Optimización de memoria y consumo de un software de procesamiento de imagen.

1. Analizar el uso de la memoria

- **¿Dónde se accede a memoria?**

  Mediante la herramienta Cachegrind podemos observar que la mayor parte de las instrucciones que acceden a memoria caché, se dan en los bucles while donde se leen y escriben las imágenes (.pgm). Con Massif podemos observar la ocupación de la pila y los porcentajes de las funciones que más acceden, en este caso, `load_image_from_file()` y `save_image_to_file()`. Ambas utilizan `fopen()` y `fclose()` para reservar y liberar memoria en la pila respectivamente.

- **¿A qué zona de memoria se accede? ¿En qué orden?**

  Primero se cargan en memoria las variables globales y las constantes del programa, luego se van guardando las variables locales y los parámetros de las funciones. Posteriormente, mientras el programa se está ejecutando, se reservará (guardando el stack pointer para volver a la pila cuando termine la tarea) y liberará la memoria necesaria.

- **¿Dónde están los fallos de caché?**

  Al ejecutar Cachegrind, obtenemos un fichero de salida, en el que podemos observar que la memoria caché de datos de primer nivel sufre unas pérdidas mínimas, pero comparadas con la caché de instrucciones son bastante grandes. Estas pérdidas llegan a la caché de último nivel (que contiene datos e instrucciones) donde el número de aciertos es muy bajo.
  Mediante cg_annotate estudiamos el fichero de salida del Cachegrind y observamos las posiciones donde se dan los 'miss' de la caché de datos, mayoritariamente se dan en los bucles de las funciones que acceden al array que contiene la imagen. para modificarlo como: `vertical_edge_detect()`, `fprintf()`, `apply_threshold()`, `invert_colours()` y `horizontal_edge_detect()`.

- **¿Se puede reducir la memoria caché de primer nivel? ¿Hasta dónde?**

  Sí, porque tenemos una tasa de fallos próxima al 0% (el número total de accesos a caché es muy elevado y aunque hay bastantes fallos, es prácticamente despreciable). Esto implica que no estamos siendo eficientes y que estamos malgastando los recursos que tenemos. Por lo que podríamos reducir la memoria caché de primer nivel para aumentar la eficiencia de los accesos a memoria, siempre y cuando la tasa de fallos sea menos al 1%.

2. Sin cambiar el programa:

- **Optimizar la memoria, mínima memoria caché necesaria (sin pérdida significativa de rendimiento), mínima RAM necesaria.**

  Podemos optimizar el programa sin cambiarlo con las opciones de optimización que incorpora el compilador gcc. Con la variable -O podemos controlar el nivel de optimización de todo el código. Al cambiar este valor, la compilación de código tomará algo más de tiempo, y utilizará mucha más memoria, especialmente al incrementar el nivel de optimización.

  Existen siete niveles de optimización: -O0, -O1, -O2, -O3, -Os, -Og y -Ofast. Se debe utilizar solo uno de ellos en /etc/portage/make.conf.

-O0	Desconecta por completo la optimización y es el predeterminado.
-O1	Es el nivel de optimización más básico. El compilador intentará producir un código rápido y pequeño sin tomar mucho tiempo de compilación.
-O2	Es el nivel recomendado de optimización. El compilador intentará aumentar el rendimiento del código sin comprometer el tamaño y sin tomar mucho más tiempo de compilación.
-O3	Es el nivel más alto de optimización posible. Activa opitimizaciones caras en términos de tiempo de compilación y uso de memoria. No garantiza una mejora de rendimiento, en muchos casos puede ralentizar un sistema debido al uso de binarios de gran tamaño y mucho uso de la memoria. Además puede romper algunos paquetes por lo que no es recomendable.
-Os	Optimizará el tamaño del código. Activa todas las opciones de -O2 que no incrementan el tamaño del código generado. Es útil para máquinas con capacidad limitada de disco o con CPUs que tienen poca caché.
-Og	Nuevo en GCC 4.8. Trata de solucionar la necesidad de realizar compilaciones más rápidas y obtener una experiencia superior en la depuración a la vez que ofrece un nivel razonable de rendimiento en la ejecución. Deshabilita optimizaciones que podrían interferir con la depuración.
-Ofast	Es nuevo en GCC 4.7. Se trata del ajuste -O3 y las opciones: -ffast -math, -fno -protect -parens y -fstack -arrays. No es recomendable ya que no cumple estándares estrictos.

  La caché de datos se puede reducir hasta 4096 Bytes con un 0.2% de tasa de error. Mientras que la caché de instrucciones se puede reducir hasta 8192 Bytes con una tasa de 0.65%.

- **Proponer cambios arquitecturales para mejorar rendimiento / consumo. Estructura de la jerarquía de memoria**

  Para mejorar el rendimiento y/o el consumo del sistema habría que adaptar de la mejor forma posible el tamaño de la caché de datos y sobretodo la de instrucciones, también se debería suprimir la caché de último nivel dada su pequeña tasa de aciertos, de forma que el tiempo medio de acceso a memoria disminuiría considerablemente a la vez que el consumo.

3. Cambiar el programa para:

- **Mejorar la tasa de aciertos y/o reducir la memoria caché. Estimar la mejora en tamaño de memoria. Estimar el impacto en rendimiento.**
- **Reducir el consumo. Estimar la mejora en consumo. Estimar el impacto en rendimiento.**

Tras el análisis anterior hemos observado que cuanto más pequeñas sean las memorias caché, la tasa de fallos se reducirá considerablemente. En este caso la mayor tasa de fallos se produce en las funciones con las que modificamos la imagen. El programa realiza cuatro veces el tratamiento de la señal para cada procesador, por lo que sería interesante mejorar este proceso, por ejemplo, accediendo una única vez a la imagen y realizando todas las funciones de tratamiento en un único paso. De esta forma, nos ahorraríamos tres cargas de la imagen que podrían ser un fallo en la caché. Otra opción podría ser procesar la imagen por zonas, por lo que tendríamos bucles más pequeños, pero luego también tendríamos que considerar las zonas intermedias lo que supondría un aumento de las líneas y la complejidad del código (como en las prácticas del laboratorio de ARQU).

También podríamos mejorar los procesos de lectura y escritura en los que se leen filas de una sola columna en vez de en varias y se recorren filas con varias columnas respectivamente, por lo que nos interesa que aumente la localidad en las cachés y por tanto el rendimiento.

Por último con la herramienta eCacti podemos estimar la disipación de potencia tanto dinámica como estática a la hora de leer y escribir en la caché, el tiempo de acceso de lectura a la misma, su área y la configuración óptima en función de algunos parámetros. Además también podemos observar las diferencias de eficiencia y de uso de recursos y las mejores configuraciones posibles antes y después de optimizar el código.