Universidad Zaragoza
1542

Trabajo Fin de Grado

Optimización de memoria y tiempo de ejecución de ORB-SLAM2
Memory and running time optimization of ORB-SLAM2

Autor

Pablo Luesia Lahoz

Director

Juan Domingo Tardós Solano

Grado de Ingeniería Informática
Escuela de Ingeniería y Arquitectura
2018

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

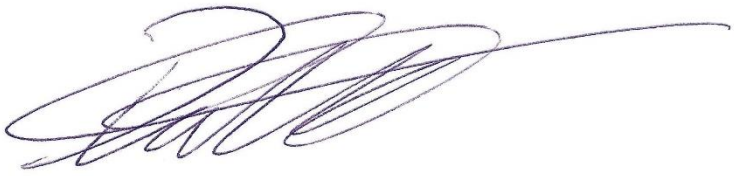
D./Dª. Pablo Luesia Lahoz,

con nº de DNI 25200520 H en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Optimización en memoria y tiempo de ejecución de ORB-SLAM2

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 28 de Junio de 2018



Fdo: Pablo Luesia Lahoz

Optimización de memoria y tiempo de ejecución de ORB-SLAM2

Resumen

ORB-SLAM2 es una librería software desarrollada por el grupo de robótica y tiempo real de la Universidad de Zaragoza que permite calcular en tiempo real la trayectoria seguida por un agente (un robot, un vehículo autónomo, un dron, etc.), mientras construye un mapa del entorno, a partir de las imágenes obtenidas por una cámara embarcada en el agente. El objetivo de este trabajo es optimizar la librería en uso de memoria y tiempo de ejecución, y extenderla con funciones de almacenamiento de mapas.

Se ha realizado un análisis de la ocupación en memoria y se ha comprobado que aumenta a medida que la cámara explora el entorno y el mapa va creciendo. Sin embargo, también se ha observado que, aunque el algoritmo elimine elementos del mapa por considerarlos redundantes, la ocupación en memoria no disminuye. Se ha implementado el uso de punteros inteligentes para los elementos del mapa, sin que la situación mejore. La conclusión es que hay una elevada fragmentación de la memoria, y aunque los datos desechados se eliminen correctamente, el sistema no es capaz de reutilizar la memoria liberada. Sería necesario un rediseño completo de las estructuras de datos del mapa, que queda fuera del alcance de este trabajo.

Para el reconocimiento de lugares ORB-SLAM2 utiliza un método de bolsa de palabras, en el que los puntos de interés de cada imagen se pasan por un árbol de vocabulario para convertirlos en palabras visuales. Para optimizar esta parte, se ha implementado el árbol mediante una estructura de datos implícita, asegurando su contigüidad en memoria. Con ello se ha disminuido drásticamente el tiempo de carga del vocabulario desde disco, su ocupación en memoria, y el tiempo de recorrido del árbol, gracias, en parte, al mejor aprovechamiento de la caché del procesador.

La librería se ha extendido con funciones de almacenar mapas en disco y cargarlos en otra ejecución, utilizando para ello la librería Boost. Con este cambio se permite compartir mapas entre distintas ejecuciones o incluso entre distintos sistemas. Esta extensión abre nuevas vías de trabajo futuro sobre esta librería, como mapeo de entornos multi-sesión o con varios agentes en paralelo.

Como baterías de prueba para comprobar el correcto funcionamiento del sistema, así como para calcular las prestaciones obtenidas, se han utilizado las secuencias de imágenes KITTI y EUROC, proporcionadas por la comunidad robótica.

ÍNDICE

1 INTRODUCCIÓN

1.1 MOTIVACIÓN

La robótica avanza cada vez más hacia la creación de sistemas de movimiento autónomo. Los coches inteligentes, cuya circulación es automática, o el uso de drones para envío de paquetes sin piloto, son algunos de los ejemplos. En estos proyectos, aparece el problema de la localización espacial del sistema. Campos como la realidad virtual o aumentada, o aún más distantes, como las asistencias a las intervenciones quirúrgicas, comparten este mismo requisito. Para resolverlo, es necesario conocer la trayectoria realizada, así como realizar un modelado del entorno. Las técnicas de *Simultaneous Localization And Mapping* (SLAM) surgieron como solución a este problema.

Las técnicas de SLAM realizan una construcción de un mapa en un entorno desconocido, por medio de uno o varios sensores, y realizan a su vez un seguimiento de la trayectoria de dicho sensor.

El uso de SLAM requiere una gran cantidad de tiempo de cálculo, así como espacio suficiente para almacenar toda la información del mapa. Esta condición limita el uso de estas técnicas en el ámbito de la robótica. Pese a que las prestaciones de los procesadores actuales están aumentando drásticamente, cada vez a más bajo precio, la mejora de los algoritmos de SLAM respecto a tiempo y espacio sigue siendo una prioridad.

SLAM también se puede aplicar a sistemas compuestos por varios subsistemas independientes. Un vuelo de varios drones sincronizados puede ser uno de los ejemplos. Es necesario en este caso, desarrollar un lenguaje común entre distintas máquinas, que permita una comunicación rápida y eficaz a la hora de compartir los datos utilizados por SLAM.

1.2 OBJETIVOS Y ALCANCE

Los objetivos de este proyecto se realizan sobre un software de SLAM desarrollado por el grupo de robótica de la Universidad de Zaragoza, llamado ORB-SLAM2 [1]. Este software consiste en una librería *open-source* (código libre), desarrollada en C++, que permite la generación de mapas de puntos dispersos utilizando cámaras RGB-D (cámara de color y profundidad), monoculares, y estereoscópicas.

Las prestaciones de la máquina necesarias para utilizar esta librería deben ser altas. Pese a que está diseñada para el funcionamiento en tiempo real, el análisis y la comparación de imágenes es costosa computacionalmente. Con la realización de los objetivos de este proyecto, se pretende optimizar la librería, mientras mantiene su funcionamiento actual, haciendo más probable el uso del sistema en máquinas de menores prestaciones.

Se pretende además extender la librería, mediante la funcionalidad de almacenar los mapas generados de manera persistente. La realización de este último objetivo implica definir un formato, entendible por ORB-SLAM2, para los mapas generados. Este puede ser un avance hacia la comunicación entre distintos sistemas que generen los mapas.

Enumerando los objetivos concretamente, son:

1. El estudio de la memoria utilizada por ORB-SLAM2 y estudiar posibles optimizaciones en el espacio usado
2. Optimizar el uso de memoria y tiempo de carga y ejecución del árbol de vocabulario
3. Analizar las partes más críticas en tiempo de ejecución de ORB-SLAM2 e implementar optimizaciones utilizando instrucciones SIMD
4. Definir una serialización que permita el almacenamiento y posterior carga de mapas

Este proyecto se realiza en paralelo a otros de optimización del tracking de la librería [2]. Por ese motivo, los objetivos de optimización en tiempo de ejecución solo se centran en el árbol del vocabulario. Con la cumplimentación de los objetivos, se espera conseguir una nueva versión base de ORB-SLAM2 que pueda utilizarse en nuevos proyectos del grupo de robótica de UNIZAR.

1.3 METODOLOGÍA

El proyecto se enfoca en el ámbito de la investigación. Se ha seguido una metodología ágil, marcando la finalización de los objetivos el final de la iteración. Si el objetivo finalizado ha resultado exitoso, la modificación asociada a dicho objetivo en el software se utiliza en la siguiente iteración.

El código se ha gestionado en un repositorio privado personal de GitHub, y se ha publicado el resultado en un repositorio privado del grupo de Robótica, Percepción y Tiempo Real de la Universidad de Zaragoza.

1.4 HERRAMIENTAS

Todas las herramientas utilizadas durante el proyecto se enumeran a continuación.

- C++: Lenguaje de programación base en el que está desarrollado ORB-SLAM2.
- ORB-SLAM2: Librería base que implementa un sistema de SLAM. Esta es sobre la que se han realizado el proyecto.
- OpenCV: Librería para C++ para la manipulación y análisis de imágenes. Proporciona a su vez mecanismos para realizar cálculo matricial y vectorial.
- DBoW2: Librería que implementa una bolsa de palabras, utilizada en la comparación de imágenes en el ámbito de visión por computador.

- Git & GitHub: Herramienta utilizada para el control de versiones, y para realizar copias de seguridad en la nube, así como compartir el código implementado.
- Valgrind: Herramienta que permite realizar perfiles de memoria.
- Boost: Librería para C++, que extiende la funcionalidad de dicho lenguaje, realizando operaciones de bajo nivel.
- CLion: Entorno de programación gratuito para estudiantes de JetBrains.
- Google & Google Scholar: motor de búsqueda utilizado para la obtención de documentación.
- KITTI & EUROC: Secuencias de imágenes utilizadas como banco de pruebas, proporcionadas por la comunidad robótica.

1.5 ESTRUCTURA DEL DOCUMENTO

Este documento recoge en su completitud, todo el trabajo realizado. Se pretende que sea comprensible por sí mismo, otorgando las explicaciones pertinentes. La información abordada por cada capítulo, se puede ver resumida a continuación:

2. *Sistema original: ORB-SLAM2*: Se describe el funcionamiento general de la librería ORB-SLAM2.

3. *Análisis de memoria y optimizaciones*: Metodología, y resultados de un análisis de memoria de ORB-SLAM2, con optimizaciones realizadas y sus resultados.

4. *Optimización del árbol del vocabulario*: Explicación del funcionamiento de la bolsa de palabras en ORB-SLAM2, y del árbol de vocabulario asociado, así como optimizaciones realizadas. Estudio de instrucciones SIMD para optimizar el cálculo de distancias entre descriptores.

5. *Almacenamiento y carga de los mapas en disco duro*: Explicación de la serialización de los mapas de ORB-SLAM2, y su uso para almacenar en disco con Boost.

6. *Conclusiones del proyecto y posibles proyectos futuros*: Conclusiones del trabajo, y posibles ámbitos de investigación.

7. *Estructura temporal del proyecto*: Muestra el tiempo dedicado a cada objetivo, y su distribución temporal.

2 SISTEMA ORIGINAL: ORB-SLAM2

2.1 INTRODUCCIÓN

ORB-SLAM2 es una librería que implementa un algoritmo de SLAM. Está desarrollada por Raúl Mur Artal para el grupo de Robótica y Tiempo Real de la Universidad de Zaragoza, como su trabajo de doctorado. Esta librería es capaz de calcular la trayectoria realizada por un sensor de visión, y generar un modelado del entorno, por medio de puntos dispersos del mapa, en tiempo real. El sensor de visión utilizado puede ser de tres tipos distintos: RGB-D, monoculares y estereoscópicas.

2.2 DETECTOR, EXTRACTOR DE DESCRIPTORES Y RECONOCIMIENTO

La librería hace uso del algoritmo de detección de esquinas FAST [3]. Los puntos de interés detectados se almacenan como descriptores ORB [4]. Los descriptores ORB son invariantes a rotación y escala, su extracción y comparación es poco costosa computacionalmente, y cada descriptor puede almacenarse como una ristra de 256 bits. La finalidad de estos descriptores es el reconocimiento, entre dos imágenes, o entre imágenes y puntos del mapa.

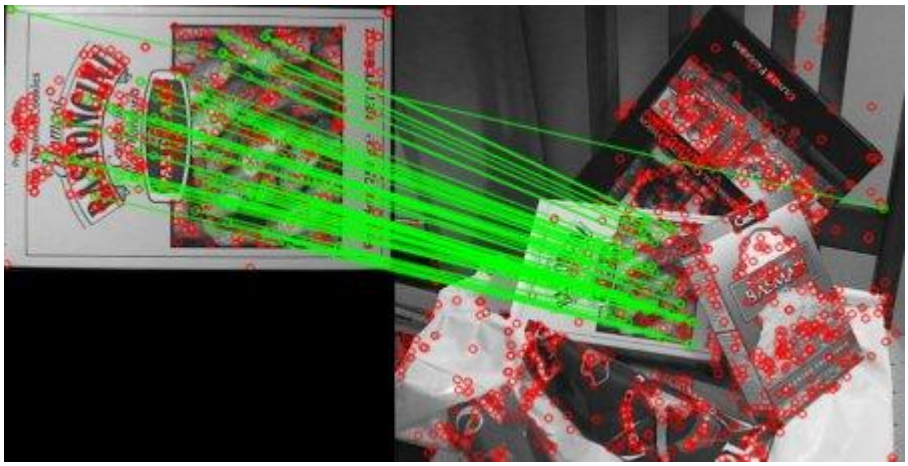


Figura 1 - Descriptores ORB (en rojo) de una imagen, y su reconocimiento en una segunda imagen (ORB coincidentes en verde). Fuente: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_matcher/py_matcher.html

El reconocimiento de lugares es computacionalmente costoso. Por ello, ORB-SLAM2 utiliza la técnica de la bolsa de palabras, basada en DBoW2. Se describe dicha técnica con detalle en el apartado 4.2. .

2.3 TRACKING, LOCAL MAPPING Y LOOP CLOSING

El funcionamiento de la librería está implementado por medio de tres hilos principales, como se puede apreciar en la figura 2. Estos se denominan *Tracking*, *Local Mapping* y *Loop Closing*.

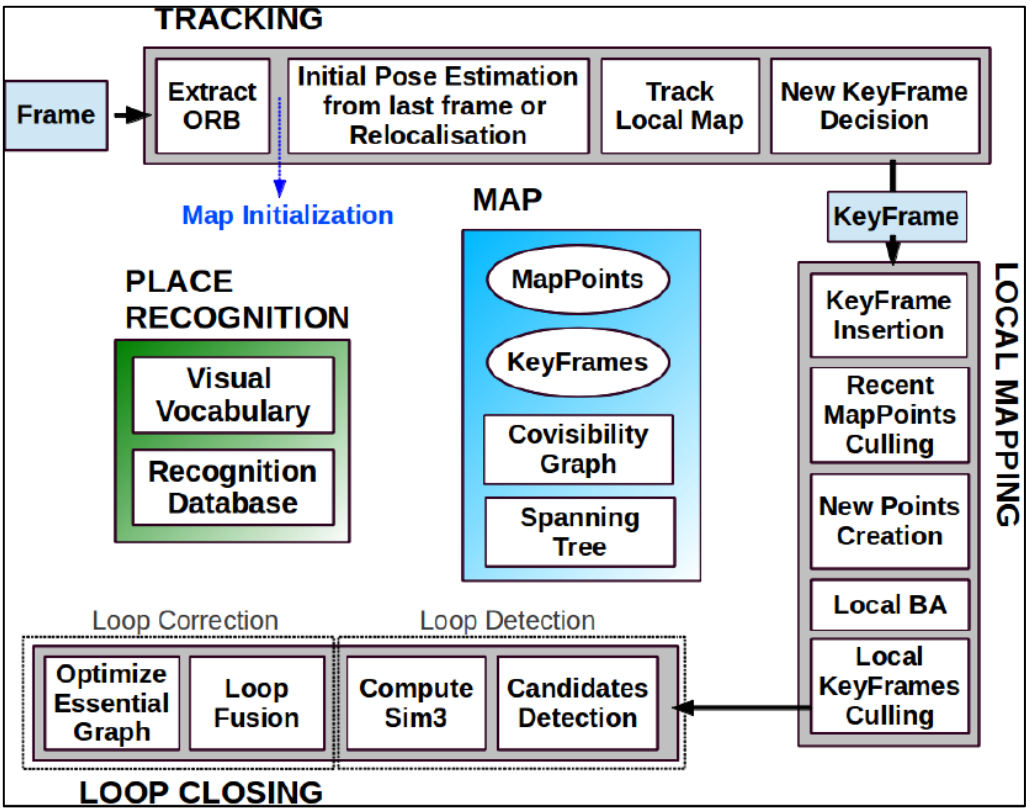


Figura 2 - Sistema completo, mostrando los distintos pasos de los hilos de Tracking, Local Mapping, Loop Closing. Se muestran también los componentes principales del módulo de reconocimiento de lugares.

En el funcionamiento estándar del hilo de *Tracking*, este se encarga de obtener las imágenes del sensor, y localizarlas en el mapa generado, mediante reconocimiento de descriptores sobre los puntos del mapa. También decide cuándo es necesario incluir una nueva imagen de referencia o *keyframe*. Tanto la inicialización del mapa, como una relocalización global de la cámara en caso de pérdida de la posición se realizan en este hilo.

Local Mapping es el nombre que recibe el hilo cuyo propósito es generar el mapa local, a partir de los nuevos *keyframes* generados por el hilo de *Tracking*. Realiza *bundle adjustment* (BA) local. Bundle Adjustment es una técnica de optimización no lineal en visión por computador, cuyo objetivo es calcular la solución óptima para las coordenadas de los puntos de interés del mapa, y para la posición y orientación de cada uno de los *keyframes* que han observado dichos puntos.

Por último, el hilo *Loop Closing* detecta grandes bucles en la trayectoria, para cada *keyframe*. Estos bucles se dan cuando la trayectoria revisita una zona ya recorrida del mapa. Una vez se detecta un bucle, *Loop Closing* ejecuta una corrección global de la deriva generada, por medio de un *bundle adjustment* global.

2.4 *MAPPOINTS* Y *KEYFRAMES*

Los dos elementos fundamentales de ORB-SLAM2 son los puntos del mapa o *mappoints* y las imágenes de referencia o *keyframes* (véase figura 3). El mapa se construye por medio de estas dos clases, siendo los *mappoints* la información del entorno, y los *keyframes* las imágenes que observan dichos puntos, posibilitando las correcciones pertinentes. El contenido de estas clases es:

- *Mappoints*:
    - Su posición en el espacio tridimensional.
    - El vector unitario que indica la dirección y sentido de su observación. El origen de este vector es el vector del centro de la cámara del *keyframe* que lo observó.
    - Su descriptor ORB más representativo, cuya distancia de Hamming a otros descriptores del mismo punto, obtenidos desde otros *keyframes*, es mínimo.
    - La distancia máxima y mínima desde la que puede ser observado. Marcado por las limitaciones de los descriptores ORB.
- *Keyframes*:
    - Posición y orientación de la cámara
    - Parámetros de calibración intrínsecos de la cámara
    - Todos los descriptores ORB obtenidos de puntos de interés, asociados o no con un *mappoint*. También almacena dicha asociación.

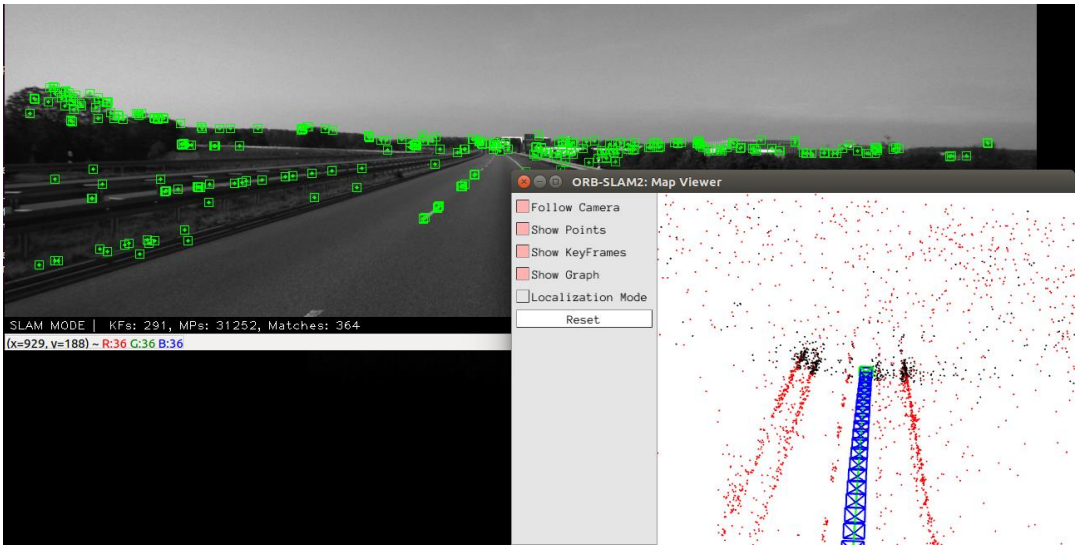


Figura 3 – Arriba izquierda, mappoints proyectados sobre la imagen. Abajo derecha, mappoints y keyframes

Los *keyframes* implementan a su vez un *grafo de covisibilidad*, y un *grafo esencial*. Ambos grafos son no dirigidos, en los cuales, sus nodos corresponden a los *keyframes*. Existe entre dos nodos una arista, si estos comparten más de n observaciones. Para el *grafo de covisibilidad* n debe ser mayor que 15, mientras que, para el *esencial*, debe ser mayor que 100. El *grafo de covisibilidad* se utiliza para definir el mapa local, mientras que las correcciones en el hilo de *Loop Closing*, se propagan un a través del *grafo esencial*. Obsérvese, que el *grafo esencial* deriva del *grafo de covisibilidad*, con objetivo de presentar una optimización. Estos grafos se generan automáticamente en la inserción de un *keyframe*.

3 ANÁLISIS DE MEMORIA Y OPTIMIZACIONES

3.1 INTRODUCCIÓN

La memoria ocupada por ORB-SLAM2 se incrementa a medida que se expande el mapa y la trayectoria. Usado en recorridos extensos, el tamaño de la memoria puede representar un impedimento si ésta es limitada. Por ese motivo, analizar y tratar de optimizar la memoria utilizada por el sistema es primordial. En este capítulo se expone la metodología utilizada para el análisis, el análisis realizado, optimizaciones derivadas del análisis, y resultados de las optimizaciones para la memoria ocupada en ejecución de ORB-SLAM2.

3.2 METODOLOGÍA DEL ANÁLISIS

La memoria en los computadores se encuentra dividida en dos zonas: estática y dinámica. La memoria estática es aquella liberada al final del dominio de la variable que la usa. Corresponde a ella todos los tipos de datos básicos y cadenas de tamaño definido. Por otro lado, corresponde al desarrollador liberar la memoria dinámica. Se utiliza para datos de tamaño no definido, cuyo espacio no se puede predecir en tiempo de compilación.

En ORB-SLAM2, la mayor parte de la memoria utilizada es dinámica. Exceptuando variables locales de las funciones y métodos, todos los demás datos se almacenan en esta zona. Analizar la memoria estática de ORB-SLAM2 carece de interés; el análisis de memoria se realiza sobre la dinámica.

Se ha utilizado Valgrind [5] para el análisis dinámico. Es un marco de trabajo con varias herramientas disponibles para sistemas operativos de base Linux. Entre ellas se encuentra *Massif*. La herramienta realiza perfiles del *heap*, que es la zona de un programa correspondiente a la memoria dinámica. La herramienta es capaz de reconocer toda la memoria ocupada, y diferenciar la memoria utilizada por cada componente del sistema. Hay que diferenciar bien estos conceptos. La memoria utilizada es la que efectivamente utiliza la ejecución. La ocupada incluye memoria muerta, efecto de que las arquitecturas suelen reservar memoria por bloques, o por fragmentaciones debido a borrados. Por consiguiente, la memoria utilizada es siempre menor que la ocupada, y la ocupada es toda la memoria que no puede ser usada por un componente externo.

Las secuencias KITTI han sido elegidas como banco de pruebas para el análisis. Estas secuencias, obtenidas por el recorrido de un vehículo, presentan una gran extensión en la trayectoria, así como la gran densidad de puntos de interés facilitados por un entorno

real. Los dos factores son los deseados en un análisis de memoria, debido a la densidad y tamaño del mapa y trayectoria generados.

Figura 4 - Mapa y trayectoria generada por ORB-SLAM2 en la secuencia KITTI 00

En síntesis, se han utilizado las secuencias KITTI para ejecutar un análisis de la memoria dinámica de ORB-SLAM2 por medio de Valgrind. Se espera encontrar con ello irregularidades, y focalizar las optimizaciones de memoria.

3.3 RESULTADOS DEL ANÁLISIS

En ORB-SLAM2, se puede dar el borrado de *keyframes* o *mappoints*. Para que se dé el borrado de un *keyframe*, el 90% de los *mappoints* vistos por el mismo, deben ser vistos por al menos otros tres *keyframes*. El borrado de un *mappoint*, se puede realizar si:

- Solo se observa en menos del 25% de las imágenes en las que se predice su observación, durante los tres primeros *keyframes* tras su creación.
- Si se observa desde menos de tres *keyframes*.

Este borrado restrictivo, permite mantener un mapa correcto en ORB-SLAM2. Con esta política, el sistema apenas debería incrementar el uso en memoria dinámica cuando se da un bucle. Si se da, los *mappoints* y *keyframes* relevantes ya se habrán añadido, y los datos erróneos o redundantes se borran. La silueta del diagrama de memoria respecto al tiempo debería ser escalonada, correspondiendo las secciones de menor pendiente a revisitas de la trayectoria en el mapa. Estas zonas podrían tener aspecto aserrado, debido a la creación y borrado iterativo de estos dos componentes.

El diagrama de la figura 5 corresponde con la secuencia 00 de KITTI, que contiene bucles de gran tamaño (véase la figura 4). Contrariamente al resultado esperado, se observa un incremento constante del uso de memoria dinámica, obviando los dos grandes picos que corresponden a *bundle adjustment* globales. Dado que el funcionamiento es correcto, se ha llegado a la conclusión de que el borrado no se realiza correctamente. Los

keyframes y *mappoints*, generados no se liberan de memoria correctamente, aunque estos dejen de ser útiles, y no sean accesibles.

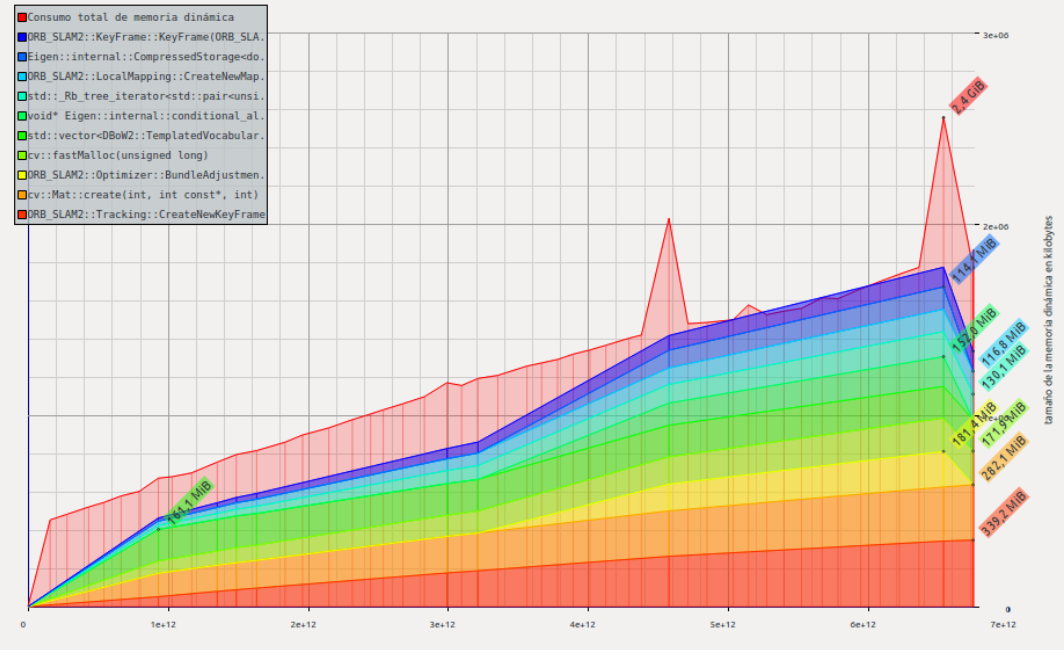


Figura 5 - Diagrama de uso del heap (memoria dinámica) en la secuencia KITTI-00. En el pico de 2,4 GiB, la memoria está ocupada por: mapoints(389,9 MiB), keyframes(779,9 MiB), DBoW2 (407,4 MiB), optimizador y bundle adjustment (599,2 MiB).

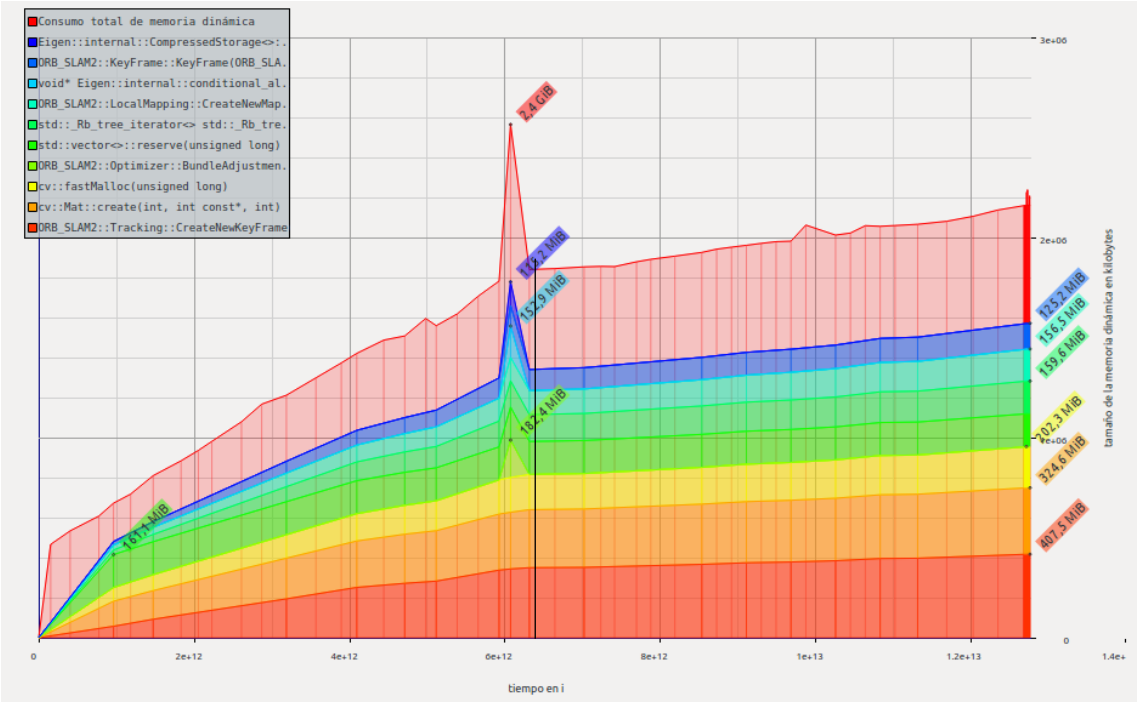


Figura 6 – Diagrama de uso del heap con dos pasadas sobre la secuencia 00 de KITTI. La segunda pasada comienza tras la línea marcada. En el pico, la memoria está ocupada por: keyframes (771,7 MiB), mappoints (319,2 MiB), DBoW2 (489,4 MiB), optimizador y bundle adjustment (631,4 MiB). Al final de la ejecución, la memoria está ocupada por: keyframes (856,3 MiB), mappoints (413,0 MiB) y DBoW2 (489,6 MiB).

Aprovechando que la secuencia KITTI 00 es cíclica, y finaliza en el punto de partida, se ha generado una prueba para corroborar que no se realiza una correcta liberación de memoria. Esta ha consistido en duplicar la secuencia, de forma que se realicen dos pasadas continuadas en una misma ejecución. La memoria dinámica utilizada se observa en la figura 6. En la tabla 1, se puede observar que la proporción de incremento en memoria y número de *keyframes* y *mappoints* no es pareja.

Durante la segunda pasada por el entorno, el número de *mappoints* añadidos es proporcionalmente menor a la memoria ocupada por estos. Esto es debido a que los nuevos *mappoints* tienden a ser redundantes y son borrados rápidamente, pero la memoria no se consigue liberar correctamente. En el caso de los *keyframes* se produce el efecto contrario. Durante la segunda pasada la inserción de un nuevo *keyframe* es rara, ya que el mapa ya contiene los resultados de la primera pasada, y se realiza un menor borrado de los nuevos *keyframes*, ya que cuando se inserta uno, no se añaden otros próximamente que hagan cumplir su condición de borrado. En cambio, durante la primera pasada, debido al desconocimiento del entorno, se crearon muchos *keyframes* que resultaron redundantes y se borraron. La mayor ocupación proporcional de los *keyframes* de la primera pasada indica que la memoria correspondiente a los borrados no se liberó adecuadamente.

		Final 1ª iteración	**Final 2ª iteración**	**Incremento**
keyframes	Número	1.426	1.736	21,73%
	Memoria	771,7 MiB	856,3 MiB	10,96%
mappoints	Número	140.831	164.827	17,03%
	Memoria	319,2 MiB	413,0 MiB	29,38%

Tabla 1 - Número de keyframes y mappoints, y memoria utilizada al final de la primera y segunda iteración de la secuencia KITTI 00.

Estos dos comportamientos observados confirman la suposición de la incorrecta liberación de memoria. El problema se ve acrecentado si se considera que estas dos clases representan el grueso del programa en memoria. Su apropiada liberación representará una mejoría global en el uso de la misma. Las optimizaciones realizadas en esta sección del proyecto se centran en mitigar el problema, que, en resumen, es generar un recolector de basura apropiado.

3.4 OPTIMIZACIONES REALIZADAS: PUNTEROS INTELIGENTES

Los punteros suelen utilizarse para gestionar la memoria dinámica. Este tipo de dato es una referencia a otro dato. Cuando termina el dominio de un puntero, la propia variable se libera, pero el dato apuntado puede continuar persistente en memoria, si ha sido declarado en memoria dinámica. Los punteros inteligentes, hacen uso de un contador de referencias. Cuando todos los punteros a un dato son borrados, el contador vale 0, y automáticamente se borra el dato referenciado.

Se han utilizado punteros inteligentes para implementar un borrado de las clases en ORB-SLAM2. Esta ha sido la solución más simple encontrada, frente a sustituir la

implementación previa, para realizar un borrado explícito. Otro detalle fundamental para el uso de punteros inteligentes, ha sido el borrado instantáneo que realiza de los datos cuando ya no se puede acceder al mismo.

La implementación con punteros inteligentes es muy sencilla. Basta con sustituir todos los punteros estándar con estos nuevos en el código. Una complicación surge con las referencias cruzadas entre *keyframes*, y entre *keyframes* y *mappoints*. Estas clases no realizan borrado automático de sus referencias, lo que provoca la existencia de punteros inteligentes residuales que impiden el borrado del dato. La dificultad ha obligado al uso de punteros débiles entre las referencias de las clases, que no afectan al contador de referencias del puntero inteligente. Se ha modificado el destructor de ambas clases, para eliminar las referencias pertinentes, y generar así la consistencia deseada.

Para probar los resultados, se ha realizado la optimización primero con *mappoints*, y luego con *keyframes*, para observar las diferencias. El éxito de estas, condiciona el uso de punteros inteligentes para ambas clases.

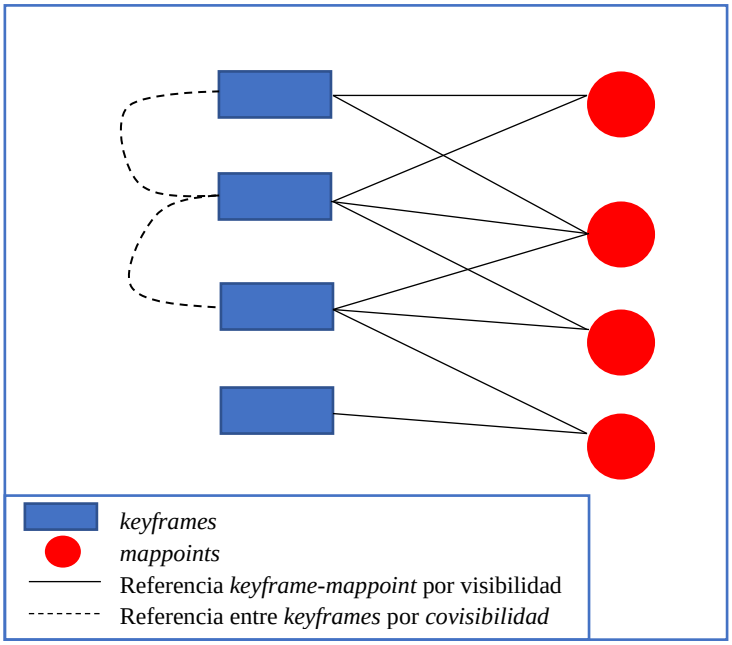


Figura 7 - Posible situación de referencias entre keyframes y mappoints en ORB-SLAM2. En este caso, existe arista en el grafo de covisibilidad entre keyframes si se comparte la visión de 2 o más mappoints.

3.5 RESULTADOS DE LAS OPTIMIZACIONES

La gestión de punteros inteligentes respecto a manipulación, creación y acceso de la variable que referencian, es más lenta que los punteros estándar. Esto se debe a que los punteros inteligentes no son tipos de datos simples de C++, y están implementados por medio de punteros estándar. Han de llamar a funciones de la clase y realizar las operaciones necesarias para su correcto funcionamiento. Por otro lado, el borrado se

realiza de manera implícita; el programa no debe recorrer sus estructuras de datos para realizar el borrado consistentemente. Como consecuencia, el borrado por medio de punteros inteligentes se realiza más rápido.

Los punteros inteligentes con los *mappoints* apenas representan cambios respecto a tiempos, como se puede apreciar en la tabla 2. Estos objetos son manipulados constantemente, creando cientos o miles por cada *keyframe* insertado. Al demostrar ser espurios, son borrados. El tiempo extra que se da en generar referencias, es compensado por el rápido borrado. Por otro lado, los *keyframes* con punteros inteligentes sí que representan una ligera mejoría en tiempo. Los *keyframes* son menos accedidos que los *mappoints*; el lento acceso de los punteros inteligentes apenas afecta frente a la mejoría del borrado.

	Local Mapping			**Tracking**		
Tiempo en ms	Media	Std	Mediana	Media	Std	Mediana
Original	273,740	181,509	296,870	11,99	2,94	11,39
P.I. para KeyFrame	268,295	187,910	271,673	11,27	3,04	11,27
P.I. para MapPoints	275,797	187,951	282,041	11,49	2,68	20,95

Tabla 2 - Tiempos del sistema original, del sistema con punteros inteligentes con keyframes, y con mappoints, para Local Mapping y Tracking en la secuencia 00 de KITTI.

Si se observan las figuras 8 y 9, y la tabla 3, no se ha conseguido ninguna mejoría respecto a la ocupación global en memoria. El motivo es la fragmentación. Para el sistema operativo, es más barato utilizar una zona de memoria virgen, que mantener una estructura de datos indicando zonas liberadas. Por lo tanto, aunque se libere memoria automáticamente con los punteros inteligentes, esta no se reutiliza por otras variables, y no se considera liberada. Obsérvese, que en el caso de los *keyframes*, la memoria ocupa incluso más. El motivo es el mayor tamaño de los punteros inteligentes frente a los punteros estándar.

	keyframes	*mappoints*	DBoW2	*Optimizador & Bundle Adjustment*	*Total*
Original	779,9 MiB	389,9 MiB	407,4 MiB	599,2 MiB	2,4 GiB
Punteros inteligentes para *keyframes*	846,0 MiB	427,2 MiB	407,3 MiB	554,9 MiB	2,5 GiB
Porcentaje mejora	-7,81%	-8,73%	0,02%	7,98%	-4%
Punteros inteligentes para *mappoints*	822,6 MiB	388,2 MiB	405,1 MiB	545,9 MiB	2,4 GiB
Porcentaje mejora	-5,19%	0,43%	0,56%	9,76%	0%

Tabla 3 - Memoria dinámica en su pico de más uso, del sistema original y del sistema con punteros inteligentes para keyframes y mappoints, en la secuencia 00 de KITTI.

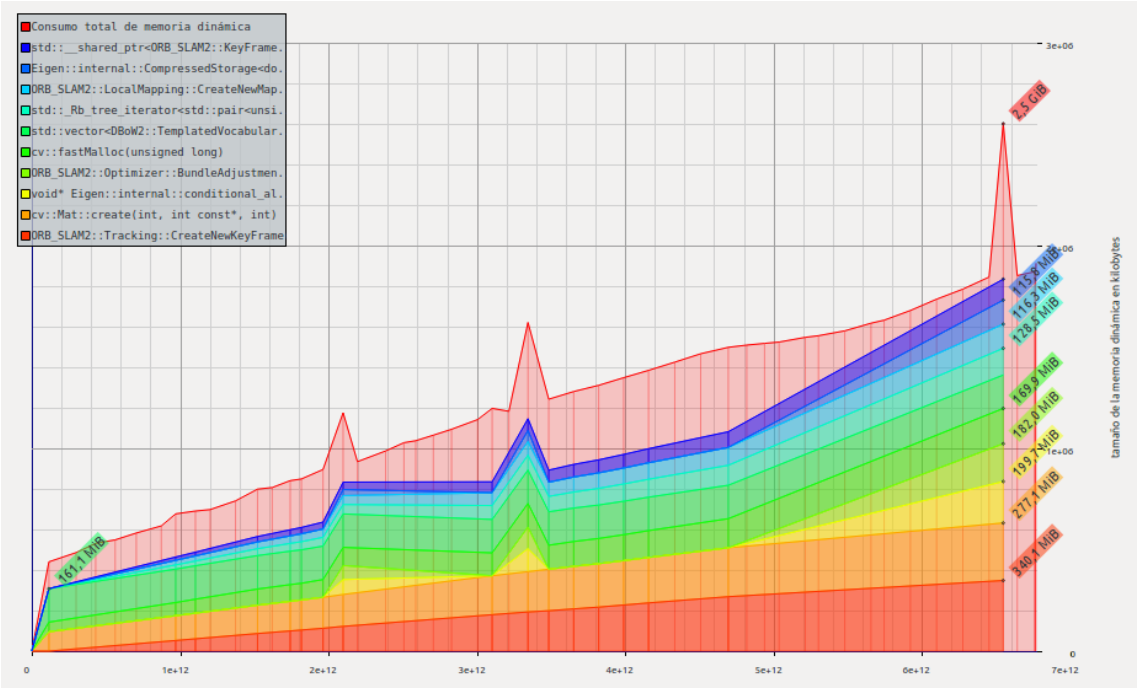


Figura 8 - Diagrama de uso del heap en la secuencia KITTI-00, tras el uso de punteros inteligentes con keyframes. En el pico de 2,5 GiB, la memoria está ocupada por: keyframes (846,0 MiB), mappoints (427,2 MiB), DBoW2 (407,3 MiB), optimizador y bundle adjustment (554,9 MiB).

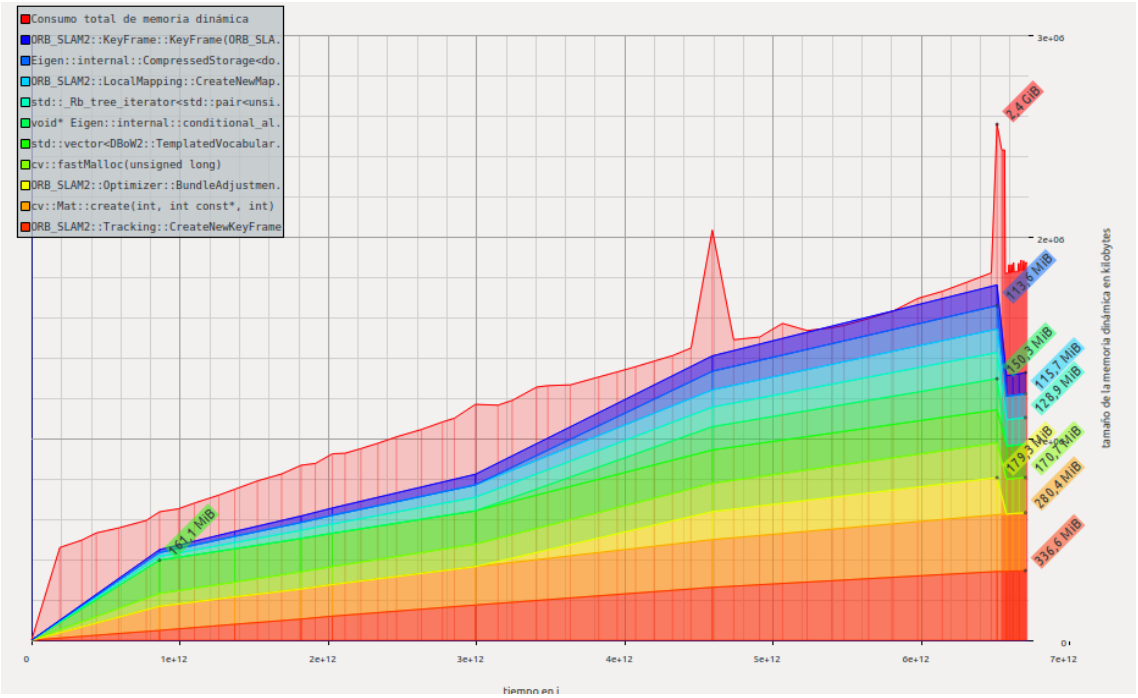


Figura 9 - Diagrama de uso del heap en la secuencia KITTI-00, tras el uso de punteros inteligentes con mappoints. En el pico de 2,4 GiB, la memoria está ocupada por: keyframes (822,6 MiB), mappoints (388,2 MiB), DBoW2 (405,1 MiB), optimizador y bundle adjustment (545,9 MiB).

4 OPTIMIZACIÓN DEL ÁRBOL DEL VOCABULARIO

4.1 INTRODUCCIÓN

El algoritmo básico para reconocimiento en visión es, dadas dos imágenes, extraer puntos de interés con sus descriptores en ambas imágenes, y comparar un descriptor de una, con todos los descriptores de la otra. Si dos descriptores tienen una distancia menor que d_{max} (distancia de Hamming para los descriptores ORB), se consideran el mismo descriptor. Es decir, el coste computacional para dos imágenes es cuadrático en el número de puntos extraídos. Para el funcionamiento en tiempo real es inviable, por el alto número de imágenes y puntos a comparar. ORB-SLAM2 utiliza por ese motivo una técnica de bolsa de palabras, basada en DBoW2 [6]. En este capítulo se describe el funcionamiento de la técnica de la bolsa de palabras en ORB-SLAM2, y las optimizaciones que se han realizado para la misma.

4.2 FUNCIONAMIENTO DE LA BOLSA DE PALABRAS

El modelo de bolsa de palabras o *bag of words*, es una técnica usada para la minería de datos. Consiste en codificar un documento como el vector de términos que contiene, ignorando el orden. Cada posición del vector corresponde a una palaba, y se almacena en cada posición el número de veces que aparece la palabra en el documento. En reconocimiento de imágenes, puede aplicarse esta misma técnica, utilizando *palabras visuales* que pueden obtenerse a partir de los descriptores de los puntos de interés.

El problema de utilizar esta técnica con los descriptores ORB, es su tamaño. En ORB-SLAM2 se almacenan como ristras de 256 bits. Un mismo punto de interés en dos imágenes distintas puede variar ligeramente en la codificación de su descriptor ORB, por cambios de luminosidad, movimiento o ruido, dificultando su reconocimiento. Por lo tanto, los descriptores ORB no pueden utilizarse directamente como *palabras visuales*.

La solución aplicada en ORB-SLAM2, ha sido el uso del Árbol de Vocabulario o *vocabulary tree* [7]. El árbol es el resultado de aplicar un agrupamiento jerárquico (k-medias jerárquico normalmente) a una cantidad masiva de descriptores. De esta manera, las *palabras visuales* utilizadas para el reconocimiento corresponden a cada uno de los agrupamientos o *clusters* obtenidos. Con el *vocabulary tree* obtenido, a cada punto de interés se le asocia la palabra del árbol más próxima a su descriptor. Para ello, se recorre el árbol, buscando en cada nivel el nodo que presenta menor distancia al descriptor buscado, y se profundiza por dicha rama, repitiendo el proceso hasta que se alcanza una hoja del árbol. Dicha hoja representa la palabra visual asociada a ese descriptor. Por tanto,

las hojas del árbol son el vocabulario utilizado para codificar el vector de la bolsa de palabras de cada imagen. En el caso del descriptor ORB, al ser un descriptor binario, se utiliza la distancia de Hamming.

El *vocabulary tree* ya se calculó para ORB-SLAM2 en su momento, y está almacenado en un fichero, para cargarlo al principio de una ejecución. Su factor de ramificación es de 10, y tiene 6 niveles de profundidad. Es decir, contiene 10^6 hojas como máximo, y, por ende, el vector de la bolsa de palabras ha de tener el mismo tamaño. Como una imagen tiene del orden de 10^3 descriptores, el vector es cuasi-vacío, y se codifica eficientemente, almacenando solamente un identificador de las hojas correspondientes a los descriptores que sí aparecen en la imagen.

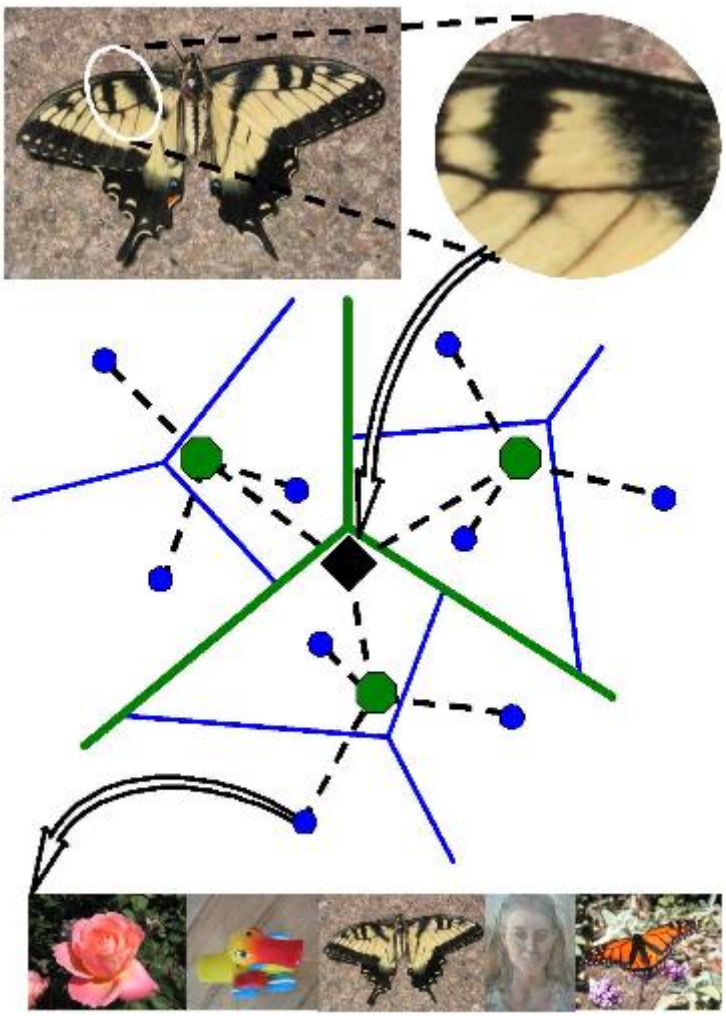


Figura 10 - Esquema del funcionamiento del vocabulary tree. Se ha extraido un descriptor de la imagen superior, y se ha recorrido el árbol, buscando la hoja más cercana. Imagen extraída de [5]

Una vez solucionado el problema del vector de términos, es necesario definir la comparación entre vectores de la bolsa de palabras. Se utiliza para ello *Term frecuency – Inverse document frecuency* (tf-idf). Esta medida, también originaria de minería de datos, mide la relevancia de un documento dada una consulta, pudiendo obtener un ranking de

los resultados. Tf-idf castiga términos comunes en todos los documentos, mientras da relevancia a los documentos en los que aparecen con frecuencia los términos de la consulta.

Este modelo permite, en su conjunto, realizar comparaciones entre distintos *keyframes*. Los descriptores ORB de un *keyframe* se codifican como hojas del *vocabulary tree*, generando así un vector de términos. Posteriormente se comparan los distintos *keyframes* mediante tf-idf para encontrar aquellos que son más similares. Se agiliza con este método el proceso de relocalización, y de detección de bucles, en los que el reconocimiento del lugar por medio de los *keyframes* disponibles es algo fundamental.

4.3 IMPLEMENTACIÓN PREVIA DEL ÁRBOL DEL VOCABULARIO

Para utilizar la bolsa de palabras, ORB-SLAM2 hace uso de DBoW2. Esta librería realiza una implementación del *vocabulary tree* recursiva. Es decir, cada nodo almacena punteros a sus nodos hijos. Por la creación recursiva del árbol, la distribución en memoria resultante es la del recorrido del árbol en profundidad (figura 11, centro). Sin embargo, el recorrido del *vocabulary tree* se realiza por medio de una búsqueda en un mismo nivel. Se realiza una comparación de distancias del descriptor sobre todos los hermanos (nodos con el mismo nodo padre), con objetivo de proseguir por aquel hermano más próximo.

La implementación recursiva comentada representa un inconveniente, si se considera acceso directo desde el procesador a memoria. Este acceso es varios órdenes de magnitud más lento que las operaciones realizadas dentro del procesador. El lento acceso originó la creación de una memoria adicional, denominada caché, de menor tamaño y mayor velocidad. Los procesadores hacen uso de la caché como almacén de datos temporal. Cuando se accede a memoria, se carga en caché un bloque de memoria que contiene el dato requerido. La hipótesis básica es que los datos de ese mismo bloque tienen una alta probabilidad de usarse a continuación. Como consecuencia, se minimiza el tiempo gastado por acceso a memoria.

La mayoría de procesadores actuales contienen caché, incluso varios niveles. La implementación recursiva no es capaz de explotar esta optimización hardware. Si se garantiza que los nodos hermanos están contiguos en memoria, el recorrido del árbol aprovechará mejor las cachés y será más eficiente.

4.4 ÁRBOL IMPLÍCITO Y NUEVA IMPLEMENTACIÓN DE LOS DESCRIPTORES

Un árbol implícito es una implementación de una estructura de datos tipo árbol sin necesidad de utilizar punteros. La implementación consiste en distribuir todos los nodos de un árbol sobre un vector o array en un orden preestablecido. La distribución debe permitir conocer el padre o los hijos de un nodo, a partir de su índice en el vector.

Distribuyendo los nodos según el recorrido en anchura del árbol, se pueden ordenar los nodos, de tal forma que se cumpla:

$$idx_{hijo} = (idx_{padre} + 1)B$$

donde idx_{hijo} es el índice del primer hijo del nodo idx_{padre}, y B es el factor de ramificación del árbol. Con dicha fórmula se puede calcular el índice del padre despejando idx_{padre}. El rango de nodos hijos del nodo correspondiente a idx_{padre}, va desde idx_{hijo} hasta $(idx_{hijo} + B - 1)$. Se consigue así la contigüidad entre nodos hermanos deseada (figura 11, derecha).

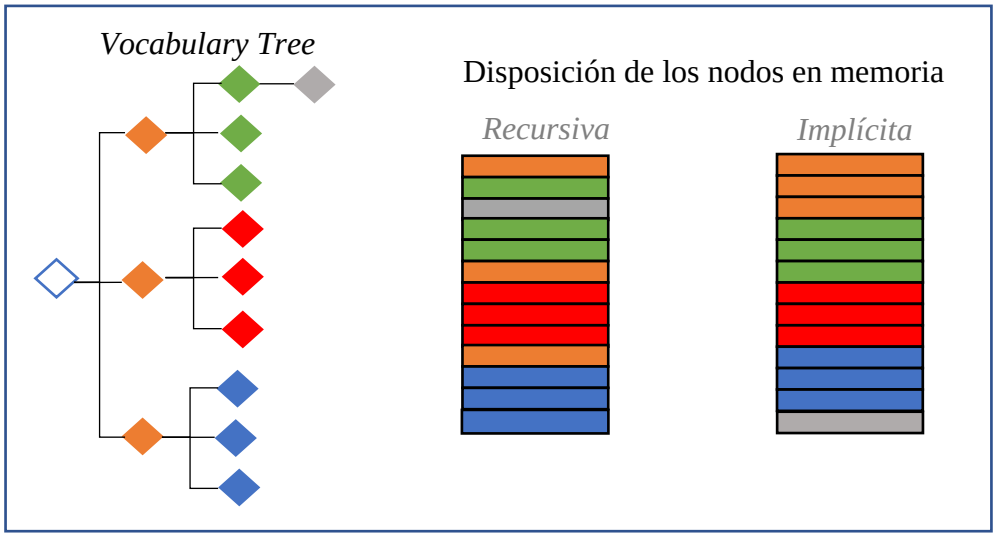


Figura 11 - Comparación de almacenamiento en memoria según la implementación recursiva o implícita del árbol

En este trabajo se ha re-implementado DBoW2 utilizando un *vocabulary tree* con estructura implícita. En caso de que el árbol no sea completo, se dejan vacíos los correspondientes elementos del vector, para asegurar que la fórmula para acceder a cada nodo sigue siendo correcta. Para evitar accesos a los posibles nodos vacíos, se ha añadido un vector auxiliar con el número de hijos para cada nodo.

Obsérvese, que puede conocer a priori el tamaño del vector, conociendo el factor de ramificación B, y el número máximo de niveles L. El tamaño se puede definir como $\sum_{i=1}^{L} B^i$. Ello permite reservar la memoria exacta que va a ocupar el árbol.

Durante toda esta sección, se menciona la continuidad en memoria del árbol. Pero esta solo será útil, en caso de que los propios términos sean contiguos. Los términos son en este caso descriptores ORB de 256 bits. En el sistema original, se implementan como matrices de OpenCV. Sin embargo, OpenCV no garantiza la continuidad en sus matrices, y, por ende, no se garantiza la continuidad entre nodos hermanos buscada. Se han re-implementado los descriptores ORB como cadenas de 32 caracteres. Es decir, los

descriptores son ahora cadenas de 256 bits. Las cadenas son contiguas, y se consigue la contigüidad deseada en el árbol.

4.5 Optimización del cálculo de la distancia entre dos descriptores

La operación de cálculo de distancia entre dos descriptores es aplicada repetidas veces en el recorrido del árbol del vocabulario. Para un árbol de factor de ramificación 10, y 6 niveles de profundidad, esta se puede llegar a calcular 60 veces para un solo descriptor. Por este motivo, se ha optado por optimizar este proceso, con intención de acelerar aún más el recorrido del árbol.

La distancia entre dos descriptores ORB se calcula mediante Hamming. La distancia de Hamming, consiste en contar el número de elementos diferentes de dos conjuntos. En este caso, el cálculo es equivalente a contar todos los bits distintos entre las cadenas que representan a los descriptores. Computacionalmente, este proceso se resume en aplicar una operación OR exclusiva o XOR entre ambas cadenas, y, posteriormente, contar todos los bits con valor 1 del resultado. Este último paso se conoce con el nombre de contar una población.

Se puede observar en el algoritmo 1, el código en C++ utilizado por DBoW2 para calcular la distancia. Obsérvese, que el algoritmo para contar la población solo funciona para datos de 32 bits, lo que requiere 8 iteraciones para los descriptores ORB de 256. La justificación para usar datos de dicho tamaño proviene de que la arquitectura común de los ordenadores actuales suele ser de 32 o 64 bits, y una sola instrucción puede aplicarse como mucho sobre registros de esos tamaños.

```
const int *pa = a.ptr<int32_t>();
const int *pb = b.ptr<int32_t>();

int dist=0;

for(int i=0; i<8; i++, pa++, pb++)
{
    unsigned  int v = *pa ^ *pb;
    v = v - ((v >> 1) & 0x55555555);
    v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
    dist += (((v + (v >> 4)) & 0xF0F0F0F) * 0x1010101) >> 24;
}

return dist;
```

Algoritmo 1 - Calculo de la distancia entre los descriptores a y b, utilizado en DBoW2 previo al trabajo.

La optimización sobre el cálculo de la distancia se ha aplicado en concreto sobre el conteo de una población. En una primera instancia se consideró el uso de instrucciones *Single Instruction Multiple Data* o SIMD. Estas funcionan mediante varios registros, a los que se le aplica la misma operación (véase la figura 12). Por ende, se podría aplicar toda la misma operación sobre el vector de 32 bytes correspondiente al descriptor ORB. No obstante, las operaciones *population count* de los procesadores estándar, son más eficientes para vectores de dicho tamaño [8]. Estas operaciones aplican el conteo de bits sobre registros de tamaño 32 o 64 bits.

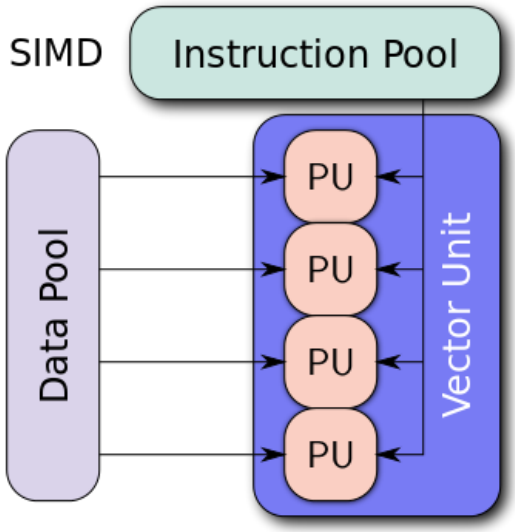


Figura 12 - Esquema de funcionamiento de las instrucciones Single Instruction Multiple Data. Fuente: https://en.wikipedia.org/wiki/SIMD

La nueva implementación del cálculo de la distancia se puede ver en el algoritmo 2 escrito en C++. Se ha eliminado el bucle, y se han aplicado directamente las operaciones *population count* del mayor tamaño disponible, que es de 64 bits. La operación XOR no se ha modificado, debido a que las operaciones booleanas son triviales para cualquier procesador.

```
unsigned long long *pa = (unsigned long long*) a.d,
                   *pb = (unsigned long long*) b.d;
int res = _mm_popcnt_u64(*(pa++)^*(pb++));
res += _mm_popcnt_u64(*(pa++)^*(pb++));
res += _mm_popcnt_u64(*(pa++)^*(pb++));
res += _mm_popcnt_u64(*(pa++)^*(pb++));
return res;
```

Algoritmo 2 - Distancia entre dos descriptores a y b, por medio de instrucciones population count de 64 bits

4.6 CARGA DEL ÁRBOL DE VOCABULARIO DESDE DISCO

El *vocabulary tree* de ORB-SLAM2 está pre-calculado, y se carga desde disco al comienzo de una ejecución. En la implementación recursiva original, el tiempo de carga del fichero y de construcción del árbol en memoria es significativo, generando un retraso de varios segundos desde el arranque del programa hasta que éste es capaz de empezar a procesar imágenes.

Con el cambio de la implementación al árbol implícito, la operación de lectura del árbol a partir de un fichero de texto también se ha cambiado, resultando más eficiente. También se ha añadido una opción para leer y escribir el árbol en formato binario, para agilizar más la operación. Este formato más eficiente en memoria y tiempo, no es compatible entre distintas máquinas. Por ello, se ha modificado ORB-SLAM2 a la hora de construir la librería, para que sea capaz de generar este fichero en formato binario a partir del fichero de texto.

Por si algún usuario quisiera hacer uso de un vocabulario propio ya escrito, se ha generado una función que permite la lectura del formato antiguo. No obstante, se pretende que todas las lecturas se realicen con el nuevo formato.

Aprovechando la nueva implementación del árbol implícito, se puede leer y escribir en bloque, debido a su adquirida continuidad en memoria. Como la consistencia del árbol se debe a su propia distribución en el vector, no es necesario almacenar datos secundarios por cada nodo en el fichero generado. Juntando todos estos factores, aparece un potencial de mejora en tiempo para la lectura desde disco.

4.7 RESULTADOS

Como batería de test para el recorrido del árbol, se han extraído varios millones de descriptores de varias secuencias de KITTI. Posteriormente se ha comprobado el funcionamiento del *vocabulary tree* implícito, realizando la comparación de resultados con la implementación previa.

Los resultados en tiempo se pueden observar en la tabla 4. Se ha observado una mejoría en tiempo en el recorrido del árbol, simplemente cambiando el tipo de implementación. Este proceso, con la nueva implementación, toma la mitad del tiempo necesario para el *vocabulary tree* como árbol recursivo. Toda esta mejoría se obtiene en gran medida gracias uso de la caché del procesador, la cual no era aprovechada por la implementación recursiva. Con el uso de las instrucciones *population count* se ha mejorado aún más el recorrido del árbol. Estas operaciones mejoran en un 1,5 el tiempo de recorrido respecto a la implementación implícita con el cálculo original de la distancia, y entre un 3 y 3,5 respecto a la implementación original.

Tiempo en µs	**Media**		**Desv. Típ.**		**Mediana**	
Número de palabras	***1***	***1.000***	***1***	***1.000***	***1***	***1.000***
Árbol Recursivo	4,950	3.697,0	0,768	51,33	5	3.697
Árbol Implícito	2,208	1.833,1	0,429	42,02	2	1.823
Árbol Implícito con *population count*	1,500	1.028,6	0,625	14,50	1	1.025

Tabla 4 - Tiempo de recorrido del árbol con 1 y 1.000 palabras, para la implementación del árbol recursiva e implícita

La memoria global del sistema también se ha mejorado, como se puede observar en la figura 13 y en la tabla 5. El *vocabulary tree*, ocupa ahora 33,9 MiB en memoria, frente a los ~406,0 MiB que ocupaba anteriormente en el pico máximo de memoria. El cambio en la implementación de los descriptores ORB ha mejorado considerablemente la memoria global utilizada. Con la implementación de OpenCV se almacenaban datos redundantes, como el tamaño del dato. Cada descriptor ocupa ahora el espacio mínimo posible, de 256 bits. La eliminación de variables adicionales del *vocabulary tree*, para mantener consistencia, como el vector de punteros a los nodos hijos, también ha contribuido a la mejora.

	keyframes	*mappoints*	DBoW2	*Optimizador & Bundle Adjustment*	*Total*
Original	779,9 MiB	389,9 MiB	407,4 MiB	599,2 MiB	2,4 GiB
***Vocabulary tree* implícito**	858,9 MiB	265,2 MiB	33,9 MiB	610,0 MiB	2,0 GiB
Porcentaje mejora	-9,19%	47,02%	1.101,76%	-1,77%	20%

Tabla 5 - Memoria dinámica máxima, utilizada por el sistema original y el sistema con la implementación implícita del árbol del vocabulario, para la secuencia 00 de KITTI

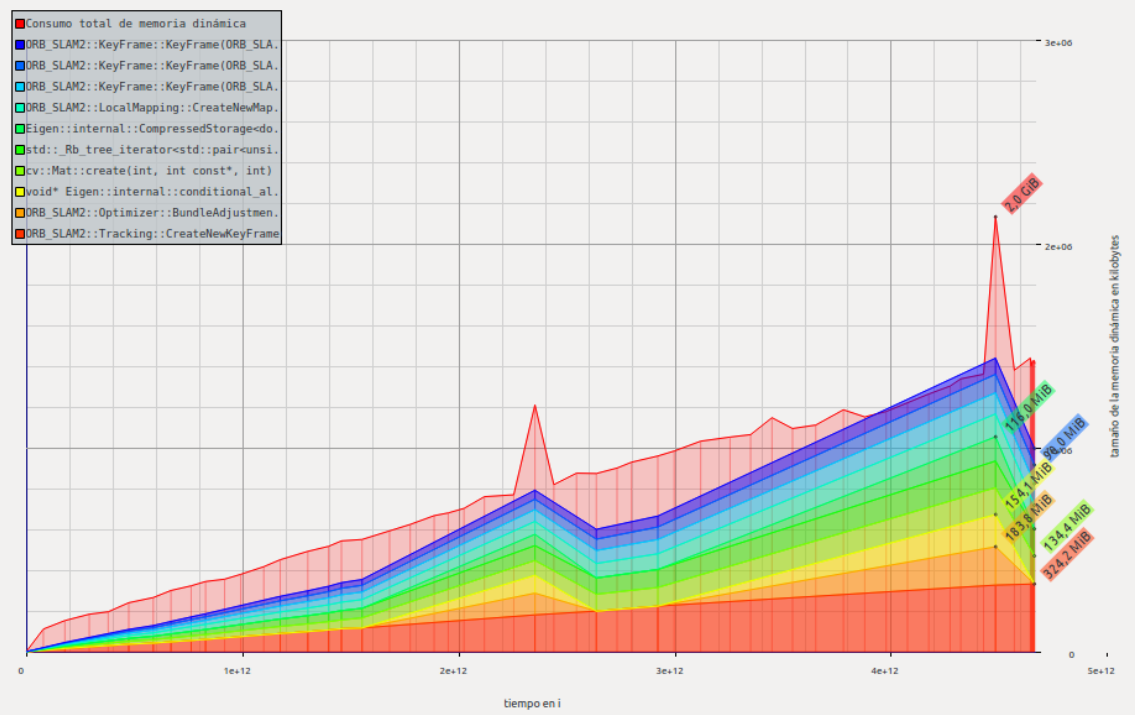


Figura 13 - Diagrama del uso del heap de ORB-SLAM2, tras las optimizaciones aplicadas en el vocabulary tree, con la secuencia KITTI 00. En el pico de 2 GiB, la memoria está ocupada por: keyframes (858,9 MiB), mappoints (265,2 MiB), DBoW2 (33,9 MiB), optimizador y bundle adjustment (610,0 MiB)

Para realizar las pruebas de lectura desde disco se ha utilizado el *vocabulary tree* por defecto de ORB-SLAM2. En la tabla 6 se observa que el tiempo de lectura ha mejorado significativamente. El nuevo formato de texto permite realizar una lectura prácticamente 2,5 veces más rápida. Pero la mayor mejoría se observa en la lectura desde el formato binario, que se realiza cerca de 360 veces más rapido. Esta mejoría se debe a que, para la lectura en binario, solo hace falta una llamada *read* del sistema, que carga todos los datos en bloque, mientras que el formato de texto requiere una lectura por cada dato, así como conversiones de texto a formato máquina.

El tamaño de los ficheros utilizados también ha variado. Al no usar datos adicionales para asegurar la consistencia, el nuevo formato de texto del *vocabulary tree* ocupa 133,0 MiB, frente a los 138,77 MiB del formato original. En binario, al almacenar los datos puros, se obtiene un tamaño de 43,44 MiB. El tamaño en formato binario depende de la arquitectura. Esta se ha obtenido de Ubuntu 16 de 64 bits.

Tiempo en s	**Tiempo**
Original	6,839
Árbol Implícito (texto)	2,657
Árbol Implícito (binario)	0,019

Tabla 6 - Tiempo de carga del vocabulary tree desde disco

5 ALMACENAMIENTO Y CARGA DE MAPAS EN DISCO DURO

5.1 INTRODUCCIÓN

Hasta el momento, el sistema solo podía utilizar el mapa que se había ido generado en esa misma ejecución. Se ha incrementado la funcionalidad de ORB-SLAM2, realizando una serialización del mapa por medio de la librería Boost. Dicha serialización permitirá guardar los mapas para posteriores ejecuciones, o incluso exportarlos entre máquinas. Este capítulo discute acerca del contenido del mapa, en qué consiste la serialización, argumenta el uso de la librería mencionada, y muestra los resultados obtenidos.

5.2 CONTENIDO DE UN MAPA Y SERIALIZACIÓN

En conclusión, el contenido del mapa consiste en los *keyframes* y *mappoints*. Almacenar estos dos componentes y su posterior carga, permitirá continuar explorando el mapa generado.

Una vez definido el contenido, el problema radica ahora en realizar una correcta serialización. La serialización es un problema de computación que consiste en codificar un objeto y enviarlo por medio de una conexión, en este caso, la salida o entrada de un fichero de disco. El objetivo de la serialización es que sea comprensible por un agente externo a la ejecución (otra ejecución del programa, otra máquina o incluso un humano). Para tipos de datos básicos, esta codificación consiste en saber interpretar el contenido de dicho dato. Pero en clases más complejas como es el caso del mapa, la serialización se vuelve un problema más complicado.

Para abordar la serialización, se debe definir un formato estándar de *keyframes* y *mappoints*. El formato, deberá resolver una serie de problemas, para su correcto funcionamiento:

- Límite de las estructuras iterativas: La información contenida en vectores, set y otros tipos de estructuras de datos iterables, deben indicar su tamaño, o marcar un límite. En una lectura de los datos, el sistema debe ser capaz de identificar cuando ha finalizado dicha estructura.
- Traducción puntero-dato: Los punteros contienen direcciones de memoria, que carecen de sentido entre ejecuciones. Para ello, suele utilizarse una estructura llamada *map*, que permite acceder rápidamente a la información identificador–dirección. Puede conocerse rápidamente el objeto a almacenar

en escritura, buscando el identificador asociado al puntero, o viceversa en lectura.

- Seguimiento de punteros: Las direcciones de memoria deben cargarse una sola vez. Si varios punteros distintos tienen la referencia al mismo dato, se debe reconocer que solo es un dato. Si no, se interpretarían como datos distintos, pero con el mismo contenido. Para ello, se debe realizar un seguimiento sobre los datos que han sido cargados, y sus direcciones de memoria.

5.3 FORMATO Y LIBRERÍA BOOST

Existen numerosas librerías para el formato de escritura en un fichero. Algunas permiten la escritura en un formato de etiquetas, como XML o YAML. Sin embargo, las librerías basadas en etiquetas deben analizar el fichero leído o generado, para comprobar la consistencia y realizar un mapeado de etiqueta-objeto. Para el mapa de ORB-SLAM, cuyo tamaño puede alcanzar del orden de 10^3 *keyframes*, y 10^5 *mappoints*, el formato de etiquetas no representa una buena alternativa.

Los formatos optados, para minimizar el tamaño del fichero resultante en disco, son binario, o texto plano. El formato binario, sirve para almacenar un mapa, cuyo uso futuro se realice en la misma máquina. Por otro lado, el formato en texto plano permite la exportación a otras arquitecturas, sacrificando tiempo de almacenamiento del mapa y espacio en disco.

Para la elección de la librería más apropiada, buscando una opción más profesional, y extendida, se ha optado para C++ Boost [9] o Cereal [10]. Ambas librerías son muy similares en uso y comportamiento. No obstante, Boost, que en realidad es un marco de trabajo, se encuentra más extendida por la comunidad y realiza un mantenimiento a largo plazo.

En síntesis, se ha realizado la serialización por medio del marco de trabajo Boost. Para la implementación de la serialización con este marco de trabajo, se define en cada clase a serializar un método específico. Se ha de indicar en este método el comportamiento de la clase al serializar. Por ello se han modificado las clases *keyframe*, *mappoint* y *map*. La última clase corresponde al mapa completo de ORB-SLAM2. En cada método se indica qué miembros de la clase se han de serializar. Se ha tratado de disminuir el tamaño del fichero generado, omitiendo todos los miembros redundantes, y almacenando solamente partes relevantes de estructuras de datos parcialmente vacías.

La comodidad de Boost radica en la simplicidad con la que se genera el método para serializar la clase. Con un simple operador, se pueden incluir cualquier tipo de estructura. Los punteros pueden serializarse del mismo modo, debido al seguimiento y exportación que realiza Boost. Se puede obviar de este modo cuestiones de consistencia. Con la función definida, la escritura en formato binario, texto o incluso XML puede realizarse cambiando simplemente el tipo de salida o entrada del fichero. Una vez

finalizada la serialización, la reconstrucción del sistema se realiza desde funciones definidas para guardar o cargar el mapa.

La versión recomendada para el uso de Boost es, como mínimo, la 1.58.0. Interesa prestar atención sobre próximas versiones, ya que Boost tiene en proyecto generar ficheros en formato binario exportables. Si tiene éxito, no serán necesarios los ficheros de texto, y se aumentará la eficiencia en la exportación.

5.4 RESULTADOS

Para el test de correcto funcionamiento, se han utilizado secuencias de KITTI. Este test ha consistido en realizar la secuencia por secciones, guardando al final de cada sección el mapa obtenido, y cargándolo al empezar la siguiente. Cada sección contiene 200 imágenes obtenidas desde la cámara. En la figura 14 se puede observar como el recorrido en una sola ejecución, y en varias son idénticos.

También se han utilizado las secuencias de EUROC, obtenidas por el vuelo de un dron en distintas habitaciones, para probar el correcto funcionamiento. Varias de estas secuencias se dan en la misma habitación, que es ideal para guardar el mapa al final de una, y usar el mapa generado para relocalizarse en otra, generando así un mapa con mayor densidad y precisión.

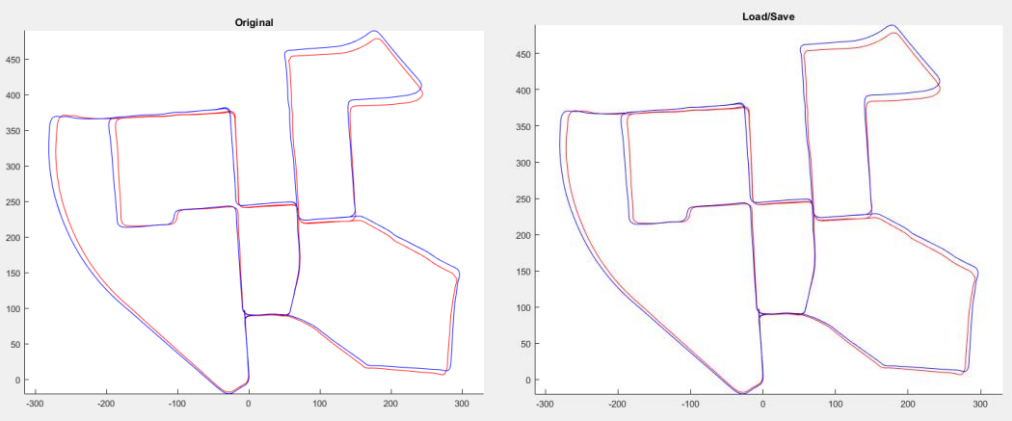


Figura 14 - Izquierda: En azul el recorrido obtenido por ORB-SLAM2 en una sola iteración, y en rojo el ground truth. Derecha: en azul el recorrido calculado por ORB-SLAM2, realizando guardado y finalización cada 200 frames, y posteriormente cargándolo y relocalizando, y en rojo su ground-truth. La secuencia utilizada es la 00 de KITTI

En la tabla 7, se puede observar como el formato binario es mucho más eficiente que el formato de texto. Alrededor de una décima parte respecto a tiempo, y en poco menos de la mitad de espacio. Pero hay que recordar que el formato binario en Boost aún no es exportable entre máquinas.

Formato	Salvar	Cargar	Tamaño fichero generado
Binario	2,584 s	2,638 s	410,064 MiB
Texto	27,521 s	20,109 s	1063,128 MiB

Tabla 7 - Tamaño, tiempo de guardar y cargar el mapa generado en la secuencia completa 00 de KITTI (1400 keyframes generados)

El cambio en el uso de memoria que utiliza la serialización por Boost respecto a una ejecución general, se ha medido con dos ejecuciones. La primera consiste en realizar un guardado completo del mapa, y la segunda un cargado del mismo, y aplicándose sobre la misma secuencia. Según la figura 15, y la tabla 8, las estructuras auxiliares que utiliza Boost para la serialización realizan un pequeño gasto adicional en memoria al guardar. Si se observa la figura 16, y la tabla 8 de nuevo, se ha adquirido mejoría a la hora de aplicar la misma secuencia, cargando mapa anteriormente generado. La carga del mapa ha conseguido realizar una desfragmentación de memoria. En la carga, todos los datos se generan en el mismo bloque. Por consiguiente, se aprovechan todos los espacios liberados, que no podían reutilizarse en la ejecución de guardado.

	keyframes	*mappoints*	DBoW2	*Optimizador & Bundle Adjustment*	*Total*
***Vocabulary tree* implícito**	858,9 MiB	265,2 MiB	33,9 MiB	610,0 MiB	2,0 GiB
Guardado del mapa con Boost	935,1 MiB	277,3 MiB	33,9 MiB	554,9 MiB	2,0 GiB
Porcentaje mejora	-8,14%	-4,36%	0,00%	7,98%	0,00%
Carga del mapa con Boost	749,2 MiB	68,5 MiB	33,9 MiB	0 MiB	1,2 GiB
Porcentaje mejora	14,64%	287,15%	0,00%	-	66,66%

Tabla 8 - Memoria dinámica utilizada como máximo por el sistema, con la implementación del árbol del vocabulario implícito y al guardar el mapa completo y al volver a cargarlo al comienzo de otra ejecución, para la secuencia 00 de KITTI

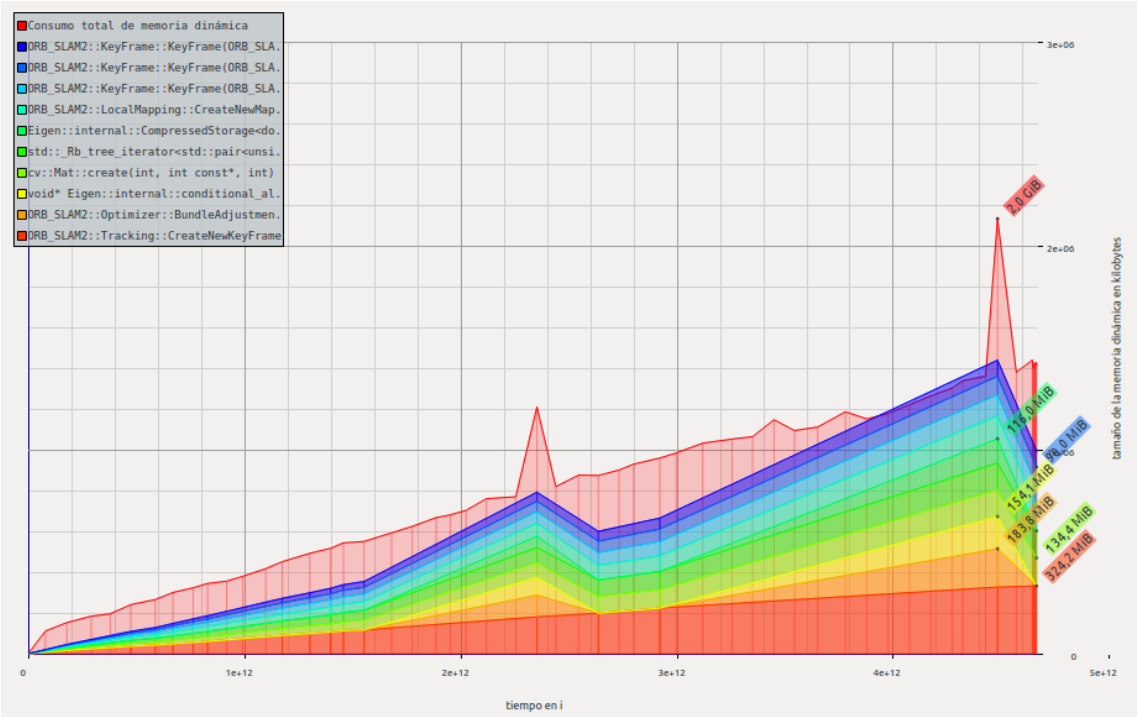


Figura 15 - Diagrama del uso del heap de ORB-SLAM2 con la secuencia KITTI 00, y realizando un guardado final del mapa. En el final de la ejecución, la memoria está ocupada por: keyframes (935,1 MiB), mappoints (277,3 MiB), DBoW2 (33,9 MiB), Boost (17,5 MiB).

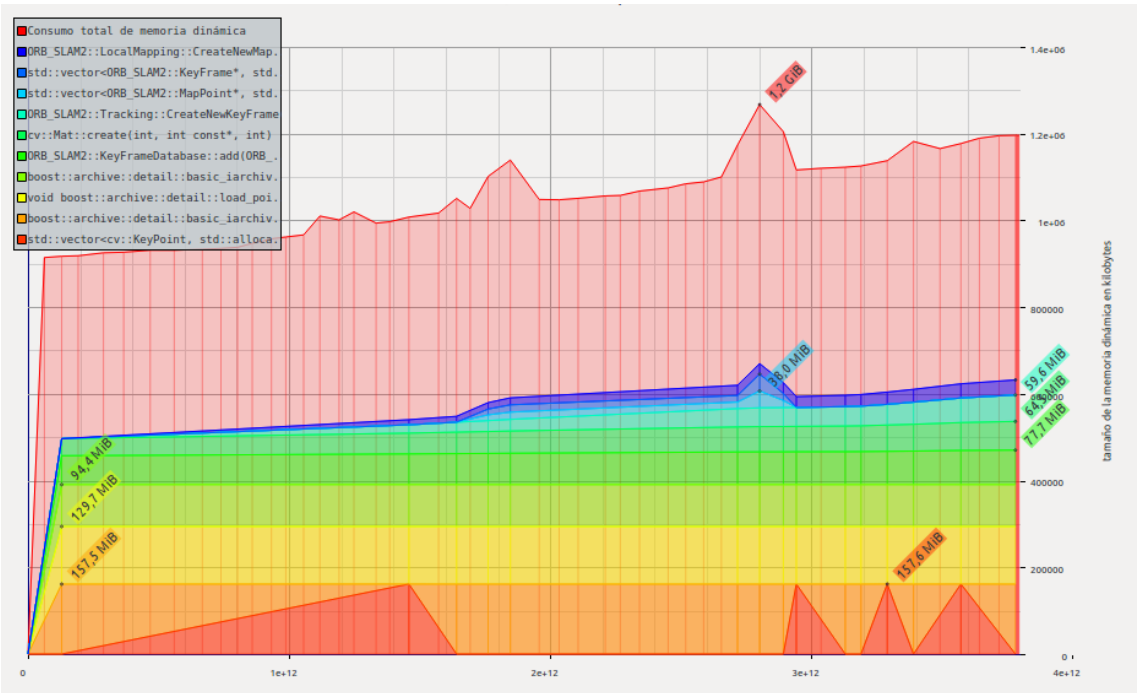


Figura 16 - Diagrama del uso del heap de ORB-SLAM2 con la secuencia KITTI 00, realizando un cargado completo del mapa, ya generado. Inmediatamente tras cargar el mapa, la memoria está ocupada por: keyframes (749,2 MiB), mappoints (68,5 MiB), DBoW2 (33,9 MiB).

6 CONCLUSIONES DEL PROYECTO Y POSIBLES PROYECTOS FUTUROS

6.1 ESTUDIO DE MEMORIA Y OPTIMIZACIONES SOBRE LA MISMA

Las optimizaciones aplicadas en ORB-SLAM2 tras el análisis del uso de memoria, han resultado poco útiles en la optimizar la ocupación de la misma. Observando los resultados, la ocupación de memoria es idéntica con las optimizaciones aplicadas. El motivo radica en la fragmentación de la memoria. Los punteros inteligentes han sido útiles para la liberar la memoria. No obstante, las arquitecturas actuales no consiguen reutilizar este espacio liberado, consumiendo más de la memoria disponible. Los *mappoints* son los componentes que más adolecen de este problema por el gran número de generaciones y borrados de los mismos. Esta conclusión se ha confirmado con los resultados obtenidos en el apartado 6.3, donde se ha conseguido desfragmentar la memoria. El trabajo aquí desarrollado, ha servido para encarrilar el futuro de las optimizaciones en memoria de ORB-SLAM2. La generación de un recolector de basura, que permita la reutilización de la memoria liberada, es ahora el objetivo principal de las optimizaciones, fundamentalmente para los *mappoints*. Realizar la declaración de los datos sobre estructuras dinámicas controladas, puede permitir la reutilización de la misma.

Se ha observado, que el uso de punteros inteligentes puede mejorar ligeramente el tiempo de ejecución. No obstante, este cambio es ínfimo. Para los punteros inteligentes con *keyframes* la mejoría es de alrededor del 2%. Esta mejoría es omisible, y más teniendo en cuenta el incremento en memoria que realizan los punteros inteligentes.

6.2 OPTIMIZACIÓN DEL ÁRBOL DEL VOCABULARIO

La nueva implementación ha resultado más eficiente. Se ha generado una notable mejoría en la memoria ocupada por todo el sistema. Ésta se debe al cambio de implementación en los descriptores ORB fundamentalmente. Ahora ocupan el espacio de 256 bits (el mínimo para descriptores ORB), frente a los 384 que ocupaban como mínimo con la implementación de matrices de OpenCV. Se ha deducido que los descriptores representan un alto porcentaje de ocupación en memoria. Disminuir el número de descriptores, o utilizar descriptores de menor tamaño, representan una alternativa adicional para la optimización de memoria, incluso en *keyframes* y *mappoints*. Además, con la nueva implementación se consigue definir una continuidad en memoria explícita. Esta continuidad permite optimizar aún más la memoria, omitiendo posibles fragmentaciones. Esta continuidad es la que ha permitido mejorar el tiempo de recorrido del árbol, al hacer uso de la caché del procesador. Se omiten cargas desde memoria al

procesador, y por ende, tiempos de espera derivados de ellas. El uso de las instrucciones *population count* para el cálculo de la distancia entre dos descriptores también contribuye a la mejoría en tiempo del recorrido. La optimización sobre esta operación individual afecta notablemente a todo el *vocabulary tree*, debido a su alta frecuencia de uso.

Se han intentado aplicar las instrucciones SIMD en el cálculo de distancias de dos descriptores, debido al tamaño de este. Pese a que se ha demostrado que no son útiles, no se descarta su uso en otras secciones del código, en las que se aplique la misma operación sobre varios datos del mismo tipo repetidamente.

El tiempo dedicado para la carga del *vocabulary tree* desde fichero en formato binario es ahora imperceptible por un humano. Se ha conseguido eliminar el tiempo de espera de inicialización, molesto para un usuario de la librería. Como se permite realizar una traducción desde el viejo formato al nuevo, o desde texto a binario, el vocabulario sigue siendo exportable, y se permite utilizar vocabularios propios que ya estuvieran usando algunos usuarios de ORB-SLAM2.

En conclusión, se ha optimizado el *vocabulary tree*. Se ha conseguido reducir a la mitad el tiempo de recorrido del árbol, además de la memoria global que utiliza ORB-SLAM2. La carga del vocabulario es ahora mucho más cómoda, y es posible inicializar el sistema mucho antes. Con todos estos resultados se espera que otras máquinas sean capaces de utilizar el sistema.

6.3 ALMACENAMIENTO Y CARGA DE LOS MAPAS EN DISCO

El sistema es ahora capaz de almacenar un mapa en un momento determinado de la ejecución, para el posterior uso en el comienzo de otra. Se abre ahora un abanico de posibilidades, en el que se permite utilizar los mapas entre distintas máquinas, o ejecuciones. La construcción de mapas combinados entre dos robots simultáneos, es posible gracias el desarrollo aquí empleado.

Con la funcionalidad de gestionar mapas desde disco, pueden investigarse ahora medios para realizar unión entre sub-mapas. Este campo sería útil para generar grandes mapas, y que los sistemas que utilizaran la librería, solo tuvieran que realizar un seguimiento de trayectoria. Esta investigación puede ser útil para el problema que tiene actualmente con la relocalización del mapa. Si no se consigue relocalizar en el mapa cargado, puede generarse un mapa auxiliar, que se una al mapa general en el momento que coincidan en alguna zona.

En los resultados obtenidos, se puede observar que la memoria consigue desfragmentarse al guardar y cargar de nuevo un mapa. Actualmente, este medio es inviable en tiempo de ejecución para realizar desfragmentación. Para cargar el mapa, el sistema se debería bloquear durante cierto tiempo, pudiendo provocar pérdida de la posición. Pero guardados y cargas parciales de zonas del mapa no relevantes en determinado momento, puede ser una solución futura para reducir la fragmentación.

7 ESTRUCTURA TEMPORAL DEL PROYECTO

Se muestra a continuación un cronograma, con la distribución temporal aplicada a cada parte del trabajo. Se incluye también el tiempo total dedicado a cada parte.

	Sep 17	Oct 17	Nov 17	Dic 17	Ene 18	Feb 18	Mar 18	Abr 18	May 18	Jun 18
Estudio inicial del sistema (35 horas)										
Análisis de memoria (45 horas)										
Optimizaciones de memoria (50 horas)										
Optimización del árbol del vocabulario (110 horas)										
Almacenamiento de los mapas en disco (60 horas)										

Total: 300 horas

8 BIBLIOGRAFÍA

[1] Raúl Mur-Artal, and Juan D. Tardós. *ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras*. in *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255-1262, Oct. 2017. doi: 10.1109/TRO.2017.2705103

[2] Jorge Martínez Romanos. *Localización del usuario en aplicaciones de realidad virtual mediante ORB-SLAM2*. Trabajo de Fin de Grado, EINA, Junio 2018

[3] Edward Rosten, Reid Porter, and Tom Drummond, "Faster and better: a machine learning approach to corner detection" in *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2010, vol 32, pp. 105-119.

[4] Ethan Rublee, Vincent Rabaud, Kurt Konolige and Gary Bradski. *ORB: an efficient alternative to SIFT or SURF*. *In* Proc. *2011 IEEE International Conference on Computer Vision*, Barcelona 2011

[5] Valgrind, http://valgrind.org/, accedido en 28-Agosto-2018

[6] D. Galvez-López and J. D. Tardos, "Bags of Binary Words for Fast Place Recognition in Image Sequences," in *IEEE Transactions on Robotics*, vol. 28, no. 5, pp. 1188-1197, Oct. 2012. doi: 10.1109/TRO.2012.2197158

[7] David Nistér and Henrik Stewénius. *Scalable Recognition with a Vocabulary Tree*, in *Proc. IEEE Conf. Comput. Vision Pattern Recog.*, New York, NY, USA, Jun. 2006, vol. 2, pp. 2161–2168.

[8] Wojciech Muła, *SSSE3: fast popcount*, http://0x80.pl/articles/sse-popcount.html, accedido en 28-Agosto-2018

[9] Boost C++ Libraries, https://www.boost.org/, accedido en 28-Agosto-2018

[10] Cereal - A C++11 library for serialization, https://uscilab.github.io/cereal/, accedido en 28-Agosto-2018