Práctica 3: Memoria caché

Arquitectura de Sistemas Integrados

Alejandro Solá
asola01@ucm.es

Pablo Suárez
pasuar03@ucm.es

Abril 2023

Índice

1. Introducción

En esta práctica, hemos entendido y estudiado el impacto del código implementado en el sistema, desde la perspectiva de la memoria caché y las métricas que determinan su rendimiento y eficiencia.

Hemos desarrollado la práctica en 3 sesiones, lo que nos ha permitido indagar en profundidad en todos los aspectos prácticos relacionados con la teoría vista en clase, desde los *commits* y los *misses* hasta el impacto de diferentes arquitecturas de memoria en el sistema embebido. Resumiendo las tareas realizadas en esta práctica, tenemso que:

- En la primera sesión hemos modificado el código base para poder medir correctamente los *hits* y los **misses**. Una vez hemos podido medir correctamente estos dos parámetros, hemos estimado teóricamente los valores, esto nos ha ayudado ha entender mejor la teoría y a repasar conceptos relacionados con el tema de memoria. Además, hemos analizado las métricas obtenidas $\forall \mathbf{N} \in \{128, 256, 512, 1024\}$ y $\forall \mathbf{K} \in \{4, 8, 16, 32\}$ y las hemos comparado con las obtenidas en la anterior práctica, mediante gráficas.
- En la segunda y tercera sesión hemos vuelto al código base, para comparar los resultados correspondientes al rendimiento de la memoria y del **SoC** cuando se implementa el código del filtro FIR en una arquitectura escalar y superescalar (i.e: desactivar/activar el pipeline), para este caso hemos hecho atención al uso de dos `bitstreams` para cada uno de los casos. Por último, hemos aplicado diferentes niveles de optimización y hemos analizado los resultados obtenidos con ellos.

2. Primera sesión

Tras cargar el proyecto `HWCounters` en VScode hemos modificado el código de la siguiente manera, para medir correctamente los *hits* y los *misses*:

```
int main(void){
    int cyc_beg, cyc_end;
    int instr_beg, instr_end;
    int Hitss_beg, Hitss_end;
    int Misses_beg, Misses_end;
    /* Initialize Uart */
    uartInit();

    pspEnableAllPerformanceMonitor(1);
    pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
    pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
    pspPerformanceCounterSet(D_PSP_COUNTER2, E_I_CACHE_HITS);
    pspPerformanceCounterSet(D_PSP_COUNTER3, E_I_CACHE_MISSES);

    cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
    instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
    Hitss_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
    Misses_beg = pspPerformanceCounterGet(D_PSP_COUNTER3);

    #ifdef FIR
        __asm("li t2, 0x001"); // Disable Pipelined Execution
        __asm("csrrs t1, 0x7F9, t2");
        int t,i;
        //init();
        for (t=K;t<N;t++)
          for (i=0;i<K;i++)
            Y[t] += h[i]*X[t-i];
    #else
        Test_Assembly();
    #endif

    Hitss_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
    Misses_end = pspPerformanceCounterGet(D_PSP_COUNTER3);
    cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
    instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);

    printfNexys("Cycles = %d", cyc_end-cyc_beg);
    printfNexys("Instructions = %d", instr_end-instr_beg);
    printfNexys("Hits = %d", Hitss_end-Hitss_beg);
    printfNexys("Mis = %d", Misses_end-Misses_beg);

    while(1);
}
```

En este paso hemos tenido que tener cuidado con la posición de los contadores, pues si queremos contabilizar solo aquello que nos interesa (en nuestro caso, solo el código del FIR) la posición de los contadores es crucial para poder saber con exactitud que está pasando en el sistema de memoria.

2.1. Entregable

Realice una estimación del número de *hits* y *misses* que debería obtener, conociendo el número de instrucciones, calculado en la anterior práctica. ¿Encuentra desviaciones entre el dato teórico y el medido? Justifique la respuesta.

Sabiendo que en la anterior práctica (gracias al uso de `breakpoints`) el código del filtro estaba comprendido entre la linea 39 y la 66. Y del laboratorio 19 sabemos que:

Characteristic	Value
I$ Size	
Data Array (without parity information) **Parity information** for data:	16 Kibytes 1 Kibyte (4 Bytes per block)
Tag Array (without parity information) **Parity information** for tags	640 Bytes 32 Bytes (1 bit per tag)
LRU State	24 Bytes (3 bits per set)
Valid Bit	32 Bytes (1 valid bit per tag)
Associativity (not configurable)	4 ways
Block Size	64 Bytes
Number of blocks (Size/Block Size=16Ki/64)	256 blocks
Number of blocks per way (# blocks/Assoc.=256/4)	64 blocks

Figura 1: Configuración del `l$`

La siguiente figura ilustra el diseño interno del `l$`

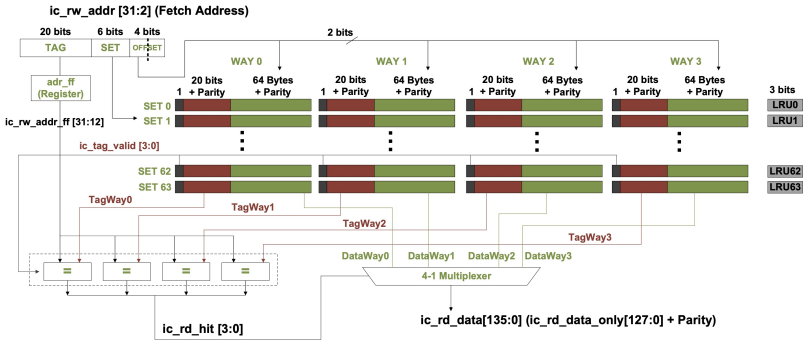


Figura 2: Diseño interno del `l$`

Y además se sabe que el tamaño de bloque es de 64 bytes, y que cada instrucción ocupa 32 bits que equivalen a 4 bytes en total, entonces se podrán almacenar 16 instrucciones en cada bloque, lo que se traduce en 1 fallo por llevar las primeras 16 instrucciones, un segundo fallo por llevar las 13 instrucciones restantes, dándonos un total de 2 *misses*. En el caso de los *hits*, los podemos calcular, sabiendo el número de iteraciones de los dos bucles y el número de accesos producidos por las variables. En este caso, hay que tener en cuenta, que el procesador escribe el resultado en la caché, por lo tanto, en la siguiente linea de código, habrá 4 accesos:

```
1 Y[t] += h[i]*X[t-i]; //Equivalente a Y[t] = Y[t]+h[i]*X[t-i]
```

- Acceso para saber `X[t-i]`.
- Acceso para saber `h[i]`.
- Acceso para saber `Y[t]`.
- Acceso para escribir en caché `Y[t]`.

Por último, para el cálculo del Miss(%), i.e., "*miss rate*" se ha hecho uso de la siguiente expresión: $miss\_rate = 1 - hit\_rate$, dónde el *hit_rate* puede calcularse como:

$$hit\_rate = \frac{hits}{hits + misses} \Rightarrow hit\_rate(\,\%) = hit\_rate \cdot 100$$

Por lo tanto:

$$miss\_rate(\,\%) = \left(1 - \frac{hits}{hits + misses}\right) \cdot 100$$

2.1.1. Métricas obtenidas con el código base

Una vez estimados los *hits* y los *misses* a mano, hemos variado los tamaños del filtro para observar cómo evoluciona el número de *hits* y *misses* de la cache de instrucciones $\forall \mathbf{N} \in \{128, 256, 512, 1024\}$ y $\forall \mathbf{K} \in \{4, 8, 16, 32\}$, asi como los *miss rates*[1] asociados. Véase la tabla 20

Cuadro 1: Resultados obtenidos con el código base.

	N = 128						**N** = 256					
K	Cycles	Instr	Hits	Misses	Miss(%)	CPI	Cycles	Instr	Hits	Misses	Miss(%)	CPI
4	32670	10870	3892	2	0.05 %	3,00	66024	22006	7860	2	0.025 %	3,00
8	61711	20122	6648	2	0.03 %	3,07	126155	41498	13688	2	0.015 %	3,04
16	113461	36706	11584	2	0.017 %	3,09	241087	78562	24768	3	0.012 %	3,07
32	194287	65170	19073	2	0.01 %	2,98	446964	145010	44624	3	0.0067 %	3,08

Cuadro 2: Resultados obtenidos con el código base.

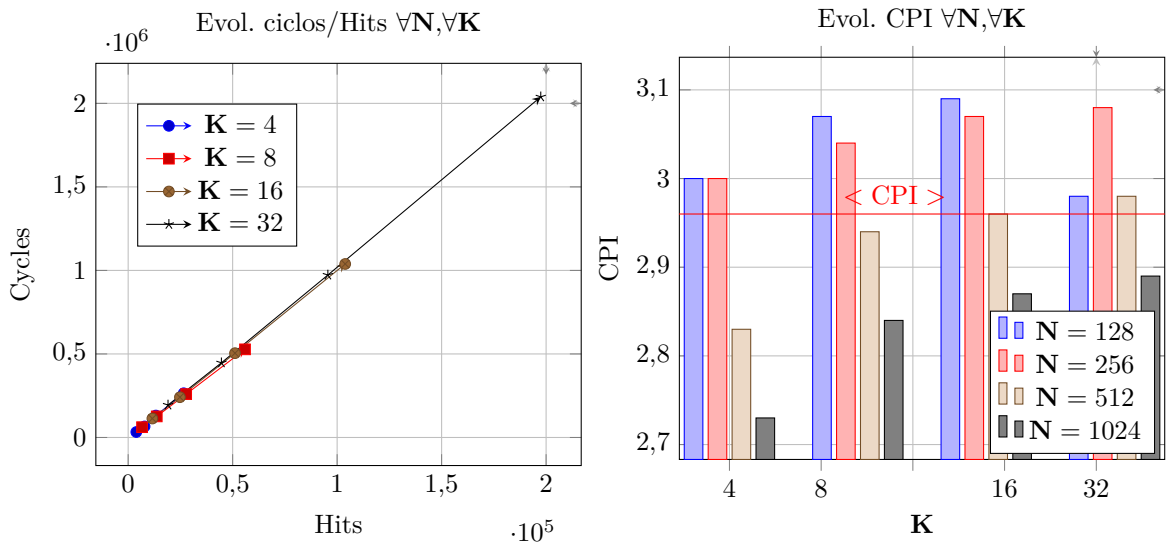
	N = 512						**N** = 1024					
K	Cycles	Instr	Hits	Misses	Miss(%)	CPI	Cycles	Instr	Hits	Misses	Miss(%)	CPI
4	131283	46310	13262	2	0.015 %	2,83	264767	96982	26567	2	0.0075 %	2,73
8	259605	88282	27762	2	0.0072 %	2,94	527596	186010	55916	2	0.0036 %	2,84
16	504660	170210	51128	2	0.004 %	2,96	1038623	361954	103860	2	0.002 %	2,87
32	972412	326002	95562	2	0.002 %	2,98	2038963	705394	197444	2	0.001 %	2,89

Cuadro 3: *Hits* y *Misses* teóricos para el código base

	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	Hits	Misses	Diff	Hits	Misses	Diff	Hits	Diff	Diff	Hits	Misses	Diff
4	1984	2	1908	4032	2	3828	8128	2	5134	16320	2	10247
8	3840	2	2808	7936	2	5752	16128	2	11634	32512	2	23404
16	7168	2	4420	15360	2	9408	31744	2	19384	64512	2	39348
32	12288	2	6785	28672	2	15952	61440	2	34122	126976	2	70468

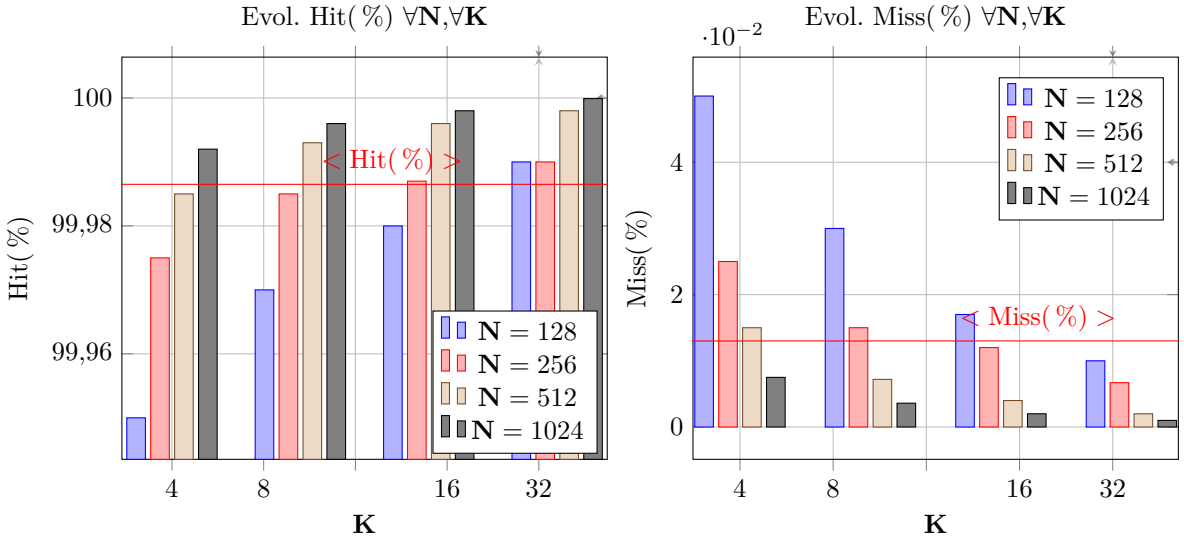
A continuación se adjuntan los gráficos correspondiente a los cuadros 20 y 21.

[1] El *miss rate* lo hemos calculado haciendo el cociente entre los *misses* y el total de lecturas de caché. Sabemos por teoría que esta cifra debería de ser como máximo $\sim 5\,\%$



En el caso del CPI vemos un comportamiento similar al estudiado en la anterior práctica. Vemos una pequeña desviación en la tendencia del CPI para **K**=32 y **N**=128. Esto podría deberse a una fluctuación a la hora de medir el valor, pues **CPI** $\propto$ **N**, pero no estamos seguros de que en el caso de repetir la medida, se obtenga un CPI mayor que para **K**=16 y **N**=128, pues esto ya lo hemos observado en la anterior práctica y se verá de nuevo a lo largo de esta. Esto es algo que no acabamos de comprender, pues solo se da para **K** = 32, y aunque para el resto de **Ns** $\in$ {**K** = 32} se obtienen CPIs mayores, puede observarse como el incremento de los mismos es inferior al sufrido cuando se pasa de **K**=4 a **K**=8, por lo que se llega a la conclusión de que según aumentan los valores de **K** y de **N** la tasa de incremento en el CPI es cada vez menor, haciendo converger al CPI según aumenta el número de instrucciones, lo cual tiene lógica, pues el denominador en la fórmula del CPI va haciéndose más grande.

Respecto a la gráfica de la derecha, se observa un aumento lineal del número de *hits* según aumentan los ciclos. Esto es lógico, pues cuantos más ciclos se llevan a cabo, hay más probabilidad de que se produzcan *hits*. El CPI medio es de 2.96.



En la figura 2.1.1 (derecha) podemos ver la tendencia creciente y lineal de los *hits* según aumenta el número de ciclos, algo que puede intuirse estudiando las tablas 20 y 21 viendo como para cada valor de **K** los ciclos y los *hits* aumentan $\times 2$. Este aumento $\times 2$ de los hits no se ve reflejado en la gráfica de la evolución de los *hits* respecto a las **Ks** porque ahí se representa el *hit rate* en (%) por lo que también depende del número de *misses*. En este caso se obtiene un <Hit(%)> del 99,9865 %, mayor que el umbral (95 %) estipulado en teoría. Además en la figura → (izquierda) se observa un crecimiento lineal de los *hits* según aumenta **K**.

En la figura → (la de la derecha) vemos representado el Miss(%) frente al valor de las **Ks**. Para ver la tendencia del *miss rate* para cada valor de **K** y **N**, hemos calculado el <Miss(%)>, así podremos ver más fácilmente las desviaciones y sus valores asociados. El *miss rate* medio es de 0.013 % lejos del 5 % por ello, es un buen resultado. Se puede observar una tendencia lineal y decreciente en los *misses* según aumenta **K**.

Este mismo análisis lo hemos repetido para las técnicas de optimización software: *Software pipeline* y *Loop unrolling*. Tanto para el caso del *Software pipeline* como para el *Loop unrolling*, hemos simulado el código $\forall \mathbf{N} \in \{128, 256, 512, 1024\}$ y $\forall \mathbf{K} \in \{8, 32\}$.

2.1.2. Métricas obtenidas con el *Loop unrolling*

La estructura del *loop unrolling* es la siguiente:

```
#ifdef FIR
    //__asm("li t2, 0x001"); // Disable Pipelined Execution
    //__asm("csrrs t1, 0x7F9, t2");
    int t,i,sum0,sum1;
    //init();

    //Aqui declaramos las variables locales que hemos creado para reducir el numero de
    accesos a memoria
    short x0,h0,x1,h1;

    for (t=K;t<N;t++){
        sum0 = 0;
        sum1 = 0;

        for (i=0;i<K;i++){
            x0 = X[t];
            h0 = h[i];
            x1 = X[t+i+1];
            sum0 += x0 * h0;
            sum1 += x1 * h0;
            x0 = X[t+i+2];
            h1 = h[i+1];
            sum0 += x1 * h1;
            sum1 += x0 * h1;
            }
         Y[t] = sum0; // tenemos que cambiar la magnitud del desplazamiento
         Y[t+1] = sum1; // tenemos que cambiar le 15 por el numero del desplazamiento que
    queramos efectuar
```

Teniendo en cuenta el código de arriba, el cálculo de los *hits* es inmediato:

$$\#_{hits,loop\_unrolling} = (N-K)\cdot K\cdot 5 + (N-K)\cdot 2$$

En la ecuación de arriba, el número 5 hace referencia al número de variables por las que se realizan accesos. En el caso del 2, este hace referencia a las dos asignaciones que se hacen al final del bucle exterior.

Cuadro 4: Resultados obtenidos con el *Loop unrolling*.

	N = 128						**N** = 256					
K	Cycles	Instr	Hits	Misses	Miss(%)	CPI	Cycles	Instr	Hits	Misses	Miss(%)	CPI
8	70547	31882	9876	3	0.03 %	2,21	145606	65802	20372	3	0.015 %	2,21
32	218815	99922	28471	3	0.01 %	2,19	505298	226098	66768	3	0.0045 %	2,23

Cuadro 5: Resultados obtenidos con el *Loop unrolling*.

	N = 512						**N** = 1024					
K	Cycles	Instr	Hits	Misses	Miss(%)	CPI	Cycles	Instr	Hits	Misses	Miss(%)	CPI
8	296019	137674	41368	3	0.0073 %	2,15	596290	285578	83348	3	0.0036 %	2,08
32	1092527	499762	143071	4	0.0028 %	2,19	2255350	1064498	295652	3	0.0014 %	2,12

En el caso del CPI vemos un comportamiento similar al estudiado en la anterior práctica. Vemos una pequeña desviación en la tendencia del CPI para **K**=32 y **N**=128, esto puede deberse al hecho de que la técnica de optimización software *loop unrolling* mejora los resultados del CPI, según aumentan las **Ns** y las **Ks** ya que al incrementar estos parámetros el número de instrucciones también lo hace. Esta desviación en la tendencia creciente del CPI para esos mismos valores de **K** y **N** también se observó en la anterior práctica. Cabe destacar la reducción de saltos que supone la implementación de esta técnica, por ello en la anterior

práctica se observaban un menor número de saltos respecto a las otras técnicas de optimización, aunque esta técnica aumenta el código de manera considerable además de aumentar la presión sobre los registros debido a un mayor uso de los mismos.

Cuadro 6: *Hits* y *Misses* teóricos para el código con *Loop unrolling*

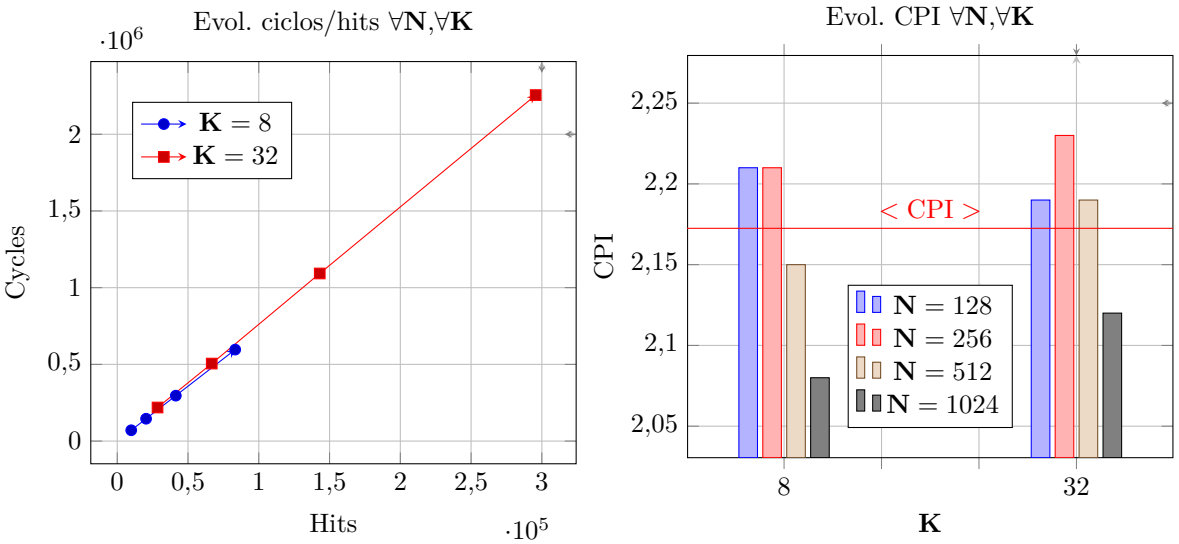
	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	Hits	Misses	Diff	Hits	Misses	Diff	Hits	Diff	Diff	Hits	Misses	Diff
8	2728	3	7148	5544	3	14828	11176	3	30192	22440	3	60908
32	15552	3	12919	36288	3	30480	77760	3	65311	160704	3	134948

Como puede observarse, la diferencia entre los *hits* medidos y teóricos es abismal, incluso más grande que el resultado teórico obtenido. Se ha investigando acerca de este asunto, pero no se ha conseguido concluir nada vinculante y seguro, que pueda explicar tal diferencia. Aunque, se piensa que pueda deberse a que lo contadores estén contemplando más instrucciones de las que deberían, pero esto tampoco tiene mucho sentido, pues los contadores se han puesto justo antes y después del código, tal y como puede verse en los códigos que se ha adjuntado en cada uno de los casos.

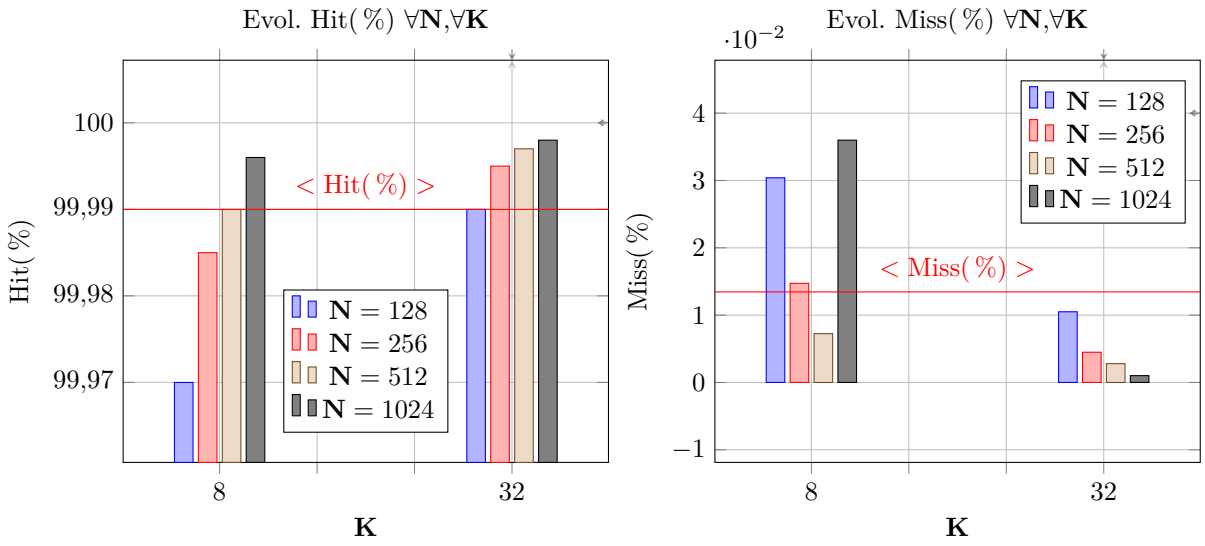
Continuando con las suposiciones, el número de *hits* podría ser muy grande debido a otras instrucciones que se encuentren en el bus y que no formen parte del código que estamos simulando, algo de lo que no estamos muy seguros, pero que se nos ha ocurrido.

Comparando los *misses* teóricos y *prácticos* no vemos diferencia, por lo tanto todo bien. Aunque en los resultados prácticos hay casos en los que se registran 4 *misses* pero eso tendrá que ver con una diferencia muy pequeña de instrucciones, entre el número de instrucciones reales que tiene en cuenta el SoC frente a las que se han contemplado teóricamente.

A continuación se adjunta el gráfico correspondiente a los cuadros 4 y 5.



Como bien se puede observar en las dos figuras de arriba, la evolución de ciclos/*hits* se mantiene creciente y lineal respecto al caso anterior. El CPI también gurada similitudes respecto al comportamiento observado en el caso anterior para el código base. El CPI se ha visto disminuido gracias al *loop unrolling*, respecto al caso anterior. En este caso, el CPI medio es de 2.17 una mejora sustancial, pues para el caso base, el CPI medio era de 2.96.



Como puede verse en las dos representaciones de la figura 2.1.2, la tendencia es creciente y lineal. En este caso el $<$ Hit(%) $>$ es del 99.99 % frente al 99.9865 %, si bien son dos cifras casi idénticas, se ve una mejora. Esta mejora es más relevante en el CPI, pues en el resto de métricas como el *miss rate*, vemos que se sigue manteniendo bajo y muy similar al obtenido en el apartado anterior.

En este caso el $<$ Miss(%) $>$ = 0.013448 %, por lo que no se ve una mejora sustancial respecto al caso anterior, pues se obtuvo un *miss rate* del 0.013 %. En este caso, para **K**=8 y **N**=1024 se observa un *miss rate* muy superior que para **N**=256 y **N**=512, cuando se supone que debería de ser inferior. Cabe destacar que este comportamiento no se vio en el caso anterior.

En la tabla 7 se exponen las mejoras del CPI que hay entre el código base y el *loop unrolling* para cada uno de los casos. La mejora entre CPIs se ha calculado de la siguiente manera: Mejora(%) $= \frac{\text{CPI}_{base} - \text{CPI}_{loop}}{\text{CPI}_{base}} \cdot 100$. Se observan mejoras similares para todos los casos, algo lógico porque son resultados producto de distintos parámetros, por lo que son proporcionales. Si uno hace la media entre las mejoras computadas, se obtiene una mejora media del 27.02 % una disminución considerable del CPI. Como ya se mencionó antes, no se observa una mejora notable en el *miss rate*.

Cuadro 7: Comparativa de `CPIs` entre código base y *loop unrolling*

	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	CPI_{base}	CPI_{loop}	%	CPI_{base}	CPI_{loop}	%	CPI_{base}	CPI_{loop}	%	CPI_{base}	CPI_{loop}	%
4	3	-	-	3	-	-	2.83	-	-	2.73	-	-
8	3.07	2.21	28 %	3.04	2.21	27,3	2.94	2.15	26,87	2.84	2.08	26,76
16	3.09	-	-	3.07	-	-	2.96	-	-	2.87	-	-
32	2.98	2.19	26.51	3.08	2.23	27.60	2.98	2.19	26.51	2.89	2.12	26.64

2.1.3. Métricas obtenidas con el *Software pipeline*

La estructura del *software pipeline* es la siguiente:

```
#ifdef FIR
    //__asm("li t2, 0x001"); // Disable Pipelined Execution
    //__asm("csrrs t1, 0x7F9, t2");
    int t,i,sum0,sum1;
    //init();

    //Aqui declaramos las variables locales que hemos creado para reducir el numero de
    accesos a memoria
    short x0,h0,x1,h1;

    for (t=K;t<N;t++){
        sum0 = 0;
        sum1 = 0;
        x0 = X[t];
        h0 = h[i];
        for (i=0;i<K;i++){
            x1 = X[t+i+1];
            sum0 += x0 * h0;
            sum1 += x1 * h0;
            x0 = X[t+i+2];
            h1 = h[i+1];
            sum0 += x1 * h1;
            sum1 += x0 * h1;
            }
         Y[t] = sum0; // tenemos que cambiar la magnitud del desplazamiento
         Y[t+1] = sum1; // tenemos que cambiar le 15 por el numero del desplazamiento que
    queramos efectuar
```

Haciendo atención al código de arriba y a la expresión implementada para el cálculo de *hits* para el *loop unrolling*, el cálculo del número de hits para este caso es inmediato:

$$\#_{hits,software\_pipeline} = (N-K)\cdot K\cdot 3 + (N-K)\cdot 4$$

El **3** de la ecuación de arriba hace referencia a los 3 accesos que se realizan por los 3 *arrays* en el bucle interior del código, mientras que el **4** hace referencia a los accesos que se producen por los dos primeros y los dos últimos *arrays* del bucle externo.

Cuadro 8: Resultados obtenidos con el *software pipeline*.

	N = 128						**N** = 256					
K	Cycles	Instr	Hits	Misses	Miss(%)	CPI	Cycles	Instr	Hits	Misses	Miss(%)	CPI
8	64298	27082	8078	4	0.05 %	2,37	132461	55882	16657	4	0.024 %	2,37
32	195602	82258	22519	4	0.0178 %	2,38	455845	184882	52681	4	0.0076 %	2,47

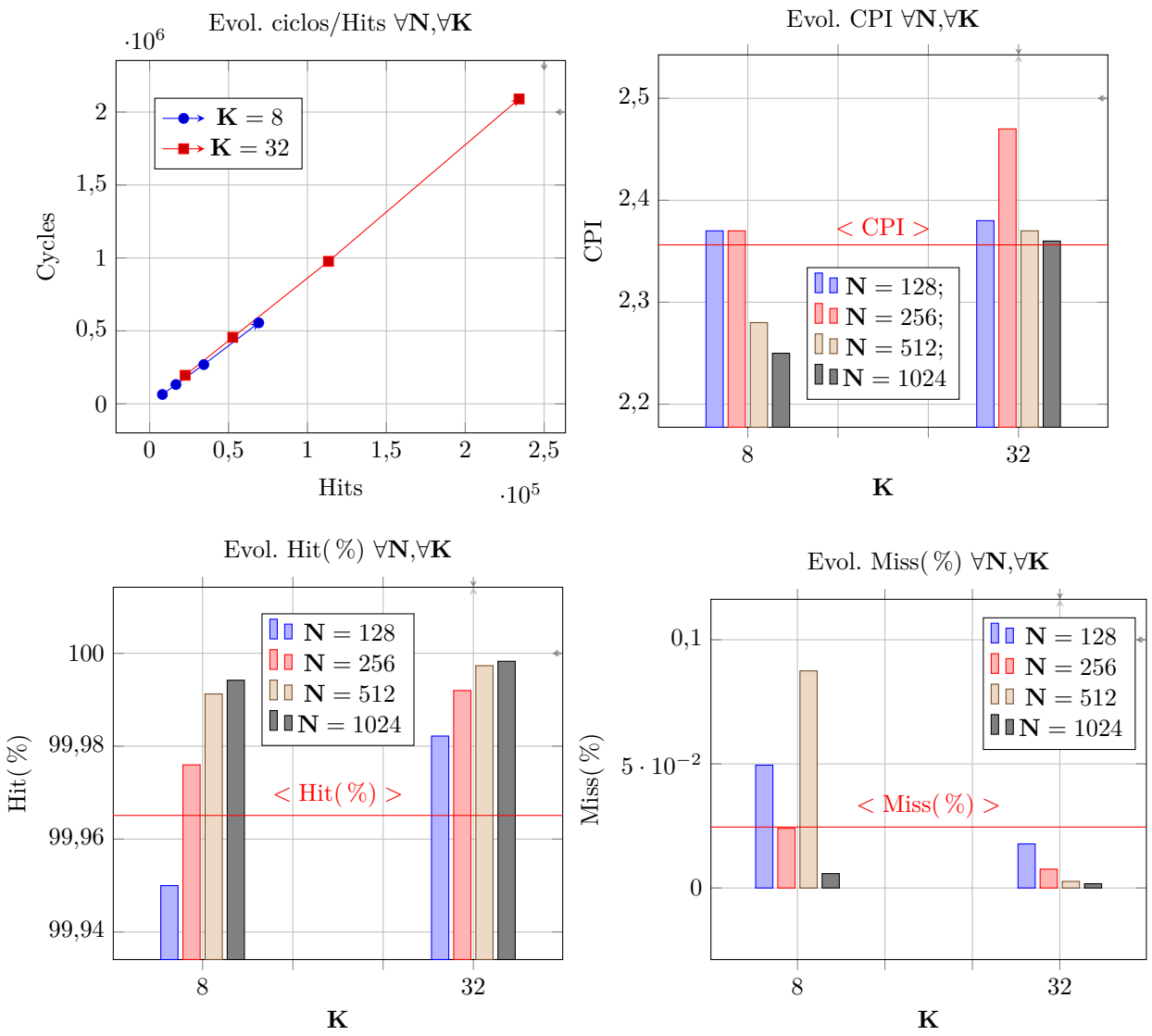
Cuadro 9: Resultados obtenidos con el *software pipeline*.

	N = 512						**N** = 1024					
K	Cycles	Instr	Hits	Misses	Miss(%)	CPI	Cycles	Instr	Hits	Misses	Miss(%)	CPI
8	269171	118018	34307	3	0.012 %	2,28	555428	246970	69121	4	0.0058 %	2,25
32	976621	411922	113315	3	0.0044 %	2,37	2089919	883954	234147	4	0.0017 %	2,36

Cuadro 10: *Hits* y *Misses* teóricos para el código con *software pipeline*

	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	Hits	Misses	Diff	Hits	Misses	Diff	Hits	Diff	Diff	Hits	Misses	Diff
8	1984	3	6094	6940	3	9717	14112	3	20195	28448	3	40673
32	9600	3	12919	22400	3	30281	48000	3	65315	99200	3	134947

Al igual que pasaba con el *loop unrolling*, la diferencia entre los resultados prácticos y teóricos es muy grande.



Como se puede ver en la figura 2.1.3 (izquierda), la tendencia de la evolución de los ciclos/hits es creciente y lineal. Observando la figura de la derecha, se obtiene un CPI medio del 2.35, peor CPI que aquel obtenido con el *loop unrolling*. La técnica del *software pipeline* no reduce el número de saltos, esta razón podría justificar un mayor CPI.

En este caso, el *miss rate* medio que se ha obtenido es del 0.02455 %, por lo tanto es un *miss rate* bueno y aún está lejos del límite del 5 % estipulado en clase.

Observando los hits, se obtiene un *hit rate* medio del 99.96 %, frente al 99.99 % obtenido con el *loop unrolling* y frente a un 99.9865 % obtenido con el código base. La única mejora que introduce el *software pipeline* frente al código base es respecto al CPI, pero comparando el *loop unrolling* con esta útima técnica, no hay mejora, pues se obtienen resultados inferiores. Aun así, es importante destacar, que aunque el *loop unrolling* obtiene mejores resultados respecto al CPI y al *hit ratio* lo hace a costa de introducir más instrucciones, ya que es el *software pipeline* la técnica de optimización que menos instrucciones introduce y menos ciclos itera. A continuación se adjunta una tabla que resume la mejora del CPI que hay entre el código base y el *software pipeline*:

La mejora media es del 20.90 %, respecto al código base. Según este resultado, la técnica de optimización software que ofrece una mejora "superior" respecto al CPI es la del *loop unrolling*, pero a costa de más instrucciones.

2.2. Entregable

¿Mejoran las técnicas anteriores el *miss rate* de la cache de instrucciones? Compare el CPI de cada técnica con el número de accesos y pérdidas de la cache.

Cuadro 11: Comparativa de `CPIs` entre código base y *software pipeline*

	N = 128			**N** = 256			**N** = 512			**N** = 1024		
K	CPI_{base}	CPI_{loop}	%	CPI_{base}	CPI_{loop}	%	CPI_{base}	CPI_{loop}	%	CPI_{base}	CPI_{loop}	%
4	3	-	-	3	-	-	2.83	-	-	2.73	-	-
8	3.07	2.37	22,80	3.04	2.37	22,04	2.94	2.28	22,45	2.84	2.25	20,77
16	3.09	-	-	3.07	-	-	2.96	-	-	2.87	-	-
32	2.98	2.38	20.13	3.08	2.47	19.81	2.98	2.37	20.47	2.89	2.36	18.34

A lo largo de los análisis anteriores, hemos podido comprobar una mejora notable respecto al CPI, pero no observamos ninguna mejora respecto a las métricas de acceso a caché (i.e: *hits* y *misses*). Si hacemos una comparativa entre CPIs y pérdidas de caché, no obtenemos una relación aparente entre ambos parámetros. Vamos a obtener entre 2 y 4 pérdidas en caché, pero el CPI apenas varía entre un caso y otro. Cabe destacar ademñas que ninguna de las dos técnicas de optimización mejora el *miss rate*

El *software pipelining* puede aumentar la localidad temporal de las instrucciones, lo que podría mejorar la tasa de fallos de caché de instrucciones. Sin embargo, también puede aumentar la cantidad de instrucciones que se ejecutan simultáneamente, lo que podría aumentar la presión sobre la caché de instrucciones y potencialmente empeorar la tasa de fallos.

El *loop unrolling* mejora la localidad espacial de las instrucciones, lo que podría reducir la tasa de fallos de caché de instrucciones. Sin embargo, también puede aumentar el tamaño del código, lo que puede causar un mayor uso de la caché de instrucciones y, en algunos casos, aumentar la tasa de fallos de caché.

3. Segunda y tercera sesión

Como ya mencionamos en la introducción del informe, en esta sección retomamos de nuevo el código original del FIR. Con él, mediremos el rendimiento del **SoC** haciendo atención a las métricas estudiadas anteriormente, para dos tipos de arquitecturas:

1. Escalar.
2. Superescalar.

Para esta sección hemos tenido que cargar el proyecto que se encuentra en la ruta:

`/home/asirv/RVfpga_v2-1/Labs/Lab20/RealBenchmarks/HWCounters_Example/`

Luego, hemos medido el número de ciclos así como el número de transacciones que se dan en el bus e instrucciones del filtro FIR para las dos jerarquías de memoria, que podemos ver en la figura 3(tomadas del pdf del Laboratorio 20):

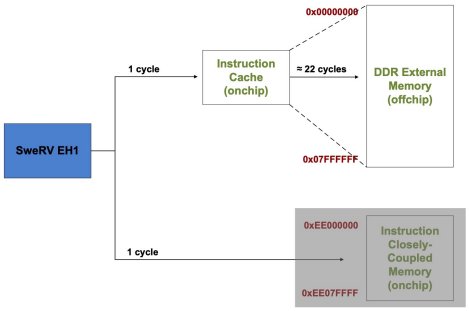
Estudiando las dos arquitecturas, vemos como en el caso de la arquitectura que se muestra en la figura 3a, el número de ciclos se ve disminuido, porque las instrucciones se quedan guardadas dentro del módulo ICCM.

Hemos leído que la memoria DCCM de la arquitectura de la figura 3b, realiza *pre-fetching*. La dinámica que sigue esta arquitectura es la siguiente; se cargan todas las instrucciones en la ICCM (en el caso de que sea de 64 bytes), en caso contrario solo cargaríamos una parte, luego tendríamos que `DataBusTransaction` $\neq$ `0` así como `InstBusTransaction` $\neq$ `0`.

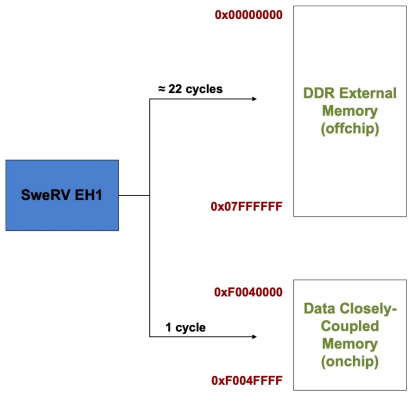
3.0.1. Procesador escalar

Un procesador escalar, respecto al superescalar (el que hemos estado implementando) se diferencia en la ausencia del pipe. Para este caso, solamente hemos tenido que descomentar las lineas que anulan la eficacia del pipe en el código, haciendo esto, y variando los parámetros del filtro, obtenemos los siguientes resultados:

A continuación se adjunta el gráfico correspondiente a los cuadros 16 y 17.



(a) Espacio de direcciones del *Instruction Memory*, conformado por una caché de instrucciones (`1$`) y una memoria DDR externa.



(b) Espacio de direcciones, conformado por dos memorias externas: DCCM y DDR.

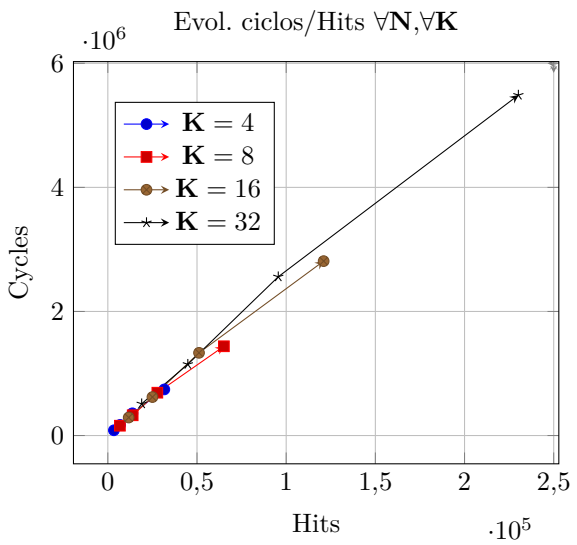
Figura 3: Las dos arquitecturas de memoria contempladas en esta sección.

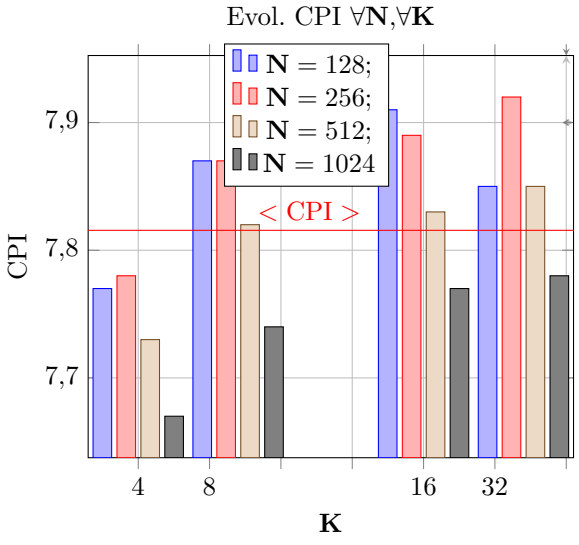
Cuadro 12: Resultados obtenidos con el código base.

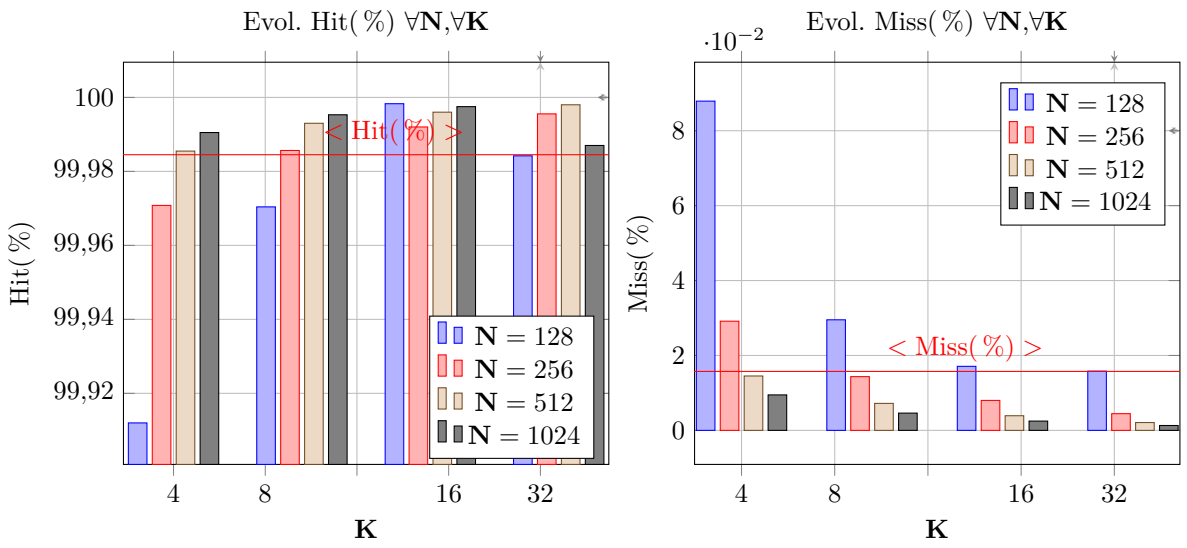
	N = 128						**N** = 256					
K	Cycles	Instr	Hits	Misses	Miss(%)	CPI	Cycles	Instr	Hits	Misses	Miss(%)	CPI
4	84501	10872	3409	3	0.0879 %	7,77	171310	22008	6858	2	0.0291 %	7,78
8	158323	20124	6772	2	0.0295 %	7,87	326584	41500	13940	2	0.0143 %	7,87
16	290369	36708	11700	2	0.0171 %	7,91	619987	78564	25012	2	0.008 %	7,89
32	511899	65172	18976	3	0.0158 %	7,85	1148575	145012	44852	2	0.0045 %	7,92

Cuadro 13: Resultados obtenidos con el código base.

	N = 512						**N** = 1024					
K	Cycles	Instr	Hits	Misses	Miss(%)	CPI	Cycles	Instr	Hits	Misses	Miss(%)	CPI
4	358207	46312	13772	2	0.0145 %	7,73	743426	96984	31675	3	0.0095 %	7,67
8	690349	38284	27760	2	0.0072 %	7,82	1438769	186012	65067	3	0.0046 %	7,74
16	1333162	170212	51131	2	0.0039 %	7,83	2811376	361956	121003	3	0.0025 %	7,77
32	2559284	326004	95563	2	0.0021 %	7,85	5487165	705396	230187	3	0.0013 %	7,78







Como ya es habitual, en la figura ⇝ vemos una tendencia lineal y creciente. Aunque, una pequeña desviación para **K**=32 es apreciable, pero nada extraño.

La evolución del CPI sigue la misma tendencia que se ah ido mencionando a lo largo de este informe. El CPI medio para el procesador escalar es de 7.8156, casi ×2,6 veces que la cifra obtenida para el procesador superescalar con el código base.

Aunque en esta arquitectura no se tenga pipe, las tendencias evolutivas respecto al procesador superescalar se mantienen. Observando la evolución lineal y creciente de los *hits* según aumenta **K**, se observa un *hit rate* medio del 99.9845 % muy parecido al obtenido con la arquitectura superescalar.

Lo mismo pasa con el *miss rate* la tendencia es decreciente y lineal (al igual que en el procesador superescalar). El *miss rate* medio que se ha obtenido es de 0.015742 %, similar al 0.013 % obtenido en la arquitectura super escalar. Cabe destacar que resulta extraño, que en ningún caso, los resultados obtenidos se aproximen al limite del 5 % visto en clase, pues se obtienen resultados muy inferiores.

3.0.2. Procesador superescalar

Los resultados de esta sección pueden verse en la primera sección del informe, pues la simulación realizada del código base es con una arquitectura superescalar.

3.1. Entregable

¿Se modifica el *miss rate*?

Si bien comparando los valores de ambas tablas vemos valores bastante parecidos, sí que obtenemos un valor de miss-rate ligeramente superior en el superescalar que en el escalar. Un procesador escalar ejecuta una instrucción a la vez, en secuencia. La tasa de fallos de caché en este tipo de procesadores dependerá principalmente de la localidad temporal y espacial del conjunto de instrucciones y datos.

En cambio, un procesador superescalar es capaz de ejecutar múltiples instrucciones en paralelo durante un solo ciclo de reloj, gracias a su diseño de múltiples unidades funcionales y a técnicas de planificación dinámica de instrucciones. Esto significa que un procesador superescalar puede requerir más accesos a la memoria caché por ciclo de reloj en comparación con un procesador escalar, lo que suele provocar un aumento en la tasa de fallos de caché si la localidad no es adecuada.

3.2. Entregable

Elabore gráficas para las diferentes jerarquías de memoria comparando el número transacciones en el bus de datos y el CPI, para los casos del filtro FIR analizados en la anterior sesión.

En esta sección hemos tenido en cuenta las dos arquitecturas de memoria mencionadas antes. La ICCM y la DCCM tienen una baja latencia de acceso es decir, que permiten leer o escribir datos en un solo ciclo. Sin embargo, a diferencia del `1$`, la ICCM y la DCCM se controlan mediante software.

3.2.1. DCCM

Ahora se habilita la DCCM en el sistema, para que la mayoría de los accesos a datos utilicen la DCCM (en lugar de la memoria DDR externa). Como se verá, este cambio aumenta el rendimiento. Además, la DCCM implementa *prefetching*.

Los resultados obtenidos con esta arquitectura se muestran en la siguiente tabla;

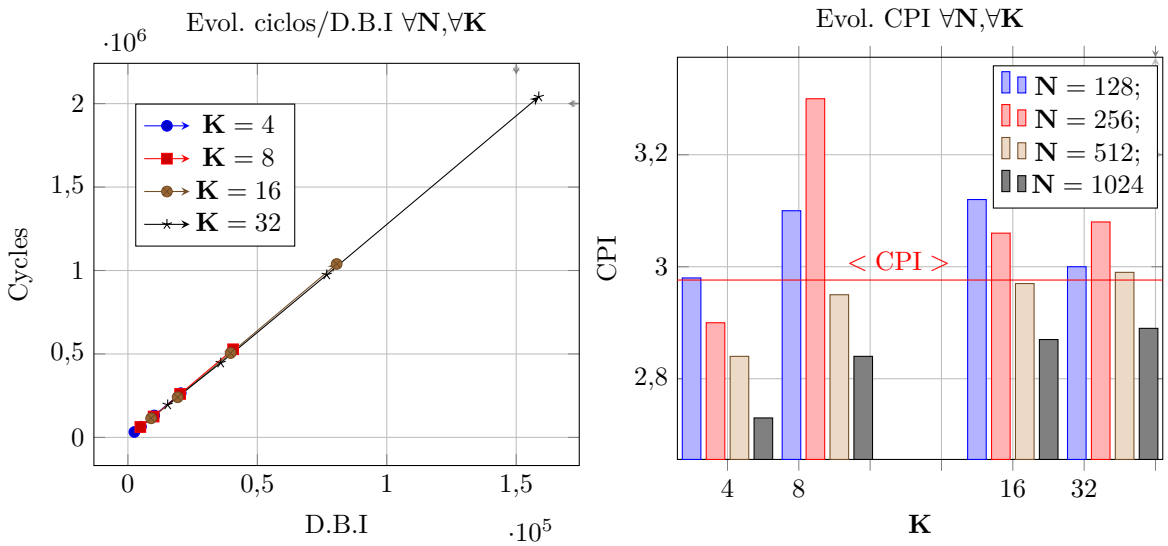
Cuadro 14: Resultados obtenidos con la arquitectura `DCCM`

	N = 128					**N** = 256				
K	Cycles	Instr	D.B.I	I.B.T	CPI	Cycles	Instr	D.B.I	I.B.T	CPI
4	32268	10844	2484	16	2,98	63900	21980	5044	16	2,90
8	62317	20096	4804	16	3,10	124958	41472	9924	16	3,30
16	114412	36680	8964	16	3,12	240467	78536	19204	16	3,06
32	195394	65144	15364	16	3,00	446428	144984	35844	16	3,08

Cuadro 15: Resultados obtenidos con la arquitectura `DCCM`

	N = 512					**N** = 1024				
K	Cycles	Instr	D.B.I	I.B.T	CPI	Cycles	Instr	D.B.I	I.B.T	CPI
4	131455	46284	10164	16	2,84	264919	96956	20404	16	2,73
8	260345	88256	20164	16	2,95	527990	185984	40644	16	2,84
16	505347	170184	39684	16	2,97	1039188	361928	80644	16	2,87
32	974258	325976	76804	16	2,99	2040636	705368	158724	16	2,89

A continuación se adjuntan las gráficas correspondientes:



Como podemos ver, si uno representa los ciclos frente a los *Data Instruction Bus* puede observar una tendencia creciente y lineal, i.e: según aumenta el número de ciclos, también aumenta el número de instrucciones de datos en el bus, de manera lineal. Comparando el CPI obtenido con esta arquitectura de memoria respecto a la arquitectura superescalar de la primera sesión y respecto a las dos técnicas de optimización, se obtiene un CPI peor respecto a las dos técnicas de optimización, pero si se compara con la arquitectura superescalar de la primera sección, se observa una ligera mejora. En este caso el CPI medio es de 2.98.

3.2.2. ICCM

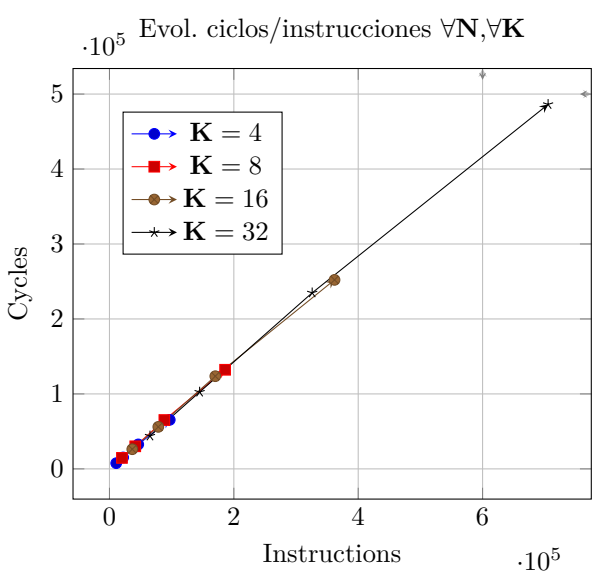
Los resultados obtenidos con esta arquitectura se muestran en las siguientes tablas;

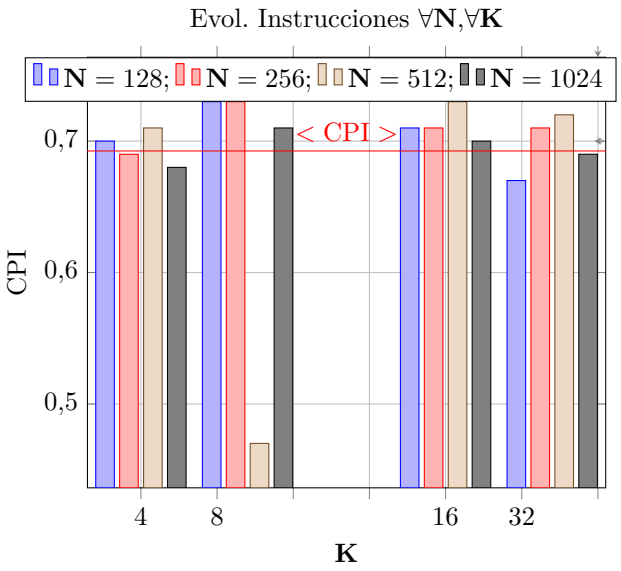
Cuadro 16: Resultados obtenidos con la arquitectura `ICCM`

	N = 128					**N** = 256				
K	Cycles	Instr	D.B.I	I.B.T	CPI	Cycles	Instr	D.B.I	I.B.T	CPI
4	7571	10844	0	0	0,70	15259	21980	0	0	0,69
8	14636	20096	0	0	0,73	30124	41472	0	0	0,73
16	26212	36680	0	0	0,71	56036	78536	0	0	0,71
32	43913	65144	0	0	0,67	102484	144984	0	0	0,71

Cuadro 17: Resultados obtenidos con la arquitectura `ICCM`

	N = 512					**N** = 1024				
K	Cycles	Instr	D.B.I	I.B.T	CPI	Cycles	Instr	D.B.I	I.B.T	CPI
4	32643	46284	0	0	0,71	65414	96956	0	0	0,68
8	65143	88256	0	0	0,47	132217	185984	0	0	0,71
16	123625	170184	0	0	0,73	252125	361928	0	0	0,70
32	234841	325976	0	0	0,72	486205	705368	0	0	0,69





El CPI ha mejorado considerablemente respecto al resto de arquitecturas, pues es $\times 5$ veces menor que el resto aproximadamente. Esto es lógico, pues no se obtiene ninguna transacción en el bus, debido a la arquitectura, pues las instrucciones se quedan cargadas en el módulo `Instruction Closely Coupled Memory (onchip)` que hace que los ciclos se reduzcan. El CPI medio en este caso es de 0.6925.

Por último, para los casos del filtro analizados en la anterior sesión, se han aplicado diferentes niveles de optimización en el compilador: **-O2** y **-O3**. Para ello se ha modificado el fichero `platformio.ini`, añadiendo el nivel de optimización en `build_flags` y activando el script `link_DCCM.ld` incluido en la línea 18. Para este último paso se ha hecho uso del `.bit` del SoC original `(rvfpganexys.bit)`.

A continuación se muestran las medidas que se han tomado:

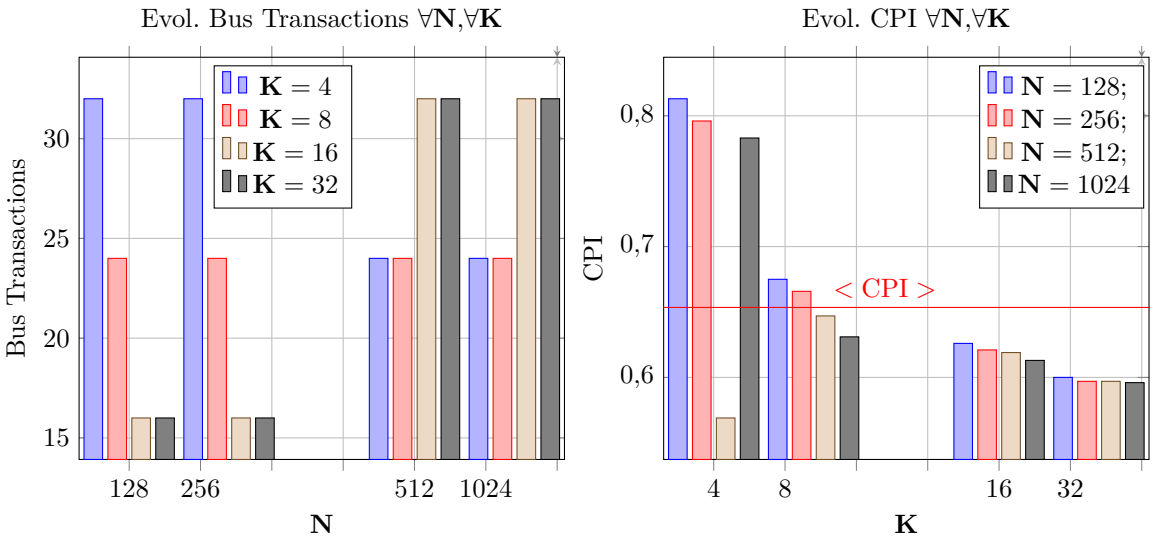
Cuadro 18: Resultados obtenidos con la optimización **-O2**.

	N = 128				**N** = 256			
K	Cycles	Instr	Bus Transac	CPI	Cycles	Instr	Bus Transac	CPI
4	4210	5179	32	0,813	8305	10427	32	0,796
8	5756	8527	24	0,675	11644	17487	24	0,6658
16	8912	14227	16	0,626	18761	30227	16	0,621
32	13894	23139	16	0,600	31941	53475	16	0,597

Cuadro 19: Resultados obtenidos con la optimización **-O2**.

	N = 512				**N** = 1024			
K	Cycles	Instr	Bus Transac	CPI	Cycles	Instr	Bus Transac	CPI
4	11897	20927	24	0,569	32831	41920	24	0,783
8	22904	35411	24	0,647	44940	71252	24	0,631
16	38508	62231	32	0,619	77408	126232	32	0,613
32	68101	114151	32	0,597	140234	235496	32	0,596

A continuación se adjuntan las figuras correspondientes;



Se observa como el CPI mejora según aumenta **K**, además en la figura de arriba a la derecha puede observarse como el CPI medio en este caso es de 0.6535. Comparando este CPI con el obtenido en casos anteriores, por ejemplo con el CPI del caso anterior para la arquitectura de memoria ICCM, se ve como hay una diferencia de 0.04 en el CPI, algo pequeño. Si se compara con el caso anterior para la arquitectura de memoria de DCCM, donde el CPI medio era 2.98, se observa una mejora considerable que porcentualmente se traduce en una mejora del $\frac{2,98-0,6535}{2,98} \cdot 100 = 78,07\,\%$ una mejora razonable, que definitivamente merece la pena. Aunque para verlo mejor habría que estudiar los *hits* y *misses*.

Estudiando la evolución de las transacciones del bus respecto a **N**, se observa como para **N**=128 y **N**=256 las transacciones en el bus son mayores para los dos valores más bajos de **k** mientras que para **N**=512 y **N**=1024 la tendencia es inversa, pues se obtienen más transacciones en el bus para los dos valores más altos de **K**. Pero se observan más transacciones en el bus para los dos valores más altos de **N**, pero estos valores no sobrepasan los valores máximo obtenidos para **N**=128 y **N**=256.

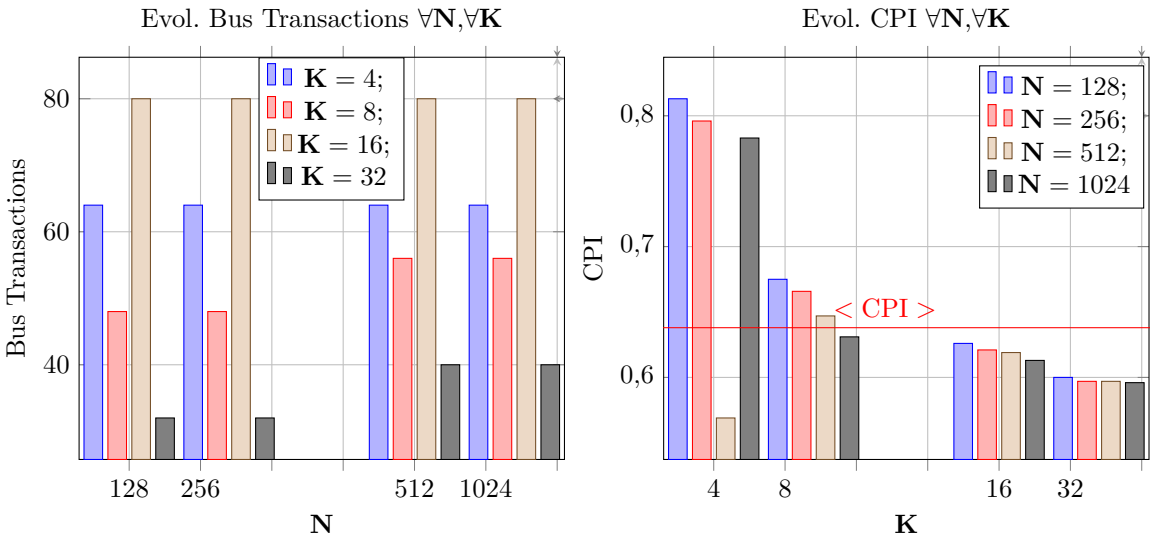
Cuadro 20: Resultados obtenidos con la optimización **-O3**.

	N = 128				**N** = 256			
K	Cycles	Instr	Bus Transac	CPI	Cycles	Instr	Bus Transac	CPI
4	2294	3634	64	0,630	4291	7218	64	0,595
8	4267	5894	48	0,724	8364	11910	48	0,702
16	5592	9172	80	0,610	11214	19156	80	0,585
32	17748	27530	32	0,645	40021	62858	32	0,637

Cuadro 21: Resultados obtenidos con la optimización **-O3**.

	N = 512				**N** = 1024			
K	Cycles	Instr	Bus Transac	CPI	Cycles	Instr	Bus Transac	CPI
4	10264	14384	64	0,714	15170	28728	64	0,528
8	16559	23947	56	0,691	31954	48010	56	0,6655
16	22471	34122	80	0,659	46025	79065	80	0,582
32	84540	133517	40	0,633	170590	274830	40	0,621

A continuación adjuntamos las figuras correspondientes;



En este caso el CPI medio es de 0.638, si comparamos este resultado con el obtenido para la optimización **-O2** observamos una cierta mejora en el CPI, pero no notable. Si el CPI obtenido en este caso es comparado con el obtenido sin aplicar las técnicas de optimización, se observa como para la arquitectura de memoria DCCM la mejora del CPI es de un $\frac{2,98-0,638}{2,98} \cdot 100 = 78,59\,\%$, esta es una gran mejora, al igual que pasaba con la optimización **-O2**. Si se compara el CPI obtenido en esta sección con el obtenido con la arquitectura de memoria ICCM se observa una mejora del $\frac{0,6925-0,638}{0,6925} \cdot 100 = 7,87\,\%$, una mejora del mismo orden de magnitud se observaba en el anterior apartado para la misma arquitectura de memoria.

En general se observa como los mejores resultados se obtienen para la arquitectura ICCM, pero ¿a costa de que? No se ha querido concluir diciendo que esta arquitectura de memoria sería la mejor para este caso, porque se desconoce el coste de su implementación y muchos otros factores que no hemos podido analizar, pues no se ha encontrado mucha información acerca de ellas en los PDFs de los laboratorios 19 y 20.

4. Conclusiones

Podemos sacar una serie de conclusiones a partir de la realización de la práctica:

- La jerarquía de memoria y su configuración tienen un impacto significativo en el rendimiento de un sistema. Un diseño adecuado y una optimización de la jerarquía de memoria pueden mejorar el rendimiento general del sistema y reducir la latencia de acceso a los datos e instrucciones.
- Los fallos de caché en la caché de instrucciones pueden ser analizados y minimizados mediante la optimización del código y ajustando los parámetros de la caché. La localidad espacial y temporal son factores clave en la tasa de fallos de caché.
- La modificación de bucles mediante transformaciones como el software pipelining y el loop unrolling puede mejorar la eficiencia de la caché de instrucciones y afectar el CPI (Ciclos Por Instrucción). Sin embargo, el impacto de estas transformaciones puede variar según el caso concreto, y es esencial analizar y medir los resultados para comprender su efecto en el rendimiento.
- Las optimizaciones del compilador pueden influir en la jerarquía de memoria y en el rendimiento del SoC (System on a Chip). Estas optimizaciones pueden mejorar el rendimiento de la memoria y reducir la latencia, pero es importante considerar las posibles compensaciones en términos de complejidad del código y tamaño del binario.

  En resumen, la práctica demuestra la importancia de comprender la jerarquía de memoria y cómo el código y las optimizaciones afectan el rendimiento final. La experimentación con diferentes configuraciones y transformaciones de código puede proporcionar información valiosa para mejorar el rendimiento del sistema y minimizar la tasa de fallos de caché.