Contents

1 Arquitectura de Computadoras - 2024

1.1 Informe Lab2: Análisis de microarquitecturas

1.2 Integrantes:

- Lautaro Bachmann
- Fabricio Longhi
- Valentino Mensio

2 Ejercicio 1

2.1 Punto b y c

Simulaciones del codigo con distitnas configuraciones de cache.

- **Tamaño:** 8KB, 16KB, 32KB
- **Asociatividad:** 1, 2, 4, y 8 vías

2.1.1 Número total de ciclos

Tamaño cache	1 via	2 vias	4 vias	8 vias
8KB	218368	189262	243159	243023
16KB	218412	188834	243071	243071
32KB	218432	188757	243071	243071

2.1.2 Ciclos ociosos

Tamaño cache	1 via	2 vias	4 vias	8 vias
8KB	129896	115878	184406	184271
16KB	129951	115413	184315	184315
32KB	129971	115292	184315	184315

2.1.3 Hits totales en la caché de datos

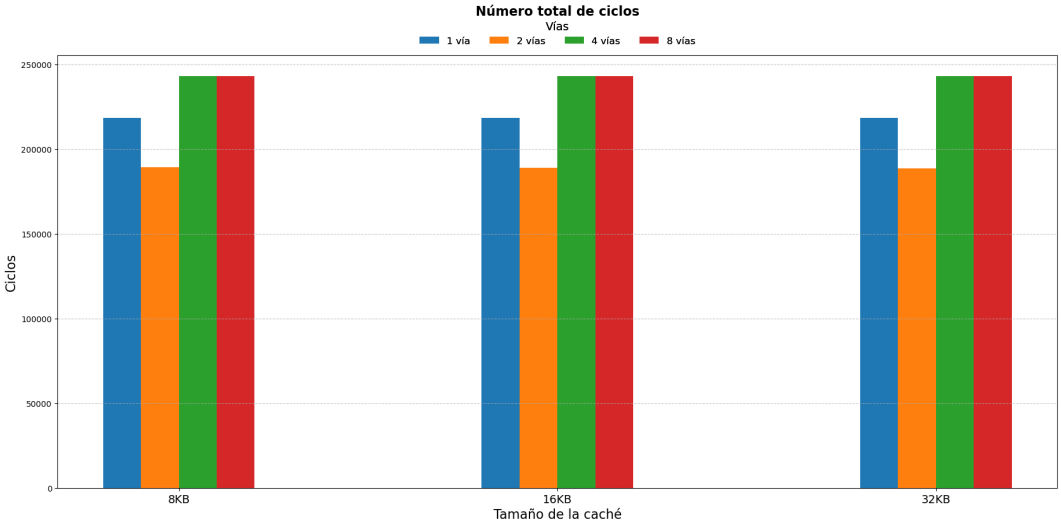

Figure 1: Imagen

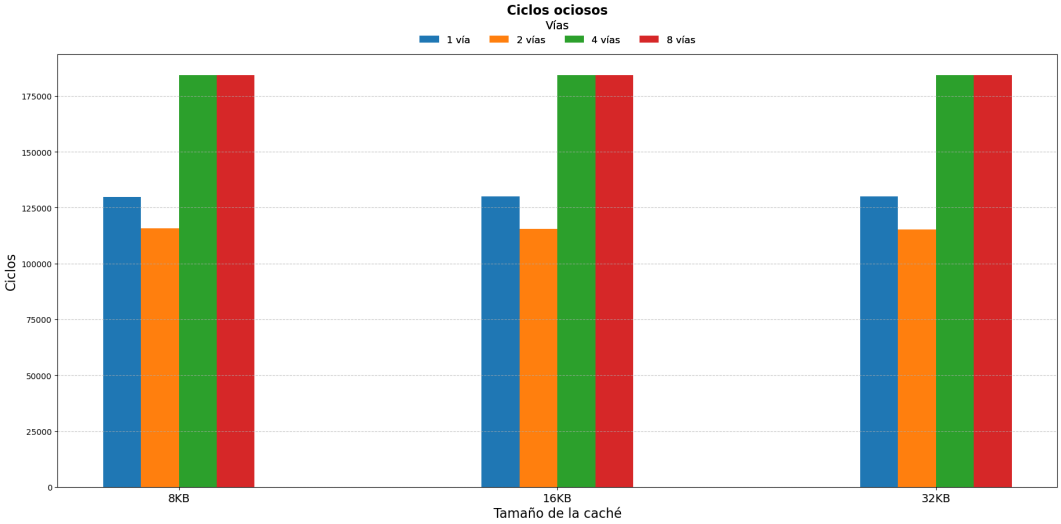

Figure 2: Imagen

Tamaño cache	1 via	2 vias	4 vias	8 vias
8KB	698	4781	10755	10755
16KB	704	4782	10755	10755
32KB	704	4769	10755	10755

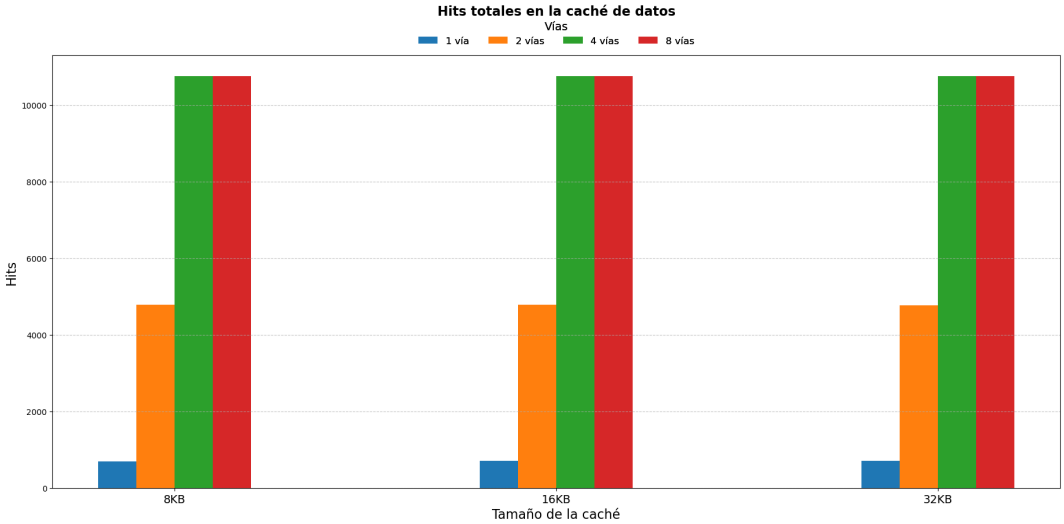

Figure 3: Imagen

2.1.4 Hits de lectura en la caché de datos

Tamaño cache	1 via	2 vias	4 vias	8 vias
8KB	470	4714	7685	7685
16KB	474	4706	7685	7685
32KB	474	4694	7685	7685

2.2 Punto d

2.2.1 Análisis de resultados

1. **Número total de ciclos**:
    - Las configuraciones con menor cantidad de ciclos totales son aquellas con 2 vias. Esto nos indica que estas configuraciones logran un equilibrio entre el

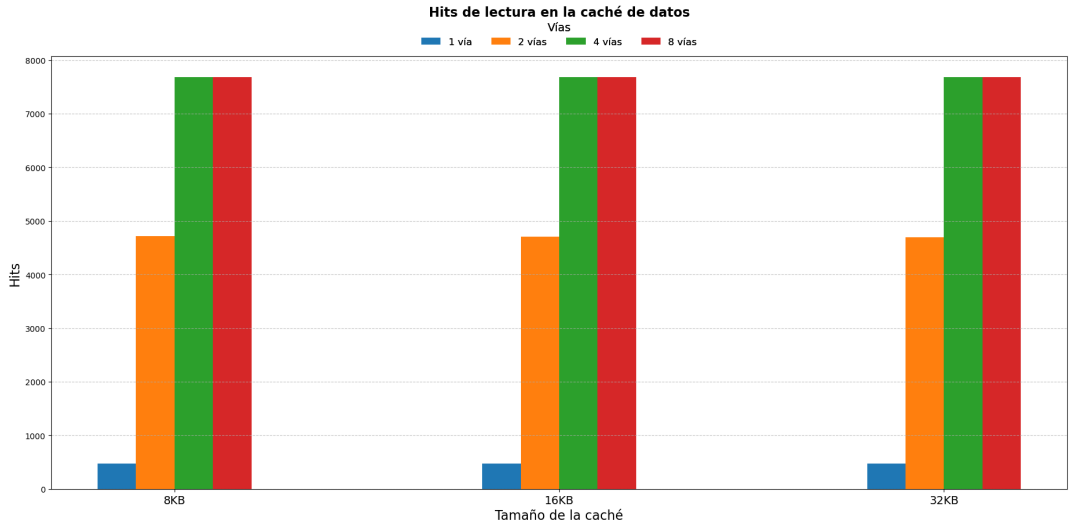

Figure 4: Imagen

tamaño de la cache y la asociatividad. Por el contrario las configuraciones con 1, 4 y 8 vias presentan un aumento significativo, esto puede deberse al costo adicional asociado a la busqueda dentro de un mayor cantidad de conjutos.

2. **Ciclos ociosos**:
   - Sigue una tendencia similar a los ciclos totales, pero siendo la mejor opción la configuración con 32KB y 2 vías, que refleja una mejor utilización del CPU en comparación a 8KB y 16KB con 1, 4 y 8 vías.
3. **Hits totales en la caché de datos**:
   - Podemos observar que la cantidad de hits totales aumenta para configuraciones de 4 y 8 vías. Pero debemos analizar la eficiencia de hits con respecto a los ciclos totales, su impacto en el rendimiento global se diluye debido al mayor costo de ciclos.
   - La configuración con 2 vías y 32KB tiene un rendimiento aceptable logrando un buen equilibrio entre eficiencia y costo.
4. **Hits de lectura en la caché de datos**:
   - Existe una similitud entre los hits de lectura y hits totales para la configuración de 2 vías, esto es debido a que en el programa la mayoría de los accesos son lecturas, y los datos se cargan eficientemente gracias al acceso secuencial y a la política de prefetching. Con 2 vías, la caché tiene suficiente capacidad para evitar conflictos significativos entre los arrays, maximizando los hits de lectura.

5. **Impacto del código ejecutado**:
    - El bucle realiza accesos frecuentes a memoria para leer y escribir datos (carga de arreglos X e Y, almacenamiento en Z), lo cual depende en gran medida del desempeño de la caché.
    - La configuración de 2 vías permite manejar estos accesos de forma eficiente sin incurrir en demasiados ciclos ociosos.

2.2.2 Conclusión Final

La configuración **32KB y 2 vías** es la más eficiente en términos globales, logrando un buen balance entre:

- **Número total de ciclos (188757, el más bajo entre todas las configuraciones).**
- **Reducción de ciclos ociosos (115292, también el menor entre todas las configuraciones).**
- **Número adecuado de hits en la caché sin comprometer el rendimiento global (4769 hits totales).**

Aunque configuraciones con mayor asociatividad (4 y 8 vías) muestran un número mayor de hits, el costo en términos de ciclos totales y complejidad no justifica su uso para este caso. Por tanto, **32KB y 2 vías** es la mejor opción para este escenario.

2.3 Punto e

2.3.1 Modificaciones realizadas

- Hacer loop unroling 4 veces
- Agrupar instrucciones que acceden a direcciones de memoria contiguas
- Separar los FMUL de los FADD para evitar dependencias de datos

2.3.2 Codigo optimizado

```
LDUR    X5, [X10, #0]   // alpha
SCVTF   D30, X5         // alpha to float
ADD     X1, XZR, XZR    // i = 0
for:
    // Load 4 elements from X
    LDUR  D0, [X2, #0]   // D0 = X[i]
    LDUR  D1, [X2, #8]   // D1 = X[i + 1]
    LDUR  D2, [X2, #16]  // D2 = X[i + 2]
    LDUR  D3, [X2, #24]  // D3 = X[i + 3]
```

```
// Load 4 elements from Y
LDUR  D4, [X3, #0]   // D4 = Y[i]
LDUR  D5, [X3, #8]   // D5 = Y[i + 1]
LDUR  D6, [X3, #16]  // D6 = Y[i + 2]
LDUR  D7, [X3, #24]  // D7 = Y[i + 3]
// Compute (alpha * X[i]) + Y[i] for 4 elements
FMUL  D8, D0, D30    // D8 = X[i] * alpha
FMUL  D9, D1, D30    // D9 = X[i + 1] * alpha
FMUL  D10, D2, D30   // D10 = X[i + 2] * alpha
FMUL  D11, D3, D30   // D11 = X[i + 3] * alpha

FADD  D8, D8, D4     // D8 = (X[i] * alpha) + Y[i]
FADD  D9, D9, D5     // D9 = (X[i + 1] * alpha) + Y[i + 1]
FADD  D10, D10, D6   // D10 = (X[i + 2] * alpha) + Y[i + 2]
FADD  D11, D11, D7   // D11 = (X[i + 3] * alpha) + Y[i + 3]

// Store results in Z
STUR  D8, [X4, #0]   // Z[i] = (alpha * X[i]) + Y[i]
STUR  D9, [X4, #8]   // Z[i + 1] = (alpha * X[i + 1]) + Y[i + 1]
STUR  D10, [X4, #16] // Z[i + 2] = (alpha * X[i + 2]) + Y[i + 2]
STUR  D11, [X4, #24] // Z[i + 3] = (alpha * X[i + 3]) + Y[i + 3]
// Update pointers and loop counter
ADD  X2, X2, #32     // Move to the next 4 elements in X
ADD  X3, X3, #32     // Move to the next 4 elements in Y
ADD  X4, X4, #32     // Move to the next 4 elements in Z
SUB  X0, X0, #4      // Decrement loop counter by 4
CBNZ X0, for         // if (i < N): loop
```

2.3.3 Resultados

2.3.4 Nomeclatura

Primero, definamos lo siguiente: - Caso A: Codigo optimizado ej1e, 32KB de cache y 1 via - Caso B: Codigo sin optimizar ej1a, 32KB de cache y 2 vias

2.3.5 Proceso de desarrollo

Primeramente intentamos hacer un loop unroling 2 veces, y sin agrupar instrucciones, lo cual nos dio un resultado peor que usando que el caso B.

Luego se nos ocurrio incrementar el loop unroling a 4 y agrupar instrucciones que acceden a direcciones contiguas de cache para maximizar cache hits.

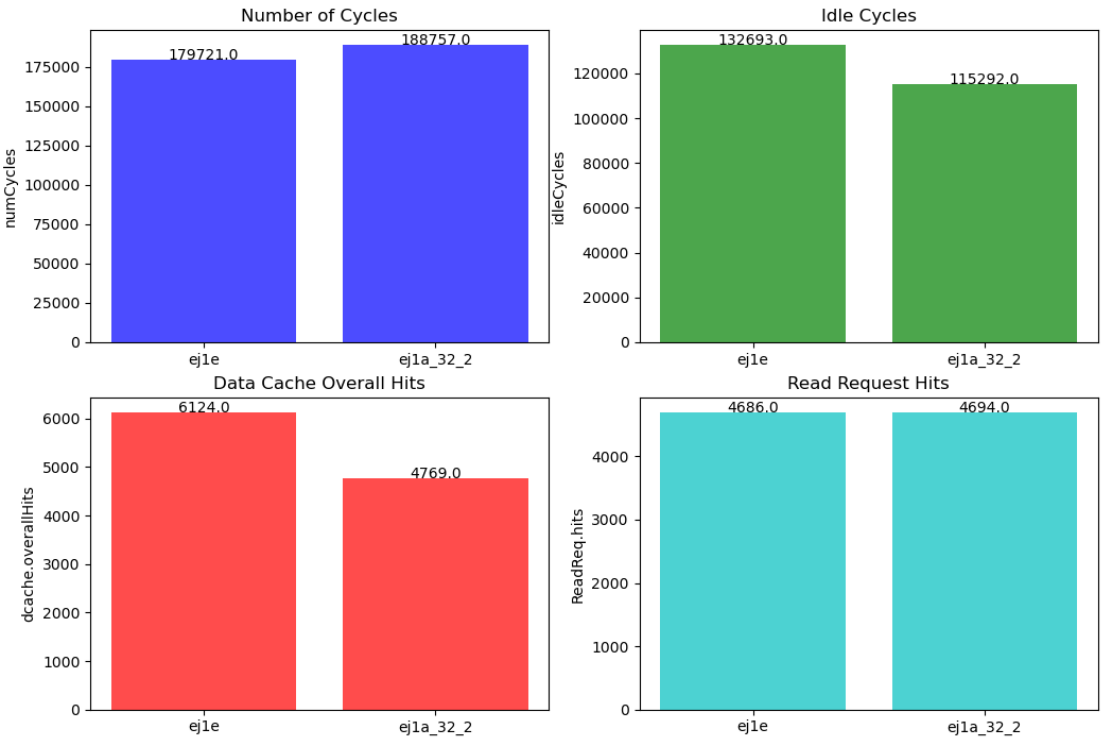

Figure 5: ej1a vs ej1e

2.3.6 Analisis

Gracias a esto, como podemos ver en el grafico, el caso A obtuvo una cantidad de ciclos menor al codigo sin optimizar caso B, de la misma manera que obtuvo muchos más hits de cache. Sin embargo, el caso A obtuvo muchos más idle cycles que el caso B.

2.4 Punto f

2.4.1 Resultados

Estadistica	Ejecución in order	Ejecución out of order
Número de ciclos de clock	173700	36992
Número de ciclos de clock ociosos	137467	225
Número de aciertos en caché de datos	6127	2756
Número total de aciertos en solicitudes de lectura a la caché de datos	4780	2235

2.4.2 Analisis

La ejecución out-of-order redujo en gran medida el número de ciclos de clock totales y la cantidad de ciclos ociosos, reflejando la capacidad de reordenar las instrucciones dinámicamente y evitar cuellos de botellas causados por dependencias de datos.

También podemos observar que aunque disminuya la cantidad de hits totales esto no afecta negativamente al rendimiento. Esto se debe a que la ejecución out-of-order maneja los accesos a memoria de forma más eficiente, reduciendo solicutudes redundantes y ocultando las latencias al paralizar operaciones independientes.

En conclusión las optimizaciones en el código (loop unrolling y la agrupacion por instrucciones), son muy efectivas para mejorar el rendimiento en arquitecturas out-of-order gracias a su capacidad de reorganizar instrucciones y paralelizar operaciones.

3 Ejercicio 2

3.1 Puntos b y c

Se considera una cache de 32 KB. Se realizaron simulaciones con distintas vias, donde obtenemos numeros de ciclos es distintos:

Numero vias	Numero de ciclos de clock
1	6208060
2	4158442
4	4156974
8	4158442

Analizando los datos obtenidos, podemos ver una mejora de desempeño cuando la simulacion utiliza 2 vias o mas. Como podemos observar, con una sola via es el peor de todos los casos, ya que en esta situacion tendremos mucha sobrescritura. Cuando tenemos 2 vias o mas obtenemos un mejor rendimiento, esto se debe a que cuando tenemos mas de dos vias, podremos guardar un arreglo por via. Con esto, se evitan sobrescrituras de elementos que comparten la misma ubicacion de cache.

Hay una diferencia de 1468 ciclos entre la simulacion con 4 vias y las de 2 y 8, esto representa un aumento de eficiencia del .035%.

Por simplicidad, como los resultados fueron practicamente identicos para los procesadores con 2 vias o más, tomaremos al procesador de 2 vias como el que dio el mejor resultado.

3.1.1 Grafico

3.2 Punto d

3.2.1 Analisis del codigo y prediccion

3.2.1.1 Bucles

- Para los bucles (`loop_k`, `loop_i`, `loop_j`), el predictor local deberia funcionar bien

3.2.1.2 Ifs

- Para los “ifs”, un predictor global va a funcionar mejor, ya eventualmente va a encontrar un patron para los saltos

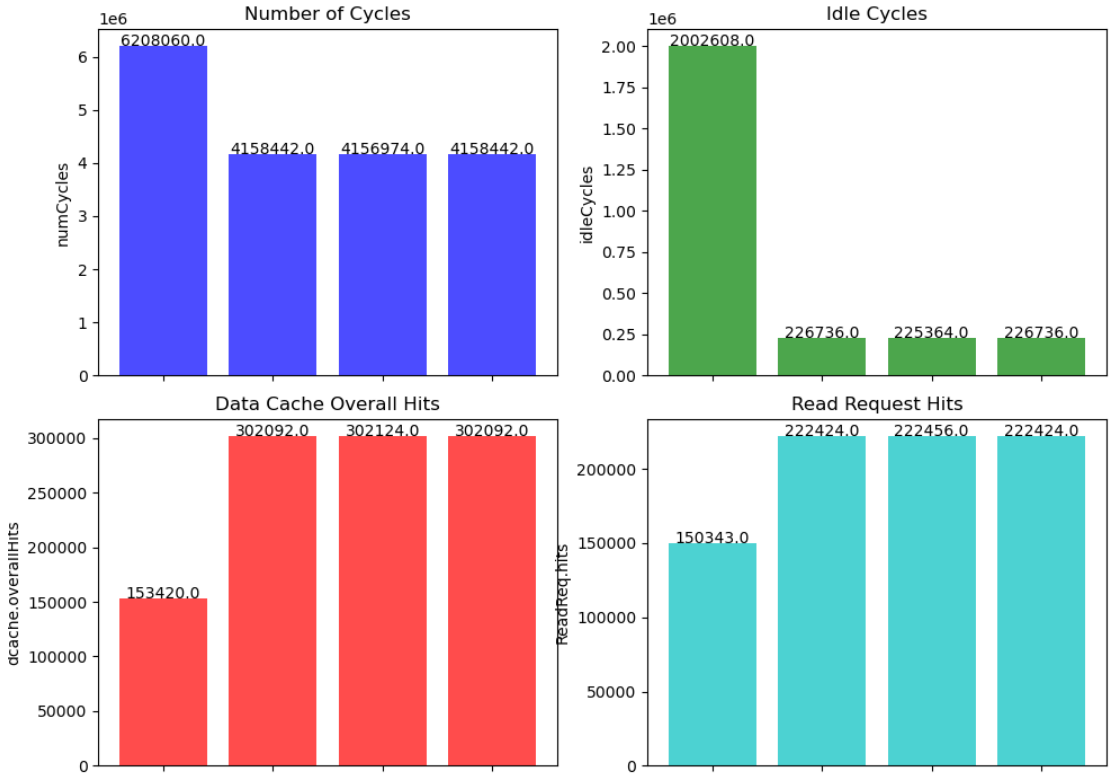

Figure 6: ej2_b_y_c

3.2.1.3 Predictor global

Creemos que el rendimiento podria mejorar considerablemente al usar el predictor por torneo, ya que este se va adaptando al predictor que mejor resultado da.

3.2.2 Resultados simulacion

Cache utilizada: 32KB, 2 vias

predictor	condIncorrect	condPredicted	missRate
local	2285	379072	0.006
tourneo	842	375531	0.002

Como podemos observar, hubo una mejora del 300% al pasar del predictor local al predictor por torneo, lo cual se alinea con nuestra prediccion de que el predictor por torneos iba a tener un desempeño mejor.

Justamente, como explicamos antes, el predictor por torneo tuvo un mejor desempeño por su capacidad de adaptarse a los distintos tipos de salto y el hecho de que haya dado un resultado mejor se alinea con nuestras expectativas.

3.3 Punto e

Tipo	predictor	condIncorrect	condPredicted	missRate
in_order	local	2285	379072	0.006
in_order	tourneo	842	375531	0.002
out_of_order	torneo	841	375398	0.002

Como podemos ver, el desempeño es practicamente el mismo entre un procesador out_of_order y uno in_order al utilizar un predictor por torneo.