Proyecto Hardware 2021-2022

- Práctica 1 -

Desarrollo de código para el procesador ARM

Jorge Lisa 774248
David Zandundo 780500

Índice

Resumen Ejecutivo

En esta práctica se ha realizado un código capaz guardar en cada celda de un sudoku los candidatos, tanto en C como en Ensamblador.

Se ha creado una función extra en el archivo de celda para que se pueda leer el valor de una celda sin que sudoku tenga que saber cómo está organizada por dentro.

Los ficheros en ensamblador se han programado cada uno en un fichero distinto para que sean accesibles tanto desde C como Ensamblador.

En la función puramente en ensamblador no se ha hecho inline de propagar, porque esta usaba todos los registros disponibles y no nos quitábamos el tener que guardar en memoria los registros de la función actualizar arm. Estuvimos buscando una forma eficiente de guardar los registros para que no se tengan que guardar en memoria, pero nos costó mucho modificar el código de propagar y al final decidimos dejarlo como antes.

Para saber en qué celda del sudoku nos toca acceder desde ensamblador, se han usado desplazamientos sabiendo cómo está organizada la memoria. Si no hubiera estado organizada de esta manera, el cambio de columna hubiera sido el mismo, debido a que el tamaño de la palabra seguiría siendo el mismo, pero para cambiar de fila tendríamos que haber usado funciones de multiplicación, que son más pesadas en temas de eficiencia temporal del código.

No se ha usado un vector de la función propagar para la versión en arm y en cambio se ha encontrado una forma de seguir el mismo procedimiento pero sin tener que acceder a memoria, haciendo parcialmente más eficiente esta función.

Cuando se midieron los tiempos del código, se encontró que el compilador optimiza mucho el código y hace inline de la función actualizar_c. Nosotros no conseguimos optimizarlo tanto, ya que no usábamos formas de optimización muy avanzadas, aunque se pensó en hacer loop unrolling.

Por último, concluimos que podríamos mejorar este código algo más, aunque aumentáramos el tamaño del mismo. Sin embargo, se han encontrado estrategias que, si en el futuro nos sobra algo de tiempo, se intentarán poner en práctica.

Introducción

En esta primera práctica vamos a optimizar el rendimiento del juego del sudoku acelerando las funciones computacionalmente más costosas. A partir del código facilitado en C, se desarrollará código para un procesador ARM y se ejecutará sobre un emulador de un procesador ARM.

Se crearán versiones ARM de las funciones ya implementadas en C, habrá distintas versiones donde las funciones en C llamaran a las funciones implementadas en ARM y viceversa. Estas funciones serán candidatos_actualizar() y candidatos_propagar().

Para ello primero buscamos entender el funcionamiento del sistema y el comportamiento de las funciones más importantes. Posteriormente evaluaremos el rendimiento y estudiaremos las opciones de optimización del compilador para mejorar las prestaciones del código generado.

Objetivo

El objetivo principal de esta práctica era desarrollar tanto en C como ARM un código capaz de inicializar y propagar los posibles candidatos de un sudoku convencional 9x9 y combinar ambos lenguajes para que desde un lenguaje se pueda llamar al otro.

Por último había que observar los tiempos de ejecución obtenidos de todas la funciones realizadas, optimizar lo máximo que se pudiese el código en ARM y examinar el funcionamiento de las opciones de optimización del compilador de C.

Metodología

Como primer paso se ha estudiado el código inicial en C y la especificación de las funciones candidatos_actualizar_c() y candidatos_propagar_c(). Después se ha implementado la función candidatos_actualizar_c() desarrollado en lenguaje C. Por último, se implementó las funciones en arm y se optimizaron.

candidatos_actualizar_c()

```
/* calcula todas las listas de candidatos (9x9)
 * necesario tras borrar o cambiar un valor (listas corrompidas)
 * retorna el numero de celdas vacías */

/* Init del sudoku en código C invocando a propagar en C
 * Recibe la cuadricula como primer parámetro
 * y devuelve en celdas_vacias el número de celdas vacías
static int candidatos_actualizar_c(CELDA cuadricula[NUM_FILAS][NUM_COLUMNAS])
{
	int celdas_vacias = 0;
	uint8_t i;
	uint8_t j;

	for (i = 0; i < 9; ++i){
		for (j = 0; j < 9; ++j){
			celda_activar_todos_candidatos(&cuadricula[i][j]);
		}
	}

	for (i = 0; i < 9; ++i){
		for (j = 0; j < 9; ++j){
			if (celda_leer_valor(cuadricula[i][j]) != 0 ){
				candidatos_propagar_c(cuadricula,i,j);
			}
			else{
				celdas_vacias++;
			}
		}
	}
return celdas_vacias;
}
```

Primero se crea un bucle que recorre todas las celdas del sudoku, para activar todos los candidatos posibles en todas las celdas. Para activar todos los candidatos se ha creado una función auxiliar llamada celda_activar_todos_candidatos() en celda.h cuyo código es el siguiente:

```
__inline static void
celda_activar_todos_candidatos(CELDA *celdaptr)
{
   *celdaptr = *celdaptr & 0x007F;
}
```

Para activar todos los candidatos solo hace falta poner todos los bits del vector de candidatos a 0 y mantener el resto de bits como están.

Se ha puesto inline, ya que es una función muy pequeña que se llama muchas veces.

Después de este bucle, para activar todos los candidatos, se crea otro que recorre también todas las celdas, si se trata de una pista (esto se comprueba si la celda tiene un valor predefinido) entonces se llama a la función candidatos_propagar_c() para propagar el valor de la celda para actualizar las listas de candidatos de las celdas en su su fila, columna y cuadrante. En el caso de que no sea una pista, se aumenta la variable que cuenta las celdas vacías. Finalmente, la función devuelve el valor de esa variable.

candidatos_actualizar_arm_c()

Para la función candidatos_actualizar_arm_c() se ha cogido el código creado en la función candidatos_actualizar_c() en C y se ha traducido al lenguaje ARM.

El código ha quedado de la siguiente manera:

```
; Calcula todas las listas de candidatos (9x9)
; necesario tras borrar o cambiar un valor (listas corrompidas)

; Recibe la cuadrícula como primer parámetro
; y devuelve el número de celdas vacías

        MOV         IP,SP
        STMDB   SP!,{R4-R10,FP,IP,LR}          ;Prólogo de la función

; INICIALIZACION
        MOV         R6,R0                       ;R0=entrada  R6=cuadrícula
        MOV         R7,#0x0                     ;R7=celdas_vacias
        MOV         R4,#0x0                     ;R4=i
        MOV         R5,#0x0                     ;R5=j
        LDR         R8, = candidatos_propagar_c ;R8= @candidatos_propagar_c
; EJECUCION
bucle1
        ADD         R1,R6,R4,LSL #5             ;valor cuadrícula + (i << 5 izq)
        ADD         R1,R1,R5,LSL #1             ;valor de r1 + (j << 1)
        LDRB        R0,[R1]                     ;valor de r1 (cuadricula[i][j])
        AND         R0,R0,#0x7F                 ;activar todos los candidatos
        STRH        R0,[R1]                     ;guarda en [r1] ½ palabra de r0

        ADD         R5,R5,#0x1                  ; j++
```

```
        CMP     R5,#0x9                 ;compara j con 9
        BLT     bucle1                  ;salta a bucle1 si es menor
        MOV     R5,#0                   ;pone j=0

        ADD     R4,R4,#0x1              ;i++
        CMP     R4,#0x9                 ;comara i con 9
        BLT     bucle1                  ;salta a bucle1 si es menor
        MOV     R4,#0                   ;pone i=0

bucle2
        ADD     R1,R6,R4,LSL #5         ;valor cuadrícula + (i << 5 izq)
        ADD     R1,R1,R5,LSL #1         ;r1 + (j << 1)
        LDRH    R0,[R1]                 ;valor de r1 (cuadricula[i][j])

        AND     R0,R0,#0xF              ;número que contiene la celda
        CMP     R0,#0x0                 ;si la celda está vacía
        ADDEQ   R7,R7,#1                ;celdas_vacias++ si no vacía

        MOVNE   R0,R6                   ;Se prepara la entrada
        MOVNE   R1,R4                   ;si se va a llamar; de lo
        MOVNE   R2,R5                   ;contrario se sigue en el bucle

        MOVNE   LR,PC                   ;Guarda el PC actual
        BXNE    R8                      ;llama candidatos_propagar_c
        ADD     R5,R5,#0x1              ;j++
        CMP     R5,#0x9                 ;compara j con 9
        BLT     bucle2                  ;salta a bucle2 si es menor
        MOV     R5,#0                   ;pone j=0

        ADD     R4,R4,#0x1              ;i++
        CMP     R4,#0x9                 ;compara i con 9
        BLT     bucle2                  ;salta a bucle2 si es menor

        MOV     R0, R7                  ;pone en return celdas_vacias
        LDMIA   SP,{R4-R10,FP,SP,PC}    ;Epílogo de la función
```

Para saltar a la función propagar en C se guarda en un registro, tras el prólogo, la dirección donde se encuentra la misma y cuando sea necesario, hacer un “branch and exchange” usando como parámetro el registro en cuestión. La función de C debe haber sido exportada con antelación al principio del archivo assembly.

Se ha predicado cada comparación, ya que es mucho más efeciente y permite llamar a los Saltos condicionados con parámetros de entrada de una forma más sencilla.

Hay dos bucles, que hacen lo mismo que en el código en C. A estos solo se salta cuando j es menor que el número de columnas (para bucle1) y cuando i es menor al número de filas (en bucle2)

Para acceder a las celdas que nos corresponden dentro de una cuadrícula, se usan desplazamientos. Esto se debe a que las direcciones de cada una de las celdas se

encuentran perfectamente alineadas de una forma que, usando desplazamientos a la izquierda constantes, podemos acceder sin problemas a cada uno de sus valores.

candidatos_actualizar_c_arm()

Esta función es idéntica a candidatos_actualizar_c() pero en vez de llamar a candidatos_propagar_c() llama a candidatos_propagar_arm() que es su versión programada en ARM.

Para esto se ha necesitado importar ateriormente la función correspondiente.

```
/* Init del sudoku en código C invocando a propagar en arm
 * Recibe la cuadricula como primer parámetro
 * y devuelve en celdas_vacias el número de celdas vacías
 */
static int
candidatos_actualizar_c_arm(CELDA cuadricula[NUM_FILAS][NUM_COLUMNAS])
{
        int celdas_vacias = 0;
        uint8_t i;
  uint8_t j;

        for (i = 0; i < 9; ++i){
                for (j = 0; j < 9; ++j){
                        celda_activar_todos_candidatos(&cuadricula[i][j]);
                }
        }

        for (i = 0; i < 9; ++i){
                for (j = 0; j < 9; ++j){
                        if (celda_leer_valor(cuadricula[i][j]) != 0 ){
                                candidatos_propagar_arm(cuadricula,i,j);
                        }else{
                                celdas_vacias++;
                        }
                }
        }
        return celdas_vacias;
}
```

candidatos_propagar_arm()

Para la función candidatos_propagar_arm() se ha cogido el código creado en la función candidatos_propagar_c() en C y se ha traducido al lenguaje ARM.

```
; Propaga el valor de una determinada celda
; para actualizar las listas de candidatos
; de las celdas en su su fila, columna y región

; Recibe como parámetro la cuadrícula, y la fila y columna de
; la celda a propagar; no devuelve nada

        MOV         IP,SP
        STMDB       SP!,{R4-R10,FP,IP,LR}       ;Prólogo de la función

; INICIALIZACIÓN
        MOV         R6,R0                       ;R0=entrada   R6=cuadrícula
        MOV         R7,R1                       ;R1=entrada   R7=fila
        MOV         R8,R2                       ;R2=entrada   R8=columna
        MOV         R4,#0x0                     ;R4=i
        MOV         R5,#0x0                     ;R5=j

; leer valor
        ADD         R12,R6,R7,LSL #5
        ADD         R0,R12,R8,LSL #1            ;R0=dir(celda[fila][columna])
        LDRB        R0,[R0]                     ;primer byte de celda[fila][columna]
        AND         R3,R0,#0xF                  ;Valor(celda[fila][columna])
        MOV         R10, #0x1

bucle_columnas
        ADD         R9,R12,R5,LSL #1
        LDRH        R0,[R9]                     ;mete en r0 la celda

;celda_eliminar_candidato(&cuadricula[fila][j],valor)
        ADD         R1, R3, #6
        MOV         R1, R10, LSL R1             ;se desplaza el bit a la pos correcta
        ORR         R0, R0, R1                  ;del candidato a eliminar, es decir,
        STRH        R0, [R9]

        ADD         R5, R5, #0x1                ;j++
        CMP         R5, #0x9                    ;compara j con 9
        BLT         bucle_columnas              ;salta a bucle_columnas si es menor

        ADD         R12,R6,R8,LSL #1            ;R12=cuadricula[i][columna]
        MOV         R10, #0x1

bucle_filas
        ADD         R9,R12,R4,LSL #5
        LDRH        R0,[R9]                     ;mete en r0 la celda

;celda_eliminar_candidato(&cuadricula[fila][j],valor)
        ADD         R1, R3, #6
        MOV         R1, R10, LSL R1             ;se desplaza el bit a la pos correcta
        ORR         R0, R0, R1
        STRH        R0, [R9]

        ADD         R4, R4, #0x1                ;i++
        CMP         R4, #0x9                    ;compara i con 9
        BLT         bucle_filas                 ;salta a bucle_filas si es menor
```

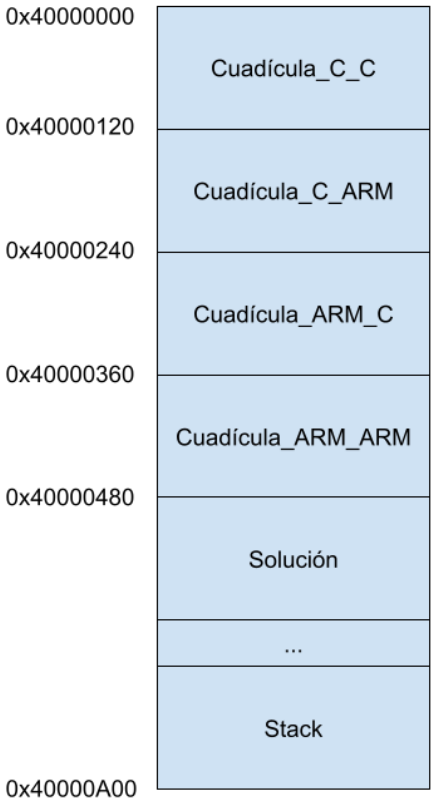
```
        MOV         R1,#0                   ;R1=init_i
        MOV         R2,#0                   ;R2=end_i
        MOV         R10,#0                  ;R10=init_j
        MOV         R11,#0                  ;R11=end_j

;init_i = init_region[fila];
        CMP         R7,#3                   ;comparamos fila>=3
        MOVHS       R1,#3                   ;init_i = init_region[fila];
        CMP         R7,#6                   ;comparamos fila>=6
        MOVHS       R1,#6
        ADD         R2,R1,#3                ;end_i = init_i + 3;

;init_j = init_region[columna];
        CMP         R8,#3                   ;comparamos fila<6
        MOVHS       R10,#3                  ;init_j = init_region[columna];
        CMP         R8,#6                   ;comparamos fila<6
        MOVHS       R10,#6
        ADD         R11,R10,#3              ;end_j = init_j + 3;

;ini de la cuadrícula a cuadricula[init_i][init_j]
        MOV         R4,R1                   ;i=init_i
        MOV         R5,R10                  ;j=init_j
        MOV         R8,#0x1

bucle_cuadrado
        ADD         R9,R6,R4,LSL #5         ;bucle para recorrer la región
        ADD         R9,R9,R5,LSL #1
        LDRH        R0,[R9]                 ;mete en r0 el contenido de la celda

;celda_eliminar_candidato(&cuadricula[fila][j],valor)
        ADD         R7,R3,#6
        MOV         R7,R8,LSL R7            ;se desplaza el bit a la pos correcta
        ORR         R0,R0,R7                ;del candidato a eliminar, es decir,
        STRH        R0,[R9]

        ADD         R4,R4,#0x1              ;i++
        CMP         R4,R2                   ;comparamos i con end_i
        BLT         bucle_cuadrado          ;salta a bucle_cuadrado si es menor

        MOV         R4,R1                   ;i=init_i
        ADD         R5,R5,#0x1              ;j++
        CMP         R5,R11                  ;comparamos j con end_j
        BLT         bucle_cuadrado          ;salta a bucle_cuadrado si es menor

        MOV         R0,#0x0
        LDMIA   SP,{R4-R10,FP,SP,PC}        ;Epílogo de la función
```

Para esta función en vez de crear un vector de enteros y guardarlo en memoria como se hace en la función de C, se ha desarrollado un algoritmo que compara el valor de la celda y asigna el valor correspondiente a las variables init_i, init_j, end_i, y end_j. Con esto conseguimos reducir las instrucciones con transferencia de datos a memoria y el espacio que ocuparía el vector en memoria.

No se ha comprobado que los enteros son de tipo uint_8, ya que en ningún momento vamos a sobrepasar los 8 bytes permitidos, dado que estas variables se crean y destruyen dentro de esta función y no son parámetros.

En esta función, los bucles tienen un nombre más descriptivo, y durante los primeros, se usa R10 para hacer de bit de eliminación de candidato. Posteriormente se cambia a R8 debido a una escasez de los mismos para realizar cálculos.

Mapa de memoria



Descripción de las optimizaciones realizadas al código ensamblador

Para la optimización del código lo primero que se ha hecho ha sido diseñar el código de las funciones antes de escribirlo, esto ha supuesto que en algunas ocasiones nos ahorremos repetir algunas funciones innecesarias.

Las funciones más costosas son las que acceden a memoria (Store y Load). Se ha intentado reducir el uso de estas instrucciones lo máximo posible. Se ha probado a utilizar las instrucciones que operan con la palabra entera (LDR y STR), sobre todo cuando estas instrucciones estaban dentro de los bucles, pero al final no se ha podido hacer porque suponía un aumento considerable de otras instrucciones.

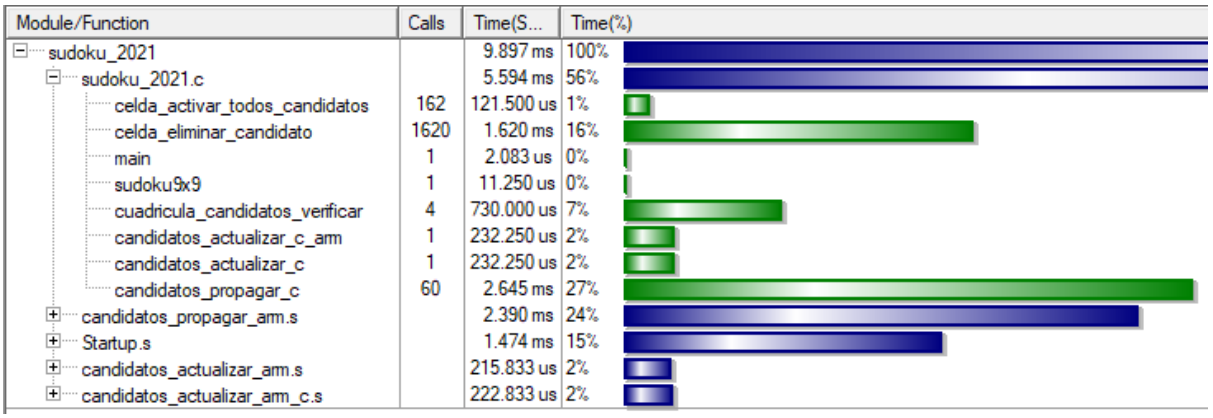
Al comienzo y final de las funciones escritas en código ARM (candidatos_propagar_arm() y candidatos_actualizar_arm() y candidatos_actualizar_arm_c()) se han utilizado instrucciones de transferencia de datos múltiples como LDMIA y STMDB que permiten que una única instrucción mueva varios datos entre la memoria y los registros, esto ha mejorado el rendimiento a la hora de llamar a las funciones.

Se han intentado utilizar lo máximo posible las instrucciones con ejecución condicional, esto ha reducido considerablemente el número de instrucciones, sobre todo las instrucciones de salto que son de las más costosas.

A la hora de calcular las direcciones de las celdas a las que se querían acceder en memoria se han utilizado las opciones de desplazamiento para multiplicar o el barrel shifter (BS), esto ha permitido reducir el número de instrucciones.

Análisis de rendimiento

Una vez completado el código de todas las funciones y comprobado que los resultados obtenidos son los correctos, se ha comprobado el tiempo de ejecución de las funciones más críticas, el resultado se muestra a continuación:



Module/Function	Calls	Time(S...	Time(%)
sudoku_2021		9.897 ms	100%
sudoku_2021.c		5.594 ms	56%
celda_activar_todos_candidatos	162	121.500 us	1%
celda_eliminar_candidato	1620	1.620 ms	16%
main	1	2.083 us	0%
sudoku9x9	1	11.250 us	0%
cuadricula_candidatos_verificar	4	730.000 us	7%
candidatos_actualizar_c_arm	1	232.250 us	2%
candidatos_actualizar_c	1	232.250 us	2%
candidatos_propagar_c	60	2.645 ms	27%
candidatos_propagar_arm.s		2.390 ms	24%
Startup.s		1.474 ms	15%
candidatos_actualizar_arm.s		215.833 us	2%
candidatos_actualizar_arm_c.s		222.833 us	2%

Como se observa la función que más tiempo se ejecuta es la de candidatos_propagar(), siendo la versión ARM más eficiente que la versión en C. Esto se debe a que esta función es llamada repetidamente por candidatos_actualizar() y además tiene un coste computacional alto.

En lo que se refiere a la función candidatos_actualizar(), se ha conseguido que la versión ARM sea más eficiente que las demás versiones en las que combinan con código en C y la propia función que solo tiene código en C.

Después de analizar el tiempo de ejecución de las funciones se ha comprobado el tamaño del código en bytes de cada versión de las funciones realizadas. Para ello se ha contado el número de instrucciones que el compilador genera cuando se compila las funciones escritas en C y las instrucciones de las funciones en ARM.

candidatos_actualizar_c()	≃ 50 instrucciones ARM
candidatos_actualizar_arm()	38 instrucciones ARM
candidatos_propagar_c()	≃ 105 instrucciones ARM
candidatos_propagar_arm()	65 instrucciones ARM

Como se observa, se ha podido reducir el número de instrucciones en las funciones de ARM en comparación a las instrucciones que genera el compilador de C.

Para calcular lo que ocupa cada instrucción en memoria solo hace falta multiplicar el número de instrucciones por cuatro que es el número de bytes que ocupa cada instrucción.

Si hacemos la comparación con las cuatro principales instrucciones que combinan ARM y C nos queda esta tabla:

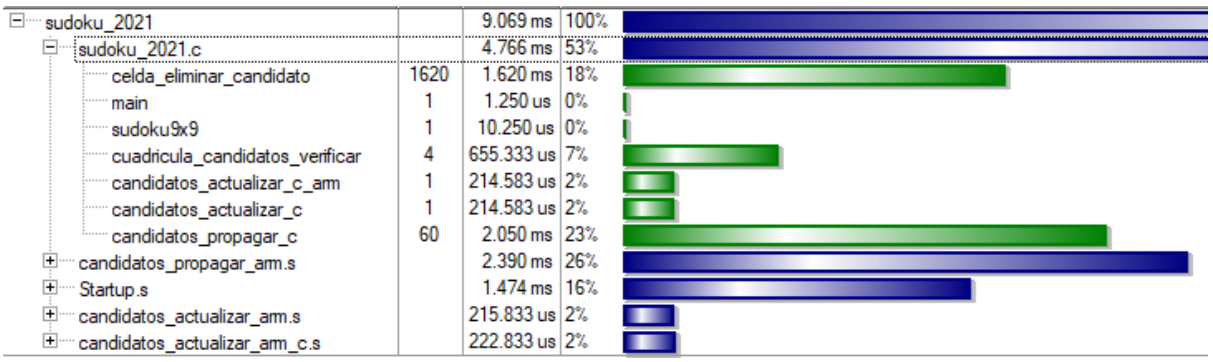
candidatos_actualizar_c()	50 instr. + 105 instr. ≃ 155 instr. ARM
candidatos_actualizar_c_arm()	50 instr. + 65 instr. ≃ 115 instr. ARM
candidatos_actualizar_arm_c()	40 instr. + 105 instr. ≃ 145 instr. ARM
candidatos_actualizar_arm_arm()	38 instr + 65 instr. ≃ 103 instr. ARM

Como se observa, la función que menos ocupa en memoria es la que ha sido programada únicamente en ARM.

Más adelante se ha compilado con las diferentes activaciones de flags, para generar código eficiente y no fácilmente depurable, estos flags aplican heurísticas de optimización de código buscando mejorar la velocidad o el tamaño del código.

-01

Este ha sido el resultado en tiempo de compilar con el flag -01:



Como se observa, las funciones que estaban escritas en C ahora tienen un tiempo de ejecución menor que antes. Incluso han llegado a tener un tiempo menor que las funciones programadas en ARM.

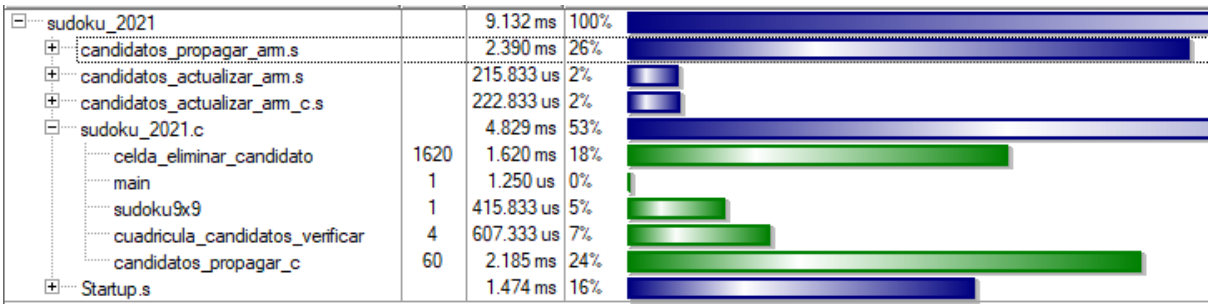
También se ha reducido el número de instrucciones que se generan al compilar las instrucciones, como se muestra a continuación:

candidatos_actualizar_c()	≃ 41 instrucciones ARM
candidatos_propagar_c()	≃ 65 instrucciones ARM

Sobre todo se ve que la función que más se ha disminuido es candidatos_propagar_c() que casi se ha reducido a la mitad de las instrucciones que se generaban en la compilación anterior. Esto más o menos iguala al número de instrucciones de la función candidatos_propagar_arm() que ha sido programada. En menor medida ha reducido el número de instrucciones en candidatos_actualizar_c().

-02

Este ha sido el resultado en tiempo de compilar con el flag -02:



Como se observa, las funciones que estaban escritas en C ahora tienen un tiempo de ejecución menor que con el anterior flag. Aunque tampoco se ha reducido tanto tiempo.

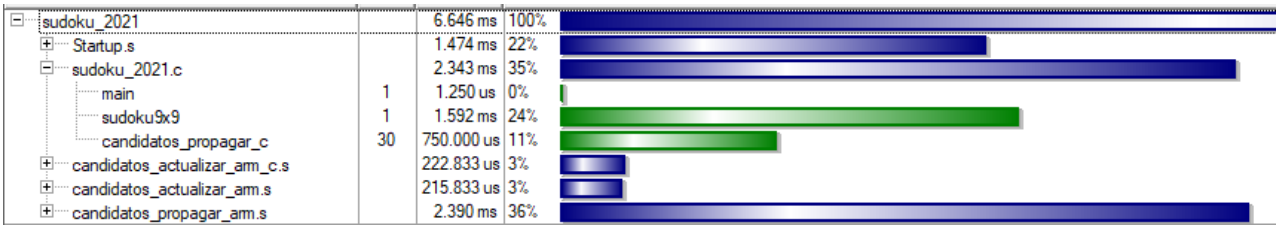
También se ha reducido el número de instrucciones que se generan al compilar las instrucciones, como se muestra a continuación:

candidatos_propagar_c()	≃ 60 instrucciones ARM

Con este flag, el compilador ha hecho inline de la función actualizar en sudoku9x9, cosa que no ha resultado ser eficiente ya que ha aumentado su tiempo de ejecución casi tanto como el flag -O0.

-03

Este ha sido el resultado en tiempo de compilar con el flag -03:



Tal y como se esperaba, se ha generado el código más eficiente en comparación con los anteriores porque el compilador aplica unas heurísticas más fuertes que las anteriores. Esta vez sí que ha supuesto una reducción en el tiempo de ejecución importante.

Con este flag, el compilador hace inline de la función actualizar_c y eliminar_candidato, llevando su tiempo de ejecución a la función sudoku9x9, aumentando considerablemente su coste en tiempo.

Finalemente, observando todas las ejecuciones, vemos que nuestro código en assembly es muy eficiente, ya que si sumamos en cada optimización el tiempo de sudoku9x9 y candidatos_propagar + candidatos_actualizar + Eliminar candidato de ambos 4 códigos, obtenemos que:

Compilación	Sudoku9x9	Actualizar	Propagar	Eliminar Candidato	Total
Nuestro en ARM	0.011ms	0.215ms	2.39ms	N/A	2.616ms
-O0	0.011ms	0.232ms	2.64ms	1.62ms	4.503ms
-O1	0.011ms	0.214ms	2.02ms	1.62ms	3.864ms
-O2	0.425ms	N/A	2.185ms	1.62ms	4.22ms
-O3	1.592ms	N/A	0.75ms	N/A	2.342ms

Podemos observar que hemos hecho un gran trabajo, ya que nosotros hacemos inline en ARM de la función eliminar candidato, y por lo tanto se el coste se encuentra integrado en la función de propagar.

Aunque tanto el tiempo en -O3 de tanto actualizar como eliminar han sido reducidos y llevados a sudoku, le aumenta tanto su coste que su optimización sería muy buena si otros programas usasen la función de propagar.

Principales problemas encontrados

Los mayores problemas han sido a la hora de trabajar con ARM. La comprensión de los Branch a otras funciones, el guardado de los registros anteriores y las optimizaciones no ha costado el mayor tiempo.
Respecto a problemas de fallos que no comprendíamos respecto al código solo hubo uno, respecto a la carga de los registros en el epílogo, que estuvimos usando LDMDB y no fucionaba, pero cambiando la instrucción por LDMIA se arregló todo.

Conclusiones

Se ha conseguido crear un código bastante eficiente, que aunque no es mejor que el flag -O3, con algún cambio creemos que seríamos capaces de acercarnos mucho o incluso superar su tiempo, ya que estos cambios se harían en la función de propagar.
También, gracias a esta práctica se ha conseguido entender el funcionamiento a bajo nivel del código en C, los flags de compilación y el funcionamiento del “inline”, el cual reduce considerablemente el tiempo de ejecución si se trata de una función que se llama muchas veces.