

Lab 2 | Ejercicio 1 (Informe)

Integrantes:

- Emanuel Nicolás Herrador (DNI 44.898.601)
- Juan Bratti (DNI 44.274.011)

Materia: Arquitectura de Computadoras 2023

Introducción

(b, c) Datos obtenidos para *daxpy* no optimizado con distintos tipos de caché

(d) Análisis de los resultados para *daxpy* no optimizado

Gráficos de los datos obtenidos para distintas cantidades de vías

Comportamiento esperado

Análisis de resultados obtenidos

(e) Optimización de la *benchmark* - *daxpy optimized*

Optimizaciones consideradas

Análisis de los resultados de la optimización para dcache de 1 vía y 32kB de dato para cada tipo de *loop unrolling*

Comparación de resultados entre *daxpy* optimizado vs. no optimizado (para 1 vía - 32kB de dato)

(f) Simulación de la dos *benchmarks* (*daxpy* optimizada con *loop unrolling* 2 vs. no optimizada) para procesador *out of order*

Resultados obtenidos para dcaché de 1 vía y 32kB de dato

Análisis de los resultados y comparativa

Introducción

En este presente informe se presentarán los resultados obtenidos para los distintos tipos de procesadores a considerar (*in order* y *out of order*), respecto a varios tipos de caché de dato principal (*dcache*, cambiando tamaño y cantidad de vías), aplicándose la benchmark *daxpy* explicitada en el enunciado del ejercicio.

Además, se consideran las métricas para el caso de optimización por “compilador” (en nuestro caso, manual) de la benchmark, usando técnicas de mejora como operadores ternarios y *loop unrolling* de 2 iteraciones.

Además de todos estos datos, se realizará su respectivo análisis y justificación para entender por qué funciona de ese modo y cuál es la mejor combinación que podemos considerar para esta *benchmark*.

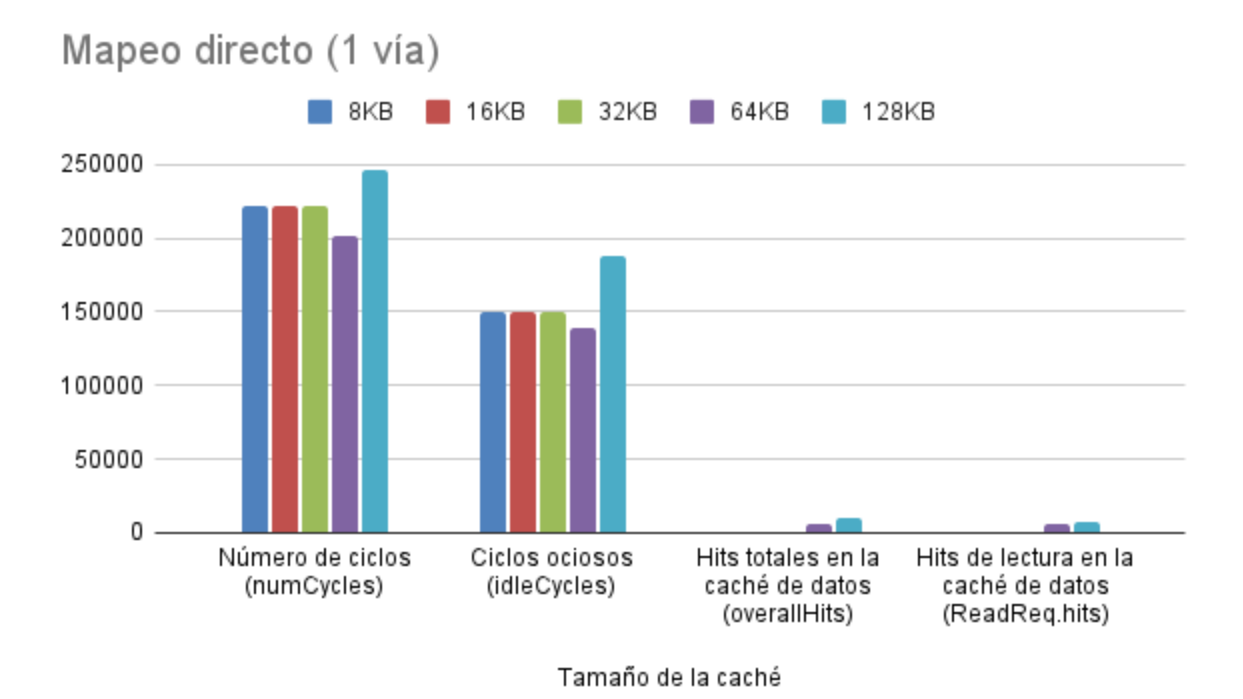
Respecto a este análisis, agregaremos las diferencias y *fallas* que consideramos que tuvo el *simulador* utilizado (principalmente frente al número de ciclos para cachés con mayor tamaño de dato y más vías).

(b, c) Datos obtenidos para *daxpy* no optimizado con distintos tipos de caché

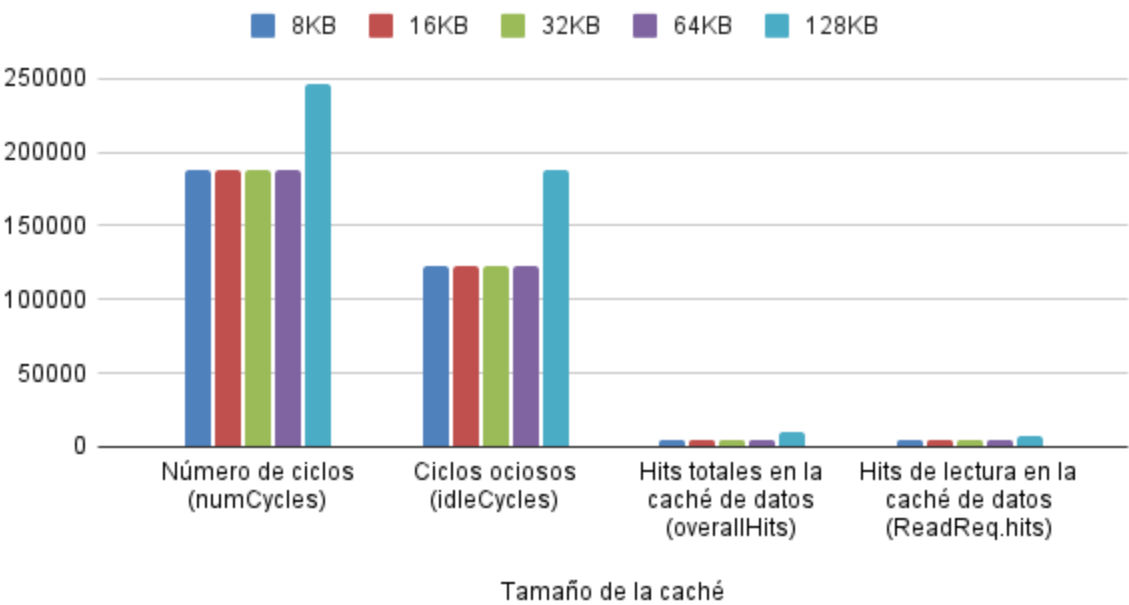
Tipo de caché	Mapeo directo (1 vía)				
Tamaño de la caché	8KB	16KB	32KB	64KB	128KB
Número de ciclos (numCycles)	221817	221861	221597	201184	246389
Ciclos ociosos (idleCycles)	149765	149809	149536	139169	187760
Hits totales en la caché de datos (overallHits)	800	800	804	6112	10761
Hits de lectura en la caché de datos (ReadReq.hits)	480	480	478	5921	7689
Tipo de caché	Asociativa por conjunto de 2 vías				
Tamaño de la caché	8KB	16KB	32KB	64KB	128KB
Número de ciclos (numCycles)	187634	187690	187690	187656	246389
Ciclos ociosos (idleCycles)	122320	122258	122258	122251	187760
Hits totales en la caché de datos (overallHits)	5335	5290	5290	5309	10761
Hits de lectura en la caché de datos (ReadReq.hits)	5189	5157	5157	5168	7689
Tipo de caché	Asociativa por conjunto de 4 vías				
Tamaño de la caché	8KB	16KB	32KB	64KB	128KB
Número de ciclos (numCycles)	246655	246407	246407	246407	246389
Ciclos ociosos (idleCycles)	188018	187771	187771	187771	187760
Hits totales en la caché de datos (overallHits)	10756	10756	10756	10756	10761
Hits de lectura en la caché de datos (ReadReq.hits)	7686	7686	7686	7686	7689
Tipo de caché	Asociativa por conjunto de 8 vías				
Tamaño de la caché	8KB	16KB	32KB	64KB	128KB
Número de ciclos (numCycles)	246375	246407	246407	246407	246389
Ciclos ociosos (idleCycles)	187739	187771	187771	187771	187760
Hits totales en la caché de datos (overallHits)	10756	10756	10756	10756	10761
Hits de lectura en la caché de datos (ReadReq.hits)	7686	7686	7686	7686	7689

(d) Análisis de los resultados para *daxpy* no optimizado

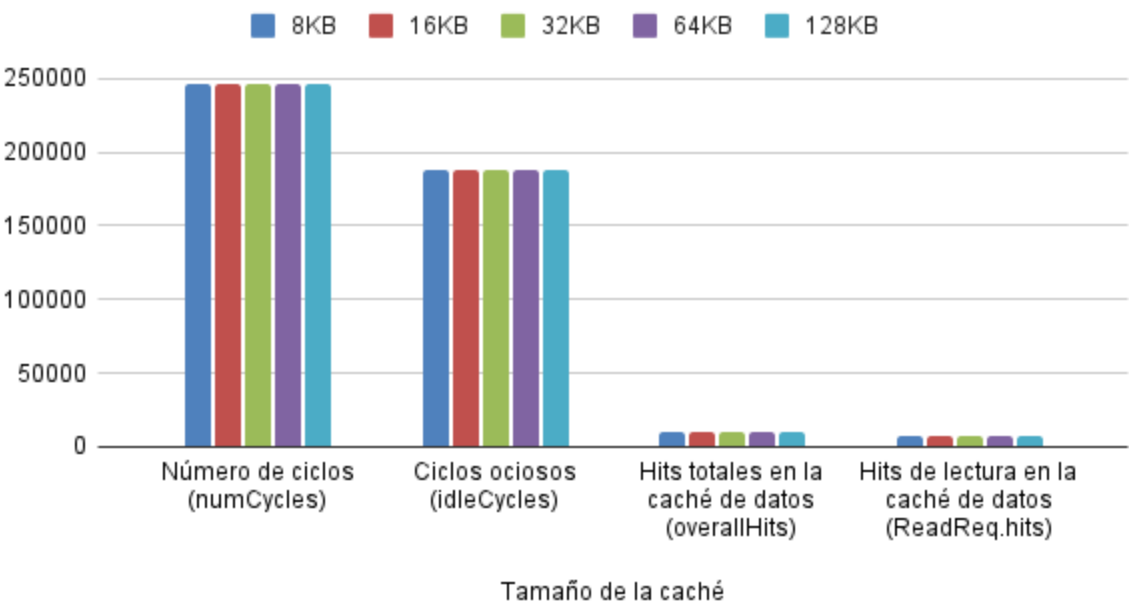
Gráficos de los datos obtenidos para distintas cantidades de vías



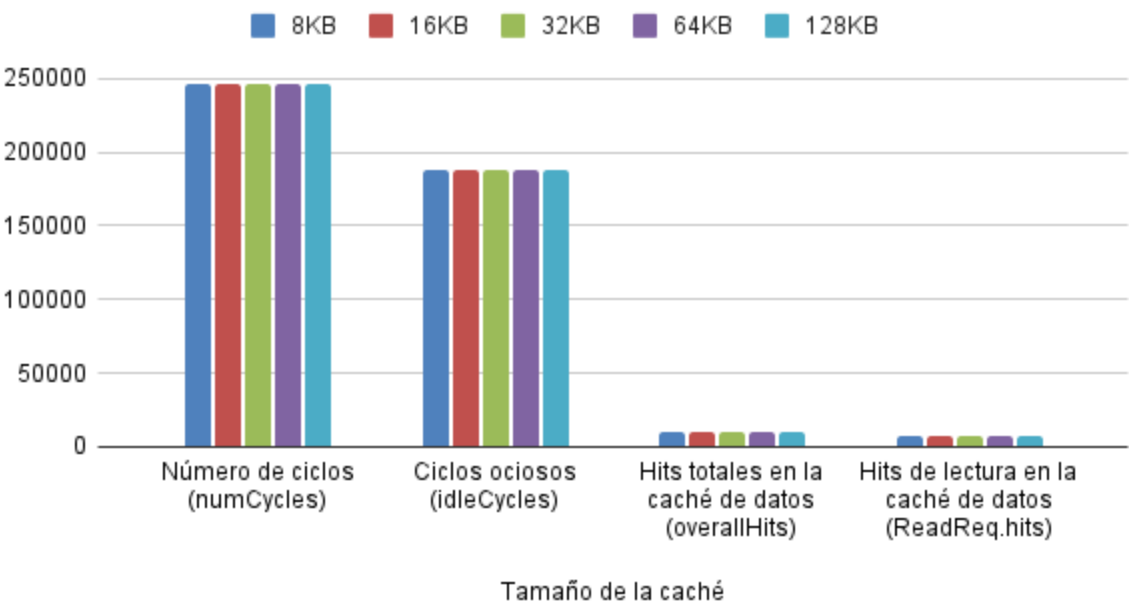
Asociativa por conjunto de 2 vías



Asociativa por conjunto de 4 vías



Asociativa por conjunto de 8 vías



Comportamiento esperado

En esta sección, veremos cuál es el comportamiento *teórico* que debería tener el procesador para la ejecución de esta benchmark para cada tipo de *dcaché* considerada. Para ello, es **muy importante** que tengamos en cuenta los siguientes datos:

- $N = 4096 = 2^{12}$
- Los números *double* son de 8 bytes (64 bits)
- Los arreglos que se considerarán (X, Y, Z) son de $2^{12} \times 8B = 2^{15}B = 2^5kB = 32kB$
 - Luego, entonces, los tres arreglos ocupan $96kB$
- Los arreglos se encuentran en el siguiente orden: X[0], ..., X[4095], Y[0], ..., Y[4095], Z[0], ..., Z[4095]



Y como parámetro super importante para el análisis, vamos a ver que, si probamos el simulador observando los *misses* de la *dcache* para varios ldr consecutivos, la *cantidad de palabras por bloque* de la caché principal de datos es de 8 double words.

Teniendo esto en cuenta, entonces, podemos observar claramente que si leemos solo el vector X, entonces tendríamos un total de 512 misses ($\frac{4096}{8}$). Sin embargo, la *benchmark* que consideramos nos obliga a leer X e Y, y a escribir Z, pisando así valores de la caché y aumentando, de ese modo, la cantidad de *overallMisses* que vamos a tener. Como consecuencia, entonces, esto aumenta la cantidad de *ciclos ociosos* y, por ende, la cantidad total de ciclos que tarda la ejecución (dado que tiene que ir a buscar el bloque entero a la memoria principal).

Como bien sabemos, la caché al ser una memoria de *alto nivel* (más cercana al procesador), son *muy rápidas* pero *muy chicas*. Por ello, mientras más grande sea la

caché, mejor suele ser la performance de nuestro procesador dado que no va a requerir irse tanto a buscar bloques de la memoria principal. Para agrandarla hay dos opciones que consideramos:

- Aumentar capacidad de datos de la caché
- Aumentar cantidad de vías consideradas

Siempre, mientras más altos sean estos valores, la performance va a ser mejor, pero se llega a un punto de “estancamiento” donde se *clava* (por decirlo de alguna forma) y agregar más vías o agrandar el tamaño no mejora sino que queda igual. En definitiva, el esquema esperado de cantidad de ciclos de ejecución respecto a alguna de las dos mejoras, tiene la *forma* de la función logarítmica (mientras más grande sea el capacidad de datos o la cantidad de vías, menos va a mejorar respecto a la configuración inmediatamente inferior).

Luego, teniendo *todo esto* en cuenta, podemos ver que lo esperado es que las combinaciones de capacidad *de datos* - *cantidad de vías* que deberían tener la mejor *performance* son aquellas que cubran **todo el espacio** para los 3 arrays (evitando el *replacement* y que se pisen los datos en la caché). Es decir, son

- 128kB - 1 vía
- 64kB - 2 vías
- 32kB - 4 vías
- 16kB - 8 vías

dado que son las combinaciones más “chicas” que cubren los 96kB de los tres arreglos juntos.

Por ello mismo, entonces, lo esperado es que mientras más grande sea el *tamaño total* de la caché que consideremos (cnt. vías * tamaño de dato), entonces vamos a tener:

- Menos misses \Rightarrow Menos replacements \Rightarrow Más hits \Rightarrow Menos ciclos ociosos \Rightarrow Menos ciclos de ejecución

Análisis de resultados obtenidos

Ahora, respecto a los *resultados reales* obtenidos por la simulación del procesador **adc-gem5**, tenemos que *sorprendentemente* no se cumple con el resultado esperado por nosotros, dado que si bien baja la cantidad de ciclos de ejecución respecto a caché de 1 y 2 vías, mientras más grande hacemos la caché para las demás vías, o si consideramos 128kB, entonces se obtienen muchos más ciclos ociosos y, por ende, más tardanza de ejecución.

Si bien eso no es en lo absoluto lo esperado por nosotros, lo que consideramos pertinente a tener en cuenta para este análisis, es entonces concentrarnos sólo en la cantidad de *hits*, *misses* y *replacements* que tiene cada configuración distinta de la caché para esta benchmark no optimizada.

Motivo de ello, entonces, nos vamos a centrar en los siguientes datos (sin considerar ciclos de ejecución):

- overallHits
- overallMisses

Luego, entonces, consideramos las siguientes tablas de datos:

Tipo de caché	Mapeo directo (1 vía)				
Tamaño de la caché	8KB	16KB	32KB	64KB	128KB
Hits totales en la caché de datos (overallHits)	800	800	804	6112	10761
Misses totales en la caché de datos (overallMisses)	11498	11498	11494	6187	1537
Tipo de caché	Asociativa por conjunto de 2 vías				
Tamaño de la caché	8KB	16KB	32KB	64KB	128KB
Hits totales en la caché de datos (overallHits)	5335	5290	5290	5309	10761
Misses totales en la caché de datos (overallMisses)	6964	7009	7009	6989	1537
Tipo de caché	Asociativa por conjunto de 4 vías				
Tamaño de la caché	8KB	16KB	32KB	64KB	128KB
Hits totales en la caché de datos (overallHits)	10756	10756	10756	10756	10761
Misses totales en la caché de datos (overallMisses)	1541	1541	1541	1541	1537
Tipo de caché	Asociativa por conjunto de 8 vías				
Tamaño de la caché	8KB	16KB	32KB	64KB	128KB
Hits totales en la caché de datos (overallHits)	10756	10756	10756	10756	10761
Misses totales en la caché de datos (overallMisses)	1541	1541	1541	1541	1537

Por lo que podemos visualizar, el comportamiento que nosotros esperábamos se da respecto a los *hits* y *misses* de la caché de datos (suponemos que la simulación tiene un error al calcular la cantidad de ciclos ociosos y totales).

Respecto a esto, se puede notar cómo claramente mientras más grande es la caché principal de datos, más *hits* va a tener (principalmente si la combinación cnt. vías - tamaño de dato hace que sea mayor a los 96kB de los 3 arrays).

Por ello, entonces, como tenemos más hits y menos misses, el comportamiento esperado es que la ejecución de la *benchmark* sea muchísimo más rápido, dado que necesita ir *muy pocas veces* a la memoria principal para cargar los bloques en la caché.

! Además, y como *dato curioso*, **la cantidad de misses que obtenemos para cachés grandes es también el esperado**. ¿Por qué?

Como mencionamos en la sección del *comportamiento esperado*, la lectura del array X para una caché lo suficientemente grande para “cubrirlo” implicaría 512 misses (dado que los bloques son de 8 double words). Por ello, entonces, notemos que como leemos de 2 arrays y escribimos en 1, la cantidad de misses esperada sería de $512 \cdot 3 = 1536$, lo cual es *muy similar* a los resultados obtenidos (1541).

(e) Optimización de la *benchmark - daxpy optimized*

Optimizaciones consideradas

Las optimizaciones que se consideraron para la *benchmark daxpy* son las siguientes:

- Cada iteración va a realizar una sola instrucción de branch, por lo que se cambia la estructura general, *permitiendo* que la última iteración siempre falle. Es decir, se ejecuta **siempre** el valor que se va a guardar en Z, pero únicamente si se cumple la condición, lo vamos a guardar
 - Esto hace que si son n iteraciones, entonces hacemos los cálculos para n+1. Al no ser tan costosa una iteración, se prefiere realizar esto para valores de N grandes, reduciendo así la cantidad de instrucciones branch totales
- Se realiza *loop unrolling* para 2, 4 y/o 8 iteraciones (se *adjuntan* los 3 códigos):
 - Como el tamaño de bloque de la caché es de 8 double words, entonces lo *mejor* sería un *loop unrolling* de 8
Lo esperado es que ande mejor si leemos los 8 valores de X todos juntos primero, luego los 8 de Y y luego escribimos los 8 de Z
 - Se consideran 2, 4, 8 dado que el N es una potencia de 2 (4096).
- Se utilizan instrucciones *ldp* y *stp* para reducir por la mitad la cantidad de instrucciones LOAD y STORE. Estas instrucciones permiten cargar y guardar en memoria valores para dos registros consecutivos
- Para las instrucciones *ldp* y *stp* se pone también el valor que se requiere que se sume luego al registro que marca la dirección (address)
 - Esto permite no tener que tener un *add* aparte para hacerlo
- Se utiliza el *n* como indicador de cuando terminar con las iteraciones, decrementándolo. La parte del SUB se hace con la activación de las flags para aprovechar y luego hacer el *branch condicional*
- Se utiliza la instrucción *fmadd* para “juntar” el *fmul* y *fadd* en una sola instrucción que toma 3 registros

Análisis de los resultados de la optimización para dcache de 1 vía y 32kB de dato para cada tipo de *loop unrolling*

En este análisis, por lo mismo que se vio en los puntos anteriores debido al extraño cálculo que se realiza sobre los ciclos ociosos y totales para la ejecución, consideraremos también la cantidad de *misses* totales de la dcache y nos concentraremos en evaluar los puntos de:

- Overall Hits
- Overall Misses


de esta caché de datos principal, dado que sabemos que mientras más hits y menos misses, el procesador debe andar mucho más rápido y tener un mejor desempeño.

Motivo de ello, los datos que recopilamos para cada tipo de *loop unrolling* (2, 4 u 8 iteraciones) son:

Tipo de caché	Mapeo directo (1 vía)		
Tamaño de la caché	32KB		
Loop unrolling	2	4	8
Número de ciclos (numCycles)	180351	212349	182418
Ciclos ociosos (idleCycles)	145647	176500	156242
Hits totales en la caché de datos (overallHits)	1426	1595	4385
Hits de lectura en la caché de datos (ReadReq.hits)	153	570	2337
Cantidad de misses (overallMisses)	6777	6609	3824

Gracias a estos datos, podemos ver que de los *loops unrollings* considerados, el mejor es el de 8 iteraciones, lo cual ya lo explicamos más arriba dado que los bloques de la caché tienen 8 *double words*. Motivo de ello, como lo que hacemos es leer primero 8 elementos de X consecutivos, luego 8 consecutivos de Y, y luego escribir 8 consecutivos de Z, apenas tengamos el *miss* para el primer elemento, los demás serán todos *hits* ya que se cargarán al principio.

Esto permite que tengamos una mejora significativa de nuestra performance cuando hacemos *loop unrolling* de 8 iteraciones, disminuyendo muchísimo los *misses* y aumentando los *hits* ⇒ La cantidad de ciclos ociosos va a ser menor (teóricamente) que la de las demás benchmarks ⇒ Va a tener mejor performance.



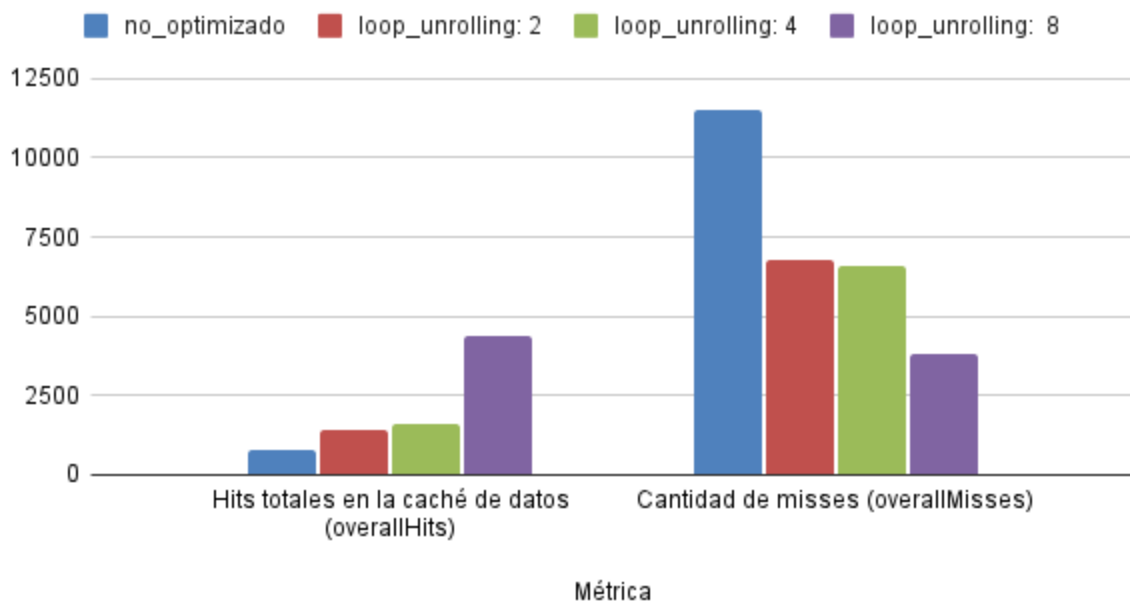
Como se mencionó varias veces antes, si bien ponemos los datos de numCycles e idleCycles, consideramos que no están bien calculados por los que no los utilizamos para el análisis, sino que nos concentramos únicamente en los *miss* y *hits*.

Comparación de resultados entre *daxpy* optimizado vs. no optimizado (para 1 vía - 32kB de dato)

Para poder comparar los valores obtenidos con las optimizaciones realizadas respecto al código de *daxpy* del punto (a), vamos a considerar los *hits* y los *misses* de la caché para medir la performance.

Tipo de código	Código no optimizado	Código optimizado		
Loop unrolling	X	2	4	8
Hits totales en la caché de datos (overallHits)	804	1426	1595	4385
Cantidad de misses (overallMisses)	11494	6777	6609	3824

Comparación optimizado - no optimizado



Claramente, como se puede visualizar según estos datos, optimizar el código sabiendo cómo es la caché del procesador y cómo funciona (en este caso que el tamaño de los bloques es de 8 double words), nos ayuda muchísimo para poder optimizar los códigos y hacer que sean **mucho más eficientes**.

En este caso, vemos una diferencia abismal en la cantidad de hits y misses entre el código de *daxpy* no optimizado vs. el código de *daxpy* optimizado con *loop unrolling* de 8 iteraciones.

Notamos que:

- La cantidad de hits aumenta en un 445%
- La cantidad de misses disminuye en un 67%

por lo que la *performance* es muchísimo mejor.

(f) Simulación de la dos *benchmarks* (*daxpy* optimizada con *loop unrolling* 2 vs. no optimizada) para procesador *out of order*

Resultados obtenidos para dcaché de 1 vía y 32kB de dato

Los resultados que obtuvimos en la simulación para un procesador *out of order* con *predictor por torneos* y caché principal de datos de 1 vía y 32kB de datos son los siguientes:

Tipo de caché	Mapeo directo (1 vía)			
Tamaño de la caché	32KB			
Loop unrolling	X	2	4	8
Número de ciclos (numCycles)	144115	82375	36225	37492
Ciclos ociosos (idleCycles)	216	214	214	455
Hits totales en la caché de datos (overallHits)	2673	656	980	1499
Hits de lectura en la caché de datos (ReadReq.hits)	2604	624	694	928
Cantidad de misses (overallMisses)	9623	7546	7223	6708

Los **gráficos** que vamos a considerar para el posterior análisis son los siguientes (incluye comparativa con el resultado de 32kB - 1 vía del punto a):

Gráfico ciclos

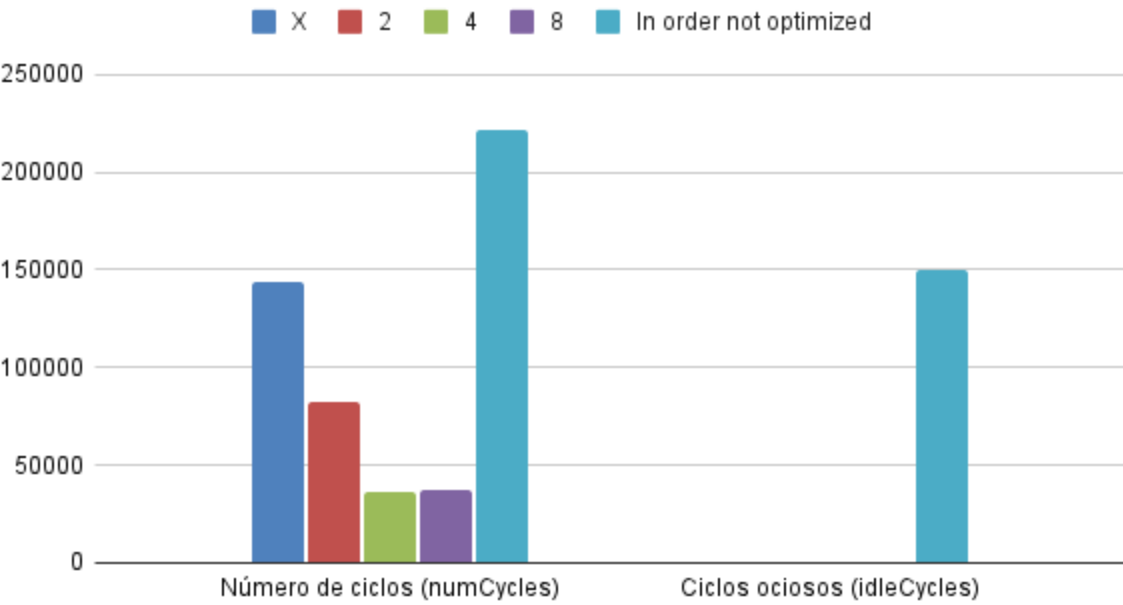
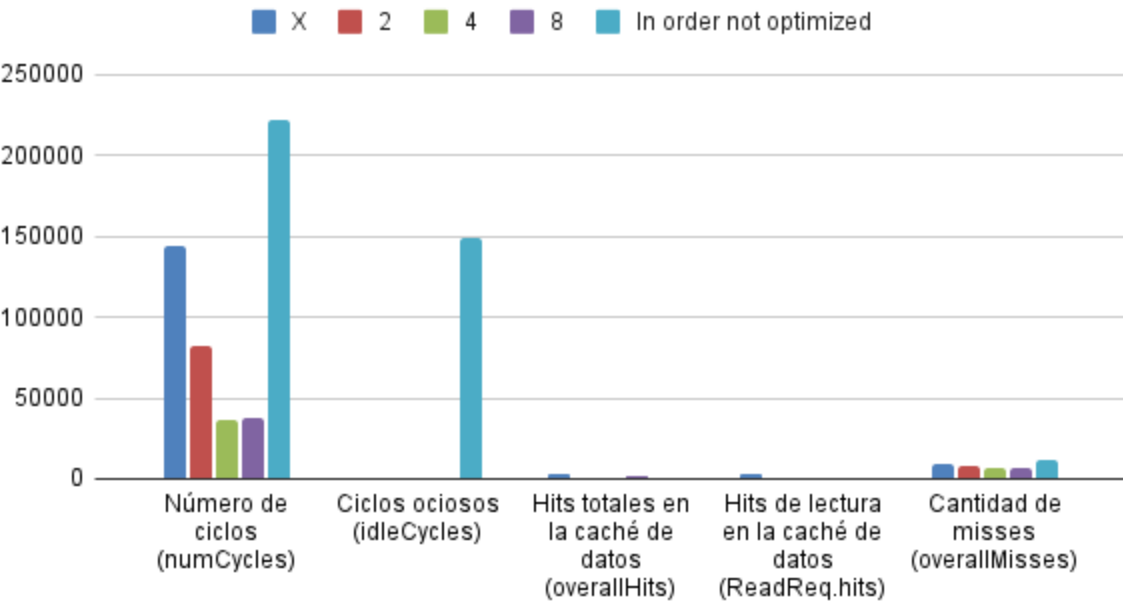


Gráfico hits, misses y ReadReq.hits



Análisis de los resultados y comparativa

! Dado que para el análisis de la simulación para los procesadores *out of order* es **muy** importante saber la cantidad de ciclos *numCycles* e *idleCycles* por sobre los valores de *hitRate* y *missRate*, vamos a considerarlos para nuestra explicación por más que hayamos visto que no son calculados correctamente por lo menos para tamaños de caché principal muy grandes.

Podemos notar claramente en los gráficos que, el hecho de que el procesador ejecute y realice las instrucciones en un orden distinto al brindado, *optimiza muchísimo* la ejecución de nuestros programas dado que se pueden hacer varias instrucciones LOAD y STORE al mismo tiempo, junto con instrucciones FMADD que estén después de estos pero ya no tengan dependencia real de dato (i.e., que tengan que esperar a la UF a que tenga calculado el dato y pase su etapa de WB).

Por ello mismo, la *performance* de nuestros programas aumenta muchísimo respecto al tener un procesador *out of order* frente al *in order*.

En términos de los datos que nos interesan para ver cómo es realmente la mejora, podemos notar que si comparamos la simulación del código de *daxpy* no optimizado con un procesador *in order* vs. la del código de *daxpy* optimizado con *loop unrolling* de 8 iteraciones con un procesador *out of order*, tenemos que:

- La cantidad de ciclos totales disminuye en un 83%
- La cantidad de ciclos ociosos disminuye en más del 99%

por lo que realmente podemos ver cómo afecta y mejora en las ejecuciones de los programas, tanto las técnicas de mejoras y optimización que realiza el compilador, como los procesadores que ejecutan instrucciones fuera de orden (*out of order*).