Programación en C y evaluación de un codificador aritmético

Edgard D. Felipe

Resumen– Actualmente, las tecnologías de compresión de la información tienen un impacto potencial de rendimiento en el ámbito de las comunicaciones punto a punto y el almacenamiento de datos, y la carga computacional que esto implica está conllevando a necesitar sistemas con prestaciones cada vez más favorables con el paso del tiempo. Hay diversas soluciones implementadas en la codificación por entropía de la información, una de ellas es la codificación aritmética. El Group on Interactive Coding of Images posee una implementación de un codificador aritmético binario adaptativo basado en contexto con palabras de longitud fija, dicho sistema tiene un rendimiento considerablemente alto respecto a otras soluciones. Este trabajo consiste en conseguir una versión más especializada y de mejores prestaciones del codificador anterior mencionado y analizar el rendimiento de ambas versiones en un tipo de procesador determinado.

Palabras clave– compresión de datos, codificación aritmética, codificación por entropía, ingeniería de rendimiento

Abstract– Nowadays, information compression technologies have a potential performance impact in end-to-end communications and data storage, such computational load that this implies is leading to the need for systems with increasingly favourable performance with the pass of time. There are several implemented solutions for entropy coding, such as arithmetic coding. The Group on Interactive Coding of Images has an implementation of a context-adaptive binary arithmetic coder with fixed-length codewords, and this new version of the system achieves a significantly high performance compared to other solutions. This work attempts to achieve a more specialized and better-performing version of the mentioned encoder and to benchmark the performance of both versions on a specific type of processor.

Keywords– data compression, arithmetic coding, entropy coding, performance engineering

1 Introducción - Contexto del trabajo

En el Departamento de Ingeniería de la Información y Comunicaciones de la Universidad Autònoma de Barcelona hay un grupo dedicado a estudiar, diseñar e implementar codificación de imágenes, técnicas de transmisión y estándards, el cual se llama Group on Interactive Coding of Images (GiCi). Este se mantiene constantemente impulsando el estado del arte de dichos ámbitos, publicando artículos y herramientas de software con regularidad. Actualmente, el GiCi tiene implementaciones de codificadores de entropía, uno de ellos es el codificador aritmético binario adaptativo basado en contexto con palabras de longitud fija[2], en el cual se centra este trabajo.

Este codificador está implementado en Java, donde se presenta un análisis de rendimiento, comparando dicho sistema respecto a otros codificadores, demostrando que este sistema consigue un ratio de compresión y eficiencia superiores. Java está demostrado como uno de los lenguajes existentes más eficientes que hay hoy en día, sin la necesidad de conocer la arquitectura del procesador específico para desarrollar. Dicho esto, es posible utilizar un lenguaje más especializado como C para conseguir más eficiencia en este sistema, tanto para un target en específico como en general, con el propósito de mejorar los tiempos de ejecución manteniendo el comportamiento original del codificador.

2 Objetivos

La meta principal de este trabajo es conseguir una versión en C de la implementación del codificador aritmético desa-

• E-mail de contacto: edgardaniel.felipe@e-campus.uab.cat
• Mención realizada: Tecnologías de la Información
• Trabajo tutorizado por: Joan Bartrina Rapesta (DEIC)
• Curs 2020/21

rrollado por el GiCi descrita en la siguiente referencia[2], la cual ha de mantenerse funcionalmente igual y reducir el tiempo de su ejecución.

Por tanto, realizar una transcripción del código de Java a una nueva versión de C para tener el nuevo codificador es uno de los objetivos de este proyecto, como también programar tests de software para validar que se mantiene funcionalmente igual y analizar su eficiencia, para poder compararla con la versión de Java y fundamentar los resultados favorables conseguidos, para lo que serán necesarios programas de benchmarking. Además, para este se deberá comprobar que el nuevo codificador consigue tener altas prestaciones en procesadores de bajos recursos y consumo energético, por tanto, se diseñará este sistema y se testeará el funcionamiento y rendimiento para un dispositivo Raspberry Pi 4.

Se ha propuesto como un objetivo adicional que consiste en optimizar la versión en C que comporten ganancias de rendimiento significativas, priorizando la integridad funcional del sistema y no alterarla a cambio de ganar eficiencia. Este trabajo extra implica realizar benchmarks y comparativas adicionales, además de fundamentar la implementación de dichas propuestas de optimización.

Finalmente, el cumplimiento de estos objetivos se han de demostrar, principalmente mediante validaciones de los testeos de software y la demostración de speedups favorables en los factores de rendimiento relevantes respecto a la versión original.

En la Tabla 1, se muestran los diferentes objetivos principales y su realización dentro de este trabajo, los cuales se describen en esta sección en el mismo orden.

ID	Descripción del Objetivo	Estado
O1	Transcripción en C del codificador aritmético	Completado
O2	Diseño y validación de tests de software	Completado
O3	Programación y evaluación de benchmarks	Completado
O3	Conseguir altas prestaciones en Raspberry Pi 4	Completado
O4	(Extra) Desarrollo de optimizaciones	Completado

TABLA 1: CUMPLIMIENTO DE LOS OBJETIVOS

3 ESTADO DEL ARTE

Actualmente, existen diversos codificadores de entropía, en esta sección, se describen implementaciones en lenguaje C de dichos métodos de codificación.

Respecto a la codificación aritmética, en la siguiente referencia[6] hay una implementación hecha en lenguaje Handle-C de esta técnica de compresión, donde solo se ha diseñado un descodificador aritmético clásico, en el cual se consigue un throughput de 3.75 MB/s y 5.375 MB/s usando dos métodos de síntesis del código en un hardware concreto.

En [7] se describe y analiza una implementación en C de una propuesta del algoritmo de compresión LZ77 para un procesador en concreto, el cual concluye con mejoras de velocidad y ratio de compresión respecto al diseño original y LZW. El profiling del rendimiento se realizó con un solid state driver (SSD) usando el compilador de C de Xtensa, el trabajo consigue una velocidad promedio de ciclos de lectura/escritura es de 22.5 MB/s en ejecuciones multi-core usando operaciones de múltiple instrucción y múltiples datos (MIMD) y volúmenes de datos de entre 4 Kb y 32 Kb.

4 METODOLOGÍAS

Desarrollar y testear parcialmente el codificador en C por medio de un software prototyping incremental ha sido viable como solución, esta técnica consiste en programar diferentes secciones independientes por separado y validarlas. De esta manera, se consigue tener partes evolucionando sin comprometer otras porciones del sistema, hasta tener el sistema completamente validado.

Los tests de software han de ser de tipo unitario, de regresión y de integración, debido a que el sistema completo, como se describirá en la sección de desarrollo, está compuesto por múltiples dependencias anidadas entre funciones y componentes, y porque hay secciones completamente independientes que también han sido testeadas. Dichos tests cubren una cantidad extensa de escenarios de uso posibles en las funciones del sistema, mediante ejecuciones que migran de modo de funcionamiento, comparaciones de ficheros de salida y a nivel de bits de los tipos de datos en ambas versiones.

Toda la información relacionada a los benchmarkings se han extraido usando la herramienta linux-perf y sus módulos stat, record y report, que han servido parar capturar código máquina y contadores de eventos a bajo nivel tanto de CPU como en memoria y entrada/salida.

5 DESARROLLO

Para comprobar que el nuevo codificador funcione de manera equivalente al original, se ha desarrollado un programa de testeo por cada componente para ambos lenguajes del codificador, donde se verifique, para cada función, que se consiguen los mismos resultados y comportamientos, que incluyen tipos de acceso y manejo de memoria, cambios en direcciones y páginas de memoria o el contenido de ficheros de salida. Para realizar esto último, ha sido necesario escoger qué funciones de cada componente pueden comprometer al sistema por ser críticos, donde no haya trivialidad al hacer la transcripción.

El procesador target de este trabajo es el del dispositivo Raspberry Pi 4 modelo B de 8GB, el cual se llama ARM Cortex-A72[5], este se cataloga en la familia A de los procesadores de ARM de aplicación específica, teniendo alta eficiencia de codificación, y por otro lado porque tiene, relativamente a la época, bajos recursos y consumo energético dentro de las arquitecturas RISC. Es necesario un sistema UNIX como Debian en este dispositivo para este trabajo.

Para realizar los benchmarks, se ha desarrollado un programa por componente en ambos lenguajes que realice una ejecución de una de sus funcionalidades, las cuales, por dependencias anidadas, pueden utilizar otras funcionalidades, como las de otros componentes. Todos los profilings se han

realizado sobre el dispositivo Raspberry Pi 4 modelo B de 8GB con un sistema operativo Debian 10 y usando la tarjeta SD del fabricante como disco duro, utilizando el compilador de C gcc 8.2 con optimizaciones agresivas y la versión de Java 11 desactivando la compilación just-in-time y utilizando la compilación ahead-of-time.

A continuación, se describen los por tres componentes principales por los que se compone este codificador, cuyo comportamiento principal se representa en la siguiente figura.

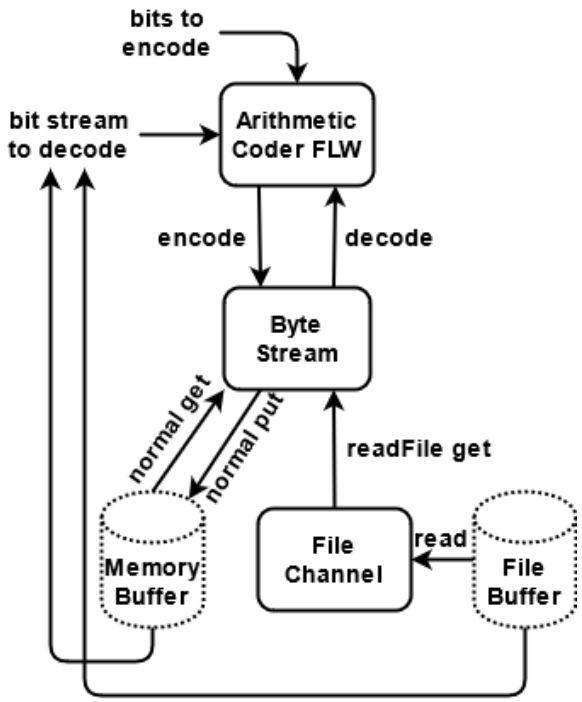

Fig. 1: Interacciones principales entre componentes del sistema

5.1. ArithmeticCoderFLW

Este componente realiza todos los procesos que corresponden a la algorítmica principal del codificador aritmético original, además de disponer de la información del estado actual de la codificación o descodificación. Este componente es capaz de codificar y descodificar migrando de modo de funcionamiento, para lo que necesita asociarse con una instancia del componente ByteStream, siendo la fuente de los datos para escribir o leer en caso de codificar o descodificar, respectivamente.

5.2. ByteStream

Esta parte del sistema se encarga de controlar el procesamiento streaming del volumen o buffer de bytes que utiliza ArithmeticCoderwFLW como datos de entrada y/o salida, para ello tendrá que inicializarse en modo normal cuando se quiera codificar y/o descodificar datos que residirán en memoria, mientras que cuando solo se necesite descodificar los datos de un fichero, la inicialización será en modo readFile, donde se le asociará una instancia de FileChannel, la cual manejará las operaciones a bajo nivel con los ficheros. Este componente, como el anterior, puede migrar entre ambos modos.

Además, este controla el buffer de datos, limitando de forma lógica la cantidad de bytes accesibles, definiendo y redefiniendo dinámicamente los límites del espacio de memoria o del fichero accesibles para ArithmeticCoderFLW.

5.3. FileChannel

En la versión original del codificador, este componente corresponde a la clase java.nio.FileChannel de Java, la cual, por su documentación[8], funciona operando sobre el equivalente de un File Stream de C como una solución para entrada-salida no bloqueante.

Por lo mencionado en las subsecciones anteriores, desarrollar los módulos funcionales de esta clase en C ha requerido comprobar que estas, usadas por un ByteStream en modo readFile, se comporten de la misma manera que en su versión de Java. Esta implementación en C ha sisdo especializada para ganar rendimiento con este codificador, por lo tanto, no es estrictamente igual a java.nio.FileChannel en todos los escenarios.

Una instancia de este componente en la nueva versión solo puede inicializarse para leer o escribir, y la forma de operar sobre los ficheros está diseñada con memory-mapping y no con un File Stream, pues el mapeo de un fichero en memoria permite realizar accesos de forma más eficiente, lo cual se comprueba en la sección de resultados. La función transferFrom de esta clase no utiliza mapeo de memoria en C, pero ha sido reemplazada por la llamada al sistema sendfile propia de los sistemas UNIX, que es su equivalente más eficiente para transferir el contenido de un fichero a otro sin necesidad de utilizar userspace, al ser una operación por kernel.

6 RESULTADOS

Se ha conseguido desarrollar el codificador aritmético en C, manteniendo un uso de las funciones y componentes similar al que tiene la versión original. En este punto, ArithmeticCoderFLW y ByteStream han permanecido equivalentes respecto a su estructura, inicialización y funciones, mientras que el componente FileChannel ha sido diseñado de forma óptima para este codificador.

Todos los tests de software han sido validados de forma individual y conjunta, comprobando así que la implementación en C es funcionalmente igual a la de Java, sin alterar sus resultados en diversos escenarios de ejecución y ratio de de compresión original.

Implementar las optimizaciones ha implicado comprobar nuevamente que el codificador permanece funcionalmente íntegro, mediante los tests de software ya desarollados. A continuación, se describen estas propuestas de optimización y sus justificaciones de implementación.

6.1. Aumento de escalabilidad

Esta primera propuesta es un diseño óptimo de la expansión del buffer de memoria del ByteStream usando la función de C realloc, cosa que en la versión de Java también se puede conseguir, con la diferencia que las estructuras de datos del codificador original se verían alteradas al introducir entidades nuevas.

La versión de Java reserva una región de memoria nueva, copia el contenido de la anterior y la libera para realizar la expansión del buffer. Esta optimización consigue reducir la paginación, overheads por copias de memoria y por tanto también mantiene la localidad temporal de las cachés, a medida que el volumen de datos va escalando. A continuación, la Fig. 2 representa esta operación con ambos métodos, poniendo como ejemplo el caso del modo normal del ByteStream.

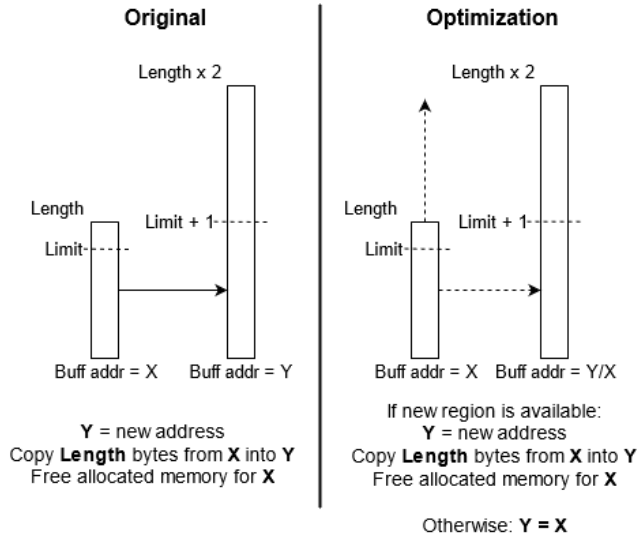

Fig. 2: Comparación de los métodos de expansión de la memoria

6.2. Acceso eficiente a ficheros

Al igual que el anterior punto, se trata de un diseño óptimo de esta implementación, el cual se centra en el componente FileChannel.

Consiste en dos aspectos, el primero es realizar las operaciones con ficheros lo más eficiente posible, de esta manera, mediante memory mapping, se consiguen realizar lecturas eficientes para el uso del ByteStream, evitando llamadas al sistema y tratando al buffer del fichero como cualquier otro buffer en memoria.

El segundo punto importante reside en el uso de la función read del FileChannel que utiliza ByteStream, el cual es leer un byte, usualmente consecutivo, a la vez. La clase java.nio.FileChannel está implementada con File Streams para operaciones buffered con volúmenes extensos según su especificación, por tanto, se ha diseñado una nueva función read1B que realiza el acceso directo a un byte que el ByteStream necesita, evitando utilizar la función de C memcpy, como se hace en la implementación genérica de esta función, read de FileChannel, en C.

6.3. Búsqueda rápida en ficheros

En este punto, se presenta una propuesta de optimización para la descodificación con ficheros cuya implementación en el sistema final ha sido rechazada de forma justificada.

Al leer un byte del ByteStream en modo readFile, se los offsets y longitud definidas por las regiones lógicas para buscar la posición del byte en el fichero dentro de un bucle. Este proceso sigue un patrón computacional de tipo scan, en el cual toda iteración del bucle depende de la finalización de toda la iteración anterior para ejecutarse, esto representa la limitación más crítica en este componente en el codificador, además que esta limitación es proporcional al número de regiones lógicas del fichero.

La optimización consiste en eliminar el patrón scan, reducir la carga de trabajo total y añadir paralelismo a nivel de iteración, utilizando solo el offset para la búsqueda. Esto último supone que todas las regiones constituyen conjuntamente el fichero completo, lo cual no es es estrictamente necesario en el codificador original.

Esto conlleva a reducir las opciones de uso del descodificador y ha sido considerado como no viable actualmente, ya que este trabajo prioriza mantener la integridad funcional del sistema por encima de comprometerla para ganar rendimiento, por tanto, esta propuesta está rechazada para implementar.

6.4. Lecturas con múltiples datos

La limitación por throughput en las obtenciones de bytes en el ByteStream puede ser mitigada utilizando operaciones de tipo single-instruction multiple-data (SIMD), de manera que, leyendo dos bytes a la vez en lugar de uno y almacenarlos en un atributo mostraría un speedup máximo teórico de 2x. Este grado de multiplicidad puede ser expandido teóricamente hasta la capacidad disponible en los registros del Raspberry.

Almacenar dos bytes leídos en un atributo, con el modelo actual del sistema, no es viable, debido a que los accesos a estos atributos, alojados en memoria dinámica, seguirán requiriendo ejecutar instrucciones de memoria, por tanto, el impacto sobre el codificador aritmético entero con esta optimización, se vería minimizado, concluyendo que para implementar esta optimización será necesario eliminar la representación de objetos con structs en el sistema completo.

Este último paso va en contra de las prioridades de este proyecto, pues de esta manera sería imposible mantener dos instancias de alguno de los componentes en un mismo programa sin realizar cambios adicionales, limitando el comportamiento y los escenarios de uso posibles del sistema original. Por estos motivos, esta propuesta, al igual que la propuesta anterior, ha sido rechazada para implementar hasta conseguir descartar los structs en el futuro.

6.5. Comparativas de rendimiento

Los benchmarks realizados utilizan un volumen de datos entre 1KB y 10MB. El diseño actual permite utilizar ficheros de más de 4 GB. Los componentes ArithmeticCoderFLW y ByteStream se limitan principalmente por dependencias de las instrucciones de memoria que corresponden a lecturas y escrituras de los atributos de los structs, saturando el único puerto de operaciones de memoria del core del Raspberry.

Por parte del ArithmeticCoderFLW, individualmente, la conversión de modelos en palabras y su proceso inverso han conseguido, respectivamente, realizar 119.47 y 149.23 M ops/s con speedups de 1.2x y 1.5x.

La versión final muestra una velocidad de codificación de 10.06 M encodes/s, con un speedup de 2.28x. Al descodificar con un ByteStream en modo normal, se consigue realizar 9.12 M decodes/s, mientras que en modo readFile, se consiguen 5.57 M decodes/s, consiguiendo speedups, respectivamente, de 2.08x y 25.32x. Si el buffer cabe en caché L1 solo favorece el rendimiento de las descodificaciones en un 1 % con un ByteStream en modo normal, y en un 11 % en modo readFile.

Por parte del ByteStream individualmente, se las inserciones en el buffer son 3.89x veces más rápidas y las obtenciones aumentan en un 5.31x en modo normal. En la Fig. 6, se muestra la comparación de velocidad de lecturas del buffer del ByteStream en modo readFile respecto a la versión de Java, en M gets/s, con la implementación inicial en C y el nuevo diseño utilizando la nueva función optimizada del FileChannel.

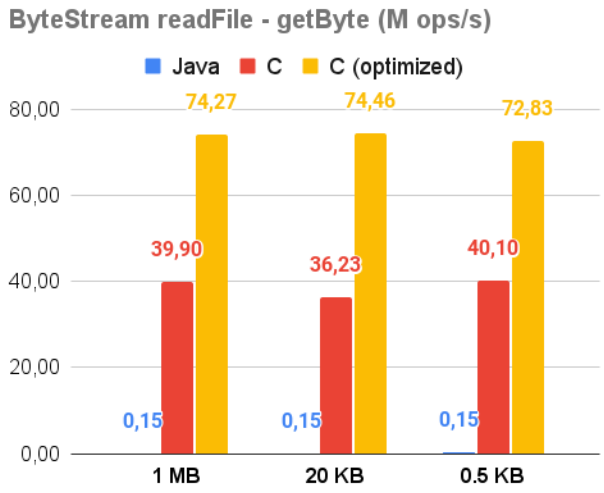

Fig. 3: Gráfica comparativa de la velocidad de lecturas por ByteStream en modo readFile

Las ejecuciones se realizaron definiendo una sola región lógica para acceder al fichero dentro del ByteStream, lo que elimina el camino crítico de dependencias mencionado en la subsección de búsqueda rápida en ficheros. Este escenario del ByteStream se limita por el ancho de banda proveniente del FileChannel, y se ve penalizado hasta en un 2 % cuando el buffer solo cabe en RAM. El speedup promedio conseguido, comparando la versión optimizada con la de Java, es de 500.74x.

Finalmente, en la Fig. 7 se comprueba que el ancho de banda conseguido por el nuevo FileChannel para el codificador es altamente más eficiente que el de Java, demostrando que se ha diseñado de forma óptima este componente para este codificador, como se describe en el punto 6.2, además de haber creado la función de optimización read1B.

El ancho de banda de la versión optimizada es 116.12x veces mayor que la versión de Java y el cuello de botella de esta función en la nueva versión es la capacidad del puerto de instrucciones de memoria del Raspberry Pi, y al tener un escenario donde el buffer del fichero cabe en la caché más cercana, la saturación del puerto de operaciones de memoria se satura en mayor medida.

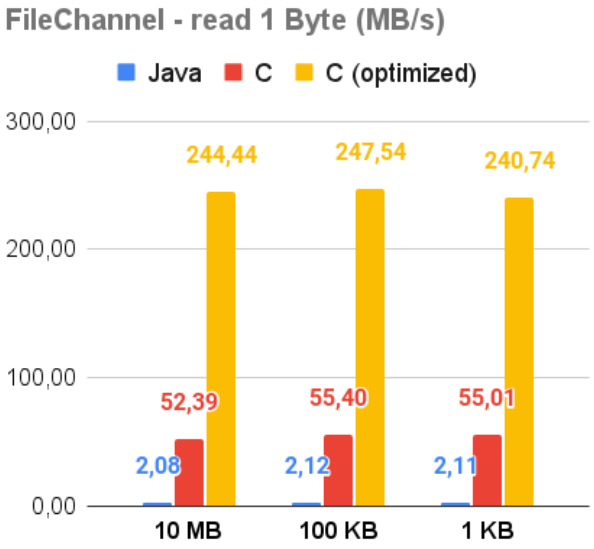

Fig. 4: Gráfica comparativa del rendimiento de las lecturas por FileChannel

7 Otras consideraciones

Debido a que un lenguaje como C utiliza una compilación ahead-of-time (AOT) y Java realiza una compilación just-in-time (JIT) además de una AOT, ha sido necesario que para todos los benchmarks se inhabilite el compilador JIT de Java, debido a que la naturaleza de los benchmarks planteados es ejecutar de forma repetida una operación que tiene un rendimiento constante. Por tanto, utilizar un compilador JIT, que recoge información de la ejecución en tiempo real para optimizar el próximo código máquina a ejecutar, elimina por completo la legitimidad de los análisis de rendimiento, invalidando el proceso de análisis. De esta manera, solo hará falta utilizar el compilador AOT de Java, comparándolo con otro de tipo AOT de C. Por este motivo, los resultados deben ser interpretadas de manera que como mínimo se consiguen los speedups que se presentan.

El nuevo codificador se limita enteramente por dependencias de datos, causadas por accesos a memoria por los atributos de los componentes, mientras que la versión original se limita por dependencias y ancho de banda en ambos modos del ByteStream. Dicha limitación puede ser mitigada descartando la opción de representar las clases de Java con structs en C, teniendo todos los atributos como variables globales, de esta manera, estos accesos ya no implicarán microoperaciones de memoria, aprovechando además la localidad espacial al residir en el stack. Esto último conlleva a que el compilador aplique register peeling, lo cual consiste en acceder al stack como con operaciones de memoria con la dirección base del stack pointer, debido a la falta de registros disponibles, para tener una cantidad de datos accesibles en registros temporalmente, pero con la diferencia que estos accesos serán, en su mayoría, operaciones en frío.

8 Conclusiones

Se ha conseguido una versión del codificador realizado por el GiCi más eficiente, de mejores prestaciones, capaz de poder trabajar con volúmenes de datos más extensos que la versión original consiguiendo reducir latencias y aumentar su ancho de banda considerablemente, además de haber validado que mantiene un funcionamiento íntegro. El grado de optimización alcanzado demuestra que el rendimiento del codificador en C son, en su totalidad, más favorables en comparación a la versión de Java.

Se ha comprobado una mejora absoluta de rendimiento en todos los aspectos del nuevo codificador, mostrando ser altamente eficiente en un procesador de bajos recursos, gracias a la implementación de las optimizaciones y el diseño específico de la nueva versión, debido al grado de especialización que un lenguaje como C permite alcanzar respecto a Java sobre ciertos escenarios y operaciones en específico.

Todas las opciones de optimización tienen la meta en común de ganar eficiencia, estas pueden requerir especializar el sistema para un número de escenarios más reducido, lo cual, en la mayoría de los casos, conlleva la responsabilidad de conocer las implicaciones que esto comporta, para así decidir o no implantarlas a los sistemas con los que trabajamos hoy en día. Este tipo de decisiones permitirán resultados igual de válidos y favorables como mejorar el rendimiento en dicho sistema sobre algún aspecto en concreto o mantener una backwards-compatibility conforme este va evolucionando. Este trabajo prioriza mantener la integridad funcional del sistema y no alterarla en cualquier proporción a cambio de ganar eficiencia.

Agradecimientos

A mis padres y los buenos amigos que encontré en el camino, incluyendo aquellos profesores que me ayudaron a comprender mejor el significado de la dedicación. También agradezco al gato de mi amigo Jhoan, por su impacto motivacional en nosotros como un memorable personaje de comparsa.

Referencias

[1] P. G. Howard and J. S. Vitter, “Arithmetic coding for data compression,” in Proceedings of the IEEE, vol. 82, no. 6, pp. 857-865, June 1994, doi: 10.1109/5.286189.

[2] F. Aulí-Llinàs, “Context-Adaptive Binary Arithmetic Coding With Fixed-Length Codewords,” in IEEE Transactions on Multimedia, vol. 17, no. 8, pp. 1385-1390, Aug. 2015, doi: 10.1109/TMM.2015.2444797.

[3] E. Felipe Polo, “Kappa-Bot/ACFLW-Java-to-C”, GitHub, 2021. [Online]. Available: https://github.com/Kappa-Bot/ACFLW-Java-to-C. [Accessed: 28-May-2021].

[4] J. Gosling, B. Joy and G. Steele, “The Java Language Specification,” Menlo Park, Calif.,, 1996.

[5] “Learn the architecture: AArch64 Instruction Set Architecture”, Developer.arm.com, 2021. [Online]. Available: https://developer.arm.com/documentation/102374/latest/. [Accessed: 22-Apr-2021].

[6] T. Zhu, J. Zhou and S. Liu, “Design and implementation of JPEG2000 arithmetic decoder based on Handel-C,” 2009 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication, 2009, pp. 505-508, doi: 10.1109/ICASID.2009.5276988.

[7] D. R. Vasanthi, R. Anusha and B. K. Vinay, “Implementation of Robust Compression Technique Using LZ77 Algorithm on Tensilica’s Xtensa Processor,” 2016 International Conference on Information Technology (ICIT), 2016, pp. 148-153, doi: 10.1109/ICIT.2016.041.

[8] “FileChannel (Java Platform SE 7)”, Docs.oracle, 2021. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/nio/channels/FileChannel.html. [Accessed: 20-Apr-2021].