la condición asociada al salto, ejecuta las instrucciones de la dirección del salto o la que hay detrás del salto según el resultado de la predicción. Utilidad: se utiliza para disminuir la penalización (en ciclos de reloj) por saltos condicionales. Dónde se implementa: Se utiliza en superescalares y en la arquitectura EPIC de los Itanium de Intel que comparte características con los procesadores VLIW (algunos autores no consideran que los procesadores con predicción de saltos sean VLIW).

◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀

Cuestión 2.(0.5 puntos). Enuncie la ley de Amdahl en el contexto de procesamiento paralelo y deduzca la expresión que la caracteriza en este contexto. Defina claramente el punto de partida que utilice en la deducción y todas las etiquetas o variables que utilice.

Solución

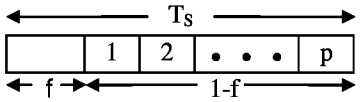
Enunciado

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.

Punto de partida deducción:

Se parte de un modelo de código secuencial (como se representa en la figura de abajo):

- Con un tiempo de ejecución secuencial, T_s, que permanece constante aunque varíe el número de procesadores disponibles `p`
- Con una parte no paralelizable (la fracción notada por `f` en la figura) que permanece constante aunque varíe el número de procesadores `p` disponibles, y una parte (1-f) que se puede paralelizar con un grado de paralelismo ilimitado y, además, repartiéndola por igual entre los `p` procesadores disponibles.



Las etiquetas de la figura se describen a continuación:

- `Ts` 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):

$$S(p) = \frac{T_S}{T_P(\mathrm{p})} = \frac{T_S}{f \times T_S + \frac{(1-f)\times T_S}{p}} = \frac{1}{f + \frac{(1-f)}{p}} = \frac{p}{fp + (1-f)} = \frac{p}{1 + f(p-1)}$$

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 paralelización puede añadir un tiempo de sobrecarga (*overhead*) debido a la necesidad de comunicación/sincronización y a las operaciones extras que puede requerir la paralelización. Teniendo en cuenta esto se utiliza menor o igual en lugar de igual en la expresión que se utiliza para representar la Ley de Amdahl:

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

◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀

Ejercicio 1. (2 puntos) Se dispone de un multiprocesador CC-NUMA con red estática y 4 nodos (numerados 0, 1, 2 y 3) con un modelo de consistencia que garantiza todos los órdenes menos W→R. Se ha implementado para este multiprocesador el siguiente código para obtener el número de componentes positivos de un vector de enteros:

```
(1) numpos_local = 0;
(2) for (i=ithread  ; i< Nmax  ; i=i+nthread) {
(3)    if (a[i]>0) numpos_local=numpos_local+1;
    }
(4) while (test_&_set(k)) {};
(5) numpos = numpos + numpos_local;
(6) if (ithread==0) printf("Valores positivos en a[]:%d\n",numpos);
```

Donde `k`, `numpos`, `Nmax` y `a[0:Nmax-1]` son variables compartidas (`k` y `numpos` están inicializadas a 0), el resto son locales: `nthread` es igual a 4 e `ithread` es el identificador del thread que ejecuta el código (`ithread`=0,1,2,3).

Conteste a las siguientes cuestiones:

(a) ¿Imprime el código el resultado esperado? Razone su respuesta. Añada lo necesario para que imprima el resultado esperado.

(b) Suponga que el CC-NUMA no tiene `test_&_set()`, pero tiene `fetch_&_or()`. Haga en el código las modificaciones necesarias para que imprima el resultado esperado usando esta segunda instrucción.

(c) Suponga ahora que el CC-NUMA sólo dispone de `compare_&_swap()`. Haga en el código las modificaciones necesarias para que imprima el resultado esperado (se valorarán las prestaciones del código resultante).

(d) Suponga ahora que el CC-NUMA dispone de `fetch_&_add()`. Haga en el código las modificaciones necesarias para que imprima el resultado esperado usando esta instrucción (se valorarán las prestaciones del código resultante).

(e) Razone si el código que ha escrito en el apartado (a) obtiene el resultado esperado en caso de que el procesador no garantice tampoco W->W. ¿Qué debe proporcionar el modelo de consistencia y dónde se usaría para que se pueda obtener el resultado esperado?

Solución

(a) Inicialmente la variable compartida `k` está a 0, la primitiva `test_&_set(k)` lo pone a 1. No hay ningún punto en el código donde se vuelva a poner `k` a 0. El flujo de control (*thread*) que realiza el primer acceso a `k`, es decir, el primero que ejecute `test_&_set(k)` obtiene un 0 (`test_&_set(k)` devueve 0) y pone `k` a 1. Como obtiene un 0 no repite el bucle del `while`, continua la ejecución (el `while` se repite si `test_&_set(k)` devuelve un 1). El resto de flujos de control obtendrán al ejecutar `test_&_set(k)` un 1, por tanto, se quedarán en el while ejecutando `test_&_set(k)`, además nunca saldrán del `while` porque no se vuelve a poner `k` a 0 y, por tanto, a ninguno obtendrá al ejecutar `test_&_set(k)` el 0 que les permita dejar el bucle. Si el flujo de control que obtiene un 0 y continúa, por tanto, la ejecución es el 0, se imprime en pantalla los valores positivos que ha contado este flujo, en caso contrario, el código no imprime nada. En cualquier caso se quedan todos los flujos menos uno bloqueados en el `while`.
Para resolver los problemas del código, hay que poner `k` a 0 tras actualizar "`numpos = numpos + numpos_local`" (en calabaza en el código modificado) y añadir una barrera (en azul en el código modificado) para que el flujo de control 0 imprima cuando todos los flujos hayan actualizado la variable compartida `num_pos`. El código modificado es el siguiente:

```
(1) numpos_local = 0;
(2) for (i=ithread  ; i< Nmax  ; i=i+nthread) {
(3)    if (a[i]>0) numpos_local=numpos_local+1;
    }
(4) while (test_&_set(k)) {};
(5) numpos = numpos + numpos_local;
    k=0;
    bandera_local= !(bandera_local) //se complementa la bandera local
    while (test_&_set(k2)) {};     //lock(k2), k2 inicializada a 0
    bar[id].cont+=1; cont_local = bar[id].cont;    //cont_local es local
    k2=0;                          //unlock(k2)

    if (cont_local ==num_procesos)  {
       bar[id].cont=0;    //se hace 0 el contador asociado a la barrera
       bar[id].bandera= bandera_local; // libera procesos en espera
    }
    else  while (bar[id].bandera!= bandera_local) {};
(6) if (ithread==0) printf("Valores positivos en a[]:%d\n",numpos);
```

(b) Hay que realizar pocas modificaciones. Donde aparece `test_&_set(variable_cerrojo)` hay que poner `fetch_&_or(variable_cerrojo,1)`:

```
(1) numpos_local = 0;
(2) for (i=ithread  ; i< Nmax  ; i=i+nthread) {
(3)    if (a[i]>0) numpos_local=numpos_local+1;
    }
(4) while (fetch_&_or(k,1)) {};
(5) numpos = numpos + numpos_local;
    k=0;
    bandera_local= !(bandera_local) //se complementa la bandera local
    while (fetch_&_or(k2,1)) {};     //lock(k2), k2 inicializada a 0
    bar[id].cont+=1; cont_local = bar[id].cont;    //cont_local es local
    k2=0;                          //unlock(k2)

    if (cont_local ==num_procesos)  {
       bar[id].cont=0;    //se hace 0 el contador asociado a la barrera
       bar[id].bandera= bandera_local; // libera procesos en espera
    }
```

```
    else  while (bar[id].bandera!= bandera_local) {};
(6) if (ithread==0) printf("Valores positivos en a[]:%d\n",numpos);
```

(c) En este caso en lugar de usar cerrojos se accederá directamente a las variables compartidas `numpos` y `bar[id].cont` con `compare_&_swap()`, evitando de esta forma tener que añadir (1) dos variables compartidas (variables cerrojos) más a las que acceder y (2) el código para implementar cerrojos. El código resultante es el siguiente (en calabaza se pueden ver las modificaciones con respecto al código del apartado (a)):

```
(1) numpos_local = 0;
(2) for (i=ithread  ; i< Nmax  ; i=i+nthread) {
(3)    if (a[i]>0) numpos_local=numpos_local+1;
    }
    do
    a = numpos;
    b = a + numpos_local;   // a y b son variables locales
    compare&swap(a,b,numpos);
    while (a!=b);

    bandera_local= !(bandera_local) //se complementa la bandera local
    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_procesos-1)  {
       bar[id].cont=0;    //se hace 0 el contador asociado a la barrera
       bar[id].bandera= bandera_local; // libera procesos en espera
    }
    else  while (bar[id].bandera!= bandera_local) {};
(6) if (ithread==0) printf("Valores positivos en a[]:%d\n",numpos);
```

(d) En este caso también se accederá directamente a las variables compartidas `numpos` y `bar[id].cont`, pero esta vez con `Fetch_&_add()`, evitando igualmente tener que añadir (1) dos variables compartidas (variables cerrojos) más a las que acceder y (2) el código para implementar cerrojos. El código resultante es el siguiente (en calabaza se pueden ver las modificaciones con respecto al código del apartado (a)):

```
(1) numpos_local = 0;
(2) for (i=ithread  ; i< Nmax  ; i=i+nthread) {
(3)    if (a[i]>0) numpos_local=numpos_local+1;
    }

(5) fetch_&_add(numpos, numpos_local);
    bandera_local= !(bandera_local) //se complementa la bandera local
    while (fetch_&_or(k2,1)) {};     //lock(k2), k2 inicializada a 0
    cont_local = fetch_&_add(bar[id].cont,1);    //cont_local es local
    if (cont_local == num_procesos-1)  {
       bar[id].cont=0;    //se hace 0 el contador asociado a la barrera
       bar[id].bandera= bandera_local; // libera procesos en espera
    }
    else  while (bar[id].bandera!= bandera_local) {};
(6) if (ithread==0) printf("Valores positivos en a[]:%d\n",numpos);
```

(e) Si no se garantiza el orden W->W el código no daría siempre el resultado esperado. Se ha añadido al código los accesos a memoria que se realizan de lectura, R, escritura, W, y lectura+escritura atómica, (R,W) en las operaciones de adquisición, liberación y en la sección crítica:

```
          (1) numpos_local = 0;
          (2) for (i=ithread  ; i< Nmax  ; i=i+nthread) {
          (3)    if (a[i]>0) numpos_local=numpos_local+1;
              }
  (R,W)   (4) while (test_&_set(k)) {};
   R, W   (5) numpos = numpos + numpos_local;
      W       k=0;
              bandera_local= !(bandera_local) //se complementa la bandera local
  (R,W)       while (test_&_set(k2)) {};     //lock(k2), k2 inicializada a 0
   R, W       bar[id].cont+=1; cont_local = bar[id].cont;    //cont_local es local
      W       k2=0;                          //unlock(k2)

              if (cont_local ==num_procesos)  {
                 bar[id].cont=0;    //se hace 0 el contador asociado a la barrera
                 bar[id].bandera= bandera_local; // libera procesos en espera
              }
              else  while (bar[id].bandera!= bandera_local) {};
          (6) if (ithread==0) printf("Valores positivos en a[]:%d\n",numpos);
```

La liberación del cerrojo permite que otro flujo de control acceda a la sección crítica. Las escrituras pueden adelantar a otras escrituras independientes que las precedan en el código porque no se garantizarse orden entre escrituras. Esto permite que las dos liberaciones de cerrojos que hay en el código (escritura en k de 0 y escritura en k2 de 0) puedan adelantar a la escritura en la variable compartida de la sección crítica que las precede. Como consecuencia, más de un flujo de control estarían en la sección crítica a la vez y podrían leer el mismo valor de la variable compartida (`numpos` en un caso y `bar[id].cont` en el otro).

Con respecto a las operaciones de adquisición, la escritura de la variable compartida no puede adelantar a la escritura del cerrojo en la adquisición porque la operación atómica que incluye la adquisición incluye tanto escritura como lectura y no se permite que las escrituras no pueden adelantar lecturas anteriores.

◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀

Ejercicio 2. (2 puntos) Suponga que en un CC-NUMA de red estática de 4 nodos (N0-N3) se implementa un protocolo MSI basado en directorios sin difusión con dos estados en el directorio (válido e inválido). Cada nodo tiene 16 GBytes de memoria principal y una línea de cache supone 64 Bytes. Considere que el directorio utiliza vector de bits completo. **(a)** Calcule el tamaño del directorio de un nodo en bytes. **(b)** Indique cual sería el contenido del directorio, las transiciones de estados (en cache y en el directorio) y la secuencia de paquetes generados por el protocolo de coherencia en los siguientes accesos sobre una dirección D que se encuentra en la memoria del nodo 3 (inicialmente D no está en ninguna cache, el orden de los accesos es el indicado por los números que les preceden):

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

Solución

Datos del ejercicio

- Tamaño Memoria principal por Nodo: 16 GB = TMN
- Tamaño Línea de Cache (bloque de memoria): 64 B = TLC
- 1 bits de estado por bloque en el directorio ya que hay que codificar dos estados (válido, inválido).
- Se accede a una dirección de la memoria del nodo 3 cuyo bloque no se encuentra en ninguna cache. Como no está en ninguna cache debe estar actualizado en memoria principal; es decir, el estado en el directorio para el bloque es válido.

(a)

$$Tamaño\_por\_nodo = \frac{TMN}{TLC} \times (1b + 4b) = \frac{2^{34} B}{2^6 B} \times 5b = 2^{28} \times 5b = 2^{20} \times \frac{2^8 \times 5b}{2^3 b / B} = (2^{20} \times 2^5 \times 5)B = 160MB$$

(b)

Intervienen los nodos N0, N1, N2 y N3. V es Válido e I inválido. BD denota el bloque de la dirección D.

ESTADO INICIAL	EVENTO	ACCIÓN	ESTADO SIGUIENTE
N0) Inválido **N1) Inválido** **N2) Inválido** **N3) Inválido** **D) Local** V | | | | |	P1 lee D	1. N1 envía petición de lectura del bloque BD (PtLec(BD)) a N3 2. N3 recibe PtLec(BD). Como tiene el bloque BD en estado Válido, (1) envía paquete de respuesta con el bloque a N1 (RpBloque(BD)) y (2) pone el bit de N1 en el directorio a 1. 3. N1 recibe RpBloque(BD) y pone el bloque en cache en estado Compartido	**N0) Inválido** **N1) Compartido** **N2) Inválido** **N3) Inválido** **D) Válido** V | | 1 | |	
N0) Inválido **N1) Compartido** **N2) Inválido** **N3) Inválido** **D) Válido** V | | 1 | |	P1 escribe en D	1. N1 envía petición de acceso exclusivo para BD (PtEx(BD)) a N3. 2. N3, cuando recibe PtEx(BD), (1) pasa BD a estado Inválido y (2) envía paquete de respuesta a N1 confirmando invalidación (RpInv(BD)). 3. N1, recibida la respuesta, modifica el bloque y lo pasa a estado Modificado.	**N0) Inválido** **N1) Modificado** **N2) Inválido** **N3) Inválido** **D) Inválido** I | | 1 | |
N0) Inválido **N1) Modificado** **N2) Inválido** **N3) Inválido** **D) Inválido** I | | 1 | |	P2 lee D	1. N2 envía PtLec(BD) a N3 porque no tiene BD. 2. N3, como tiene BD en estado Inválido: (1) reenvía (RvPetLec(BD)) la petición al nodo N1 (que según el directorio tiene copia válida del bloque), y (2) pone en la entrada del bloque en el directorio estado pendiente de Válido y activa el bit de N2. 3. N1 recibe RvPetLec(BD) y: (1) envía a N3 un paquete de respuesta con el bloque (RpBloque(BD)), y (2) pasa BD en su cache a Compartido. 4. N3 recibe la respuesta de N1 (RpBloque(BD)) y: (1) responde con el bloque a N2 (RpBloque(BD)) y (2) activa el bit de N2 y pone el estado del bloque en el directorio a Válido 5. N2, cuando recibe RpBloque(BD), introduce el bloque en su cache en estado Compartido	**N0) Inválido** **N1) Compartido** **N2) Compartido** **N3) Inválido** **D) Válido** V | | 1 | 1 |
N0) Inválido **N1) Compartido** **N2) Compartido** **N3) Inválido** **D) Válido** V | | 1 | 1 |	P3 lee D	N3 lee BD de su propia memoria, lo mete en su cache en estado Compartido y activa el bit de copia de N3 en el directorio	**N0) Inválido** **N1) Compartido** **N2) Compartido** **N3) Compartido** **D) Válido** V | | 1 | 1 | 1
N0) Inválido **N1) Compartido** **N2) Compartido** **N3) Compartido** **D) Válido** V | | 1 | 1 | 1	P0 escribe en D	1. N0 envía a N3 petición de lectura de BD con acceso exclusivo PtLecEx(BD) 2. N3 recibe PtLecEx(BD) y, como tiene el bloque en estado Válido: (1) envía los paquetes RvInv(BD) a los nodos que, según el directorio, tienen copia del bloque, (2) pone en el directorio el estado de BD en pendiente de Inválido.	**N0) Modificado** **N1) Inválido** **N2) Inválido** **N3) Inválido** **D) Inválido** I | 1 | | | |

		3.N1, N2 y N3, cuando reciben RvInv(BD): (1) invalidan su copia de BD y (2) responden a N3 confirmando la invalidación RpInv(BD). 4.N3, cuando recibe todas las RpInv(BD) (espera a todas las invalidaciones para garantizar un orden en los accesos a BD y así garantizar coherencia): (1) envía respuesta con el bloque a N0 confirmando invalidación RpBloqueInv y (2) activa el bit de copia de N0 en el directorio y pone el estado de BD en el directorio a Inválido 5.N0, cuando recibe RpBloqueInv(BD) de N3, introduce el bloque en su cache, escribe en la copia de BD de su cache y lo pone en estado Modificado en el directorio cache

◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀

Ejercicio 3. (1 punto) 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 50% de las instrucciones son operaciones con la ALU (1 CPI), el 20% LOADs (4 CPI), el 10% STOREs (1 CPI) y el 20% BRANCHs (4 CPI).

Además, un 25% de las operaciones con la ALU utilizan operandos en registros, que no se vuelven a utilizar. ¿Se mejoran o se empeoran las prestaciones si, para sustituir ese 25% 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 4? ¿En qué tanto por ciento mejora o empeora el tiempo de ejecución del conjunto de programas.

Solución

Datos del ejercicio:

Alternativa 1 (i: tipo de instrucción en la alternativa 1, puede ser load, store, alu y br (instrucción de salto); CPI_i^1: Ciclos por Instrucción para instrucciones del tipo i en la alternativa 1; NI_i^1: Número de Instrucciones del tipo i en la alternativa 1; NI^1: Número de Instrucciones en total en la alternativa 1):

i=	CPI_i^1	Fracción NI_i^1/NI^1	NI_i^1	Comentarios
LOAD	4	0,2	$0,2*NI^1$	
STORE	1	0,1	$0,1*NI^1$	
ALU	1	0,5	$0,5*NI^1$	25 % inst. que usan opernados en registros que no se vuelven a utilizar
BR	4	0,2	$0,2*NI^1$	
TOTAL		**1**	NI^1	

Alternativa 2 (i: tipo de instrucción en la alternativa 2, puede ser load, store, alu r-r (operación alu con dos operandos en registro), alu r-m (operación alu con un operando en registro y otro en memoria) y br (instrucción de salto); CPI_i^2: Ciclos por Instrucción para instrucciones del tipo i en la alternativa 2; NI_i^2: Número de Instrucciones del tipo i en la alternativa 2; NI^1: Número de Instrucciones en total en la alternativa 1; NI^2: Número de Instrucciones en total en la alternativa 2):

i=	CPI_i^2	NI_i^2/NI^1	NI_i^2	Comentarios
LOAD	4	0,2 - **0,25 * 0,5** = 0,2-**0,125**= 0,075	$0,2*NI^1$ - **$0,25*NI^1*0,5$** = $0,075*NI^1$	El 25 % de las instrucciones con la ALU de la alt. 1 operan con un dato en memoria que no hay que cargar con load en un registro en la alt. 2; por tanto, disminuyen las instrucciones load en la alt. 2
STORE	1	0,1	$0,1*NI^1$	

ALU r-r	1	0,75 * 0,5 = 0,375	$0,75*NI^1 * 0,5 = 0,375*NI^1$	75% de las instrucciones con la ALU de la alt. 1 siguen estando en la alt. 2
ALU r-m	4	**0,25 * 0,5 = 0,125**	**$0,25*NI^1 * 0,5 = 0,125*NI^1$**	25% de las instrucciones con la ALU de la alt. 1, es decir, el 25% del 50% del total de instrucciones de la alt. 1 son, en la alt. 2, instrucciones con un operando en memoria
BR	4	0,2	$0,2*NI^1$	
TOTAL		**0,875**	**$NI^2 = 0,875*NI^1$**	

En primer lugar se calcula el tiempo de CPU para la situación inicial (utilizando el número de instrucciones de cada tipo en la alternativa 1, NI_i^1, que aparece en la cuarta columna de la primera tabla):

$$T_{CPU}(1) = (0,2*NI^1*4+0,1*NI^1*1+0,5*NI^1*1+0,2*NI^1*4) * T_{ciclo} =$$

$$(0,8+0,1+0,5+0,8) * NI^1 * T_{ciclo} = 2,2 * NI^1 * T_{ciclo}$$

En segundo lugar se calcula el tiempo de CPU para la alternativa 2 (utilizando el número de instrucciones de cada tipo en la alternativa 2, NI_i^2, que aparece en la cuarta columna de la segunda tabla):

$$T_{CPU}(2) = (0,075*NI^1*4+0,1*NI^1*1+0,375*NI^1*1+0,125*NI^1*4+0,2*NI^1*4) * T_{ciclo} =$$
$$(0,3+0,1+0,375+0,5+0,8) * NI^1 * T_{ciclo} = 2,075 * NI^1 * T_{ciclo}$$

Los dos tiempos obtenidos se pueden comparar porque están en función de las mismas variables. Las prestaciones mejoran, puesto que el tiempo se reduce con la alternativa 2. El tiempo se ha reducido un

$$[2,2*NI^1*T_{ciclo} - 2,075*NI^1*T_{ciclo}] * 100/ 2,2*NI^1*T_{ciclo} = 0,125 * 100/2,2 \sim 5,68\%$$