Lab 2 | Ejercicio 3 (Informe)

Integrantes:

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

Materia: Arquitectura de Computadoras 2023

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 una caché de datos principal con 32kB de dato y de mapeo directo, y un predictor de saltos por torneos, teniendo en consideración el código de la benchmark **bubbleSort** tanto optimizado como no.

Además de todos estos datos, se realizará su respectivo análisis y justificación para entender por qué funciona de ese modo y cómo mejora la optimización de un programa evitando los saltos condicionales y usando las instrucciones *csel*.

(c) Datos obtenidos para *predictor por torneos* y caché de datos principal de 32kB con mapeo directo (1 vía) y procesador *in order*

Datos obtenidos (resultados + gráficos)

En la siguiente tabla tenemos los rendimientos de ambos tipos de códigos con respecto a las variables obtenidas de las benchmarks:

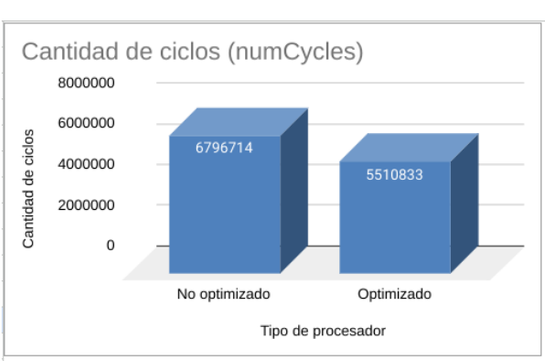
Cantidad de vías	Mapeo directo (1 vía)	
Tamaño de la caché	**32KB**	
Tipo de predictor	**Por torneos**	
Tipo de procesador	**In order**	
Código	No optimizado	Optimizado
Cantidad de ciclos (numCycles)	6796714	5510833
Cantidad de ciclos ociosos (idleCycles)	268	263
CPI	1,36	0.85
Conditional branchs predicted (condPredicted)	1073537	501502
Conditional branchs not predicted (condIncorrect)	144903	1007
MissRate de la predicción	11,89%	0,20%
Hits de la caché principal de datos (overallHits)	1508480	1999003
Misses de la caché principal de datos (overallMisses)	0	0
MissRate de la caché principal de datos	0,00%	0,00%

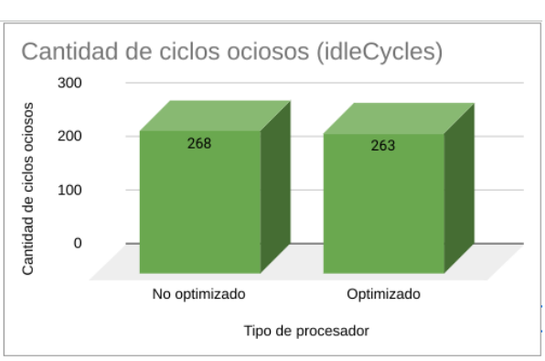
Si bien colocamos varios datos, para nuestro análisis nos concentraremos principalmente en **numCycles, idleCycles, CPI** y **MissRate de la predicción.**

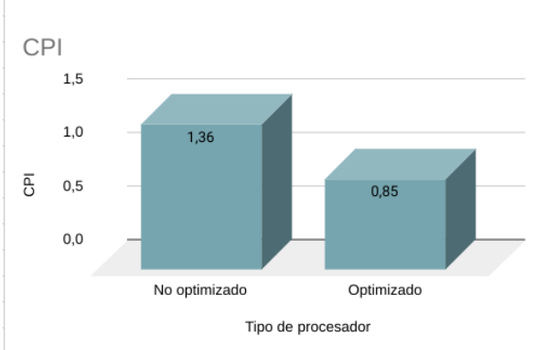
La cantidad de *misses* de la caché es igual a 0 dado que nuestro arreglo es de un poco menos de 8kB y la preparación de las *estadísticas / métricas* se realiza **luego** de la generación random de los valores para el arreglo.

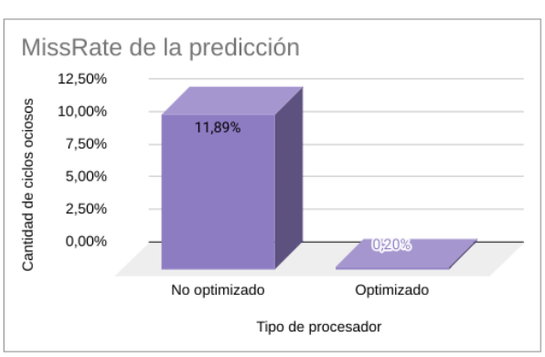
Luego, esto significa que una vez que comenzamos con el *bubbleSort*, absolutamente TODOS los valores del arreglo ya se encuentran en la caché por lo que no se generan misses para estos casos.

Motivo de ello, entonces, adjuntamos los gráficos que se utilizarán para referencia en el análisis:









Análisis de los resultados - Comparación entre código optimizado vs. no optimizado

Primero, antes de realizar el análisis de los datos que obtuvimos por la simulación de la ejecución de nuestros códigos de *bubbleSort*, tenemos que tener en cuenta que la optimización que realizamos consta de utilizar *operadores ternarios* para evitar la concurrencia de saltos (evitar bloques *if/else*) y, así, poder bajar el *missRate* de la predicción, usándose el predictor por torneos únicamente para la predicción de los *takens* de los *loops*.

Por ello mismo, tenemos que notar que como vamos a hacer uso de las instrucciones condicionales (en este caso, de **csel**), entonces nos podemos evitar los bloques if/else disminuyendo así instrucciones de salto condicionales y los posibles errores de predicción que pueda tener nuestro predictor por estos.

Visualizamos entonces, que en el caso del código optimizado:

- La cantidad de **numCycles** disminuye en un 18.9192%
- La cantidad de **idleCycles** disminuye en un 1.8657%
- La cantidad de **CPI** disminuye en un 37.5%
- El **missRate** de la predicción disminuye en un 98.3179%

lo cual significa un aumento **significativo** de la *performance* en la ejecución de la benchmark dado que nuestro CPI tiene una gran mejora pasando de 1,36 clocks por instrucción a solo 0,85.

(d) Datos obtenidos para *predictor por torneos* y caché de datos principal de 32kB con mapeo directo (1 vía) y procesador *out of order*.

Datos obtenidos (resultados)

En la siguiente tabla tenemos los rendimientos de ambos tipos de códigos con respecto a las variables obtenidas de las benchmarks:

Cantidad de vías	**Mapeo directo (1 vía)**			
Tamaño de la caché	**32KB**			
Tipo de predictor	**Por torneos**			
Tipo de procesador	**In order**		**Out of order**	
Código	No optimizado	Optimizado	No optimizado	Optimizado
Cantidad de ciclos (numCycles)	6796714	5510833	6663907	5513693
Cantidad de ciclos ociosos (idleCycles)	268	263	112	112
CPI	1,36	0.85	1,33	0,85
Conditional branchs predicted (condPredicted)	1073537	501502	1290019	501506
Conditional branchs not predicted (condIncorrect)	144903	1007	145024	1008
MissRate de la predicción	11,89%	0,20%	10,11%	0,20%
Hits de la caché principal de datos (overallHits)	1508480	1999003	1508474	1499501
Misses de la caché principal de datos (overallMisses)	0	0	0	0
MissRate de la caché principal de datos	0,00%	0,00%	0,00%	0,00%

> 💡 La cantidad de *misses* de la caché es igual a 0 dado que nuestro arreglo es de un poco menos de 8kB y la preparación de las *estadísticas / métricas* se realiza **luego** de la generación random de los valores para el arreglo.
>
> Luego, esto significa que una vez que comenzamos con el *bubbleSort*, absolutamente TODOS los valores del arreglo ya se encuentran en la caché por lo que no se generan misses para estos casos.

Análisis de los resultados - Comparación entre procesador *in order* vs. *out of order*

Podemos ver en los datos obtenidos que, aún así habiendo cambiado el tipo de procesador a *out_of_order*, no encontramos una diferencia **significativa** de *performance* en comparación con el tipo de procesador *in_order* (respecto a la comparación entre cada versión).

> 💡 La diferencia “más significativa” que se encuentra es que el CPI disminuye un 0.03 en el caso del código no optimizado para el procesador *out of order*.

Visualizamos entonces, que en el caso del código optimizado para el tipo de procesador *out_of_order* (en comparación con el código no optimizado del mismo tipo de procesador) tenemos:

- La cantidad de **numCycles** disminuye en un 17.2604%.
- La cantidad de **CPI** disminuye en un 36.0902%.
- El **missRate** de la predicción disminuye en un 98.0218%.

obteniendo una clara diferencia, la cual se explica con la gran reducción de saltos que se tiene a la hora de considerar operadores ternarios condicionales de assembler.

Por ello mismo, se puede notar también la optimización generada si se sacan saltos condicionales y se “reduce” la cantidad de misses de predicción por salto.

Respecto a la comparación entre los dos tipos de procesadores, consideramos que la diferencia *no es significativa* dado que por más que no haya orden en las instrucciones, para cada iteración **siempre** voy a necesitar cargar el valor de memoria que se asignó en la anterior (por ejemplo, en la iteración *i*, leo el dato en *i+1* y capaz también lo escribo, mientras que en la siguiente lo vuelvo a leer).

Esto hace que haya una dependencia *real* de dato y no podamos avanzar con la ejecución de las instrucciones fuera de orden dado que necesitamos que se calculen los valores anteriores para saber cómo continuar.