2º curso / 2º cuatr.

Grado en Ing. Informática

Arquitectura de Computadores: Ejercicios y Cuestiones

Tema 1. Arquitecturas Paralelas: Clasificación y Prestaciones

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

1 Ejercicios

Ejercicio 1. En un procesador no segmentado que funciona a 300 MHz, hay un 20% de instrucciones LOAD que necesitan 4 ciclos, un 10% de instrucciones STORE que necesitan 3 ciclos, un 25% de instrucciones con operaciones de enteros que necesitan 6 ciclos, un 15% de instrucciones con operandos en coma flotante que necesitan 8 ciclos por instrucción, y un 30% de instrucciones de salto que necesitan 3 ciclos. **(a)** ¿Cuál es la ganancia que se puede obtener por reducción en el tiempo de las instrucciones con enteros a 3 ciclos? y **(b)** ¿cuál es la ganancia que se puede obtener por reducción en el tiempo de las instrucciones en coma flotante a 3 ciclos?

Solución

Datos del ejercicio:

Tipo i de instr.	CPI_i (c/i)	NI_i	CPI_i^a	CPI_i^b
LOAD	4	0.20*NI	4	4
STORE	3	0.10*NI	3	3
ENTEROS	6	0.25*NI	3	6
FP	8	0.15*NI	8	3
BR	3	0.30*NI	3	3
TOTAL		**NI**		

La frecuencia de reloj no hará falta.

El tiempo total de procesamiento antes de la mejora es igual a:

$$T_{sin\_mejora} = NI \times CPI \times T_{ciclo} = (\, NI \times 0.2 \times 4 + NI \times 0.1 \times 3 + NI \times 0.25 \times 6 + NI \times 0.15 \times 8 + NI \times 0.3 \times 3 \,) \times T_{ciclo} =$$
$$NI \times (0.2 \times 4 + 0.1 \times 3 + 0.25 \times 6 + 0.15 \times 8 + 0.3 \times 3) \times T_{ciclo} =$$
$$NI \times (0.8 + 0.\,3 + 1.5 + 1.2 + 0.9) \times T_{ciclo} = NI \times 4.7 \times T_{ciclo}$$

donde NI es el número de instruciones y T_{ciclo}=1/frecuencia

(a) En el caso de mejoras en la ejecución de enteros, el tiempo sería:

$$T_{mejora\_ent} = NI \times CPI \times T_{ciclo} = NI \times (0.2 \times 4 + 0.1 \times 3 + 0.25 \times 3 + 0.15 \times 8 + 0.3 \times 3) \times T_{ciclo} =$$
$$NI \times (0.8 + 0.\,3 + 0.75 + 1.2 + 0.9) \times T_{ciclo} = NI \times 3.95 \times T_{ciclo}$$

y por tanto la ganancia sería:

$$S_{mejora\_ent} = T_{mejora\_ent} / T_{sin\_mejora} = NI \times 4.7 \times T_{ciclo} \,/\, NI \times 3.95 \times T_{ciclo} = 4{,}7/3{,}95 = 1{,}8987$$

$$S_{mejora\_ent} = 1.19$$

(b) En el caso de mejoras en operaciones de coma flotante, el tiempo sería:

$$T_{mejora\_fp} = = NI \times CPI \times T_{ciclo} = NI \times (0.2 \times 4 + 0.1 \times 3 + 0.25 \times 6 + 0.15 \times 3 + 0.3 \times 3) \times T_{ciclo} = NI \times (0.8 + 0.3 + 1.5 + 0.45 + 0.9) \times T_{ciclo} = NI \times 3.95 \times T_{ciclo}$$

y por tanto la ganancia sería:

$$S_{mejora\_ent} = T_{mejora\_ent} / T_{sin\_mejora} = NI \times 4.7 \times T_{ciclo} \; / \; NI \times 3.95 \times T_{ciclo} = 4{,}7/3{,}95 = 1{,}8987$$

$$S_{mejora\_fp} = 1.19$$

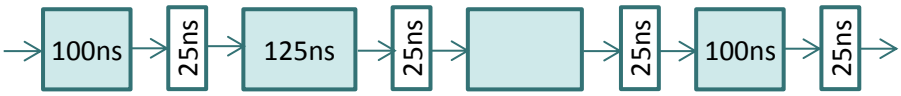
En ambos casos la ganancia conseguida es la misma.

Ejercicio 2. Un circuito que implementaba una operación en T_{op}=450 ns. se ha segmentado mediante un cauce lineal con cuatro etapas de duracción T1=100 ns., T2=125 ns., T3=125 ns., y T4=100 ns. respectivamente, separadas por un registro de acoplo que introduce un retardo de 25 ns. **(a)** ¿Cuál es la máxima ganancia de velocidad posible? ¿Cuál es la productividad máxima del cauce? **(b)** ¿A partir de qué número de operaciones ejecutadas se consigue una productividad igual al 90% de la productividad máxima?

Solución

Datos del ejercicio:

Sin segmentar



Con segmentación

(a) La ganancia en velocidad para n entradas se obtiene dividiendo en tiempo que tardan en ejecutarse n entradas en el circuito sin segmentar entre el tiempo que supone la ejecución en el circuito segmentado:

$$S(n) = \frac{T^{ss}(n)}{T^{cs}(n)}$$

El tiempo de ejecución de n entradas en el circuito sin segmentar es:

$$T^{ss}(n) \; = \; T_{op} \times n = 450 \text{ ns} \times n$$

El circuito se ha segmentado en cuatro etapas separadas por un registro de acoplo con un retardo de d=25 ns. El tiempo de ciclo del cauce, t, se obtiene a partir de la expresión:

$$t = \max\{T1, T2, T3, T4\} + d = \max\{100, 125, 125, 100\} + 25 \text{ ns} = 150 \text{ ns}$$

En ese cauce, una operación tarda un tiempo $TLI = 4 \times 150$ ns. (tiempo de latencia de inicio) en pasar por las cuatro etapas del mismo. El tiempo de ejecución de las n entradas en el circuito segmentado es:

$$T^{cs}(n) = TLI + (n-1) \times t = 4 \times 150\ ns + (n-1) \times 150\ ns$$

La ganancia en prestaciones conseguida al segmentar sería:

$$S(n) = \frac{T^{ss}(n)}{T^{cs}(n)} = \frac{450\ ns \times n}{4 \times 150\ ns + (n-1) \times 150\ ns} = \frac{3 \times n}{3+n}$$

El valor máximo de la ganancia de velocidad se obtiene aplicando el límite cuando n tiende a infinito:

$$S(n) = \lim_{n\to\infty} \frac{3 \times n}{3+n} = \mathbf{3}$$

La productividad del cauce es:

$$P(n) = \frac{n}{T^{cs}(n)} = \frac{n}{4 \times 150\ ns + (n-1) \times 150\ ns} = \frac{n}{(3+n) \times 150\ ns}$$

La productividad máxima es:

$$P(n) = \lim_{n\to\infty} \frac{n}{(3+n) \times 150\ ns} = \frac{1}{150\ ns} = \mathbf{6.67\ Mop./ns}$$

(c) El valor de n para el que se consigue una productividad igual al 90% de la productividad máxima es:

$$P(n) = \frac{n}{(3+n) \times 150\ ns} = 0.9 \times \frac{1}{150\ ns} \Rightarrow \frac{n}{3+n} = 0.9 \Rightarrow n = 2.7 + 0.9 \times n \Rightarrow n = \frac{2.7}{0.1} = \mathbf{27\ op.}$$

Con 27 operaciones se alcanza el 90% de la productividad máxima. La productividad aumentará conforme se incremente n.

Ejercicio 3. En un procesador sin segmentación de cauce, determine cuál de estas dos alternativas para realizar un salto condicional es mejor:

•ALT1: Una instrucción COMPARE actualiza un código de condición y es seguida por una instrucción BRANCH que comprueba esa condición.

•ALT2: Una sola instrucción incluye la funcionalidad de las instrucciones COMPARE y BRANCH.

Hay que tener en cuenta que para ALT1, en el conjunto de programas de prueba, el 20% de las instrucciones son instrucciones BRANCH asociadas a salto condicional; que las instrucciones BRANCH en ALT1 y COMPARE+BRANCH en ALT2 necesitan 4 ciclos mientras que todas las demás necesitan sólo 3; y que el ciclo de reloj de la ALT1 es un 25% menor que el de la ALT2, dado que en éste caso la mayor funcionalidad de la instrucción COMPARE+BRANCH ocasiona una mayor complejidad en el procesador.

Solución

Datos del ejercicio:

$T^1_{ciclo} = T^2_{ciclo} - 0{,}25 \times T^2_{ciclo} = 0{,}75 \times T^2_{ciclo}$

ALT1

Tipo i de instr.	ciclos	NI_i	Proporción (%)	Comentarios
BRANCH en ALT1	4	$0{,}2 \times NI^1$	20	
RESTO en ALT1	3	$0{,}8 \times NI^1$	80	Incluye instrucciones COMPARE
TOTAL		**NI^1**	**100**	**NI^1 : nº instr. en ALT1**

ALT2

Tipo i de instr.	ciclos	NI_i	Proporción (%)	Comentarios
COMPARE+BRANCH en ALT2	4	$0{,}2\times NI^1$	20×100/80 = 25	ALT2 no tendrá aparte las instrucciones COMPARE, que son un 20% en ALT1
RESTO en ALT2	3	$0{,}6\times NI^1$	(80-20) ×100/80 = 75	ALT2 no tendrá aparte las instrucciones COMPARE, que son un 20% en ALT1
TOTAL		**$NI^2 = 0{,}8\times NI^1$**	**100**	Se reducen el nº de instr. en ALT2, **NI^2,** al no tener instr. COMPARE aparte

El tiempo de CPU se puede expresar como:

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

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.

Para realizar la comparativa se va a calcular T^1_{CPU} y T^2_{CPU} en función de las mismas variables:

- Para la alternativa primera, *ALT1*, se tiene:

$$T^1_{CPU} = NI^1 \times CPI^1 \times T^1_{ciclo} = NI^1 \times (0.2\times 4 + 0.8\times 3) \times T^1_{ciclo} = NI^1 \times 3.2 \times 0.75 \times T^2_{ciclo} = NI^1 \times 2.4 \times T^2_{ciclo}$$

(CPI para ALT1 es $CPI^1 = 0.2\times 4 + 0.8\times 3 = 3.2$)

- Para la alternativa segunda, *ALT2*, se tiene:

$$T^2_{CPU} = NI^2 \times CPI^2 \times T^2_{ciclo} = NI^1 \times (0.2\times 4 + 0.6\times 3) \times T^2_{ciclo} = NI^1 \times 2.6 \times T^2_{ciclo}$$

(CPI para ALT2 es $CPI^2 = \frac{0.2\times NI^1\times 4 + 0.6\times NI^1\times 3}{NI^2} = (0.2\times 4 + 0.6\times 3) \times \frac{NI^1}{NI^2} = \frac{0.8+1.8}{0.8} = \frac{2.6}{0.8} = 3.25$)

Los tiempos de CPU para *ALT2* y para *ALT1* se han expresado en función de las mismas variables (número de instrucciones de *ALT1*, NI^1, y tiempo de ciclo de *ALT2,* T^2_{ciclo}) para poder comparar entre ambos. Como se puede ver $T^1_{CPU} < T^2_{CPU}$ y, por tanto, se tiene que *ALT2* no mejora a *ALT1*. Es decir, aunque un repertorio con instrucciones más complejas puede reducir el número de instrucciones de los programas, reduciendo el número de instrucciones a ejecutar, esto no tiene que suponer una mejora de prestaciones si al implementar esta opción la mayor complejidad del procesador lo hace algo más lento.

◀

Ejercicio 4. ¿Qué ocurriría en el problema anterior si el ciclo de reloj fuese únicamente un 10% mayor para la ALT2?

Solución

En este caso $T^2_{CICLO} = 1.10 * T^1_{CICLO}$; por tanto:

$$T^1_{CPU} = NI^1 \times CPI^1 \times T^1_{ciclo} = NI^1 \times (0.2\times 4 + 0.8\times 3) \times T^1_{ciclo} = NI^1 \times 3.2 \times T^1_{ciclo}$$

$$T^2_{CPU} = NI^2 \times CPI^2 \times T^2_{ciclo} = NI^1 \times (0.2\times 4 + 0.6\times 3) \times 1.1 \times T^1_{ciclo} = NI^1 \times 2.86 \times T^1_{ciclo}$$

Así, ahora $T^2_{CPU} < T^1_{CPU}$ y por lo tanto, la segunda alternativa sí que es mejor que la primera. Es decir, el mismo planteamiento que antes no mejoraba la situación de partida ahora sí lo consigue. Sólo ha sido necesario que el aumento del ciclo de reloj que se produce al hacer un diseño más complejo del procesador sea algo menor.

La situación reflejada en estos dos ejercicios (3 y 4) pone de manifiesto la necesidad de estudiar la arquitectura del computador desde un enfoque cuantitativo, y no sólo teniendo en cuenta razonamientos que pueden ser correctos, pero que pueden llevar a resultados finales distintos según el valor de las magnitudes implicadas.

◀

Ejercicio 5. 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 43% de las instrucciones son operaciones con la ALU (3 CPI), el 21% LOADs (4 CPI), el 12% STOREs (4 CPI) y el 24% BRANCHs (4 CPI).

Se ha podido comprobar que un 25% de las operaciones con la ALU utilizan operandos en registros que no se vuelven a utilizar. Compruebe si mejorarían las prestaciones si, para sustituir ese 25% de operaciones, se añaden instrucciones con un dato en un registro y otro en memoria. Tengan en cuenta en la comprobación que para estas nuevas instrucciones el valor de CPI es 4 y que añadirlas ocasiona un incremento de un ciclo en el CPI de los BRANCHs, pero no afectan al ciclo de reloj.

Solución

Datos del ejercicio:

Alternativa 1

I_i^1	CPI_i^1	NI_i^1	Comentarios
LOAD	4	$0{,}21\times NI^1$	
STORE	4	$0{,}12\times NI^1$	
ALU	3	$0{,}43\times NI^1$	25 % inst. que usan opernados en registros que no se vuelven a utilizar
BR	4	$0{,}24\times NI^1$	
TOTAL		NI^1	

Alternativa 2

I_i^2	CPI_i^2	NI_i^2	Comentarios
LOAD	4	$(0{,}21-0{,}25\times0{,}43)\times NI^1 = 0{,}1025\times NI^1$	El 25 % de 43% desaparece al usarse ese mismo número de operaciones con la ALU que acceden a memoria
STORE	4	$0{,}12\times NI^1$	
ALU r-r	3	$0{,}75\times0{,}43\times NI^1 = 0{,}3225\times NI^1$	
ALU r-m	4	$0{,}25\times0{,}43\times NI^1 = 0{,}1075\times NI^1$	25% de las operaciones con la ALU; es decir el 25% del 43%
BR	5	$0{,}24\times NI^1$	
TOTAL		**$NI^2=0{,}8925\times NI^1$**	**Es decir, NI^2 es igual a $0{,}8925\times NI^1$**

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

- Se reduce el 25% de las $0.43*NI^1$ instrucciones de operación con la ALU y operandos en registros a las que las nuevas instrucciones sustituyen, donde NI^1 es el nº de instrucciones para la arquitectura original.
- Se reduce en $0.25*0.43*NI^1$ el número de 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 $0.25*0.43*NI^1$ nuevas instrucciones de operación con la ALU que se introducen en el repertorio (y que sustituyen a las 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 $NI^2=0.8925*NI^1$.

En primer lugar se calcula el tiempo de CPU para la situación inicial:

$$T^1_{CPU} = CPI^1\times NI^1\times T_{ciclo} = (0.43\times3+0.21\times4+0.12\times4+0.24\times4)\times NI^1\times T_{ciclo} = (1.29+2.28)\times NI^1\times T_{ciclo} = 3.57\times NI^1\times T_{ciclo}$$

(CPI para ALT1 es $CPI^1=0.43\times3+0.21\times4+0.12\times4+0.24\times4 = 1.29+0.57\times4 = 1.29+2.28 = 3.57$)

En segundo lugar se calcula el tiempo de CPU para la alternativa 2:

$$T^2_{CPU} = CPI^2 \times NI^2 \times T_{ciclo} = [0.3225 \times 3 + (0.12+0{,}1025+0{,}1075) \times 4 + 5 \times 0{,}24] \times NI^1 \times T_{ciclo} = (0{,}9675+0{,}33 \times 4+0{,}24 \times 5) \times NI^1 \times T_{ciclo} = 3.4875 \times NI^1 \times T_{ciclo}$$

Como se puede ver, en este caso, $T^1_{CPU} > T^2_{CPU}$, y por lo tanto se mejoran las prestaciones, pero si el porcentaje de instrucciones sustituidas fuese un 20% en lugar de un 25%, en ese caso, se puede ver que

$$T^2_{CPU} = CPI^2 \times NI^2 \times T^2_{ciclo} = 3.59 \times NI^1 \times T_{ciclo}$$

Ahora, en cambio, la segunda opción no mejora 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 tiene el proceso de definición de conjuntos de *benchmarks* para evaluar las prestaciones de los computadores, y las dificultades que pueden surgir para que los fabricantes se pongan de acuerdo en aceptar un conjunto de *benchmarks* estándar.

Ejercicio 6. Se ha diseñado un compilador para la máquina LOAD/STORE del problema anterior. Ese compilador puede reducir en un 50% el número de operaciones con la ALU, pero no reduce el número de LOADs, STOREs, y BRANCHs. Suponiendo que la frecuencia de reloj es de 50 Mhz. ¿Cuál es el número de MIPS y el tiempo de ejecución que se consigue con el código optimizado? Compárelos con los correspondientes del código no optimizado.

Solución

Datos del ejercicio:

Alternativa 1

I_i^1	CPI_i^1	NI_i^1/NI^1	Comentarios
LOAD	4	0,21	
STORE	4	0,12	
ALU	3	0,43	
BR	4	0,24	
TOTAL		**1**	

Alternativa 2

I_i^2	CPI_i^2	NI_i^2/NI^1	Comentarios
LOAD	4	0,21	
STORE	4	0,12	
ALU r-r	3	0,5 * 0,43 = 0,215	Se reducen las instrucciones que usan la ALU en un 50%
BR	4	0,24	
TOTAL		**0,785**	**Es decir, $NI^2 = 0{,}785 * NI^1$**

En la situación inicial del problema anterior se tenía que

$$T^1_{CPU} = CPI * NI^1 * T_{ciclo} = 3.57 * NI^1 * T_{ciclo}$$

y, por lo tanto

$$MIPS^1 = \frac{NI^1}{T^1_{CPU} * 10^6} = \frac{NI^1}{CPI * NI^1 * Tciclo * 10^6} = \frac{f(hz)}{CPI * 10^6} = \frac{50 * 10^6 \text{ c/s}}{3.57 \text{ c/i} * 10^6} = \mathbf{14.005\ MIPS}$$

El tiempo de CPU para la alternativa 2 es:

$$T^2_{CPU} = CPI^2 * NI^2 * T_{ciclo} = [\ 0.215*3 + (0.21+0.12+0.24)*4\ ] * NI^1 * T_{ciclo} = (0.645+0.57*4) * NI^1 * T_{ciclo} = (0.645+2.28) * NI^1 * T_{ciclo} = 2.925 * NI^1 * T_{ciclo}$$

$$T^2_{CPU} = 2.925 * \frac{NI^1}{NI^2} * NI^2 * T_{ciclo} = 2.925 * \frac{NI^1}{NI^2} * NI^2 * T_{ciclo} = 3.726 * NI^2 * T_{ciclo}$$

$$MIPS^2 = \frac{NI^2}{T^2_{CPU} * 10^6} = \frac{NI^2}{CPI * NI^2 * Tciclo * 10^6} = \frac{f(hz)}{CPI * 10^6} = \frac{50 * 10^6\ c/s}{3.726\ c/i\ * 10^6} = \mathbf{13.42\ MIPS}$$

Como se puede ver, se consigue una reducción de tiempo de ejecución (T^2_{CPU} es menor que T^1_{CPU}), pero sin embargo, el número de MIPS para la segunda opción es menor. Se tiene aquí un ejemplo en el que los MIPS dan una información inversamente proporcional a las prestaciones. La razón de esta situación, en este caso, es que se ha reducido el número de las instrucciones que tenían un menor valor de CPI. Así, se incrementan las proporciones de las instrucciones más lentas en el segundo caso (por eso crece el valor de CPI), y claro, el valor de los MIPS se reduce. No obstante, hay que tener en cuenta que aunque las instrucciones que se ejecutan son más lentas, hay que ejecutar un número menor de instrucciones, y al final, el tiempo se reduce.

Ejercicio 7. En un programa que se ejecutan en un procesador no segmentado que funciona a 100 MHz, hay un 20% de instrucciones LOAD que necesitan 4 ciclos, un 15% de instrucciones STORE que necesitan 3 ciclos, un 40% de instrucciones con operaciones en la ALU que necesitan 6 ciclos, y un 25% de instrucciones de salto que necesitan 3 ciclos. **(a)** Si en las instrucciones que usan la ALU el tiempo en la ALU supone 4 ciclos, determine cuál es la máxima ganancia que se puede obtener si se mejora el diseño de la ALU de forma que se reduce su tiempo de ejecución a la mitad de ciclos. **(b)** ¿Con qué porcentaje de instrucciones con operaciones en la ALU se podría haber obtenido en los cálculos del apartado (a) una ganancia mayor que 2? Razone su respuesta.

Solución

Datos del ejercicio:

Alternativa 1

I_i^1	CPI_i^1	NI_i/NI	Comentarios
LOAD	4	0,2	
STORE	3	0,15	
ALU	6	0,4	4 ciclos en la ALU
BR	3	0,25	
TOTAL		**1**	

Alternativa 2

I_i^2	CPI_i^2	NI_i/NI	Comentarios
LOAD	4	0,2	
STORE	3	0,15	
ALU	2+2=4	0,4	Se reducen en 2 ciclos lo que consumen en la ALU
BR	3	0,25	
TOTAL		**1**	

(a) El tiempo de ejecución sin la mejora sería igual a

$$T^1_{CPU} = NI*(0.2*4 + 0.15*3 + 0.4*6 + 0.25*3)*T_c = NI * 4.4 * T_c$$

donde f es la frecuencia y NI el número de instrucciones promedio de los programas. Puesto que la mejora consiste en que las operaciones con la ALU necesitan la mitad de tiempo, dado que dicha operación

consume 4 ciclos, tras la mejora pasaría a necesitar 2 ciclos, y por tanto, las instrucciones con la ALU necesitarían 2+2=4 ciclos. Por tanto, el tiempo con la mejora sería:

$$T^2_{CPU} = NI*(0.2*4 + 0.15*3 + 0.4*4 + 0.25*3)*T_c = NI * 3.6 * T_c$$

De esta forma, la ganancia de velocidad sería:

$$S = T^1_{CPU} / T^2_{CPU} = 4.4/3.6 = \mathbf{1.22}$$

(b) La ganancia en prestaciones conseguida en una instrucción de la ALU es de 6 ciclos/4 ciclos=1.5. Aunque todas las intrucciones fuesen instrucciones de la ALU la ganancia nunca podría llegar a 2, llegaría a 1.5, que es lo que han mejorado las instrucciones de la ALU. Supongamos que todas las instrucciones son de la ALU, entonces:

$$T^2_{CPU} = 4 \text{ ciclos} * NI$$

$$T^1_{CPU} = 6 \text{ ciclos} * NI$$

$$S = T^1_{CPU} / T^2_{CPU} = 1.5$$

Ejercicio 8. Suponga que, en los programas que constituyen la carga de trabajo habitual de un procesador, las instrucciones de coma flotante consumen un promedio del 13% del tiempo del procesador.

(a) Ha aparecido en el mercado una nueva versión del procesador en la que la única mejora con respecto a la versión anterior es una nueva unidad de coma flotante que permite reducir el tiempo de las instrucciones de coma flotante a tres cuartas partes del tiempo que consumían antes. ¿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?

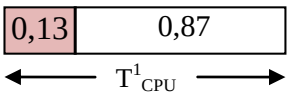
(b) ¿Cuál es la máxima ganancia de velocidad que, en promedio, puede esperarse en los programas debido a mejoras en la velocidad de las operaciones en coma flotante?

(c) ¿Cuál debería ser el porcentaje de tiempo de cálculo con datos en coma flotante en los programas que ejecuta el procesador para esperar una ganancia máxima de 4?

(d) En la situación anterior, ¿cuánto debería reducirse el tiempo de las operaciones en coma flotante con respecto a la situación inicial para que la ganancia sea 2?

Solución

Datos del ejercicio:



(a) Resolución 1:

$$T^2_{CPU} = 0.13 \times \frac{3}{4} \times T^1_{CPU} + 0.87 \times T^1_{CPU} = 0.0975 \times T^1_{CPU} + 0.87 \times T^1_{CPU} = 0.9675 \times T^1_{CPU}$$

$$S \leq \frac{T^1_{CPU}}{T^2_{CPU}} = \frac{T^1_{CPU}}{0.9675 \times T^1_{CPU}} \cong \mathbf{1.0336}$$

(a) Resolución 2 (se usa la expresión que caracteriza la ley de Amdahl):

Como el tiempo de las instrucciones de coma flotante se reduce tres cuartas partes, la mejora de velocidad es *p=4/3* (la inversa de la reducción del tiempo).

Como el porcentaje de instrucciones de coma flotante es el 13%, la fracción a la que se puede aplicar la mejora es:

(1-f) = 0.13, y por tanto, f =0.87

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

$$S \leq \frac{p}{1+f\times(p-1)} = \frac{4/3}{1+0.87\times(4/3-1)} = \frac{4/3}{1+0.87\times 1/3} = \frac{4}{3+0.87} \cong \mathbf{1.0336}$$

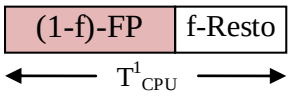
(b) Resolución 1: La ganancia será mejor cuanto menor sea el tiempo que supongan las operaciones FP. La mayor ganancia se conseguiría haciendo ese tiempo despreciable, es decir, si la mejora de las unidades FP se hace muy muy grande, infinito. En este caso:

$$T^2_{CPU} = 0 + 0.87\times T^1_{CPU} = 0.87\times T^1_{CPU} \qquad S \leq \frac{T^1_{CPU}}{T^2_{CPU}} = \frac{T^1_{CPU}}{0.87\times T^1_{CPU}} = \frac{1}{0.87} \cong \mathbf{1.149}$$

(b) Resolución 2 (se usa la expresión que caracteriza la ley de Amdahl): Para determinar el valor de la ganancia máxima que se pide, se obtiene el límite cuando la mejora del recurso (instrucciones en coma flotante) tiende a infinito:

$$S_{max(f=cte)} \leq \lim_{p\to\infty} \frac{p}{1+f\times(p-1)} = \frac{1}{f} = \frac{1}{0.87} \cong \mathbf{1.149}$$

(c) Resolución 1:



Si se cambia la fracción de código con FP (la notamos por (1-f)) y tenemos en cuenta que para obtener la mayor ganancia se debe hacer el tiempo de ejecución de las FP igual a 0, entonces:

$$T^2_{CPU} = 0 + f\times T^1_{CPU} \qquad S \leq \frac{T^1_{CPU}}{T^2_{CPU}} = \frac{T^1_{CPU}}{f\times T^1_{CPU}} = \frac{1}{f} = 4 \Rightarrow f = 0.25 \text{ y } (1-f) = \mathbf{0.75}$$

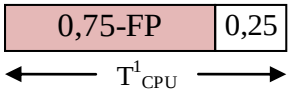
f debería ser al menos 0,25 y, por tanto, 1-f=0,75; es decir,las operaciones FP deberían ser un 75%.

(c) Resolución 2 (se usa la expresión que caracteriza la ley de Amdahl): Para que la ganancia máxima (es decir cuando la mejora, *p*, tiende a infinito) sea igual a 4, la fracción del tiempo inicial que no se reduce con la mejora se obtendría a partir de:

$$S_{max(f=cte)} \leq \frac{1}{f} = 4$$

Por lo que *f=1/4=0.25*, y la fracción de tiempo que tendría que poder reducirse con la mejora tendría que ser *(1-f)=1-0.25=**0.75***.

(d) Resolución 1:



$$T^2_{CPU} = 0.75\times\frac{1}{p}\times T^1_{CPU} + 0.25\times T^1_{CPU}$$

$$S \leq \frac{T^1_{CPU}}{T^2_{CPU}} = \frac{T^1_{CPU}}{0.75\times\frac{1}{p}\times T^1_{CPU} + 0.25\times T^1_{CPU}} = \frac{1}{0.75\times\frac{1}{p} + 0.25} = 2 \Rightarrow p = \frac{1.5}{0.5} = 3$$

El tiempo se debería reducir a 1/3 del tiempo original. Es decir, las instrucciones de coma flotante deberían hacerse tres veces más rápidas.

(d) Resolución 2 (se usa la expresión que caracteriza la ley de Amdahl): Si, al utilizar el valor de *f* anterior se deseara obtener una mejora de velocidad de 2 en los programas, la mejora de velocidad en las instrucciones de coma flotante (es decir *p*) se puede obtener a partir de la ley de Amdahl:

$$S \leq 2 = \frac{p}{1 + f \times (p-1)} = \frac{1}{f + \frac{f-1}{p}} = \frac{1}{25 + \frac{0.75}{p}} \Rightarrow p = \frac{1.5}{0.5} = \mathbf{3}$$

Es decir, las instrucciones de coma flotante deberían hacerse tres veces más rápidas.

Ejercicio 9. Suponga que, en el código siguiente, `a[]` es un array de números de 32 bits en coma flotante y `b` un número de 32 bits en coma flotante, y que debería ejecutarse en menos de 0,5 segundos para `N`=10^9:

```
for (i=0; i<N; i++)  a[i+2]=(a[i+2]+a[i+1]+a[i])*b;
```

(a)¿Cuántos GFLOPS se necesitan para poder ejecutar el código en menos de 0,5 segundos?

(b) Suponiendo que este código en ensamblador tiene `7N` instrucciones y que se ha ejecutado en un procesador de 32 bits a 2 GHz. ¿Cual es el número medio de instrucciones que el procesador tiene que poder completar por ciclo para poder ejecutar el código en menos de 0,5 segundos?

(c) Estimando que el programa pasa el 75% de su tiempo de ejecución realizando operaciones en coma flotante, ¿cuánto disminuiría como mucho el tiempo de ejecución si se redujesen un 75% los tiempos de las unidades de coma flotante?

Solución

(a) Puesto que hay tres operaciones en coma flotante por iteración (dos sumas y un producto, con igual coste para el producto y la suma) el número de operaciones en coma flotante a realizar tras las N iteraciones es $N_{flot}=3\times N = 3\times 10^9$.

Por lo tanto, GFLOPS = $N_{flot}/(t\times 10^9)$ donde t es el tiempo en segundos. Así:

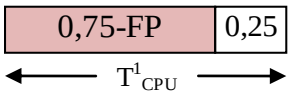
$$\text{GFLOPS} = N_{flot}/(t\times 10^9) = 3\times 10^9/0.5\times 10^9 = 6 \text{ GFLOPS}$$

(b) Si el programa tiene NI = 7N = 7 10^9 instrucciones, teniendo en cuenta que

$$T = NI\times CPI\times Tciclo = NI\times CPI/f = (7\ 10^9 \text{ instrucciones}) \times CPI \,/\, (2\times 10^9 \text{ ciclos/s}) < 0.5 \text{ s}$$

Al despejar CPI < 0.5/3.5, y el número de instrucciones por ciclo IPC = 1/CPI > 7

(c) Resolución 1:



$$T^2_{CPU} = 0.25 \times T^1_{CPU} + 0.25 \times 0.75 \times T^1_{CPU} = 0.25 \times T^1_{CPU} \times (1 + 0.75)$$
$$S \leq \frac{T^1_{CPU}}{T^1_{CPU}} - \frac{T^2_{CPU}}{T^1_{CPU}} = 1 - 0.25 \times 1.75 = 1 - 0.4375 = 0.5625 \Rightarrow 56.25\%$$

Por tanto, el tiempo se reduce un 56.25%

Resolución 2: Para resolver la última cuestión se puede recurrir a la ley de Amdahl. En este caso f=0.25 es la fracción de tiempo en el que no se puede aprovechar la mejora; el incremento de velocidad es p=1/0.25 (si el tiempo de las operaciones en coma flotante era t, el tiempo de las operaciones en coma flotante con la mejora es 0.25t, dado que se ha reducido un 75%). La mejora de velocidad que se observaría se puede obtener aplicando la ley de Amdahl, sustituyendo convenientemente:

$$S \leq p/(1+f(p-1)) = 4/(1+0.25\times 3) = 2.286$$

Como $S = T_{sin\_mejora}/T_{con\_mejora} \leq 2.286$ se tiene que $T_{sin\_mejora}/2.286 = 0.4375*T_{sin\_mejora} \leq T_{con\_mejora}$. Es decir, el tiempo con la mejora podría pasar a ser el 43.75% del tiempo sin la mejora y, por tanto, se reduce el tiempo en un 56.75%.

◀

Ejercicio 10. Un compilador ha generado un código máquina optimizado para el siguiente programa

```
par=0; impar=0;
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 (porque se ha usado la técnica de desenrollado el bucle que veremos en el Seminario 4): el código tiene un número de iteraciones de N/2, 7 instrucciones fuera del bucle (2 de almacenamiento en memoria, 5 instrucciones para inicializar registros), 9 instrucciones dentro del bucle (3 instrucciones para implementar el salto y la actualización de la variable de control `i`, 4 instrucciones coma flotante y dos instrucciones de carga desde memoria a registro -se leen dos componentes de `x`).El computador donde se ejecuta dispone de:

- Un procesador superescalar de 32 bits a 2 GHz capaz de terminar dos instrucciones de coma flotante por ciclo y dos instrucciones de cualquier otro tipo por ciclo, excepto instrucciones de carga, cuyo tiempo depende de si hay o no fallo de cache.

- Dos caches integradas en el chip de procesamiento (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.

- Una memoria principal con latencia de 30 ns y ciclos burst 6-1-1-1 a través de un bus de memoria de 200 Mhz.

Conteste a las siguientes cuestiones: **(a)** ¿Cuál es la velocidad pico del procesador (en GFLOPS)? **(b)** ¿Cuál es el tiempo mínimo que tarda en ejecutarse el programa para $N=2^{11}$. **(c)** ¿Cuántos MFLOPS alcanza el programa?

(NOTA: Considere que el vector x se almacena en memoria en una dirección múltiplo del tamaño de una línea de cache y que ningún componente está en cache cuando se referencia; `N`, `i` estarán en registros de enteros, `par`, `impar`, `c`, y `x[]` son números de 32 bits en coma flotante,dentro del bucle `c`, `par` e `impar` estarán en registros).

Solución

Datos del ejercicio:

Código optimizado	Instrucciones máquina	Ciclos (ver tercera tabla del ejercicio)
`par=0; impar=0;`	5 instr. inialización de registros (par, impar, i, N, c)	5 instr. * 0.5 ciclos/instr. = 2.5 ciclos
`for (i=0;i<N;i=i+2)`	4 instr. para control del bucle (ADD, CMP, BRCND, BR)	4 instr. * 0.5 ciclos/instr.= 2 ciclos
`par=par+c*x[i];` `impar=impar-c*x[i+1];`	2 instr. LOAD (x[i],x[i+1]) 4 instr. FP	- Con fallo de cache: 61 ciclos ª1 componente, 62 ciclos 2ª, 71 ciclos 3ª, 72 ciclos 4ª (ver tabla segunda). - Sin fallo de cache: 2 instr. * 1 ciclo/instr. = 2 ciclos - 4 instr. * 0.5 ciclos/instr. = 2 ciclos
	2 instr. STORE (par, impar)	2 instr. * 1 ciclo/instr. = 2 ciclos
	9 * N/2 + 7 instr. en total	

En la tabla, ADD nota instrucción de suma, CMP instrucción de comparación, BR instrucción de salto y BRCND instrucción de salto condicional. El bucle for se implementa en ensamblador con 4 instrucciones: 3 instrucciones (incremento variable de control `i`, ADD, comparación variable de control con `i`, CMP, salto condicional si `i` no alcanza `N`, BRCND) antes del código que ejecutan las iteraciones del bucle y una instrucción detrás de estas instrucciones (salto incondicional, BR, a la instrucción que incrementa `i`).

burst	6 – 1 – 1 – 1	Frecuencia del bus = 200 MHz => ciclo de 1/200 MHz = 5 ns => 6 ciclos = 30ns
Flanco reloj	↑ ↓ ↑ ↓ ↑ ↓ ↑ ↓	Procesador de 32 bits (4B): se supone entonces que el bus trasfiere 4B en paralelo (1 palabra)
Bytes transferidos	4B-4B 4B-4B 4B-4B 4B-4B 8B 8B 8B 8B =32B	Línea de cache de 32B: se supone entonces que un *burst* obtiene una línea => hay transferencias en el flanco subida y en el flanco bajada del reloj (8B)
Ciclos procesador	60 10 10 10 ciclos	Ciclos procesador por ciclos bus: $\frac{2\ GHz\ proc.}{0.2\ GHz\ bus} = 10\ ciclos$
Ciclos componente	61 71 81 91 ciclos x[0] x[2] x[4] x[6] 62 72 82 92 ciclos x[1] x[3] x[5] x[7]	Se supone que x comienza en un múltiplo de una línea de cache. Los datos se obtienen en un ciclo más debido a que la latencia de la cache es de un ciclo. En 60 ciclos estarán en la cache x[0] y x[1]. En 61 ciclos estará x[0] en el procesador y un ciclo más tarde estará x[1]. En 70 ciclos estarán x[2] y x[3] en la cache.

Tipo i	$1/CPI_i$	CPI_i
LOAD	1 instr./ciclo 1/61 instr./ciclo	1 si no hay fallo de cache 61 (para la 1ª componente en la línea de cache) si hay fallo de cache
Resto	2 instr./ciclo	0.5
TOTAL		**1**

Puesto que el programa en ensamblador tiene 5 instrucciones fuera del bucle y 9 dentro (el resultado queda en un registro) y se ha desenrollado el bucle, el número de iteraciones es

(a) Velocidad pico del Procesador:

V_{pico}=2 inst/ciclo × 2.10^9 ciclos/seg = 4.10^9 inst/sg = 4 GFLOPS

(b) El número de iteraciones es la mitad del número de iteraciones del bucle original:

Iteraciones = N / 2 = $2^{11}/2=2^{10}$

El número de instrucciones ejecutadas es (ver primera tabla de datos del ejercicio):

NInstrucciones = $9*2^{10}+7=9223$

El número de fallos de cache será de:

Fallos de cache = nº de líneas de cache en x =

$$\frac{2^{11} comp.\ \times 2^2\ B/comp\,.}{2^5\ B/LC} = 2^8\ Líneas\ de\ cache$$

Tiempo mínimo de procesamiento (ver primera tabla del ejercicio):

Ciclos = 5 instr. * 0.5 ciclos/instr. *(iniciar 5 registros)* + ciclos del bucle + 2 instr. * 0.5 ciclos/instr. *(2 STORE)*

Ciclos = 2.5 ciclos + ciclos del bucle + 1 ciclos = 3.5 ciclos + ciclos del bucle

Ciclos necesarios para acceder a los datos:

- Los datos en la 1ª iteración, `x[0]` y `x[1]`, estarán disponibles en cache pasados 60 ciclos de su acceso, ya que hay un fallo de cache. Cada uno se lee de cache en un ciclo de reloj. La instr. LOAD de `x[0]` supone 61 ciclos debido al fallo de cache y al ciclo de lectura desde la cache.
- Los datos en la 2ª iteración, `x[2]` y `x[3]`, estarán disponibles en cache 10 ciclos después de la llegada a cache de los datos de la primera iteración, `x[0]` y `x[1]` (ver segunda tabla).
- Los datos en la 3ª iteración, `x[4]` y `x[5]`, estarán disponibles en cache 10 ciclos después de la llegada a cache de los datos de la 2ª iteración, `x[2]` y `x[3]`.
- Los datos en la 4ª iteración, `x[6]` y `x[7]`, estarán disponibles en cache 60 ciclos después de la llegada a cache de los datos de la 3ª iteración, `x[4]` y `x[5]` (hay un nuevo fallo de cache).

Ciclos que supone la 1ª iteración del bucle:

Ciclos 1ª iteración del bucle = 3i *(ADD, CMP, BRCND)* *0.5c/i + 60 c *(fallo de cache)* + 1c *(load de cache)*+ 2i*0.5c/i *(2 op. FP)* + 1c *(load de cache)* + 2*0.5c/i *(2 op. FP)* + 1i*0.5c/i *(BR: salto incondicional)*

Ciclos que supone la ejecución de instrucciones entre la instr. de carga de x[0] y la instr. de carga de x[2] :

Ciclos entre instr. carga de x[0] e instr. carga de x[2] *(1 ciclo instr. LOAD de x[0] + 4 instr. FP de la iteración 1 + instr. LOAD de x[1] iteración 1+ instr. BR iteración 1+ ADD, CMP y BRCND de la iteración 2)* = 1c *(load de cache)* + 2i*0.5c/i *(2 op. FP)* + 1c *(load de cache)* + 2*0.5c/i *(2 op. FP)* + 1i*0.5c/i *(BR: salto incondicional)* + 3i *(ADD, CMP, BRCND)* *0.5c/i = 1c + 1c + 1c + 1c + 0.5c + 1.5 c = 6 c

Hay 6 ciclos, pero `x[2]` no estará disponible en cache hasta pasados 10 ciclos desde que `x[0]` y `x[1]` llegaron a cache. Luego la carga y, por consiguiente, el resto de instrucciones de la 2º iteración no podrán ejecutarse hasta que pasen 10 ciclos, no 6 ciclos. Igual ocurre con la 3ª iteración. En la 4ª iteración hay que esperar más, ya que se produce un fallo de cache, lo que supone 60 ciclos.

Teniendo todo esto en cuenta, los ciclos del bucle serían:

Ciclos del bucle =
3i *(ADD, CMP, BRCND iteración 1)* *0.5c/i + 60 c *(fallo de cache x[0])* + max (6c,10c) * 3 *(línea de cache 1)* +

$$\sum_{i=1}^{2^8-1} (\max(6c, 60c) + \max(6c, 10c) * 3)\ (2^8 - 1\ líneas\ de\ cache)$$

+ 1c *(load de cache iteración N/2)* + 2i*0.5c/i *(2 op. FP)* + 1c *(load de cache)* + 2*0.5c/i *(2 op. FP)* + 1i*0.5c/i *(BR: salto incondicional)* + 3i *(ADD, CMP, BRCND)* *0.5c/i

Ciclos del bucle = 1.5c + (60c + 30c)+ (2^8-1) * 90c + 6c = 7.5c + 2^8 * 90c = 7.5c+ 23040c = 23047.5c

Los ciclos en total serían:

$$\text{Ciclos} = 2.5 \text{ ciclos} + \text{ciclos del bucle} + 1 \text{ ciclos} = 3.5 \text{ ciclos} + \text{ciclos del bucle} = 23051\text{c}$$

El tiempo en segundos sería:

$$\text{T} = \frac{23051 \text{ c}}{2 \times 10^9 \text{ c/s}} = 11524.5 \text{ ns}$$

(c) MFLOPS que alcanza el programa (una operación FP de multiplicación y otra de resta por iteración, no se van a tener en cuenta en el cáluclo las operaciones de acceso a memoria):

$$\text{MFLOPS} = \frac{2 \ op./iter. \times 2^{11} \text{ iter.}}{11524.5 \times 10^{-9} \text{ s} \times 10^6} \cong 355.42$$

◀

2 Cuestiones

Cuestión 1. Indique cómo se podría aprovechar un computador MISD para acelerar la determinación de si `n` números son primos o no. Considere que se conocen los M números primos entre 1 y el máximo valor que puede tener un número de entrada.

◀

Cuestión 2. Indique cuál es la diferencia fundamental entre una arquitectura CC-NUMA y una arquitectura SMP.

Solución

Ambos, SMP y CC-NUMA, son multiprocesadores, es decir, comparten el espacio de direcciones físico. También en ambos se mantiene coherencia entre caches de distintos procesadores (CC- *Cache-Coherence*).

En un SMP (*Symmetric Multiprocessor*) la memoria se encuentra centralizada en el sistema a igual distancia (latencia) de todos los procesadores mientras que, en un CC-NUMA (*Cache-Coherence Non Uniform Memory Access*), cada procesador tiene físicamente cerca un conjunto de sus direcciones de memoria porque los módulos de memoria están físicamente distribuidos entre los procesadores y, por tanto, los módulos no están a igual distancia (latencia) de todos los procesadores.

La memoria centralizada del SMP hace que el tiempo de acceso a una dirección de memoria en un SMP sea igual para todos los procesadores y el tiempo de acceso a memoria de un procesador sea el mismo independientemente de la dirección a la que se esté accediendo; por estos motivos se denomina multiprocesador simétrico (*Symmetric Multiprocessor*) o UMA (*Uniform Memory Access*).

Los módulos de memoria físicamente distribuidos entre los procesadores de un CC-NUMA hacen que el tiempo de acceso a memoria dependa de si el procesador accede a una dirección de memoria que está en la memoria física que se encuentra en el nodo de dicho procesador (cercana, por tanto, al procesador que accede) o en la memoria física de otro nodo. Por este motivo el acceso no es uniforme o simétrico y recibe el nombre de NUMA.

◀

Cuestión 3.¿Cuándo diría que un computador es un multiprocesador y cuándo que es un multicomputador?

Solución. Será un multiprocesador si todos los procesadores comparten el mismo espacio de direcciones físico y será un multicomputador si cada procesador tiene su espacio de direcciones físico propio.

◀

Cuestión 4. ¿Un CC-NUMA escala más que un SMP? ¿Por qué?

Solución. Sí, un CC-NUMA escala más que un SMP, porque al añadir procesadores al cálculo el incremento en el tiempo de acceso a memoria medio aumenta menos que en un SMP si el sistema operativo, el compilador y/o el programador se esfuerzan en ubicar en la memoria cercana a un procesador la carga de trabajo que va a procesar. La menor latencia media se debe a un menor número de conflictos en la red en el acceso a datos y a la menor distancia con el módulo de memoria físico al que se accede. Al aumentar menos la latencia se puede conseguir un tiempo de respuesta mejor en un CC-NUMA que en un SMP al ejecutar un código paralelo o múltiples códigos secuenciales a la vez.

◀

Cuestión 5. Indique qué niveles de paralelismo implícito en una aplicación puede aprovechar un PC con un procesador de 4 cores, teniendo en cuenta que cada core tiene unidades funcionales SIMD (también llamadas unidades multimedia) y una microarquitectura segmentada y superscalar. Razone su respuesta.

Solución

Paralelismo a nivel de operación. Al ser una arquitectura con paralelismo a nivel de instrucción (ejecuta instrucciones en paralelo) por tener una arquitectura segmentada y superescalar, puede ejecutar operaciones en paralelo, luego puede aprovechar el paralelismo a nivel de operación.

Paralelismo a nivel de bucle (paralelismo de datos). Las operaciones realizadas en las iteraciones del bucle sobre vectores y matrices se podrían implementar en las unidades SIMD de los cores, lo que reduciría el número de iteraciones de los bucles. Además las iteraciones de los bucles, si son independientes, se podrían repartir entre los 4 cores.

Paralelismo a nivel de función. Las funciones independientes se podrían asignar a cores distintos para que se puedan ejecutar a la vez.

Paralelismo a nivel de programa. Los programas del mismo o distinto usuario se pueden asignar a distintos cores para así ejecutarlos al mismo tiempo.

◀

Cuestión 6. Si le dicen que un ordenador es de 20 GIPS ¿puede estar seguro que ejecutará cualquier programa de 20000 instrucciones en un microsegundo?

Solución. Los MIPS se definen como el número de instrucciones que puede ejecutar un procesador (en millones) divididos por el tiempo que tardan en ejecutarse.

$$MIPS = \frac{Numero\_de\_Instrucciones}{tiempo(sg.)*10^6}$$

Los MIPS suelen utilizarse como medida de las prestaciones de un procesador, pero realmente sólo permiten estimar la velocidad pico del procesador, que solo permitiría comparar procesadores con el mismo repertorio de instrucciones en cuanto a sus velocidades pico. Esto se debe a que esta medida

- No tiene en cuenta las características del repertorio de instrucciones de procesador. Se da el mismo valor a una instrucción que realice una operación compleja que a una instrucción sencilla.
- Pueden tenerse valores de MIPS inversamente proporcionales a la velocidad del procesador. Si un

procesador tiene un repertorio de instrucciones de tipo CISC que permite codificar un algoritmo con, por ejemplo, 1000 instrucciones, y otro con un repertorio de tipo RISC lo codifica con 2000, si el primero tarda 1 microsegundo y el segundo 1.5 microsegundos, tendremos que el primer procesador tiene 1000 MIPS y el segundo 1333 MIPS. Por tanto, si nos fijamos en los MIPS, el segundo procesador será mejor que el primero, aun precisando un 50% más de tiempo que el primero, para resolver el mismo problema.

De ahí que incluso se haya dicho que MIPS significa *Meaningless Information of Processor Speed* (información sin sentido de la velocidad del procesador).

En relación a la pregunta que se plantea, la respuesta puede tener en cuenta dos aspectos:

- El número de instrucciones que constituyen un programa (número estático de instrucciones) puede ser distinto del número de instrucciones que ejecuta el procesador finalmente (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. Si la pregunta, al hacer referencia al número de instrucciones, se refiere al número estático, la respuesta es que NO se puede estar seguro puesto que puede que al final no se ejecuten 20000 millones de instrucciones.
- Si el repertorio de instrucciones contiene instrucciones que tardan en ejecutarse tiempos diferentes, para que el programa tardase el mismo tiempo es preciso que se ejecutase el mismo número de instrucciones y del mismo tipo (en cuanto a tiempo de ejecución de cada una). En este caso, también, la respuesta es NO.

◀

Cuestión 7. ¿Aceptaría financiar/embarcarse en un proyecto en el que se plantease el diseño e implementación de un computador de propósito general con arquitectura MISD? (Justifique su respuesta).

Solución: El tipo de procesamiento que realiza un procesador MISD puede implementarse en un procesador MIMD con la sincronización correspondiente entre los procesadores del computador MIMD para que los datos vayan pasando adecuadamente de un procesador a otro (definiendo un flujo único de datos que pasan de procesador a procesador). Por esta razón un computador MISD **no tiene mucho sentido** como computador de propósito general. Sin embargo, puede ser útil un diseño MISD para un dispositivo de propósito específico que sólo tiene que ejecutar una aplicación que implica un procesamiento en el que los datos tienen que pasar por distintas etapas en las que sufren ciertas transformaciones que pueden programarse (en cada uno de los procesadores). La especificidad del diseño puede permitir generar diseños muy eficientes desde el punto de vista del consumo, de la complejidad hardware, etc.

◀

Cuestión 8. Deduzca la ley de Amdahl para la mejora de la velocidad de un procesador suponiendo que hay una probabilidad f de no utilizar un recurso del procesador cuya velocidad se incrementa en un factor p.

Solución: El tiempo necesario para ejecutar un programa en el procesador sin la mejora es igual a:

$$T_{sin\_mejora}=f\times T_{sin\_mejora}+(1-f)\times T_{sin\_mejora}$$

donde $f*T_{sin\_mejora}$ es el tiempo que no podría reducirse al aplicar la mejora, y $(1-f)*T_{sin\_mejora}$ es el tiempo que podría reducirse como máximo en un factor igual a p al aplicar la mejora. Por tanto, el tiempo al aplicar la mejora sería:

$$T_{con\_mejora} \geq f\times T_{sin\_mejora}+((1-f)\times T_{sin\_mejora})/p$$

El signo “>” se usa porque al aplicar la mejora es posible añadir algún tiempo de sobrecarga extra. Según esto, la ganancia de velocidad que se podría conseguir sería:

$$S = T_{sin\_mejora}/T_{con\_mejora} \leq T_{sin\_mejora}/(f*T_{sin\_mejora}+((1-f)\times T_{sin\_mejora})/p) = 1/(f+(1-f)/p) = p/(1+f(p-1))$$

y así llegamos a la expresión que se utiliza para describir la Ley de Amdahl: $\mathbf{S \leq p/(1+f(p-1))}$

◀

Cuestión 9. ¿Es cierto que para una determinada mejora realizada en un recurso se observa experimentalmente que, al aumentar el factor de mejora, llega un momento en que se satura el incremento de velocidad que se consigue? (Justifique la respuesta)

Solución: Para responder a esta pregunta recurrimos a la ley de Amdahl, que marca un límite superior a la mejora de velocidad. Como se tiene que S ≤ p/(1+f(p-1)), donde p es el factor de mejora y f la fracción de tiempo sin la mejora en el que dicha mejora no puede aplicarse, se puede calcular la derivada de p/(1+f(p-1)) con respecto a p, y ver qué pasa cuando cambia p. Así:

$$d(p/(1+f(p-1))/dp = ((1+f(p-1))-pf)/(1+f(p-1))^2 = (1-f)/(1+f(p-1))^2$$

y como puede verse, a medida que aumenta p el denominador se va haciendo mayor, y como (1-f) se mantiene fijo, la derivada tiende a cero. Eso significa que la pendiente de la curva que describe la variación de p/(1+f(p-1)) con p va haciéndose igual a cero y por lo tanto no aumenta la ganancia de velocidad: se satura, o lo que es lo mismo, llega a una asíntota que estará situada en 1/f (que es al valor al que tiende p/(1+f(p-1) cuando p tiende a infinito).

◀

Cuestión 10. ¿Es cierto que la cota para el incremento de velocidad que establece la ley de Amdahl crece a medida que aumenta el valor del factor de mejora aplicado al recurso? (Justifique la respuesta).

Solución: La respuesta a esta pregunta se deduce de la misma expresión de la derivada de la cota que establece la ley de Amdahl, calculada al responder la cuestión 4:

$$d(p/(1+f(p-1))/dp = ((1+f(p-1))-pf)/(1+f(p-1))^2 = (1-f)/(1+f(p-1))^2$$

dado que (1-f) es positivo (solo en el peor de los casos, si no se pudiera aplicar el factor de ganancia a ninguna parte del programa, tendríamos que (1-f)=0 y no se obtendría ninguna ganancia de velocidad), y puesto que $(1+f(p-1))^2$ siempre es positivo, la derivada es positiva. Eso significa que 1+f(p-1) crece al crecer p. Lo que ocurre es que, tal y como se ha visto en la cuestión 4, este crecimiento es cada vez menor (tiende a 0).

◀

Cuestión 11. ¿Qué es mejor, un procesador superescalar capaz de emitir cuatro instrucciones por ciclo, o un procesador vectorial cuyo repertorio permite codificar 8 operaciones por instrucción y emite una instrucción por ciclo? (Justifique su respuesta).

Solución. Considérese una aplicación que se codifica con N instrucciones de un repertorio de instrucciones escalar (cada instrucción codifica una operación). La cota inferior para el tiempo que un procesador superescalar podría tardar en ejecutar ese programa sería

$$T_{superescalar} = (N\times T_{ciclo\_superesc})/4$$

Un procesador vectorial ejecutaría un número de instrucciones menor puesto que su repertorio de instrucciones permitiría codificar hasta ocho operaciones iguales (sobre datos diferentes) en una instrucción. Si suponemos que la aplicación permite que todas las operaciones que hay que ejecutar se puedan "empaquetar" en instrucciones vectoriales, el procesador vectorial tendría que ejecutar N/8 instrucciones.

Por tanto, considerando también que el procesador vectorial consigue terminar instrucciones según su velocidad pico (como supusimos en el caso del superescalar), la cota inferior para el tiempo que tarda la ejecución del programa en el procesador vectorial sería:

$$T_{vectorial} = (N/8) \times T_{ciclo\_vectorial}/1$$

Por tanto

$$T_{superescalar} / T_{vectorial} = 8 \times T_{ciclo\_superesc}/4 \times T_{ciclo\_vectorial} = 2 \times T_{ciclo\_superesc}/T_{ciclo\_vectorial}$$

Según esto, si los dos procesadores funcionan a la misma frecuencia, el procesador vectorial podría ser mejor (siempre y cuando las hipótesis que se están considerando de que están procesando instrucciones según su máximo rendimiento se puedan considerar suficientemente aproximada a la realidad).

No obstante, hay que tener en cuenta que usualmente, los procesadores superescalares han sido los que más rápidamente han incorporado las mejoras tecnológicas, y normalmente, los procesadores superescalares funcionan a frecuencias más altas que los procesadores vectoriales contemporáneos. En el ejemplo, si la frecuencia del superescalar fuera más del doble de la del vectorial, sería mejor utilizar un procesador superescalar.

Además, hay que tener en cuenta que para que los procesadores vectoriales sean eficientes (se puedan admitir las hipótesis en cuanto a codificación de las operaciones y funcionamiento según la velocidad pico) es necesario que el programa tenga un alto grado de paralelismo de datos (son procesadores más específicos que los superescalares). Por eso, para dar una respuesta más precisa habría que conocer más detalles de la carga de trabajo que se piensa ejecutar en los procesadores.

Cuestión 12. En la Lección 2 de AC se han presentado diferentes criterios de clasificación de computadores y en el Seminario 0 de prácticas se ha presentado atcgrid. Clasifique atcgrid, 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. Razone su respuesta.

Solución

Los dos nodos de atcgrid, atcgrid1 y atcgrid2, están conectados entre sí mediante un conmutador Ethernet, esto supone que por hardware un nodo no puede acceder a la memoria que se encuentra en otros nodos. El software del sistema de comunicación ofrece funciones para copiar datos de la memoria de un nodo a la memoria de otro nodo. Por tanto, en el nivel de empaquetamiento/conexión de sistema, atcgrid es una arquitectura MIMD (clasificación de Flynn) y, en particular, un multicomputador (clasificación del sistema de memoria) o arquitectura NORMA.

Los dos nodos de atcgrid tienen una placa con dos socket (encapsulados) que comparten memoria, aunque cada socket tiene acceso mediante un enlace (conexión punto-a-punto dedicada exclusivamente a la comunicación entre esos puntos) a un conjunto de direcciones de memoria del sistema de memoria compartido. Por tanto, en el nivel de placa, atcgrid es una arquitectura MIMD (clasificación de Flynn) y, en particular, un multiprocesador CC-NUMA (clasificación del sistema de memoria). Es NUMA por tener el espacio de memoria compartido físicamente distribuido entre los sockets (encapsulados), lo que hace que un núcleo tarde menos en acceder a la memoria que está directamente conectada con un enlace al encapsulado en el que se encuentra que a la memoria conectada al otro encapsulado de la placa. Además el hardware mantiene coherencia entre las caches de último nivel de los encapsulados, de ahí que aparezca CC (*Cache-Coherent*) en el nombre.

Los encapsulados de atcgrid constan de un único dado de silicio con 6 cores/núcleos que comparten el último nivel de cache. Por tanto, en el nivel de encapsulado, atcgrid es una arquitectura MIMD (clasificación

de Flynn) y, en particular, un multiprocesador UMA o SMP en un chip (clasificación del sistema de memoria), también llamado en la bibliografía científica CMP (*Chip MultiProcessor*). Es UMA porque la latencia de acceso al último nivel de memoria cache del encapsulado es igual para todos los cores.

Los cores/núcleos de atcgrid tiene repertorio de instrucciones vectoriales porque tiene unidades funcionales multimedia que permiten ejecutar en paralelo múltiples operaciones escalares (suma, and, multiplicación, etc.). Las unidades funcionales multimedia implementan una arquitectura SIMD (la memoria en la que están los datos con los que se opera es un registro). Además, cada núcleo de atcgrid ejecuta dos flujos de control en paralelo (dos threads del SO en paralelo) debido a la implementación de multithread simultáneo (*HyperThreading*). Desde el punto de vista del SO un núcleo de atcgrid son dos procesadores independientes (es como si el núcleo tuviera dos procesadores lógicos). Por tanto, desde el punto de vista del SO un núcleo tiene dos procesadores de una arquitectura MIMD, aunque en realidad físicamente los dos flujos de control de un núcleo comparten una unidad de control y una unidad de procesamiento (como se verá en el Tema 3).

Cuestión 13. En la Lección 1 de AC se han presentado diferentes criterios de clasificación del paralelismo implícito en una aplicación y en el Seminario 0 de prácticas se ha presentado atcgrid. ¿Qué tipos de paralelismo aprovecha atcgrid? Razone su respuesta.

Solución

- Paralelismo a nivel de operación:
  - Al ser una arquitectura con paralelismo a nivel de instrucción, por tener una arquitectura segmentada y superescalar, puede ejecutar operaciones en paralelo, luego puede aprovechar el paralelismo a nivel de operación.
- Paralelismo a nivel de bucle (paralelismo de datos):
  - Las operaciones realizadas en las iteraciones del bucle sobre vectores y matrices se podrían implementar en las unidades SIMD de los cores de atcgrid, lo que reduciría el número de iteraciones de los bucles del código máquina.
  - Las iteraciones de los bucles se podrían repartir entre threads del SO. Los cores/nucleos de atcgrid pueden ejecutar dos threads del SO en paralelo debido a la implementación de multithread simultánea (*HyperThreading*). Los threads de un mismo código también se pueden repartir entre los 6 cores de un encapsulado de atcgrid (6 cores * 2 threads/core = 12 threads) y entre los cores de los dos encapsulados de una placa atcgrid (2 encapsulados * 6 cores/encapsulado * 2 threads/core * = 24 threads). Por tanto, las iteraciones de un bucle se podrían repartir entre 24 threads del SO que se pueden ejecutar en paralelo en un nodo de atcgrid.
  - Si las iteraciones de los bucles se reparten además entre procesos del SO se podrían aprovechar los cores de los dos nodos de atcgrid (2 nodos * 2 encapsulados/nodo * 6 cores/encapsulado * 2 threads/core * = 48 threads repartidos en dos procesos). En total se podrían ejecutar en paralelo 48 flujos de control repartidos entre un mínimo de dos procesos (uno por nodo).
- Paralelismo a nivel de función (paralelismo de tareas):
  - Las funciones independientes se podrían ejecutar en paralelo usando los dos threads de los cores, usando todos los cores de los dos encapsulados (24 flujos de control en paralelo como máximo), e incluso, si el paralelismo se hace explícito a nivel de proceso, se podrían usar los dos nodos (48 flujos de control en paralelo como máximo).

- Paralelismo a nivel de programa:
  - Los programas del mismo o distinto usuario se pueden asignar a distintos cores de atcgrid estén o no en el mismo nodo para así ejecutarlos al mismo tiempo (48 flujos de control en paralelo como máximo en atcgrid). El paralelismo a nivel de programa sólo se puede hacer explícito a nivel de proceso del SO.