|  |
| --- |
| Universidad blas pascal |
| Trabajo Final |
| Programación eficiente |
|  |
| **Cenzano Sebastian. D’Alessandro Leonardo** |
| **03/02/2019** |

|  |
| --- |
|  |

# Consigna

El objetivo de este Trabajo Final es tomar un programa de alguna materia anterior, proyecto personal o descargado de Internet y analizarlo en detalle suﬁciente como para poder determinar qué correcciones, mejoras y optimizaciones se deberían aplicar. Se deberán analizar los siguientes puntos:

* Cobertura del código fuente,
* Detección de las funciones que realizan el mayor consumo de CPU,
* Redundancia de operaciones aritmético–lógicas,
* Modo de utilización de la memoria asignada en forma dinámica,
* Modo de acceso a la memoria caché y principal.

Las pruebas a realizar deberán estar sistematizadas para así tener una medición objetiva y repetible. Una vez analizado el software, se deberán proponer mejoras a realizarle. Luego, se procederá a su implementación. Finalmente, se deberá hacer un estudio comparativo entre el software original y su versión mejorada aplicando, en caso de ser posible, las mismas pruebas de software sistematizadas. En esta etapa se incluirán todas las tablas y gráﬁcos necesarios para mostrar el impacto de los cambios realizados.

# Problemática abordada

El programa a utilizar fue provisto en la materia “Algoritmos y estructuras de datos 2”. El mismo, utiliza el Algoritmo de Huffman para comprimir un archivo provisto como entrada. Además, cuenta con la opción de descomprimir el archivo utilizando el mismo Algoritmo.

En la fase de compresión, se toma un archivo binario, se analiza su contenido y se crea un árbol de Huffman cargando los nodos con la información de los bytes y su frecuencia de aparición. Luego, información como la tabla de símbolos y el vector que representa al árbol se guardan en el archivo a comprimir, terminando con el proceso de compresión.

Para descomprimir el proceso es, a grandes rasgos, el inverso.

Con el objetivo de utilizar las herramientas vistas durante el dictado de la materia, el programa se compilara, ejecutara y testeara en una distribución del sistema operativo Linux. El software a testear, originalmente programado para Windows, requirió cambios mínimos para que su uso sea el esperado, como ejemplo corregir las instrucciones “system(‘pause’)” mediantes las cuales el programa espera alguna acción de parte del usuario para seguir con la ejecución.

# Herramientas

Las herramientas a utilizar para el “*profiling”* son las siguientes:

* Gcov: Para medición de cobertura de código. Se incluirán los reportes generados por la herramienta.
* Gprof: Para obtener información sobre consumo de CPU, discriminado por función. Orden y cantidad de llamada de las funciones. Se incluirán los reportes generados.
* Valgrind: Para obtener información sobre manejo de memoria.

# Ejecución & diagnostico

Realizaremos pruebas, con las herramientas previamente mencionadas, en el software sin modificaciones para poder obtener un primer diagnostico.

Luego analizaremos los resultados y en base a estos, realizaremos las mejoras vistas durante la materia.

## Compilación

Para Compilar el software se utilizara el compilador “g++” en su versión 7.3.0 la cual utiliza por defecto el standard c++14.Realizamos una primera prueba de compilación con los siguientes parámetros:

g++ -o HuffmanCompressor.exe \*.cpp  
Y obtenemos los siguientes resultados.

object.cpp: In member function ‘virtual std::\_\_cxx11::string Object::toString()’:object.cpp:23:30: error: cast from ‘Object\*’ to ‘int’ loses precision [-fpermissive]  
 sprintf( cad, "%x", (int)this );  
 ^~~~  
object.cpp: In member function ‘virtual int Object::hashCode()’:  
object.cpp:32:17: error: cast from ‘Object\*’ to ‘int’ loses precision [-fpermissive]  
 return (int)this;  
 ^~~~

Estos errores se deben a que la implementación del software utiliza clases “Objeto” (emulando las superclases provistas por Java) que, en este caso, calculan el hash de un objeto simplemente “casteándolo” a un objeto del tipo int.Habiendo confirmado el correcto funcionamiento del programa, procederemos a utilizar el flag de compilación “-fpermissive” para seguir con las pruebas.

## Planificación de las pruebas

Los archivos de prueba consisten en archivos de imágenes con la extensión .jpg de tamaños variados, por lo que se creó un script de bash para la compilación y el uso de cada una de las herramientas de profiling y optimización:

* Script-GCOV.sh
* Script-GPROF.sh
* Script-Valgrind.sh

## Cobertura

### Configuración

Para realizar las pruebas dinámicas de cobertura del cogido, utilizaremos la herramienta de GCC llamada GCOV. Con los resultados de estas pruebas podremos remover código que no se utiliza y entender cuáles son las áreas más utilizadas del código, y que por lo tanto, mejores resultados traerán al mejorar su performance.Como mencionamos previamente, las pruebas se normalizaron con la utilización de scripts y los pasos se describen a continuación.  
  
La compilación se realizo con los siguientes argumentos:

g++ -ftest-coverage -fprofile-arcs -fprofile-dir=./Corrida-$nombre ../../HuffmanOriginal/\*.cpp -fpermissive -o HuffmanCompressor

Los flags *–fprofile-arcs y –ftest-coverage* son mandatorios para utilizar GCOV ya que, entre otras cosas, crean un grafico de flujo del programa.  
Por una cuestión de comodidad, se agregó también el flag *–fprofile-dir* para almacenar los archivos *.gcda* en un directorio separado.Para obtener realmente un indicador de cobertura del programa, el mismo se corrió tres veces en cada archivo de prueba. Una vez para comprimir el archivo, una segunda para descomprimirlo y una tercera sin argumentos para imprimir el mensaje de cómo debe utilizarse el programa. Esto permite ejecutar todo el código “Alcanzable” del programa.

./HuffmanCompressor comprimir "$nombre.jpg" ./HuffmanCompressor descomprimir "$nombre.cmp"  
./HuffmanCompressor

Resultados cobertura

El análisis general de cobertura demuestra que se ejecuto un 66.86% de un total de 528 lineas para la compresión y descompresión de los 3 archivos.

Del informe final procederemos a analizar los números de los archivos que no han sido cubiertos al 100% para luego ir al código fuente. Obviaremos archivos que no corresponden al repositorio, tales como “*iostream*” ignoraremos también a efectos prácticos de este trabajo los archivos “.h”.

* File '../../HuffmanOriginal/FilaDePrioridad.cpp'  
  Lines executed:3.23% of 31  
  Creating 'FilaDePrioridad.cpp.gcov'
* File '../../HuffmanOriginal/Frequency.cpp'

Lines executed:57.14% of 14

Creating 'Frequency.cpp.gcov'

* File '../../HuffmanOriginal/Heap.cpp'

Lines executed:66.95% of 118

Creating 'Heap.cpp.gcov'

* File '../../HuffmanOriginal/HuffmanCode.cpp'

Lines executed:28.57% of 21

Creating 'HuffmanCode.cpp.gcov'

* File '../../HuffmanOriginal/HuffmanCompressor.cpp'

Lines executed:80.63% of 160

Creating 'HuffmanCompressor.cpp.gcov'File

* ../../HuffmanOriginal/HuffmanNode.cpp'

Lines executed:84.21% of 19

Creating 'HuffmanNode.cpp.gcov'

* File '../../HuffmanOriginal/HuffmanTree.cpp'

Lines executed:75.95% of 79

Creating 'HuffmanTree.cpp.gcov'

* File '../../HuffmanOriginal/main.cpp'

Lines executed:100.00% of 12

Creating 'main.cpp.gcov'

* File '../../HuffmanOriginal/Nodo.cpp'

Lines executed:5.00% of 20

Creating 'Nodo.cpp.gcov'

* File '../../HuffmanOriginal/object.cpp'

Lines executed:33.33% of 15

Creating 'object.cpp.gcov'

De estos resultados podemos corroborar que ciertos archivos no están influyendo en el funcionamiento del programa. Luego de realizar todos los análisis, en la etapa de corrección realizaremos pruebas removiendo estos archivos del repositorio y comprobaremos si es que el software mantiene su funcionamiento.  
Los archivos no utilizados son los siguientes:

* FilaDePrioridad.cpp
* Nodo.cpp

El archivo “Main.cpp” que creamos para estas pruebas se encuentra entre los archivos que han sido cubiertos en un 100%. Este archivo utiliza las principales funcionalidades del software que son las de comprimir y descomprimir por lo que deberemos analizar la cobertura archivo por archivo.

## Uso de CPU y tiempo de ejecución

Para obtener los resultados de tiempo de ejecución y uso de CPU, se utilizo la herramienta llamada GPROF. Mediante un análisis de los resultados provistos por la herramienta podremos identificar que partes del código priorizar al momento de realizar la optimización.

Las pruebas se encuentran automatizadas en el script *script-GPROF.sh*.

La compilación se realizo con los siguientes argumentos:

g++ -pg ../../HuffmanOriginal/\*.cpp -fpermissive -o HuffmanCompressor

Al igual que con las pruebas de GCOV, los archivos generados por la herramienta se mueven automáticamente a otra carpeta luego de que se genere el informe final.

Se realizaron pruebas tanto para el caso de compresión como para el caso de descompresión.

Se debió agregar un nuevo archivo de prueba de un tamaño mayor que el resto para que los resultados sean considerables y pueda brindar mejor información para analizar.

### Resultados uso de CPU

Los resultados del análisis de llamadas y uso de CPU demuestran que en los archivos más pequeños el *profiling* no puede realizarse adecuadamente, mientras que para los archivos de mayor tamaño el tiempo de ejecución permite un mejor análisis.

Cuando utilizamos los archivos de mayor tamaño, la función que más se utilizo y que además consumió más tiempo del programa fue *HuffmanTree::buscar(unsigned char)* con 775671 llamadas y 57.91% de tiempo de ejecución.

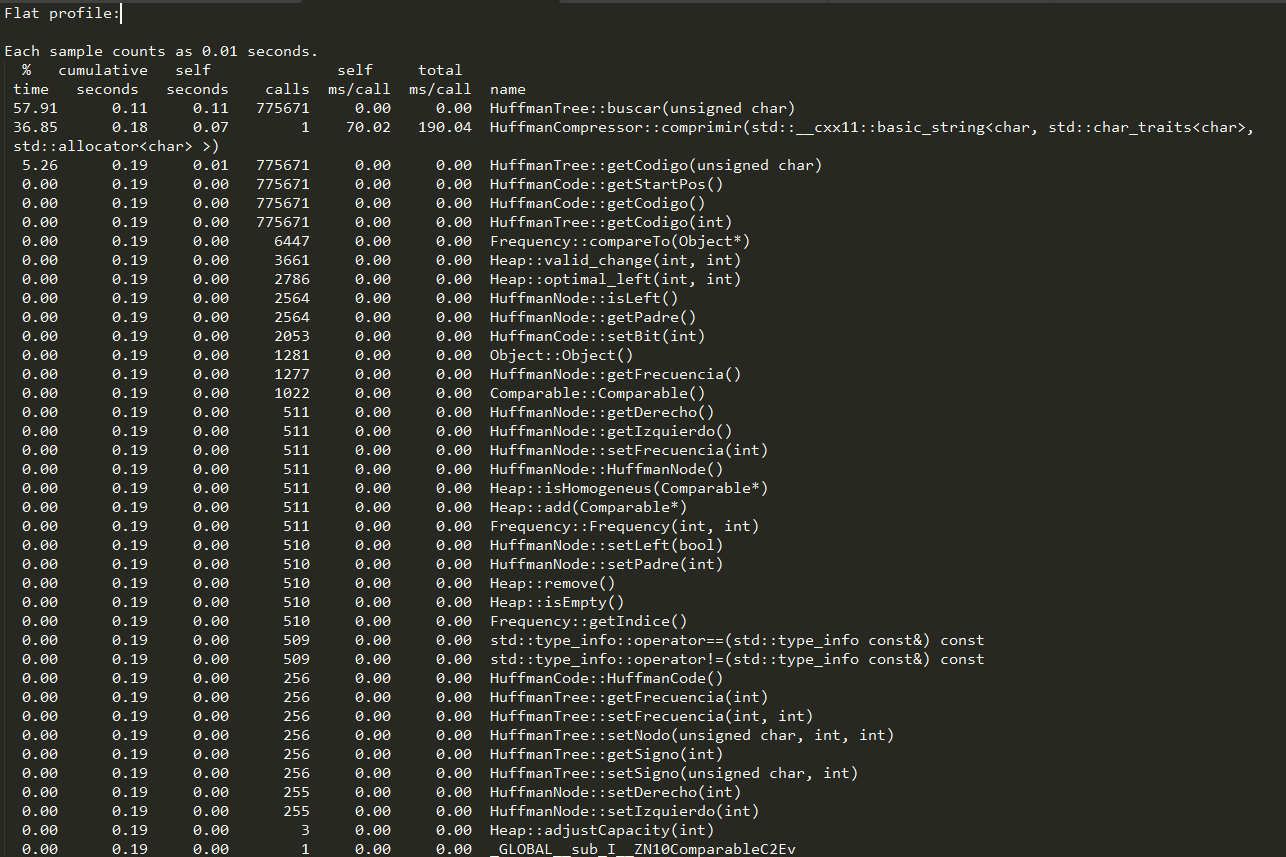
Como el principal factor de ordenamiento de la salida de GPROF es el tiempo de ejecución, en segundo lugar tenemos *HuffmanCompressor::comprimir()*pero es la función “padre” del resto (sin contar main).

En el caso de descompresión la función con más tiempo de ejecución es HuffmanCompressor::descomprimir(), pero no es información relevante si tenemos en cuenta que se da lo mismo que en el caso de la función de compresión mencionada previamente.

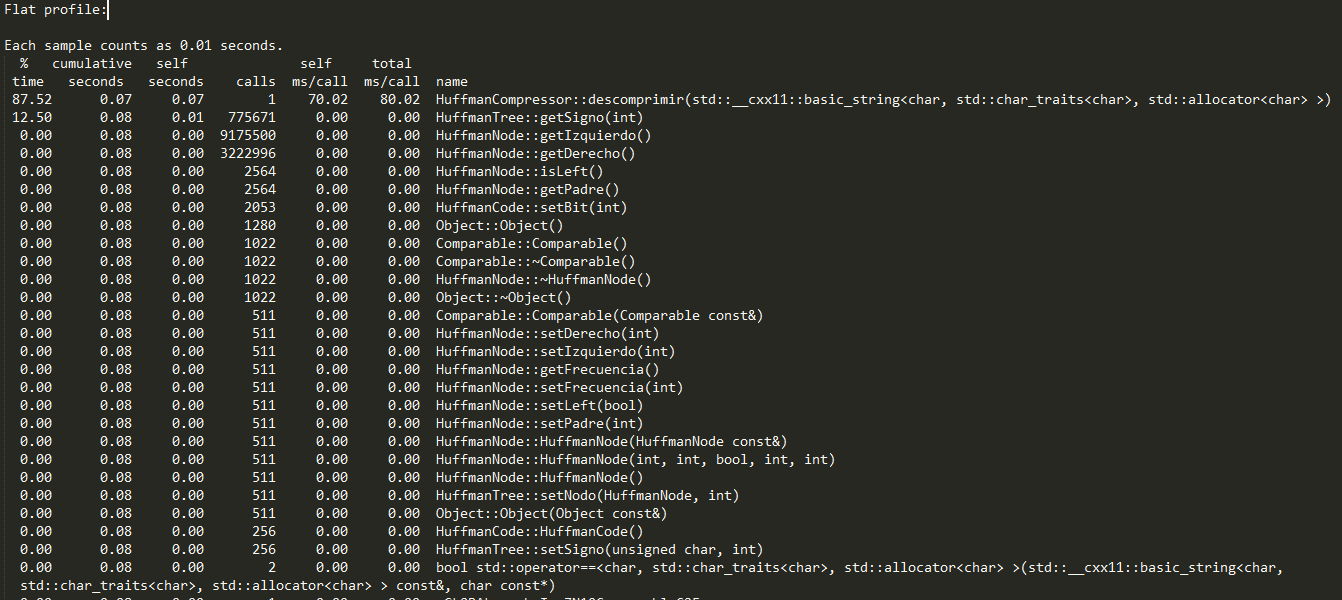
Las funciones más utilizadas son *HuffmanNode::getIzquierdo()* y *HuffmanNode::getDerecho()* con 9175500 y 3222996 llamadas respectvamente.

A continuación se comparten los gráficos para un mejor entendimiento.

### Compresión



### Descompresión



## Uso de memoria

La herramienta utilizada para realizar *profiling* de uso de memoria es Valgrind.  
Las mediciones se realizaron con dos herramientas que brinda. Memcheck y Cachegrind.

Estas herramientas fueron elegidas por lo siguiente:

* Memcheck es detector de errores en manejo de memoria. Puede detectar problemas que son comunes en C/C++ a la hora de utilizar memoria dinámica.
* Cachegrind mide el rendimiento de la cache durante la ejecución de acuerdo a sus características (capacidad, tamaño de bloque de datos, grado de asociatividad).

Las pruebas se encuentran automatizadas en el script *script-VALGRIND.sh*.

### Memcheck

Para el uso de la herramienta Memcheck la compilación se realizo de la siguiente manera.

g++ -g ../../HuffmanOriginal/\*.cpp -fpermissive -o HuffmanCompressor

A diferencia de la compilación standard que veníamos realizando, se agrego el flag –g para que Valgrind pueda indicarnos donde están los errores en los archivos fuente.

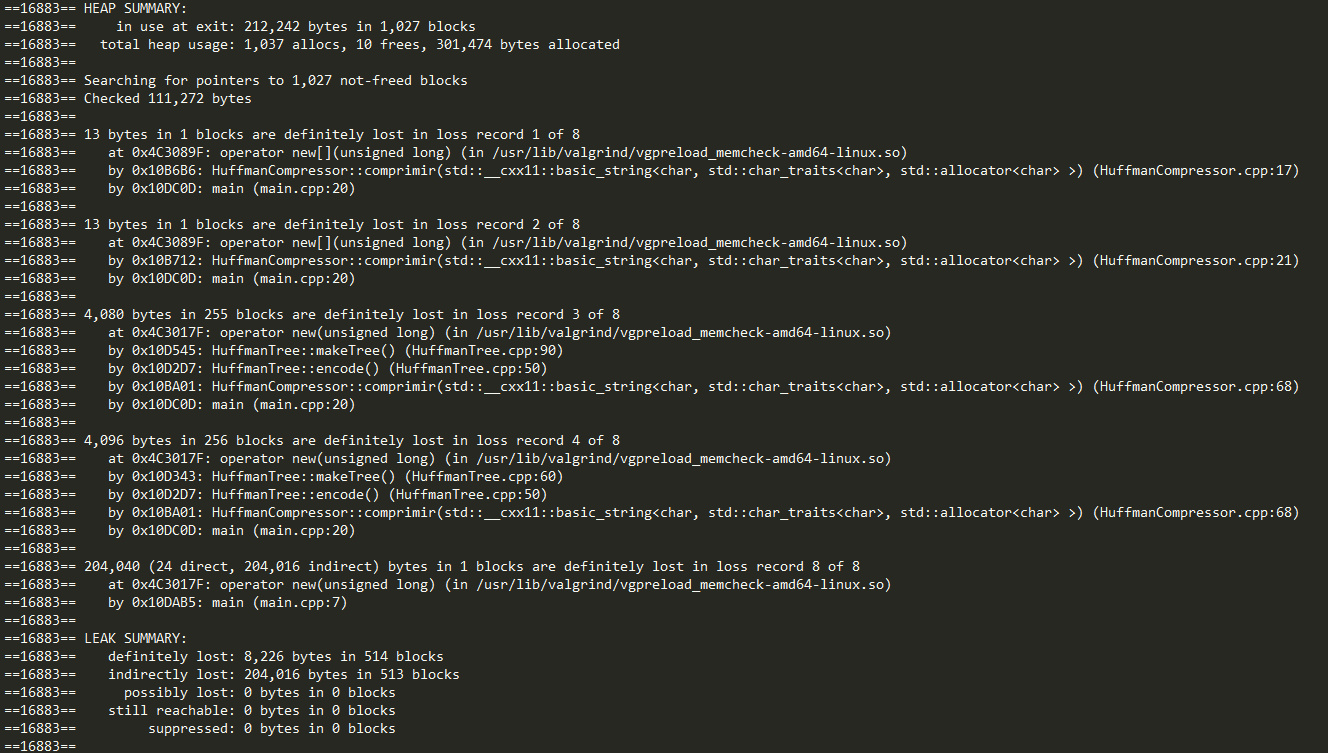
Luego, para utilizar la herramienta Memcheck, corrimos el programa de la siguiente manera.

valgrind --leak-check=full --log-file=./Resultados$nombre-compresion -v ./HuffmanCompressor comprimir "$nombre.jpg"

valgrind --leak-check=full --log-file=./Resultados$nombre-descomprimir -v ./HuffmanCompressor descomprimir "$nombre.cmp"

### Resultados Memcheck

El análisis del uso de memoria dinámica, demuestra a primera vista que la mayoría de las solicitudes de memoria mediante el operador new, no son liberadas.



Podemos ver en el informe de Valgrind que de 1037 solicitudes, solo 10 son liberadas.   
Identificando las líneas de código en donde se realiza el pedido de memoria, realizaremos la liberación de la misma en la etapa de mejoras.

### Cachegrind

Nuevamente hay que compilar el programa con la opción -g para obtener información de *debugging* y luego ejecutar los siguientes comandos.

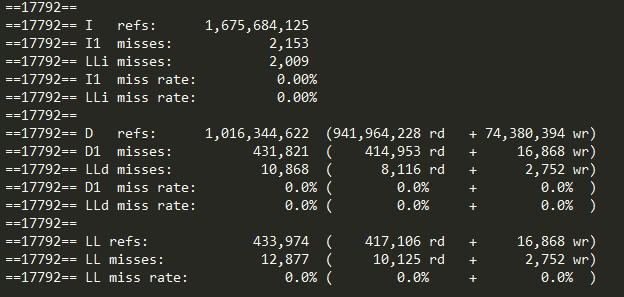
valgrind -v --tool=cachegrind --cache-sim=yes ./HuffmanCompressor comprimir "$nombre.jpg" 2>&1 | tee ./Resultados-$nombre-compresion.txt

valgrind -v --tool=cachegrind --cache-sim=yes ./HuffmanCompressor descomprimir "$nombre.cmp" 2>&1 | tee ./Resultados-$nombre-descompresion.txt

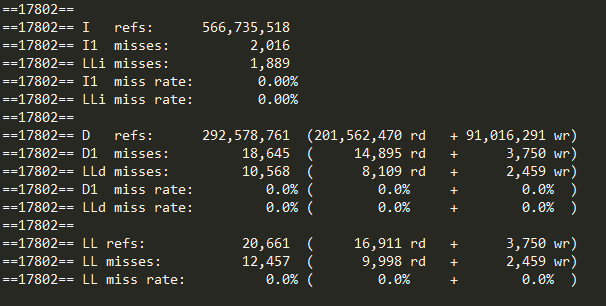
-v para activar modo verbose, --tool para elegir la tool a utilizar y –cache-sim para activar los resultados de acceso o miss a la cache.

Los resultados son los siguientes:

### Compresión



### Descompresión



## Correcciones

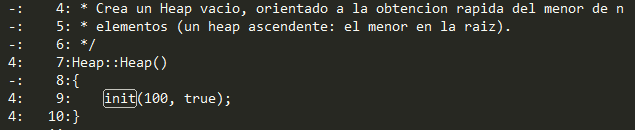
### Cobertura

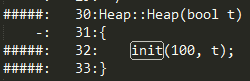
Luego del análisis de cobertura pudimos detectar que dos de los archivos fuente, no estaban influyendo en el funcionamiento del programa. Los mismos son:

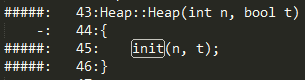
* FilaDePrioridad.cpp
* Nodo.cpp

Luego de corroborar que el programa funcionaba como era esperado, como primer medida de mejora se removieron estos archivos junto con sus archivos de cabecera.

Siguiendo con el análisis de cobertura, notamos que en muchos archivos fuentes donde se realiza la declaración de los métodos de una clase, la cobertura es baja ya que de una clase que posee sobrecarga de constructores, solo se utiliza uno. A efectos prácticos de este trabajo podríamos remover las sobrecargas, pero no es una buena idea ya que disminuye la reutilización de código.  
Un ejemplo de esto último es el archivo *Heap.cpp* cuyo informe de Gcov mostraremos a continuación.







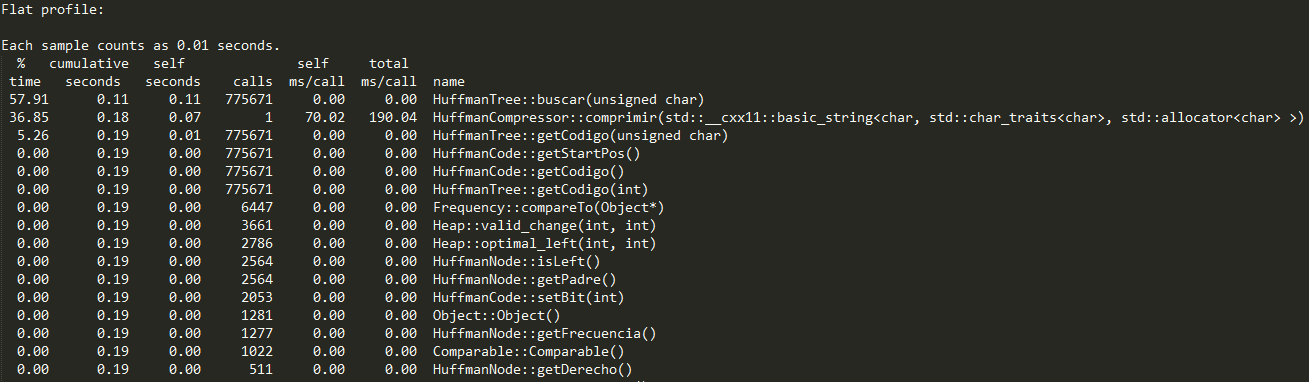
Otro detalle que influye en la baja de la cobertura de algunos archivos, es la inclusión de un método “*toString*” para imprimir por pantalla el contenido de una clase. Si bien también es una función de utilidad que ayuda a la reutilización del código, en este caso removeremos estos métodos para observar un impacto en la cobertura.

Un método que no se utilizo es el llamado “*Desafio3*” que es parte de la clase “*HuffmanCompressor*”. El mismo también fue removido.



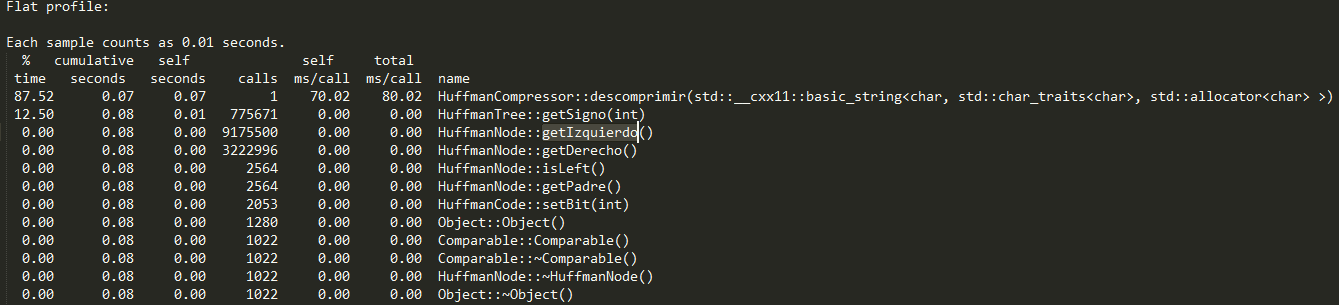
Como prueba final volvemos a compilar el programa y luego de verificar que su comportamiento es el esperado, seguimos con las siguientes mejoras.

### Uso de Cpu

  
Como se puede observar en el informe de uso del programa en el caso de compresión, los métodos no se ejecutan durante mucho tiempo, y siendo “self seconds” el principal índice para ordenar esta tabla, los resultados deben analizarse con cuidado.   
A primera vista podemos observar que el método en la cual transcurre más tiempo es el método HuffmanTree::buscar(). El mismo consiste básicamente en una búsqueda de una coincidencia usando un ciclo “*for*”. Para optimizar esta función utilizaremos las mejoras vistas en la materia para ciclos de repetición, tales como desdoblamiento de ciclos o recorrido de los ciclos en orden inverso.  
Otras de las funciones que podemos ver en el output no serán trabajadas por lo siguiente:

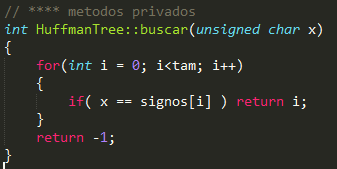
* HuffmanTree::getCodigo(): Es básicamente un “*wrapper*” de HuffmanTree::buscar()  
  la cual ya fue discutida.
* HuffmanCode::getStartPos(): Devuelve uno de los atributos de la clase.
* HuffmanCode::getCodigo(): Devuelve uno de los atributos de la clase.

Lo mismo ocurre para el informe del caso de descompresión

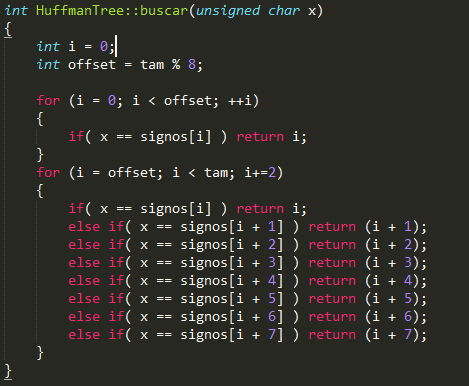


### Mejoras en ciclos de repetición

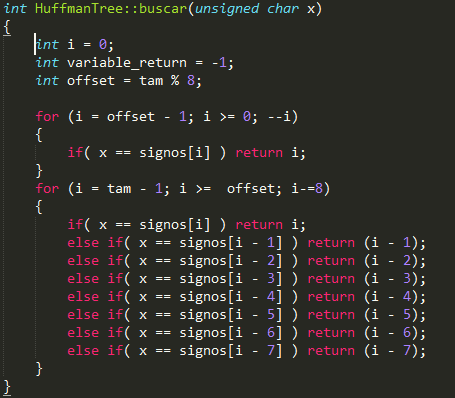
En el caso del método, HuffmanTree::buscar(), el código original era el siguiente



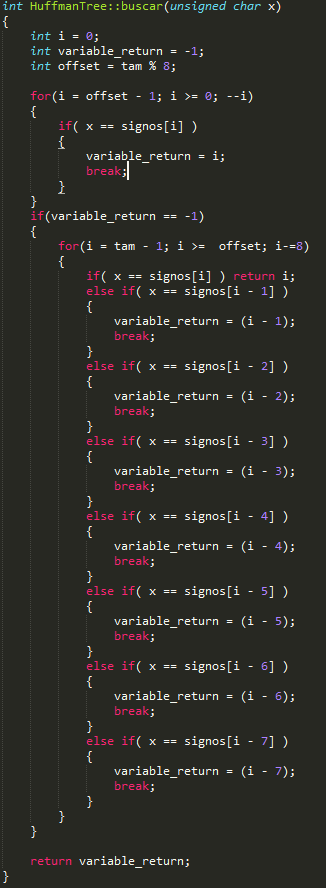
La primera mejora a implementar será la de desdoblamiento de ciclos. El número elegido para desdoblar será 8.  
El resultado es el siguiente:



Y por ultimo basándonos en otra de las mejores vistas en clase, recorreremos los ciclos en sentido inverso.



Se realizan unos cambios extras para evitar tantas instrucciones de “*return”* y mantener un solo “*return”* al final de la función.

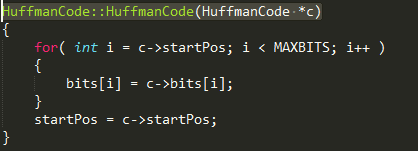


Si bien durante la materia se discutió que debían evitarse las instrucciones “*if*” dentro de un ciclo de repetición, en este caso es difícil trasladar estos chequeos fuera del ciclo porque para acceder al arreglo necesitamos el índice provisto por el ciclo.

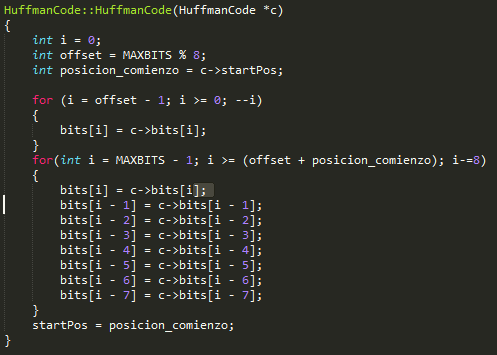
Una vez que confirmamos que el funcionamiento del programa sigue siendo el esperado, buscaremos otros ciclos de repetición en el código y procederemos a aplicar mejoras.

Realizaremos cambios similares al ciclo que se puede encontrar en el método “HuffmanCode::HuffmanCode(HuffmanCode \*c)”.

Código original:



Código luego de mejoras de performance:



Luego de, una vez más constatar que el programa funciona como es esperado, procedemos a realizar cambios similares en el código. Estos cambios son, recorrido de los ciclos en orden inverso y desdoblamiento en la medida de que estos cambios no dificulten demasiado la posterior lectura de código por un tercero.

### Uso de Memoria

Como pudimos observar previamente, el principal déficit en el manejo de memoria, es la falta de liberación de memoria solicitada dinámicamente. En base al informe de Memcheck, agregamos instrucciones de borrado en los destructores de la clase para que liberen los recursos una vez que ya no sean necesarios.

# Segundo ciclo de pruebas

Luego de realizar todas las mejoras descriptas en la sección anterior, procederemos a ejecutar nuevamente todos los chequeos previamente realizados en el software original y compararemos los resultados.

Comparación cobertura

Cobertura total en Programa original:



Cobertura total en Programa modificado:

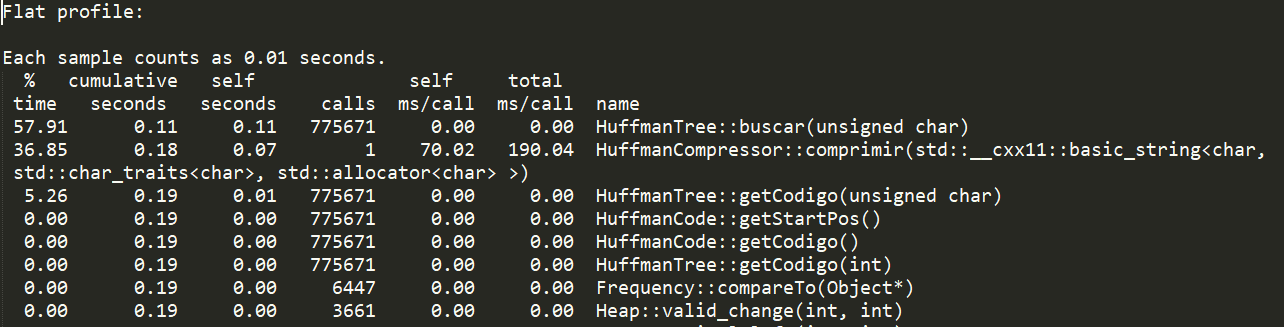


Algunas comparaciones entre archivos fuente del software original y modificado.

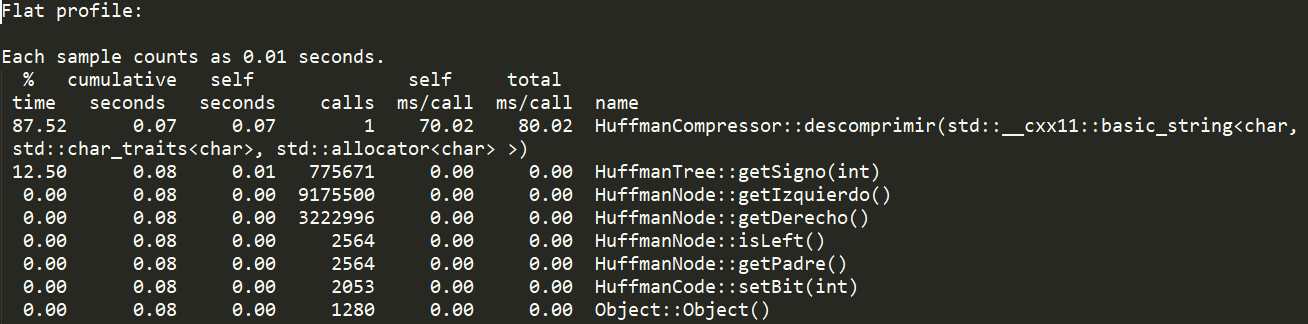
Original  
  
  
Modificado  
  


### Comparación uso de CPU

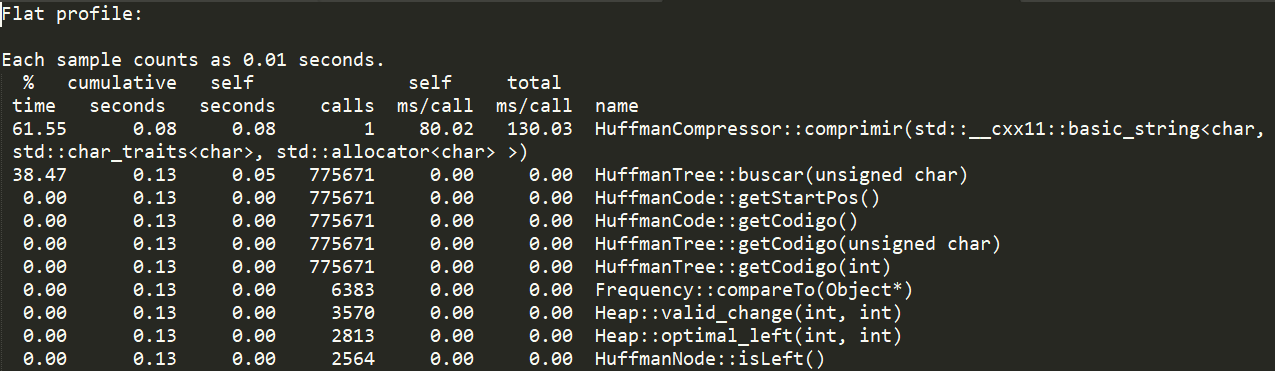
El método donde más cambios se realizaron fue HuffmanTree::buscar(). Compararemos los resultados para ver si hubo una mejora.

Original  
Compresión  


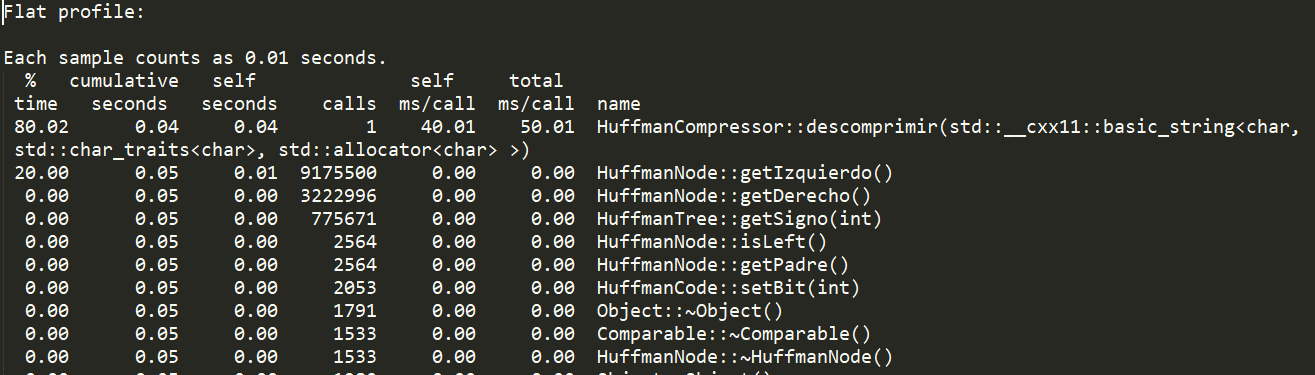
##### Descompresión



#### Modificado Compresión



##### Descompresión



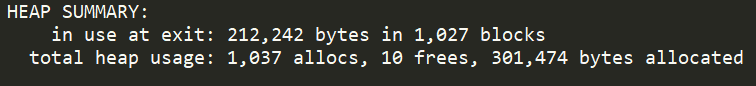
Luego de comparar los resultados podemos ver lo siguiente:

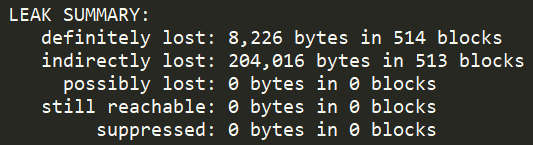
El tiempo acumulado de ejecución para la compresión en el programa original, fue de 0.19 segundos, mientras que para, el mismo caso, en el programa modificado, el tiempo acumulado de ejecución fue de 0.13 segundos.

Para el caso de descompresión, el tiempo acumulado del programa, antes de las modificaciones fue de 0.08 segundos, mientras que luego de las modificaciones, de 0.05 segundos.

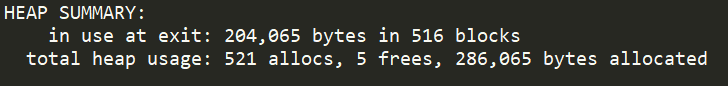
La mala decisión a destacar, son los cambios realizados en la función HuffmanTree::buscar(), los cuales empeoraron el tiempo de ejecución. A pesar del desdoblamiento de ciclos y el recorrido en orden inverso, la cantidad de condicionales dentro de la función, impactan en la performance.

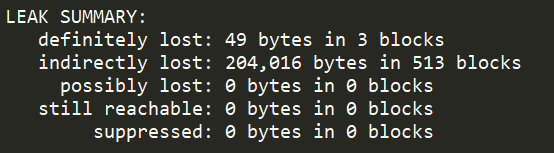
Comparación Uso de memoria

Original  
Compresión  


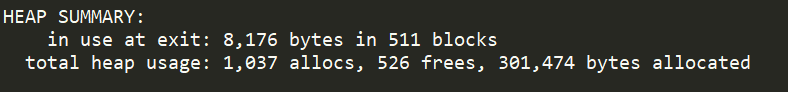


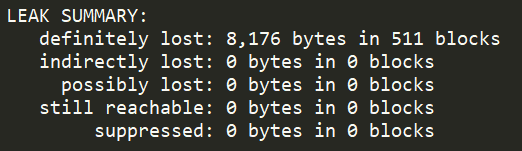
##### Descompresión



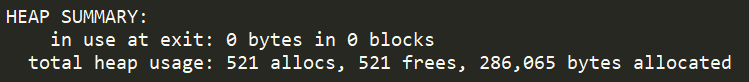


#### Modificado Compresión

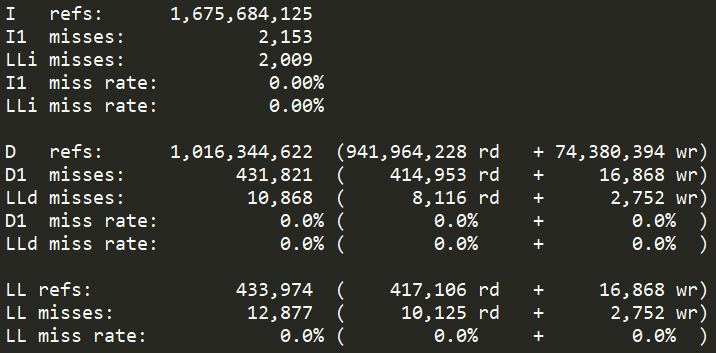




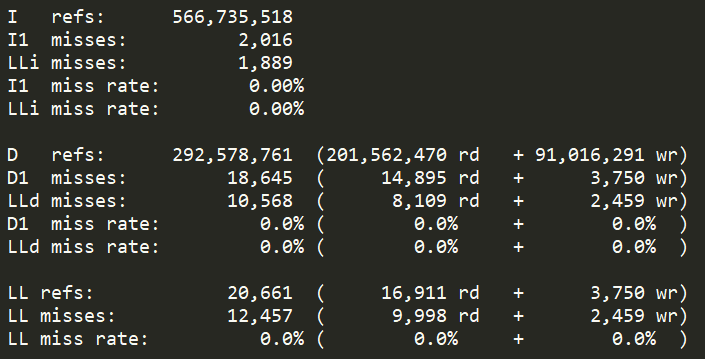
##### Descompresión



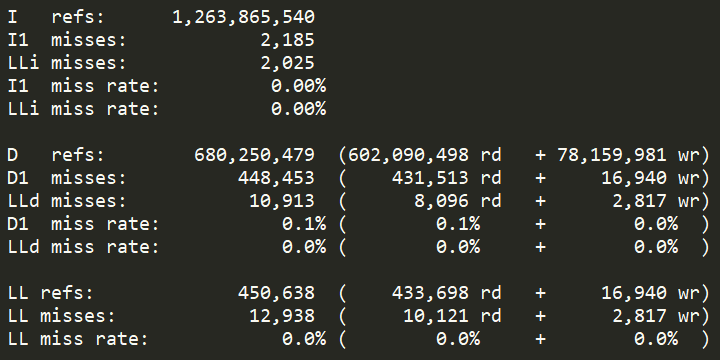
Comparación Uso de memoria Cache

Original  
Compresión  


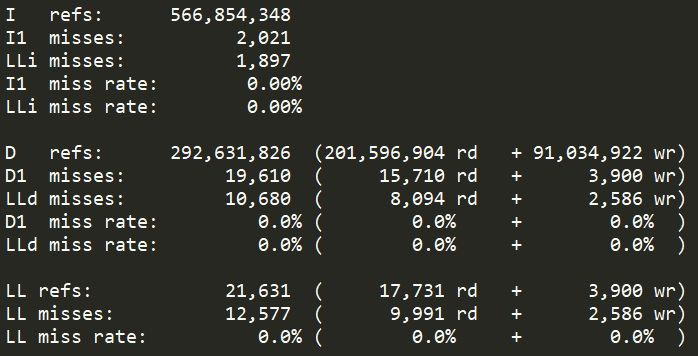
##### Descompresión



#### Modificado Compresión



##### Descompresión



# Conclusión

Luego de realizar las comparaciones pertinentes entre la versión original y luego de las mejoras, podemos concluir que, a la hora de optimizar el código, debe hacerse un balance entre el rendimiento esperado y la mantenibilidad del código.

Algunas pruebas, como la prueba de cobertura, nos brindaron un resultado certero que nos facilito remover el código muerto sin mayores problemas. Mientras que en el caso de tiempo de ejecución, las soluciones no fueron tan directas o automáticas. Esto se confirma ya que en nuestro caso, una de las funciones sobre la cual aplicamos gran parte de las mejoras, no mejoro su tiempo de performance, pero las mismas estrategias en otras partes del código, lograron mejorar el tiempo de ejecución total del programa.