

# **Titulo del trabajo**

**Diego Alejandro Londoño Jiménez**  
*diegoa.londono@udea.edu.co*

Despartamento de Ingeniería Electrónica y Telecomunicaciones  
Universidad de Antioquia  
Medellín  
Mayo de 2022

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Preguntas orientadoras</b>	<b>4</b>
2.1. En sus propias palabras defina que es <i>profiling</i> . . . . .	4
2.2. ¿Qué papel desempeña la creación de los CFG y los CallGraph en el “profiling” de un determinado código? . . . . .	4
2.3. Consulte y describa brevemente en qué consisten las herramientas Kcachegrind y Valgrind. . . . .	4
2.4. En el contexto de profiling, ¿en qué consisten las fugas de memoria ( <i>memory leaks</i> ) . . . . .	4
2.5. ¿Por qué cree usted que las fugas de memoria se deben evitar en los Sistemas de Tiempo Real? . . . . .	5
<b>3. Profiling de un programa de prueba</b>	<b>5</b>
3.1. Compilación . . . . .	5
3.1.1. Qué hace el programa <i>prueba.c</i> . . . . .	5
3.1.2. ¿Cuál es el papel que desempeña el <code>#include &lt;valgrind/callgrind.h&gt;</code> , así como las instrucciones de preprocessamiento <i>CALLGRIND_START_INSTRUMENTATION</i> y <i>CALLGRIND_STOP_INSTRUMENTATION</i> ? . . . . .	5
3.2. Generación del Callgraph del programa “Prueba” . . . . .	6
3.2.1. Consulte e indique qué hacen las opciones <code>--dump-instr=yes</code> y <code>--instr-atstart=no</code> . . . . .	6
3.2.2. En el entorno gráfico del Kcachegrind observe el contenido de la ventana inferior llamada “Call Graph”. Analice este grafo presentado y interprete los datos numéricos presentados sobre dicho grafo. . . . .	6
3.2.3. Análisis de Callee Map, Source Code, Machine Code, Callees, All Callers . . . . .	6
<b>4. Profiling de un programa con Fuga de Memoria</b>	<b>7</b>
4.1. Compilación . . . . .	7
4.1.1. ¿Qué hace este programa <i>memoryleak.c</i> ? . . . . .	7
4.1.2. Consulte e indique qué hace las opciones <code>-leak-check=full</code> . . . . .	7
4.1.3. Explique la salida obtenida luego de ejecutar el anterior comando. ¿Qué falla presenta el programa <i>memoryleak.c</i> ? . . . . .	7
4.1.4. Resuelva la falla de memoria presentada. . . . .	8
4.1.5. ¿Cuál es el lugar adecuado para colocar estas dos instrucciones? . . . . .	8
4.1.6. En su análisis, interprete los datos numéricos presentados sobre dicho grafo. . . . .	9

4.1.7. Análisis de Callee Map, Source Code, Machine Code, Callees, All Callers, con relación al programa <i>Memoryleak</i> . . . . .	9
--	---

<b>5. Conclusiones</b>	<b>11</b>
------------------------	-----------

## **1. Introducción**

## 2. Preguntas orientadoras

### 2.1. En sus propias palabras defina que es *profiling*

Análisis detallado del código que se tiene para saber dónde se está consumiendo más ciclos de reloj el procesador, teniendo claro lo anterior se puede proceder a remplazar la función o las bibliotecas en caso de que se pueda y de esta manera poder optimizar el código.

### 2.2. ¿Qué papel desempeña la creación de los CFG y los CallGraph en el “profiling” de un determinado código?

- **CFG:** Es un modelo abstracto del código fuente sin importar el lenguaje del código fuente, modela la operación de datos (aritméticas, matriciales) y operación de control(condicionales). Permite ver como evolucionana y cuál es la dependencia que hay de los datos, una de las ventajas de ver esa dependencia y de poder contar varios *core* en el procesador, es que se pueden usar los diferentes *core* para las operaciones que no tengan dependencia.
- **CallGraph:** Muestra cuál es el ordenamiento de llamadas a funciones, qué función llama a cuál función,tanto funciones propias, como funciones de bibliotecas que hagan parte del lenguaje.

### 2.3. Consulte y describa brevemente en qué consisten las herramientas Kcachegrind y Valgrind.

“**Valgrind** es un programa flexible para depurar y hacer análisis detallado al código ejecutable de Linux. Consta de un núcleo, que proporciona una CPU sintética en software, y una serie de herramientas de depuración y profiling. La arquitectura es modular, de modo que se pueden crear nuevas herramientas fácilmente y sin alterar la estructura existente.” [1] En pocas palabras, es una herramienta que me permite hacer un profiling detallado del código y a encontrar dónde se está consumiendo más instrucciones de reloj el código

**KCachegrind**, es una herramienta que permite visualizar los datos de los profiling realizados al código.

### 2.4. En el contexto de profiling, ¿en qué consisten las fugas de memoria (*memory leaks*)

“Una fuga de memoria (más conocido por el término inglés memory leak) es un error de software que ocurre cuando un bloque de memoria reservada no es liberada en un programa de computación. Comúnmente ocurre porque se pierden todas las referencias a esa área de memoria antes de haberse liberado.

Dependiendo de la cantidad de memoria perdida y el tiempo que el programa siga en ejecución, este problema puede llevar al agotamiento de la memoria disponible en la computadora. Este problema se da principalmente en aquellos lenguajes de programación en los que el manejo de memoria es manual (C o C++ principalmente), y por lo tanto es el programador el que debe saber en qué momento exacto puede liberar la memoria. Otros lenguajes utilizan un recolector de basura o conteo de referencias que automáticamente efectúa esta liberación. Sin embargo todavía es posible la existencia de fugas en estos lenguajes si el programa acumula referencias a objetos, impidiendo así que el recolector llegue a considerarlos en desuso.” [2]

## 2.5. ¿Por qué cree usted que las fugas de memoria se deben evitar en los Sistemas de Tiempo Real?

Si se llega al punto de agotar la memoria disponible de la computadora y dependiendo del proceso que se esté ejecutando puede traer consecuencias muy delicadas, hasta llegar al punto de tener perdidas humanas, por tal motivo el programador debe de garantizar siempre que esto no suceda,

## 3. Profiling de un programa de prueba

### 3.1. Compilación

#### 3.1.1. Qué hace el programa *prueba.c*

Se importa la biblioteca < valgrind/callgrind.h >, es la que se encarga de ejecutar los comandos *CALLGRIND\_START\_INSTRUMENTATION* y *CALLGRIND\_STOP\_INSTRUMENTATION*. El programa consta de tres funciones enteras que retornan las operaciones correspondientes. la función int add() retorna la suma de la variable val de cero hasta 799, la función int mult() retorna la multiplicación de la variable val de cero hasta 799, la función foo() retorna el valor de la función add() dos veces más la función mult()

#### 3.1.2. ¿Cuál es el papel que desempeña el #include < valgrind/callgrind.h >, así como las instrucciones de preprocesamiento

*CALLGRIND\_START\_INSTRUMENTATION* y *CALLGRIND\_STOP\_INSTRUMENTATION*?

#include < valgrind/callgrind.h > se encarga de traer del programa valgrind la biblioteca *callgrind.h* la cual ejecuta las instrucciones *CALLGRIND\_START\_INSTRUMENTATION* y *CALLGRIND\_STOP\_INSTRUMENTATION*, los macros anteriores se ubican en la

posición del código en la que debe comenzar y finalizar la elaboración del profiling y se usa para hacer el profiling especialmente a una parte del código.

### 3.2. Generación del Callgraph del programa “Prueba”

#### 3.2.1. Consulte e indique qué hacen las opciones $--dump-instr = yes$ y $--instr-atstart = no$

$--dump-instr = yes$ : Permite ver el código ensamblador, los datos del perfil resultante sólo se pueden ver con KCachegrind.  $--instr-atstart = no$  Inicializa Callgrind con el modo de instrumentación desactivado

#### 3.2.2. En el entorno gráfico del Kcachegrind observe el contenido de la ventana inferior llamada “Call Graph”. Analice este grafo presentado y interprete los datos numéricicos presentados sobre dicho grafo.

Los valores hacen referencia a la cantidad de ciclos de reloj que se demora cada bloque en realizar lo que le corresponde ejecutar.

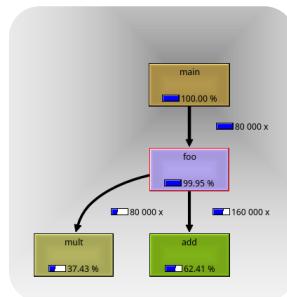


Figura 1: CallGraph

#### 3.2.3. Análisis de Callee Map, Source Code, Machine Code, Callees, All Callers

**Callee Map:** Estando en esta pestaña, el usuario se puede desplazar en la lista de Flat Profile y se puede ver en Callee Map las funciones que son llamados y cuál función contiene a cuál función a medida que se desplaza en Flat Profile.

**Source Code:** Muestra la función que se le esté haciendo el profiling, dicha función se encarga de llamar las otras dos funciones, por lo tanto, es la que consume más ciclos de reloj en ejecución diferente al main

**Callees:** Muestra la siguiente función que será llamada con, por esta razón cuando se selecciona la función add y mult no muestra nada en la pestaña *Callees*

**All Callers:** En esta pestaña se puede ver la diferencia entre Incl y self, la diferencia entre ellas se trata de atributos de costos para las funciones relativas a algún tipo de evento. Como las funciones pueden llamarse entre sí, tiene sentido distinguir el costo de la propia función (“Costo propio”) y el coste que incluye todas las funciones llamadas (“Costo inclusivo”). A veces también se habla de costos “*propios*” como costos “exclusivos”<sup>1</sup>.

## 4. Profiling de un programa con Fuga de Memoria

### 4.1. Compilación

#### 4.1.1. ¿Qué hace este programa memoryleak.c?

Se importan los ficheros de cabecera pertinentes para usar las funciones requeridas en el programa. La cabecera `< stdio.h >`, se encarga de manejar las entradas y salidas del programa, `< stdlib.h >` se encarga de la generación de los numeros aleatorios y reservar memoria `< time.h >` contiene funciones para el manejo y conversión de fechas y horas entre diferentes formatos.

Se reserva un espacio de memoria con la instrucción `malloc`, con el ciclo `for`, se inicializa el espacio de memoria que se reservó anteriormente, con la instrucción `srand (time(NULL))`, se inicializa el generador de números aleatorios para poder almacenarlos en la variable `randNum`, luego ese número se ubica en el arreglo que se reservó en el espacio de memoria y se imprime la posición y valor que tiene esa posición.

#### 4.1.2. Consulte e indique qué hace las opciones `-leak-check=full`

Da detalles de cada bloque definitivamente perdido o posiblemente perdido, incluyendo dónde fue asignado. (En realidad, combina los resultados de todos los bloques que tienen el mismo tipo de fuga y trazas del stack suficientemente similares en un único “registro de pérdidas”.<sup>2</sup>

#### 4.1.3. Explique la salida obtenida luego de ejecutar el anterior comando. Qué falla presenta el programa memoryleak.c?

Se está reservando un bloque de memoria con la instrucción `int *intArray = malloc(sizeof(int) * ARR_SIZE);` y en ningún momento se esté liberando la memoria, por lo tanto el programa en algún momento va a colapsar por falta de memoria para ejecutarse.

---

<sup>1</sup>Esta definición fue tomada del manual de Kcachegrind

<sup>2</sup>Esta definición fue tomada del manual de Valgrind

#### 4.1.4. Resuelva la falla de memoria presentada.

Se libera la memoria con la instrucción `free(intArray);` y la ejecutar nuevamente la instrucción `valgrind --leak-check = full./memoryleak` se tiene como resultado: “**==3880== For lists of detected and suppressed errors, rerun with: -s ==3880== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)**”

#### 4.1.5. ¿Cuál es el lugar adecuado para colocar estas dos instrucciones?

```
int *intArray = malloc(sizeof(int) * ARR_SIZE);

CALLGRIND_START_INSTRUMENTATION;
for (int i = 0; i < ARR_SIZE; i++)
{
    intArray[i] = i;
}
srand(time(NULL));
CALLGRIND_STOP_INSTRUMENTATION;
```

se elige este lugar para hacer el profiling, ya que, viendo el código se puede observar que es la parte donde se hará más uso de los ciclos del reloj.

**4.1.6. En su análisis, interprete los datos numéricos presentados sobre dicho grafo.**

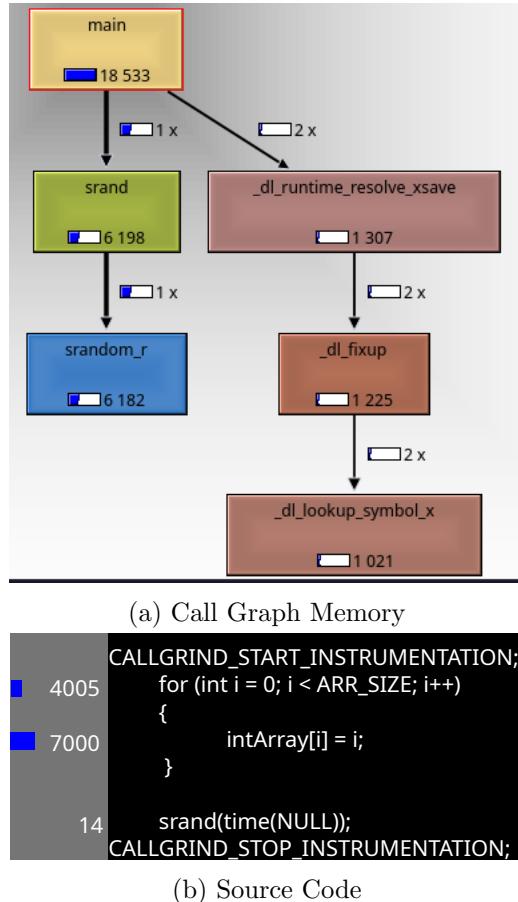


Figura 2: Análisis del Código

Si se observa la figura 2b, se puede apreciar la cantidad de ciclos de reloj en el bucle For y en la inicialización del puntero.

**4.1.7. Análisis de Callee Map, Source Code, Machine Code, Callees, All Callers, con relación al programa *Memoryleak***

**Callee Map:** Estando en esta pestaña, se observa las funciones que se encuentran en el análisis del profiling, allí se puede observar cuantos ciclos de reloj consume cada función, que librerías propias del SO linux esta usando y código fuente y cabeceras del lenguaje C

**Source Code:** Muestra la función que se le esté haciendo el profiling, dicha función se encarga de llamar las otras dos funciones, por lo tanto, es la que consume más ciclos de reloj en ejecución diferente al main

**Callees:** Muestra la siguiente función que será llamada con, por esta razón cuando se selecciona la función add y mult no muestra nada en la pestaña *Callees*

**All Callers:** En esta pestaña se puede ver la diferencia entre Incl y self, la diferencia entre ellas se trata de atributos de costos para las funciones relativas a algún tipo de evento. Como las funciones pueden llamarse entre sí, tiene sentido distinguir el costo de la propia función (“Costo propio”) y el coste que incluye todas las funciones llamadas (“Costo inclusivo”). A veces también se habla de costos “*propios*” como costos “*exclusivos*”<sup>3</sup>.

---

<sup>3</sup>Esta definición fue tomada del manual de Kcachegrind

## 5. Conclusiones

- XXXXXXXX
- YYYYYYY
- ZZZZZZZ

## **Referencias**

- [1] Linux, “Valgrind documentation,” 2022. [Online]. Available: [https://valgrind.org/docs/manual/valgrind\\_manual.pdf](https://valgrind.org/docs/manual/valgrind_manual.pdf)
- [2] Wikipedia, “Fuga de memoria,” 14 nov 2020 a las 17:58. [Online]. Available: [https://es.wikipedia.org/wiki/Fuga\\_de\\_memoria](https://es.wikipedia.org/wiki/Fuga_de_memoria)