UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Entorno integrado para el análisis de transacciones de datos de un decodificador de vídeo escalable

Autor: **D. Abelardo Báez Quevedo**

Tutor(es): **Dr. D. Gustavo I. Marrero Callicó**

Dr. D. Sebastián López Suárez

Fecha: **Julio de 2013**

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Entorno integrado para el análisis de transacciones de datos de un decodificador de vídeo escalable

HOJA DE FIRMAS

Alumno/a: D. Abelardo Báez Quevedo Fdo.:

Tutor/a: Dr. D. Gustavo I. Marrero Callicó Fdo.:

Tutor/a: Dr. D. Sebastián López Suárez Fdo.:

Fecha: Julio de 2013

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Instituto Universitario de Microelectrónica Aplicada
Sistemas de información y Comunicaciones

Máster en
Tecnologías de Telecomunicación

Trabajo Fin de Máster

Entorno integrado para el análisis de transacciones de datos de un decodificador de vídeo escalable

HOJA DE EVALUACIÓN

Calificación: .....

Presidente: Fdo.:

Secretario: Fdo.:

Vocal: Fdo.:

Fecha:

IUMA
t +34 928 451 086
f +34 928 451 083
iuma@iuma.ulpgc.es
www.iuma.ulpgc.es
Campus Universitario de Tafira
35017 Las Palmas de Gran Canaria

Versión: 2.0 – 14 jun 2011

Índice general

Índice de figuras

Listado de Acrónimos

ADSL Asymmetric Digital Subscriber Line

API Application Programming Interface

ARM Advanced RISC Machines

CABAC Context-Adaptive Binary Arithmetic Coding

CAVLC Context-Adaptive Variable Length Coding

CDROM Compact Disc Read Only Memory

CGS Coarse-Grain Quality Scalability

CPU Central Processing Unit

CVS Concurrent Versions System

DSI División de Diseño de Sistemas Integrados

DSP Digital Signal Processor

DVD Digital Versatile Disc

FGS Fine-Grain Quality Scalability

FIR Finite Impulse Response

GOP Group Of Pictures

IDE Integrated Development Environment

IEC International Electrotechnical Commission

IETR Institute of Electronics and Telecommunications-Rennes

INSA Institut National des Sciences Appliquées

ISO International Organization for Standardization

IUMA Instituto Universitario de Microelectr ?nica Aplicada

JSVM Joint Scalable Video Model

JVT Joint Video Team

MGS Medium-Grain Quality Scalability

MPEG Moving Picture Expert Group

NAL Network Abstraction Layer

PDA Personal Digital assistant

RAM Random Access Memory

SNR Signal to Noise Ratio

SVC Scalable Video Coding

TCPMP The Core Pocket Media Player

TDT Televisión digital terrestre

TFM Trabajo Fin de Master

TV Televisión

UIT-T Sector de Normalización de las Telecomunicaciones de la UIT

UIT Unión Internacional de Telecomunicaciones

VCEG Video Coding Experts Group

VCL Video Codinging Layer

W3C World Wide Web Consortium

XML eXtensible Markup Language

Capítulo 1

Introducción

1.1. Introducción

La era digital, en donde las telecomunicaciones intervienen de una forma significativa, ha supuesto para el vídeo un importante incremento en su uso y su público. El sistema DVD, el Blu-ray, la TDT, los servicios como Youtube, el *broadcasting* de vídeo a través de Internet, la telefonía móvil, o los dispositivos de reproducción Full-HD, son un buen ejemplo de ello. Además hoy en día contamos con numerosos estudios, como el realizado recientemente por Cisco, "*Hyperconnectivity and the Approaching Zettabyte Era*" [1], en el que se refleja entre otros datos que la transmisión de vídeo a través de Internet supuso en el 2012 más de la tercera parte del tráfico en Internet, y sugiere que en el 2017 pueda exceder el 70 %, tal como se observa en la Figura 1.1

El auge en la demanda de las aplicaciones de vídeo es consecuencia directa en gran medida a la mejora en los sistemas de transmisión, y sobre todo en la investigación y desarrollo de los sistemas de codificación, pues de poco sirve esforzarse en crear una infraestructura de telecomunicaciones con tecnología innovadora, si el canal de transmisión no se aprovecha correctamente, debido a un uso inadecuado en la codificación de la información.

Los estándares en la codificación de vídeo se han esforzado en aprovechar de forma eficiente el canal con el objetivo de conseguir la máxima calidad con la menor cantidad posible de información. Como resultado han surgido multitud de estándares, tales como H.261 (1990), MPEG-1 Vídeo (1993), MPEG-2 Vídeo (1994), H.263 (1995,1997), MPEG-4 Visual (1998) [2] o H.264/AVC (2003) [3], y más recientemente H.265/HEVC [4] que han logrado aprovechar el sistema de transmisión con mayor o menor eficiencia.

En la actualidad H.264/AVC ha logrado demostrar que ofrece una calidad y una eficiencia a niveles muy aceptables, siendo el estándar que mayor difusión ha sufrido en los últimos

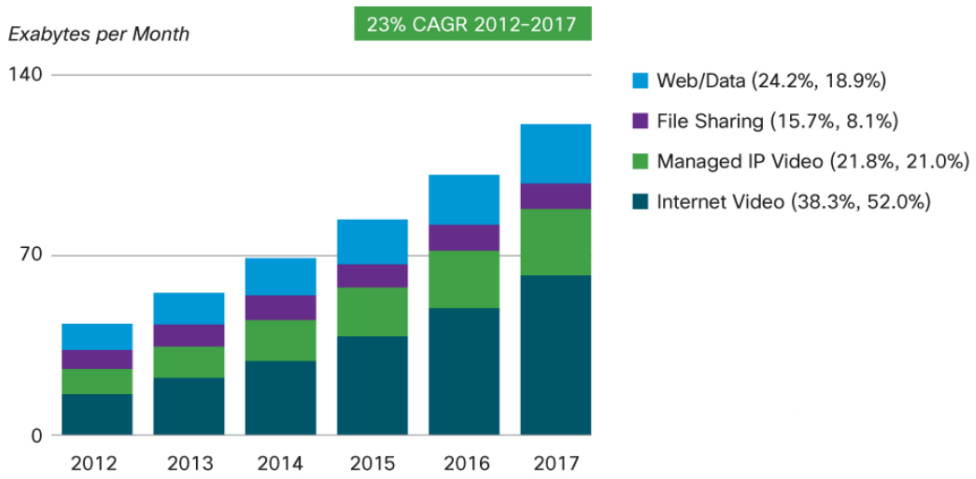


Figura 1.1: Gráfica del uso de Internet según un estudio de Cisco.

años. Sin embargo, todos estos estándares carecen de un sistema eficiente para ofrecer una calidad degradada de su información.

SVC (*Scalable Video Coding*) [5], es un una extensión del estándar H.264/AVC, y se presenta como la solución ideal a las carencias de los actuales sistemas de codificación de vídeo.

El término escalabilidad en SVC se refiere al hecho de que se puede eliminar parte de la información original codificada, generando por lo tanto menor información, y en consecuencia obteniendo una información degradada, pero que puede ser decodificada, y por ende, adaptada a las preferencias del usuario, o a las características de su terminal.

SVC mejora la respuesta ante fallos y añade robustez al sistema, pues no es necesario disponer de toda la información en el receptor. El decodificador, puede usar parte de la información original y obtener un vídeo degradado, esto quiere decir que el receptor consigue adaptarse a los problemas que existan en el sistema de transmisión, o en el sistema de decodificación. Este hecho resulta especialmente útil en las aplicaciones donde se utiliza transmisión en tiempo real, como la videovigilancia, o las aplicaciones basadas en RTP/IP.

SVC reduce drásticamente el ancho de banda empleado en la multidifusión de vídeo, pues al contrario que en los estándares anteriores, SVC permite reunir en una única codificación los distintos perfiles que demanda el usuario, ya que el receptor descartará la información que no necesita, y decodificará sólo aquella que se ajuste a sus características o a sus necesidades.

Otro aspecto clave en el éxito de SVC es el hecho de que se establece como una extensión, lo cual implica que utiliza los mismos bloques funcionales que se utilizan en H.264/AVC, in-

corporando otros nuevos que dotan al sistema de la escalabilidad, pero aprovechando todo el trabajo de investigación y desarrollo que se ha invertido en H.264/AVC.

El diseño de un sistema empotrado comienza con una descripción software del sistema, generalmente en lenguaje C. En muchos casos el diseñador genera un software que implementa las especificaciones que previamente han establecido organizaciones que desarrollan un determinado estándar. En este caso, el software no está optimizado, ya que sirve únicamente para cumplir con las especificaciones y comprobar el funcionamiento de la aplicación.

Adaptar este software al sistema empotrado final, requiere un diseño del sistema en el que se establecen entre otros aspectos, el diseño de la arquitectura, la partición hardware/software, la optimización del software, o la implementación de aceleradores hardware. Todas las partes del diseño requieren un estudio de rendimiento de las distintas partes del software, ya sea debido a grandes necesidades de cálculo, o al uso intensivo de la memoria, que finalmente pueden influir en aspectos tan relevantes como el consumo de potencia.

1.2. Open SVC Decoder

Open SVC Decoder [6] es un decodificador de código abierto escrito en C que implementa el estándar SVC, y que ha sido incluido en al menos dos de los reproductores de código abierto mas famosos, TCPMP y Mplayer.

Ha sido desarrollado por el IETR-INSA "*Institute of Electronics and Telecommunications - Rennes*" [7], y está diseñado en C de forma que pueda implementarse en varias plataformas basadas en x86, PDA's con procesadores ARM o xscale, y en la familia de procesadores DSP-c64x.

Además, al tratarse de un *software* de código abierto, se ofrece la posibilidad de adaptar su código a la investigación y el desarrollo, lo que le confiere cierta flexibilidad a la hora de mejorarlo o adaptarlo, según las necesidades.

1.3. Herramientas de *profiling*

Desarrollar un programa de forma óptima no es tarea sencilla, y suele ser uno de los objetivos principales en cualquier empresa que se dedique al desarrollo de *software* [8] [9]. A menudo, cuando se desarrolla un proyecto de *software*, gran parte del esfuerzo se emplea en

optimizar y detectar aquellas partes del código que exigen un mayor uso de los recursos.

Las herramientas de *profiling* facilitan el trabajo de identificar aquellas partes del código que llevan una mayor carga computacional, o que usan la memoria de manera intensiva. De esta forma, se permite actuar en consecuencia modificando el código, con el objetivo de disminuir el número de recursos necesarios.

Existen diversas herramientas de *profiling* en el mercado, algunas de código abierto como Gprof, otras específicas para determinadas arquitecturas como ARM RealView Profiler o VTune, y otras comerciales como Rational Purify de IBM o Insure++ de PARASOFT.

Valgrind es un conjunto de herramientas de código abierto que se ejecutan bajo linux, y que permiten realizar tareas de profiling sobre código C o C++, con el que se puede obtener un uso detallado del consumo de CPU, el uso de la memoria, detectar errores en los hilos de ejecución o incluso errores de memoria.

1.4. Objetivos

El presente Trabajo Fin de Master (TFM) se encuadra dentro de una de las líneas de investigación sobre técnicas de reescalabilidad, dentro del proyecto de investigación del plan nacional de I+D+i denominado DREAMS, "*Dynamically Reconfigurable Emdedded Platforms for Networked Context-Aware Multimedia*" en el que participa la División de Diseño de Sistemas Integrados (DSI), perteneciente al Instituto Universitario de Microelectrónica Aplicada (IUMA).

El objetivo de este TFM es colaborar con la línea de investigación creando un entorno de análisis de prestaciones de un decodificador de vídeo escalable, que facilite el estudio de las transacciones de datos.

Para el desarrollo del entorno se utilizará el lenguaje Python [10] bajo el IDE Eclipse [11] que se ejecuta bajo el Sistema Operativo (SO) linux. Por otro lado, como herramientas de *profiling* se utilizará la herramienta Cachegrind de Valgrind, mientras que el decodificador de vídeo escalable que se desea analizar es el Open SVC Decoder.

El entorno permitirá realizar tareas de *profiling* de forma automatizada de distintos *bit streams*, ofreciendo los resultados en un fichero Excel que contiene el uso de la memoria de las distintas funciones que componen cada código funcional, permitiendo configurar las distintas opciones de decodificación para cada *bit stream*, lo que permitirá estudiar las transacciones

de datos entre las funciones. Además, el entorno debe permitir modificar de manera sencilla las funciones que se someten a estudio, por si fuera necesario incluir nuevas funciones, o modificar las existentes.

1.5. Peticionario

Tras haber superado satisfactoriamente las asignaturas especificadas en el Plan de Estudios del Máster en Tecnologías de Telecomunicación, impartido por el IUMA, se solicita, como requisito indispensable para la obtención del título de Máster en Tecnologías de la telecomunicación, el desarrollo, la redacción, la exposición y la defensa de un Trabajo Fin de Máster.

1.6. Organización del documento

La memoria se compone de seis capítulos, con los siguientes contenidos:

En el capítulo 1 se exponen los objetivos del presente Trabajo Fin de Máster, y se establece el entorno del mismo.

En el capítulo 2 se describe brevemente la extensión SVC del estándar de compresión de vídeo H.264/AVC, en el que se distinguen diversos subapartados donde se explican los aspectos básicos de su funcionamiento, los tipos de escalabilidad, y sus aplicaciones.

En el capítulo 3 se describen las herramientas usadas para desarrollar el entorno, en donde se detalla su funcionamiento y las herramientas externas que se han utilizado para tratar los datos, así como las librerías utilizadas en Python para tratar los archivos XML y Excel.

En el capítulo 4 se describe la herramienta de *profiling* Valgrind, en donde se detallan las características de la herramienta Cachegrind, y los pasos para utilizarla mediante línea de comandos (CLI).

En el capítulo 5 se describen las características del entorno desarrollado, las características básicas y los ficheros de configuración, así como la metodología empleada para interpretar los resultadosobtenidos con Valgrind.

En el capítulo 6 se incluyen los resultados y las conclusiones obtenidas a partir de la

realización del presente Trabajo Fin de Master, así como las líneas futuras de investigación.

Al final de la memoria se encuentra la bibliografía, en donde se detallan las referencias bibliográficas utilizadas como documentación durante el desarrollo del TFM.

Capítulo 2

El estándar H264/AVC y SVC

2.1. Introducción

La evolución en los sistemas multimedia está fuertemente ligada a la evolución de la sociedad de la Información, que cada día exige una mayor calidad de los contenidos, lo que motiva al desarrollo de nuevos sistemas de transmisión, visualización y almacenamiento de los datos en formato digital. Concretamente la explosión de la demanda de contenido multimedia, tanto en internet como en los nuevos dispositivos móviles o a través de vídeo-conferencias, ha sido posible gracias al desarrollo de los sistemas de codificación de datos para la transmisión de vídeo con calidad y con tasas de transferencias relativamente bajas.

La evolución de estos sistemas se basa en la codificación de los datos a transmitir , que en muchos de los casos asumen pérdidas de información para poder adaptarse a los canales de transmisión maximizando de esta forma la relación entre calidad y tasa binaria.

El sistema de codificación de datos multimedia más popular fue el MPEG-1 Audio Layer 3, más conocido como mp3, desarrollado por el MPEG (*Moving Picture Expert Group*), basado en un sistema de compresión de datos con pérdidas. Este tipo de sistemas de codificación dio paso a la compresión del vídeo digital, de donde han surgido multitud de formatos de codificación entre los que destacan:

- MPEG -1: Orientado al almacenamiento y reproducción de audio y vídeo en soporte CD ROM.
- MPEG-2: Codificación de audio y vídeo para transmisión de vídeo de calidad Televisiva.
- MPEG-4 : Ha ido evolucionando a lo largo del tiempo, y han aparecido más de 20 variaciones llamados “partes”.

De todas las partes de MPEG-4, es la “parte 10” la que define un *codec* de vídeo de alta

compresión, también conocido como H.264 o MPEG- 4 AVC (*Advanced vídeo Coding*). Para el desarrollo de este estándar han intervenido el ISO/IEC *Moving Picture Expert Group* (MPEG) y el ITU-T *Video Coding Expert Group*, lo que dio nombre al JVT (*Joint Video Team*) en diciembre del 2001. Inicialmente la intención era la de crear un estándar capaz de proporcionar una buena calidad de imagen con tasas de transferencia mucho menores que las empleadas por los estándares MPEG-2 o MPEG-4 parte 2.

Estos primeros objetivos no eran suficientes para emplear el sistema en ambientes profesionales, que requieren resoluciones y tasas binarias mucho más elevadas, representación de partes de vídeo sin pérdidas, etc., por lo que surgieron nuevas extensiones para cubrir las necesidades del sector profesional. De aquí surgió la necesidad de proveer de un estándar escalable que permitiera adaptar los datos a los distintos canales de transmisión y a los dispositivos encargados de reproducir dichos datos. De aquí nace una nueva extensión del H.264/AVC . El objetivo de la estandarización del SVC (*Scalable Video Coding*) ha sido la capacidad de codificar tramas de vídeo de alta calidad que contengan uno o varios conjuntos de *bit streams* que puedan ser decodificados con una complejidad y calidad similar, a la que se obtiene empleando el estándar H.264/AVC, y con la misma cantidad de datos.

2.2. Funcionamiento de la extensión SVC

Básicamente un sistema de codificación de vídeo se puede denominar escalable cuando se puede eliminar parte de un *bit stream*, para formar un nuevo *bit stream* que se pueda reproducir gracias a un decodificador, y que contiene la información del *bit stream* original pero con una calidad inferior. Un *bit stream* que no cumple esta propiedad se denomina de capa simple.

La escalabilidad se puede ver como un conjunto básico de capas de calidad, donde cada capa aporta un mayor refinamiento a la calidad final de las imágenes. Existen diversos tipos de escalabilidad:

- *Escalabilidad Espacial*: reducción del tamaño del vídeo.
- *Escalabilidad Temporal*: reducción del número de imágenes por segundo o *framerate*.
- *Escalabilidad de Calidad*: reducción de la fidelidad (informalmente denominada SNR) empleada en la extensión SVC del H.264.

Los distintos tipos de escalabilidad se pueden combinar entre sí.

El criterio de diseño más importante en los estándares de escalabilidad de codificación de vídeo, es la eficiencia del código y su complejidad. Este fue uno de los principios fundamentales del desarrollo del SVC, donde se estableció que únicamente se añadirían nuevas herramientas al estándar H.264/AVC siempre y cuando estas soporten de forma eficiente los distintos tipos de escalabilidad. Esto era precisamente lo que evitaba el uso de escalabilidad espacial y de calidad de los sistemas de codificación anteriores, ya que la aplicación de este tipo de escalabilidades disminuía considerablemente la eficiencia de la codificación, incrementando además la complejidad de los decodificadores.

Para el desarrollo de SVC se establecieron los siguientes requisitos:

- Eficiencia de código similar.
- Pequeño incremento en la complejidad de los decodificadores.
- Soporte de todos los tipos de escalabilidad.
- Soporte del estándar H.264/AVC.
- Soporte para adaptar un *bit stream* simple después de la codificación.

2.3. Aspectos básicos de H.264/AVC

Puesto que SVC es una extensión del estándar H.264 resulta necesario conocer los conceptos básicos este último, con el objetivo de entender las características de SVC.

Al igual que para los estándares anteriores, en el estándar H.264 se definen perfiles y niveles, que especifican restricciones en el *bit stream*. Cada perfil especifica un conjunto de características. Los niveles especifican los límites de los valores que deben tomar los elementos de la sintaxis del estándar. Por lo general, la carga de procesamiento del decodificador y la capacidad de memoria para un perfil dado se desprende de los diferentes niveles.

La última versión del estándar H.264 define siete perfiles:

- *Perfil básico o baseline*, empleado en servicios de conversación en tiempo real, como vídeo conferencia y vídeo teléfono.
- *Perfil principal o main*, para aplicaciones de almacenamiento digital de vídeo y datos, así como de transmisión de televisión.
- *Perfil extendido o extended*, aplicable también a servicios de multimedia en Internet.

- *Perfil alto High*, para procesar vídeo de 8 bits y aplicaciones de alta resolución.
- *Perfil High 10*, para procesar vídeo de hasta 10 *bits* y aplicaciones de alta resolución.
- *El perfil High 4:2:2* soporta el formato de muestreo de los cuadros de crominancia de 4:2:2 y hasta 10 *bits* por muestra de exactitud.
- *El perfil High 4:4:4* soporta el formato de muestreo de los cuadros de crominancia 4:4:4 y hasta 12 *bits* por muestra de exactitud

La introducción de la extensión de escalabilidad (SVC) supuso la introducción de tres niveles:

- *Perfil Escalable Baseline* - Orientado a aplicaciones móviles o de vigilancia.
- *Perfil Escalable High* - Orientado a aplicaciones de almacenaje, *broadcasting* y *streaming*.
- *Perfil Escalable High-Intra* - Destinado para aplicaciones profesionales.

2.3.1. Estructura de las capas de H .264 /AVC

El estándar H.264/AVC está compuesto por dos capas, la *Capa de Abstracción de Red* (NAL, *Network Abstraction Layer*) y la *Capa de codificación de vídeo* (VCL, *Video Coding Layer*)

2.3.2. Capa de Abstracción de Red: NAL

Los datos generados tras la codificación de vídeo se organizan en "unidades NAL", compuestos por un conjunto entero de *bytes*. Estas unidades NAL se pueden clasificar en dos tipos, unidades VNL-NAL y las unidades no VNL-NAL. Las primeras contienen datos de las imágenes, y las segundas contienen información adicional, como parámetros e información suplementaria, pero que no son estrictamente necesarias para la decodificación de las secuencias de vídeo.

2.3.3. Capa de Codificación de vídeo: VCL

La capa de codificación de vídeo en el estándar H.264/AVC presenta diversas mejoras de compresión que permiten incrementar la eficiencia de codificación respecto a los sistemas de codificación anteriores, además de aportar una mayor flexibilidad y adaptabilidad.

Las imágenes se dividen en pequeñas trozos denominados *macrobloques*, consistentes en un bloque de 16x16 píxeles de *luminancia*, un bloque de 8x8 píxeles de *crominancia* roja y

otro de 8x8 píxeles de *crominancia* azul. Las muestras de este *macrobloque* se predicen espacial o temporalmente, y la señal residual de la predicción es codificada. La ventaja de este estándar con respecto a los anteriores es que los *macrobloques* se pueden dividir en bloques menores, lo que proporciona una mayor exactitud en la estimación de movimientos.

Dependiendo del grado de libertad para la generación de la señal de predicción, el estándar soporta tres tipos básicos de imágenes:

- Tipo I: Imágenes-intra, se codifican empleando predicción espacial.
- Tipo P: Imágenes generadas a empleando codificación inter-cuadro.
- Tipo B: Imágenes generadas utilizando codificación predictiva bidireccional.

Para reducir los efectos de los bordes en los bloques, se implementa en el bucle de compensación de movimiento, un filtro adaptable denominado "*deblocking filter*".

El estándar H.264/AVC soporta *dos métodos de codificación de entropía* basados en la adaptación del contexto para incrementar el rendimiento con los anteriores estándares. *La codificación adaptativa de longitud variable* (CAVLC) que emplea códigos de longitud variable y su adaptabilidad se restringe a codificar los niveles de los coeficientes de la transformada. Por otro lado *la codificación binaria aritmética adaptativa* (CABAC), emplea codificación aritmética con un mecanismo mucho más sofisticado, que emplea dependencia estadísticas, lo que le permite ahorrar tasas de *bits* en torno al 10 %-15 % con respecto al CAVLC a costa de un considerable incremento en el coste computacional.

2.4. Tipos de escalabilidad

La extensión de escalabilidad del estándar H.264/AVC puede implementar tres tipos de escalabilidad: *Temporal*, *Espacial* y de *Calidad* (Figura 2.1). Esta implementación debe basarse en el principio de la eficiencia y la complejidad establecidos para la extensión, por lo que el diseño de estos mecanismos deben pasar un estricto control para su implementación.

2.4.1. Escalabilidad Temporal

Se denomina *escalabilidad temporal* cuando un *bit stream* se puede dividir en unidades temporales con sus correspondientes propiedades. Las unidades temporales las podemos denotar como T_K, donde K, representa la capa temporal a la que pertenece la unidad de datos. (Figura 2.2)

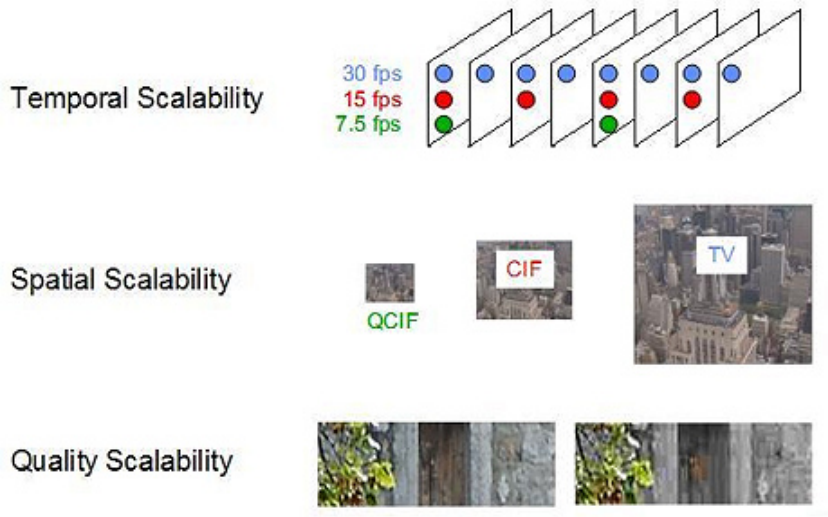


Figura 2.1: Resumen de tipos de escalabilidad en SVC

Los primeros estándares MPEG-1, MPEG-2, H.263 y MPEG-4 ya implementaban la *escalabilidad temporal*, pero raramente se usaron debido a su baja eficiencia y a su complejidad.

El estándar H-264/AVC provee un incremento en la flexibilidad de la escalabilidad temporal debido al control de memoria de imágenes de referencia, lo que permite la codificación de secuencias de imágenes con una dependencia temporal arbitraria. De esta forma el soporte de escalabilidad temporal no cambia el diseño del estándar, aprovechando la codificación del estándar.

2.4.1.1. Estructura de predicción jerárquica

La predicción jerárquica consiste principalmente en predecir las imágenes correspondientes a las capas temporales superiores. Una vez se reciben todas las tramas pertenecientes a un *grupo de imágenes* (GOP), en donde se ha definido una escalabilidad temporal k, se procede a la predicción de las restantes imágenes, pertenecientes a las capas temporales superiores a la definida para el grupo de imágenes.

Los tres ejemplos de predicción jerárquica se pueden observar en la Figura 2.2, en el ejemplo "(a)", se presenta una predicción con tramas tipo B que posee una eficiencia de código excelente para implementar una escalabilidad temporal. En el ejemplo "(b)" se presenta una predicción jerárquica no vinculada, donde se predicen las capas temporales superiores de forma no simétrica, dando prioridad a las imágenes que se representarán primero. En último lugar, en el ejemplo "(c)" , se observa cómo se van generando las imágenes en el mismo orden en el que se deben mostrar, lo cual reduce el retardo, pero tiene el inconveniente que disminuye su eficiencia en el código.

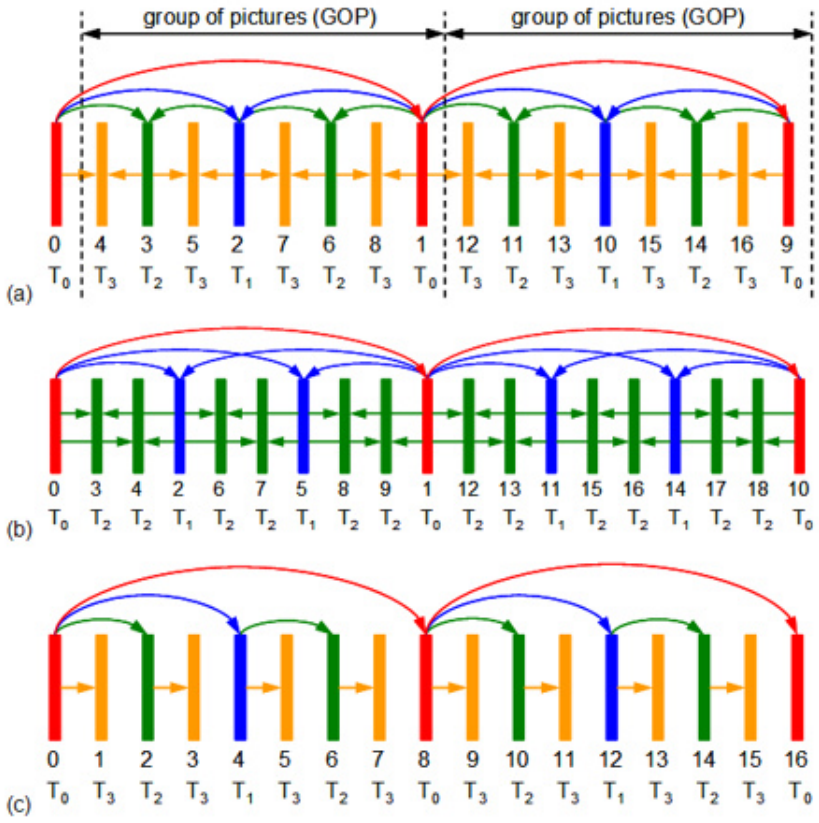


Figura 2.2: Estructura de predicción jerárquica

2.4.2. Escalabilidad Espacial

La extensión SVC del H.264/AVC emplea *la escalabilidad espacial* en forma de capas, donde cada capa corresponde a una resolución espacial soportada y definida por un identificador de dependencia "D". Este indicador posee el valor 0 para la capa base, y aumenta en incrementos de 1 con cada capa espacial. De esta forma se soportan ratios arbitrarios de resolución.

En cada capa espacial se emplea *predicción de compensación de movimiento* y *predicción intra-capa* para la codificación de capas-simples (Figura 2.3)

El principal objetivo a la hora de diseñar herramientas de predicción es la de emplear la información de las capas inferiores para incrementar la eficiencia de las capas superiores.

La señal de predicción es creada a partir de *la compensación de movimiento* dentro de la capa superior por sobre-muestreo de la capa base, o realizando la media de la sub-muestreada con una señal de predicción temporal.

Aunque la reconstrucción de las muestras de las capas inferiores represente completamente la reconstrucción de la capa inferior, estas no son necesariamente los mejores datos

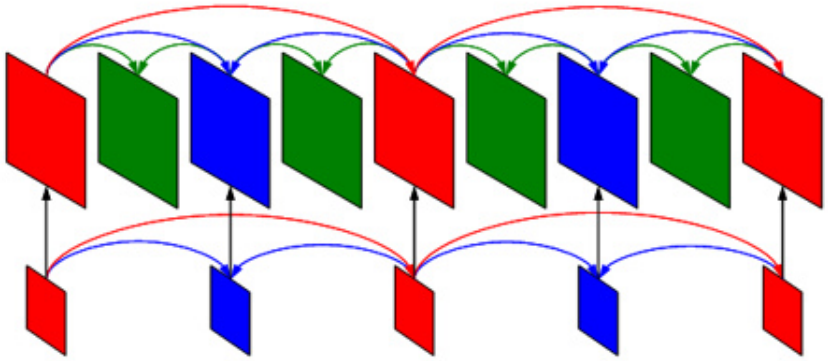


Figura 2.3: Escalabilidad espacial, predicción intra-capa

que se pueden emplear para una *predicción inter-capa*. Generalmente *la predicción inter-capa* compite con la *predicción temporal* y especialmente para secuencias de cámara lenta y alto detalle espacial, la señal de predicción temporal representa una mejor aproximación de la señal original que la reconstrucción de las capas inferiores.

Para mejorar la eficiencia de código en la escalabilidad espacial, se añaden dos conceptos a SVC: *predicción de modos de macrobloques* y sus parámetros asociados al movimiento y *la predicción de la señal residual*. Un codificador SVC puede elegir libremente entre *predicción intra* o *inter-capa* basándose en las características de la señal original.

2.4.2.1. Predicción de movimiento *Inter-capa*

Para las capas superiores, SVC incluye un nuevo tipo de macrobloque, señalizado por un elemento denominado "*Base Mode Flag*". Para este tipo de *macrobloques* sólo se transmite una señal residual sin ningún tipo de información adicional.

Cuando el *macrobloque* de la capa de referencia es *inter-codificado*, el *macrobloque* de la capa de mejora también es *inter-codificado*. En ese caso, la división de los datos del *macrobloque* de la capa de mejora, junto con los índices de referencia asociados, y los vectores de movimiento, se obtienen de los datos correspondientes del bloque 8x8 en la capa de referencia denominada *predicción de movimiento inter-capa*.

La división del *macrobloque* se obtiene del bloque de 8x8 en la capa de referencia correspondiente. Cuando el bloque no se divide en bloques más pequeños, el *macrobloque* de la capa de mejora tampoco es dividido. Así, cada MxN *sub-macrobloque* en la capa de referencia, corresponde a 2Mx2N divisiones de *macrobloques* en el *macrobloque* de la capa de mejora.

El concepto de SVC incluye la posibilidad de emplear vectores de movimiento escalables

del bloque de 8x8 superpuesto, situado en el bloque de la capa de referencia, como herramienta de predicción del *vector de movimiento* para los tipos de *macrobloques* convencionales *inter-codificados*. Se emplea un indicador para señalar si se usa la predicción del *vector de movimiento* llamado "*motion prediction flag*".

2.4.2.2. Predicción residual *Inter-capa*

Esta predicción puede ser usada para el resto de *macrobloques inter-codificados*, aunque no estén codificados empleando el nuevo tipo de *macrobloque* del SVC, indicado por el indicador de modo base denominado "*reference base mode*". Se añade un indicador al *macrobloque* para capas de mejora espacial, empleada para *la predicción residual inter-capa*.

Cuando se emplea este tipo de predicción, la señal residual del correspondiente *sub-macrobloque* de 8x8 de la capa de referencia, es interpolada empleando un filtro bilineal. La interpolación del residuo de la capa de referencia se realiza mediante una transformación básica del bloque para asegurar que no se aplica ningún filtro a lo largo de los bordes del bloque, evitándose de esta forma distorsión en la señal.

2.4.2.3. Intra-predicción *Inter-capa*

Cuando un *macrobloque* de la capa de mejora se codifica con "*base mode flag*" igual a 1 y los *sub-macrobloques* 8x8 en su capa de referencia son *intra-codificados*, la señal de predicción del *macrobloque* de la capa de mejora es obtenida por *intra-predicción inter-capa*, para la cual la reconstrucción *intra* de la capa de referencia es interpolada. En el sobre muestreo de la componente de luminancia se emplea un filtro FIR unidimensional de 4 etapas horizontal y vertical. Las crominancias son interpoladas empleando un filtro bilineal simple. El filtrado se realiza a través de los bordes de los *macrobloques* empleando las muestras de los *macrobloques* adyacentes. Cuando los bloques vecinos no son *intra-codificados* las muestras necesarias son generadas por algoritmos de replicación de bordes.

2.4.2.4. Consideraciones de complejidad

Se ha demostrado que la complejidad del codificador puede ser reducida al restringir el uso de la *intra-predicción inter-capa*. Esto se ha denominado "restricciones de la predicción *inter-capa*", por lo tanto no se suelen utilizar con el objetivo de evitar operaciones con coste computacional alto y accesos intensos a memoria en la *compensación de movimiento*.

Como consecuencia, el uso de la *intra-predicción inter-capa* sólo se permitirá para *macrobloques* de la capa de mejora, donde el *macrobloque* correspondiente es *intra-codificado* en la capa de referencia. Además, se requiere que todas las capas que se utilizan para la predicción

inter-capa de las capas superiores sean codificadas empleando una *intra-predicción* limitada, por lo que el *macrobloque intra-codificado* de la capa de referencia puede ser construido sin la reconstrucción de cualquier *macrobloque inter-codificado*.

Estas restricciones son obligatorias en SVC, así como el hecho de que cada capa debe ser decodificada con un solo bucle de compensación de movimiento. De este modo, el incremento en la complejidad del decodificador para SVC, en comparación con una sola capa de codificación, es menor que en los estándares de codificaciones anteriores. Cada unidad NAL de capa de mejora de calidad temporal o espacial puede ser analizada independientemente a la capa inferior, lo que brinda nuevas oportunidades para reducir la complejidad del decodificador.

2.4.3. Escalabilidad de Calidad

La *escalabilidad de Calidad* puede ser considerada como un caso especial de escalabilidad espacial con idénticos tamaños de imágenes para las capas base y las capas de mejora. Este tipo de escalabilidad se basa en el concepto de codificación espacial escalable y es conocida también como "*Coarse-Grain quality Scalable coding*" (CGS). Se emplean los mismos mecanismos de predicción *inter-capas* que los empleados en la *codificación espacial escalable*, pero sin realizar las operaciones de sobre muestreo y el "*deblocking filter*" para *macrobloques intra-codificados* de la capa de referencia.

Sin embargo, este concepto de multicapa para la codificación de escalabilidad de calidad, sólo permite soportar unas determinadas tasas de transferencia en un flujo de *bits* escalable e igual al número de capas. El cambio entre diferentes capas CGS sólo puede hacerse en puntos definidos en los *bit streams*. Por otra parte el concepto de codificación escalable de calidad multicapa pasa a ser menos eficiente, cuando las diferencias de las tasas relativas entre las sucesivas capas CGS son más pequeñas.

En el diseño del SVC, se crea también una modificación del CGS, denominada "*Escalabilidad de calidad de grano medio*" (MGS), que permite aumentar la flexibilidad del *bit stream*, la robustez frente a errores, y la eficiencia del código. Las diferencias con el concepto de CGS es un nivel de señalización mayor, que permite un cambio entre diferentes capas de MGS en cualquier unidad de acceso, junto con el concepto de imagen clave, que permite el ajustar el compromiso entre la deriva y la codificación eficiente, de la capa de mejora para estructuras de jerárquicas.

Con el concepto MGS, cualquier unidad NAL de la capa de mejora puede ser descartada

de un flujo de bits de calidad escalable, y por tanto, se provee la codificación escalable de calidad.

2.4.3.1. Control de deriva "*drift*"

El proceso de *predicción de movimiento compensado* para codificación de calidad escalable, basada en paquetes, debe ser diseñado cuidadosamente, ya que determina el compromiso entre la eficiencia de la codificación de la capa de mejora y *la deriva*.

La deriva o "*Drift*" describe el efecto que se produce cuando los bucles de predicción de compensación de movimiento en los codificadores y en el decodificador no están sincronizados debido a que los paquetes de refinamiento de calidad son descartados de un *bit stream*. La Figura 2.4 muestra diferentes conceptos de negociación de la eficiencia del código de la capa de mejora, y la deriva de la codificación escalable de calidad basada en paquetes. Para

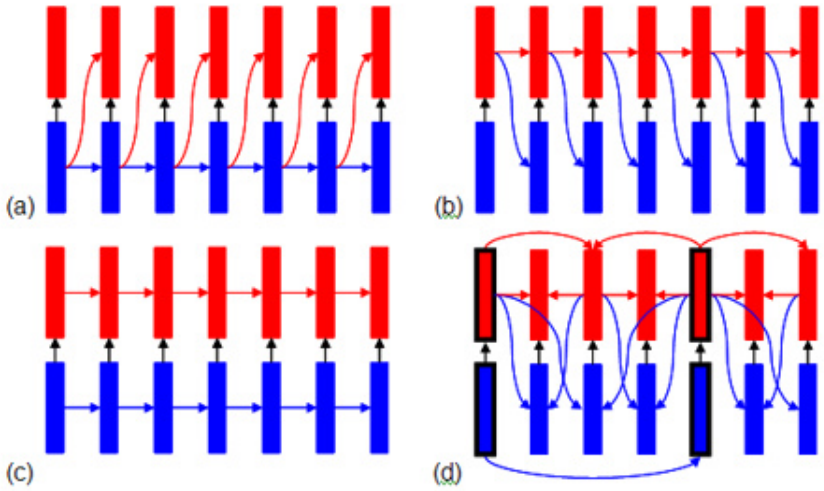


Figura 2.4: Concepto de negociación de eficiencia de código y deriva

la *codificación escalable de calidad de grano fino* (FGS) del MPEG-4 Visual, la estructura de predicción se escogió eliminando completamente *la deriva*. Tal y como muestra en el ejemplo "(a)" de la Figura 2.4, *la compensación de movimiento* solo se realiza empleando la reconstrucción de la capa base como referencia. Por lo tanto, cualquier pérdida o modificación del refinamiento de calidad, no tiene ningún impacto en el bucle de *compensación de movimiento*, pero disminuye significativamente la eficiencia de la codificación de la capa de mejora respecto a una codificación de capa simple.

Para la *codificación escalable de calidad* en H.262 MPEG-2 vídeo, la referencia con la calidad más alta se emplea siempre para la predicción de compensación de movimiento, como se muestra en el ejemplo "(b)" de la Figura 2.4. Esto permite una alta eficiencia en la codificación de la capa de mejora y asegura una baja complejidad. Sin embargo, cualquier

pérdida de paquetes de refinamiento de calidad, produce una *deriva* que solo puede ser corregida mediante una *intra-actualización*.

Como alternativa, se puede emplear un concepto basado en dos bucles de *compensación de movimiento*, tal como se observa en el ejemplo "(c)" de la Figura 2.4, similar al de la codificación escalable espacial, especificada en H.262 MPEG-2 vídeo, H.263 y MPEG-4 Visual. Aunque la capa de base no se ve influenciada por las pérdidas de paquetes en la capa de mejora, cualquier pérdida de un paquete de incremento de calidad, se traduce en una *deriva* en la reconstrucción de la capa de mejora.

Para la codificación de MGS en SVC se emplean *las imágenes clave*. Para cada imagen se transmite un "*flag*", que indica si se emplea la reconstrucción de calidad base, o la reconstrucción de la capa de mejora de las imágenes de referencia en la predicción de compensación de movimiento. Para limitar el coste computacional del decodificador para las imágenes clave, SVC especifica que los parámetros de movimiento no deben cambiar en la representación de las imágenes clave de la capas base y la capa de mejora. Además, para las imágenes clave, la codificación no se puede hacer con un bucle único de compensación de movimiento. El ejemplo "(d)" de La figura 2.4 muestra como el concepto de imagen clave puede ser combinado eficientemente con una estructura jerárquica de predicción.

2.5. Aplicaciones e implementaciones

Las aplicaciones en las que se emplea el vídeo digital son cada día más numerosas, gracias a las mejoras introducidas en la codificación, lo cual ha permitido principalmente el aumento de calidad disminuyendo la tasa de bits. De esta forma, el uso del vídeo puede llegar a emplearse cada vez más en un mayor número de dispositivos, lo que hace tan sólo unos años parecía inviable.

La codificación de vídeo se emplea en multitud de entornos, por ejemplo en sistemas de circuito cerrado de televisión empleando vídeo IP. Dada la disminución de los requerimientos de ancho de banda para una misma calidad de imagen, este tipo de sistemas permite niveles de almacenamiento entre el 25 % y el 50 % respecto a los sistemas de codificación anteriores, o una mayor calidad de imagen empleando el mismo ancho de banda.

Otro tipo de aplicación es la retransmisión de vídeo de señales de televisión digital sobre TDT. Actualmente en España existen muchas cadenas de TV que han comenzado a emitir en H.264, alcanzando una resolución de 1440x1080i. La emisora RTVE emitió con gran éxito en *Alta Definición* los juegos olímpicos de Pekín 2008, a través de satélite y cable, y

actualmente disponen de un canal en *Alta Definición* a través de la TDT. Es importante destacar que para la recepción de estos canales de alta definición es necesario decodificadores que soporten H.264, no siendo válido la gran mayoría de los decodificadores que existen en la actualidad que utilizan MPEG-2. En cualquier caso la situación actual sugiere que el número de decodificadores que soportan H.264, y que actualmente ya vienen incorporados en los aparatos de TV va en aumento.

SVC todavía no se ha difundido lo suficiente, y su éxito en la TV dependerá del coste y la modificación que puedan sufrir los actuales receptores que ya soporten H.264. Sin duda, aquellas aplicaciones en las que se exija una fiabilidad importante en la conexión, como por ejemplo en las relacionadas con la vídeo vigilancia, encontrarán en SVC una valiosa alternativa.

Otra aplicación interesante es la que pueden emplear las compañías de telefonía en colaboración con las televisiones para emitir sus contenido a través de plataformas de telefonía móvil. En este aspecto, SVC es la respuesta al gran consumo de recursos que supone emitir una misma señal de vídeo para cada uno de los perfiles estándar que poseen los usuarios, sin mencionar la robustez que añade SVC cuando un terminal no recibe la suficiente potencia de señal.

El vídeo bajo demanda es otra aplicación interesante. Actualmente existen tiendas online que permiten obtener contenido audiovisual de alta calidad, ya sean comerciales, o gratuitas, como VIMEO o WUAPI, (*http://www.vimeo.com/* o *http://wuapi.com/*), donde se puede comprobar la gran calidad que se puede alcanzar empleando el estándar H.264 con tasas de transferencia típicas de un acceso básico mediante ADSL.

2.6. Conclusiones

SVC es un anexo que aumenta de manera significativa la eficiencia del estándar H.264. La escalabilidad permite entre otras cosas disminuir significativamente el ancho de banda utilizado por parte del emisor, ya que éste no tiene que enviar distintos *bit streams* para cubrir las necesidades de los distintos perfiles. También reduce el ancho de banda utilizado por el receptor, ya que aunque el receptor sea capaz de decodificar el *bit stream* completo, puede interesarle decodificar sólo aquellas capas que le permitan finalizar la reproducción, ya sea debido a falta de potencia en la señal, o por falta de batería en el dispositivo del receptor. Además, añade robustez al sistema, pues en el caso de que no se reciba el *bit stream* en su totalidad, existe la posibilidad de recibir partes del *bit stream* que sí puedan decodificarse por si mismas. Las ventajas de la escalabilidad son tan relevantes, que probablemente se

implementen en el nuevo estándar HEVC.

Capítulo 3

Especificaciones del entorno de análisis

3.1. Elección de la interfaz del entorno

Existen ciertos requisitos a tener en cuenta cuando se eligen las herramientas que se van a utilizar para el desarrollo de un entorno destinado al ámbito de la investigación. Un aspecto importante es la eficiencia y versatilidad de las herramientas utilizadas para el desarrollo. Otro aspecto importante es conseguir que el impacto en el equipo de investigación no suponga un problema entre el tiempo necesario para aprender a usarlo, y la compatibilidad con las otras herramientas ya desarrolladas por el equipo de investigación.

Además, en este caso existen algunas exigencias que obligan a utilizar determinadas herramientas y entornos.

En el pasado se desarrolló una herramienta de profiling en Windows, utilizando las herramientas de profiling de Visual Studio 2010, y esperando que los resultados ofrecidos por la herramienta abarcaran los dos ámbitos principales en el consumo de recursos de un sistema, el uso de la CPU y el uso de la memoria.

Las herramientas de profiling consiguen determinar el uso de memoria consultando los contadores que monitorizan el uso de las caches, sin embargo, puesto que las herramientas de profiling consultan los contadores de la CPU, fue imposible determinar con exactitud el consumo de memoria, pues no se tiene acceso a la cache de nivel 1 (L1).

Tras consultar las distintas herramientas del mercado, se detectó que el acceso real al contador que memoriza el uso de la cache L1 no es posible, debido a la cercanía de dicha

memoria al núcleo del procesador, ya que la simple gestión de su lectura mediante un contador ralentizaría en exceso todo el sistema, y debido a esto los diseñadores no incluyen un contador para ese nivel.

Sin embargo, se detectó que existen otras herramientas que no utilizan los contadores de la CPU, sino que se basan en emular una plataforma hardware en la que se establecen las memorias y el procesador, ofreciendo la posibilidad de monitorizar lo que ocurre con L1.

Esto es vital para conocer el uso real de la memoria por parte de cualquier software, pues por ejemplo, una matriz sobre la que se realicen múltiples operaciones, se mantendrá en la cache L1 mientras el software la necesite, y si no se tiene acceso a dicha memoria, no se podrían contabilizar todos los accesos a memoria de las operaciones realizadas con la matriz.

Una de las herramientas más extendidas en el ámbito del desarrollo de software es Valgrind, un conjunto de herramientas de codigo abierto que permiten emular una arquitectura, y ofrecen la posibilidad de monitorizar lo que ocurre con la caché L1. Debido a su importancia su explicación detallada de características y uso se resumen en el Capítulo 4.

El lenguaje de programación elegido es Python, debido entre otros aspectos a la flexibilidad que ofrece para ser usado en otros sistemas operativos, variando la parte del código que hace referencia a la herramienta de profiling. Además, Python ha demostrado ser un lenguaje de alto nivel con una curva de aprendizaje muy suave, y con una amplio catálogo de APIS, librerias, y manuales que facilitan su aprendizaje.

3.2. El lenguaje Python

Python fue creado a finales de los ochenta por *Guido van Rossum* en el Centro para las Matemáticas y la Informática (*CWI, Centrum Wiskunde & Informatica*), en los Países Bajos. Su nombre se debe a la afición de su creador por los humoristas *Monty Python*.

Es un lenguaje multiparadigma e interpretado, lo que garantiza que el software que se genere pueda ser usado en distintas plataformas, siempre y cuando exista un intérprete para la plataforma, y además el código generado no utilice librerías que usen aspectos básicos del sistema. Como ejemplo, el registro de Windows, o las rutas propias de un sistema operativo Unix.

En cualquier caso, esto puede tenerse en cuenta cuando se crea un programa, y decidir en función del sistema operativo dónde se ejecuta, qué acciones se pueden llevar a cabo.

Uno de sus propósitos es el de ser un lenguaje con una sintáxis muy limpia, lo que es de agradecer en programas muy extensos, donde la limpieza en el código es escencial.

Existe un amplio catálogo de APIS y librerias, una de las más famosas es Django, que permite crear aplicaciones web utilizando el paradigma modelo-vista-controlador, y que ha demostrado ser una excelente herramienta muy extendida entre los desarrolladores, o las librerias wxForm, que permiten crear y usar entornos gráficos.

3.3. El entorno de desarrollo *Eclipse*

Eclipse es un IDE ampliamente extendido entre la comunidad, compuesto por un conjunto de herramientas de programación de código abierto. Utiliza módulos o también llamados *plug-ins* para ofrecer nuevas funcionalidades según el lenguaje que se desee utilizar.

Utiliza un editor de texto con resaltado de síntaxis, y la posibilidad de autocompletar parte del código, como los métodos de los objetos ya conocidos, las funciones creadas por el usuario, etc.

Principalmente se usó para generar código Java, pero hoy en día su uso está muy extendido, y soporta multitud de lenguajes, en función de los módulos instalados, contiene un control de versiones (CVS)

La primera versión (*Eclipse 3.0*) se lanzó en 2004 y en la actualidad existe la version 4.3 *Eclipe Kepler* que es el entorno utilizado para desarrollar el entorno de análisis del presente TFM.

3.4. El estándar XML

XML es un metalenguaje extensible de etiquetas desarrollado por el *World Wide Web Consortium* (W3C), aunque cabe destacar que no ha nacido sólo para su aplicación en Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo, etcétera.

La tecnología XML busca dar solución al problema de expresar información estructurada de la manera más abstracta y reutilizable posible, entendiéndose por estructurada aquella

información que se compone de partes bien definidas, y que esas partes a su vez puedan contener de otras partes. Al final lo que se obtiene es un árbol de trozos de información estructurada.

Es extensible, ya que después de diseñado y puesto en producción, es posible extender XML con la adición de nuevas etiquetas, de modo que se pueda continuar utilizando sin complicación alguna, y portable, ya que si un tercero decide usar un documento creado en XML, es sencillo entender su estructura y procesarlo en consecuencia, lo que permite comunicar aplicaciones de distintas plataformas. Por ejemplo, podríamos tener una aplicación en Linux con una base de datos PostgresSQL y comunicarla con otra aplicación en Windows y Base de Datos MS-SQL Server, o MySQL.

3.5. *BitStreamExtractorStatic* (JSVM)

El *Joint Video Team* (JVT), un esfuerzo de colaboración colectivo entre la ITU-T VCEG y el ISO/IEC, ha desarrollado el *software* JSVM (*Joint Scalable Video Model*), un *software* de referencia para el estándar de codificación de vídeo escalable SVC.

Está desarrollado en C++ y se ofrece su código fuente, de manera que puede compilarse en plataformas *Windows* y *Linux*, y está compuesto de varias aplicaciones que implementan la extensión SVC, como un codificador, un decodificador y varias utilidades para analizar y modificar *bit streams* que cumplan con el modelo estándar de SVC.

BitStreamExtractorStatic es una de las utilidades que se incorporan en el *software* JSVM, y sirve para extraer *sub streams* de un *bit stream* que cumpla el estándar SVC. Cuando se ejecuta *BitStreamExtractorStatic* especificando un *bit stream* sin añadir ninguna opción se muestran las distintas capas escalables que contiene el *bit stream*, lo que la hace especialmente útil cuando se desea comprobar si un *bit stream* contiene errores, ya sea debido a que no contenga las capas deseadas, o debido a una codificación incorrecta.

3.6. Herramientas auxiliares

3.6.1. *LibreOffice Calc* o *Microsoft Excel 2003*

El entorno ofrece genera como resultado una hoja de cálculo en formato *Excel* (.xlsx), en el que se detallan los resultados obtenidos con la herramienta de profiling, por lo que necesario disponer al menos de *Microsoft Excel 2003*, LibreOffice Calc, o cualquier otro programa que permita visualizar documentos xlsx.

3.7. Conclusiones

Python es un lenguaje que agrada desde el primer momento, la sencillez y la elegancia de su sintaxis están en armonía con su potencia y su flexibilidad. Los creadores de Python buscaban esa belleza en el lenguaje, los tipos de datos contienen muchísimos métodos a través de sus clases, que permiten realizar acciones de manera sencilla y eficiente, la exigencia en la escritura mediante la sangría obligatoria impone la legibilidad del código fuente, y la enorme comunidad científica e investigadora que apoyan el lenguaje, hacen que su aprendizaje resulte una tarea sencilla. En la actualidad existen numerosas librerías que simplifican el desarrollo, y los manuales, tutoriales y guías on-line abundan en la red, de manera que no resulta difícil encontrar material didáctico.

XML es un estándar muy flexible que permite gestionar y almacenar datos en un fichero de texto plano, que puede ser editado con cualquier procesador de textos. Esta característica hacen que sea un método ideal para tratar los datos de manera externa a la aplicación que los maneja, y que además garantice su portabilidad, además, XML no contiene restricciones en el tipo de datos almacenado, pues al contener cadenas alfanuméricas, es el intérprete del archivo XML quien decide qué tipo de datos contiene, y cómo tratarlos.

Capítulo 4

Valgrind

4.1. Introducción

Una de las grandes inversiones realizada por las empresas desarrolladoras de *software* es la destinada a la optimización del código, máxime si tenemos en cuenta que algunas abarcan el ámbito de la implementación *hardware*, donde entran en juego aspectos tan importantes, como el propio coste *hardware*, su escalabilidad, su diseño, o su consumo energético. Una herramienta de *profiling*, a menudo llamadas también *performance tools* o *profiler tools*, es básicamente un conjunto *software* que es capaz de analizar una aplicación, obteniendo como resultado una visión global del uso de los recursos del sistema por parte de la aplicación a estudio.

Existen varias herramientas de *profiling*, algunas de código abierto como *Valgrind*, y otras comerciales y de código privativo como *Rational Purify* o *Insure++*, entre otras. Todas se centran en la detección de errores o en la optimización del código, por ejemplo, registrando el consumo de CPU de la aplicación, o de las funciones del código de la aplicación, o llevar un control de los accesos a la memoria.

Garantizar que una herramienta de *profiling* es más adecuada que otra en ocasiones se vuelve una tarea difícil, si se tiene en cuenta que el criterio de decisión viene condicionado por el análisis de los resultados de la herramienta a lo largo del tiempo. Por lo tanto, generalmente una empresa que se dedica al desarrollo de *software* no se limita a utilizar una única herramienta, sino que compara los resultados obtenidos entre distintas herramientas.

Una vez analizados los resultados, se procede a identificar aquellas partes en el código que suponen una carga en la aplicación, y se modifican si ese es el propósito, o simplemente se identifican, si tras un código optimizado lo que se desea es realizar un estudio con el objetivo de una implementación *hardware*.

Por ejemplo, en la actualidad cuando se diseña un sistema empotrado se suele combinar una serie de arquitecturas con procesadores dedicados a acelerar las partes que realizan un uso intensivo de los recursos. En el caso de un decodificador de vídeo, generalmente se delegan todas aquellas funciones en las que interviene un uso intensivo de los recursos debido a los algoritmos de procesamiento de la señal de vídeo. Este tipo de arquitecturas heterogéneas se encuentra habitualmente en los dispositivos móviles, y suelen denominarse a menudo como aplicaciones procesador, un ejemplo típico es el que se implementa en las arquitecturas de NVIDIA Tegra, STMicroelectronic Nomadik ST8820 o Broadcom BCM2820.

4.2. Características de las herramientas de Valgrind

Valgrind es un conjunto de herramientas de *profiling*, que se llevan desarrollando desde finales de 2008, y que han aumentado su efectividad a lo largo de los años, muestra de ello, son los numeroso proyectos que han utilizado Valgrind para mejorar su código o parte de su código, entre los que podemos destacar servicios tan ampliamente conocidos como CUPS o Samba, programas como Gimp, Blender, Opera o Firefox, motores para bases de datos como MySQL o PostgreSQL, o en el programa científico *Mars Exploration Rover* de la NASA.

La principal ventaja de Valgrind, es poder obtener resultados del uso de memoria de la aplicación analizada, incluidos los accesos a la memoria caché de nivel 1. Para ello Valgrind emula una arquitectura en la que ejecuta la aplicación, y establece registros de control mediante los cuales puede recopilar el número de accesos a dicha memoria.

La aplicación que se desea analizar debe prepararse previamente con el propósito de que la herramienta de *profiling* reconozca qué aplicación se desea analizar y adaptarla a las herramientas de *profiling*, por lo que debe tenerse en cuenta que cualquier aplicación debe ser adaptada en un paso previo a las herramienta.

Básicamente la aplicación analizada se debe compilar en modo *debug*, esto se consigue en *gcc* con el parámetro *-g*, o en su defecto habilitar la opción en el IDE que se utilice, para compilar la aplicación que se desea analizar, en modo *debug*.

4.2.1. Las distintas herramientas de Valgrind

Las herramientas de *profiling* de Valgrind ofrecen distintos resultados en función del tipo de datos que se deseen analizar. Aunque la única herramienta utilizada en este TFM es *Cachegrind* a continuación se ofrece una descripción muy breve de las herramientas más

relevantes.

4.2.1.1. *Memcheck*

Si no se indica una herramienta específica, la herramienta por defecto que usará Valgrind es *Memcheck*.

Memcheck permite detectar errores de memoria, como:

- El acceso no permitido a determinadas zonas de memoria, por ejemplo, si se ha excedido en la reserva de la memoria, el desbordamiento de pila o los accesos a memoria que ya ha sido liberada previamente.
- Uso de valores no definidos, por ejemplo, valores que no se han iniciado o valores que se han generado como consecuencia de otros valores no definidos.
- Liberación incorrecta de memoria dinámica, esto ocurre cuando se libera dos veces bloques de memoria dinámica.
- Fugas de memoria.

Es difícil detectar este tipo de problemas con otros métodos, por lo que a menudo se mantienen sin detectar durante mucho tiempo, hasta que ocasionalmente la aplicación genera un error inexplicable y difícil de detectar, por lo que esta herramienta juega un papel principal en la detección de este tipo de errores.

4.2.1.2. *Cachegrind*

Cachegrind permite obtener el uso de la memoria caché, simula cómo la aplicación interactua con la jerarquias de caches del sistema. Para ello , simula un sistema diferenciando dos niveles de caché, la caché de nivel 1 y el resto de cachés.

Cachegrind diferencia entre los accesos que corresponden a las intrucciones, y los accesos a memoria que corresponden a los datos, de esta manera diferencia entre I1, D1 y LL (*Last-Level* cachés).

Entre otros, *Cachegrind* obtiene los siguientes datos:

- Lecturas de caché **I**, **Ir** corresponde a las lecturas en caché de **instrucciones** ejecutadas, **I1** corresponde con las intrucciones ejecutadas en la caché de nivel **1**, **I1mr** (*read misses*) corresponde con los fallos de lectura de instrucciones en la caché de nivel 1, y **LL** corresponde a los fallos de lectura en las cachés de nivel 2 y 3, por ejemplo, **Ilmr**

corresponde con los fallos de lectura de las intrucciones (*read misses*) en las cachés de nivel 2 y 3.

- De forma análoga **Dr** equivale al numero de lectura de datos, **D1mr** correponde con el número de fallos de lectura de datos en la cache de nivel 1, y **DLmr** corresponde con el número de fallos de lectura de datos en las cachés de nivel 2 y 3.
- **Dw** corresponde con las escrituras, de manera que **D1mw** corresponde a los fallos de escritura de la memoría caché de nivel 1 y **Dlmw** a los fallos de lectura de datos del resto de cachés.
- Además de la memoria caché, *Cachegrind* puede obtener los datos relativos a los saltos que se producen en el código, tanto los saltos ejecutados, como los saltos que no se han predicho correctamente.

De manera que si queremos obtener todos los accesos de datos de la cache de nivel 1, necesitamos tanto los fallos de lectura D1mr, como los de escruta D1mw.

Además *Cachegrind* puede obtener el acceso de cada función en el código, así como el acceso línea a línea de cada una de las instrucciones del código, aunque no sean funciones.

Cabe destacar que *Cachegrind* ofrece muchísima información que es difícil de analizar sin una herramienta que unifique los resultados, de manera que se hace recomendable el uso de la herramienta *cg_annotate* que simplifica la salida, unificando los accesos de cada función.

4.2.1.3. *Callgrind*

Callgrind permite generar un árbol de llamadas entre las funciones, mostrando la jerarquía que existen entre ellas, similar al programa de documentación Doxygen. Por defecto, genera el número de instrucciones que se han ejecutado, la relación con las líneas de código que han llamado a estas instrucciones, la relación entre las funciones, y el número de llamadas a cada función.

También puede obtener el uso de la memoria caché y las predicciones de salto, al igual que *Cachegrind*, añadiendo esta información al árbol de llamadas.

4.2.1.4. Otras Herramientas

Valgrind tiene otras herramientas como *Hellgrind* o *DRD* que permiten detectar errores con los hilos y multihilos del código, como lo errores de sincronización entre hilos.

Massif y *DHAT* son herramientas de Valgrind que permiten obtener datos de rendimiento de la memoria dinámica, como la memoria dinámica en uso, etc.

SGCheck, *BBV*, *Lackey* y *Nullgrind* son también otras herramientas de Valgrind.

4.3. Análisis de los resultados de *Cachegrind*

Una vez que se analiza el tipo de resultados que se desean obtener en función del tipo de aplicación y del nivel de detalle que se desea en el análisis, *Cachegrind* ofrecerá los resultados en un archivo en modo texto.

Para ello, resulta conveniente entender qué tipo de resultados ofrece *Cachegrind*, conocer en qué formatos ofrece la información, y comprender los resultados en relación a los tipos de datos ofrecidos en el modo seleccionado.

4.3.1. El archivo de salida

Si no se especifica lo contrario *Cachegrind* ofrecerá los resultados generales por pantalla, y la información detallada de las funciones en un archivo de texto llamado `cachegrind.out.PID`, donde PID será un número que corresponde con el número del proceso en el que se ha ejecutado el ejecutable con Valgrind.

Si se desea, se puede especificar otro archivo de salida, utilizando el argumento:

`--cachegrind-out-file=`*`fichero`*

Este comando será imprescindible en nuestra herramienta, pues es buena idea disponer de los datos de sesión una vez se ha finalizado el análisis de cada *bit stream* , con el objetivo de disponer de los datos para futuros análisis.

Una vez finalizado el análisis, *Cachegrind* nos mostrará en pantalla el resultado final del uso de la memoria como se puede observar en la figura 4.1.

Esta información sólo nos sirve para obtener una idea global del uso de la memoria de nuestra aplicación, pero no ofrece ninguna información detallando las funciones de nuestro código.

La información detallada de cada función se encontrará en el archivo de salida, como se puede apreciar en la figura 4.2.

```
==3485== I   refs:      255,956
==3485== I1  misses:        786
==3485== LLi misses:        779
==3485== I1  miss rate:    0.30%
==3485== LLi miss rate:    0.30%
==3485==
==3485== D   refs:      123,867  (82,117 rd   + 41,750 wr)
==3485== D1  misses:      1,088  (   938 rd   +    150 wr)
==3485== LLd misses:        995  (   860 rd   +    135 wr)
==3485== D1  miss rate:     0.8% (   1.1%     +    0.3%  )
==3485== LLd miss rate:     0.8% (   1.0%     +    0.3%  )
==3485==
==3485== LL refs:         1,874  ( 1,724 rd   +    150 wr)
==3485== LL misses:       1,774  ( 1,639 rd   +    135 wr)
==3485== LL miss rate:      0.4% (   0.4%     +    0.3%  )
```

Figura 4.1: Salida en pantalla de *Cachegrind*

```
...
...
...
desc: I1 cache:         32768 B, 64 B, 4-way associative
desc: D1 cache:         32768 B, 64 B, 8-way associative
desc: LL cache:         4194304 B, 64 B, 16-way associative
cmd: ./a.out
events: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
fl=/home/abaez/Dropbox/TFM/prueba.c
fn=funcion1
5 300 0 0 0 0 0 100 0 0
8 300 0 0 100 0 0 100 0 0
9 100 0 0 100 0 0 0 0 0
10 200 0 0 200 0 0 0 0 0
fn=funcion2
12 300 0 0 0 0 0 100 0 0
14 500 1 1 100 0 0 300 0 0
15 100 0 0 100 0 0 0 0 0
16 200 0 0 200 0 0 0 0 0
fn=funcion3
18 3 0 0 0 0 0 1 0 0
20 1204 0 0 701 0 0 101 0 0
21 2 0 0 2 0 0 0 0 0
...
...
...
```

Figura 4.2: Archivo de salida de *Cachegrind*

4.3.2. La herramienta *cg_annotate*

Cuando se trata el análisis de la aplicación desde un punto de vista funcional, entendiendo funcional al hecho de analizar cada una de las funciones por separado en la aplicación a estu-

dio, *Cachegrind* obtiene los resultados en relación a la cantidad de hilos que se han ejecutado, lo que dificulta la interpretación de los resultados, pues el valor total del uso de caché de una función, será la suma del uso de memoria de dicha función en todos los hilos de ejecución.

Debido a esta característica, interpretar el archivo de salida no es una tarea sencilla, para ello Valgrind suministra una herramienta llamada *cg_annotate* que simplifica esta tarea, unificando los accesos de cada ejecución para cada función, y ofreciendo como resultado el uso total de la memoria para cada función.

Para ejecutar *cg_annotate* simplemente debemos indicar el nombre de la salida que se ha obtenido con *Cachegrind*:

```
cg_annotate fichero
```

Obteniendo en pantalla una salida como la de la figura 4.3.

```
--------------------------------------------------------
 Ir  I1mr  ILmr      Dr  D1mr  DLmr      Dw  D1mw  DLmw
--------------------------------------------------------
255,960   786   779 82,117   938   860 41,750   150   135  PROGRAM TOTALS

--------------------------------------------------------
 Ir  I1mr  ILmr      Dr  D1mr  DLmr      Dw  D1mw  DLmw  file:function
--------------------------------------------------------
123,901   631   625 42,241   924   851 16,270   133   128  ???:???
 48,999    48    48 17,900     6     6 13,101     4     0  ???:vfprintf
 45,231     8     8 10,474     0     0  5,887     1     0  ???
    :_IO_file_xsputn
```

Figura 4.3: Salida por pantalla de *cg_annotate*

Se debe tener en cuenta que *cg_annotate* ofrece la salida por pantalla, y que no dispone de ningún argumento con el que especificar una salida a un fichero externo con el que luego analizar los datos, por lo tanto es obligatorio el uso del *pipe* > para indicar un fichero de salida, es decir que se debe ejecutar la herramienta siguiendo la siguiente sintaxis:

```
cg_annotate fichero > ficherosalida
```

Además, en ocasiones resulta efectivo centrarse en un tipo de datos, pues si no se indica lo contrario *cg_annotate* ofrecerá tanto el resultado de las instrucciones como el resultado de los datos, tanto para la caché L1, como para el resto de caches (LL).

cg_annotate dispone de un argumento que nos permite mostrar únicamente los datos que necesitamos facilitando la lectura de los resultados, esto se consigue con el siguiente argumento:

`--show=XX,YY,ZZ,...`

Donde XX, YY, y ZZ corresponden con el tipo de datos que queremos mostrar. En nuestro caso como únicamente queremos estudiar el uso de la caché debido a los datos, resulta cómodo visualizar exclusivamente Dr y Dw, es decir, incluir el argumento: `--show=Dr,Dw`

Otro argumento interesante es:

`--threshold=`*`number`*

Donde *number* indica el uso mínimo que ha de tener una función para que *cg_annotate* la tenga en cuenta, de esta manera `--threshold=0` tendrá en cuenta todas las funciones con independencia del uso en el código, por defecto el valor es de 0.1 %

Por último se ha de tener en cuenta que si la función a estudio es una de las funciones principales en un módulo del código, y las funciones a las que llama son funciones secundarias que no tienen que ver con otro módulos o bloques, se suelen tener en cuenta las funciones secundarias con el propósito de obtener una idea global del consumo de la función, pues al fin y al cabo, las funciones secundarias forman parte del módulo o bloque. En el caso contrario, si la función analizada utiliza a su vez otras funciones que pertenecen a otros módulos o bloques funcionales, obtener el resultado de la función junto a sus funciones secundarias implica que parte del recurso analizado se ha empleado en otro bloque funcional, y por lo tanto, debe tenerse en consideración qué tipos de datos analizar y cómo interpretar los resultados del análisis.

4.4. Conclusiones

Valgrind ha demostrado a lo largo de los años que se ha vuelto una herramienta indispensable para detectar los problemas de memoria, o mejorar la eficiencia en la gestión de la memoria. Posee una interfaz relativamente sencilla, y su potencia queda demostrada con la enorme cantidad de usuarios y empresas que la han utilizado para mejorar el rendimiento de sus aplicaciones. Otra característica de Valgrind es la posibilidad de utilizar distintas herramientas en función de las necesidades del estudio de rendimiento que se desee, esta característica, lejos de suponer un problema, demuestra la versatilidad de las herramientas de profiling que contiene, y la mejora que supone utilizar exclusivamente aquellos recursos indispensables para realizar un análisis en el rendimiento de una aplicación determinada.

Existe una amplia comunidad de usuarios que utilizan Valgrind y que siempre están dispuestos a solucionar los problemas que pudieran surgir, y los desarrolladores están continuamente en contacto con los usuarios mediante listas de distribución, foros especializados, y blogs de desarrollo.

Capítulo 5

El entorno de análisis *PySVCVal*

5.1. Origen y necesidades

PySVCVal es el nombre del entorno de análisis desarrollado en este TFM que permite decodificar una serie de *bit streams* de manera automática, utilizando el decodificador *Open SVC Decoder*, mediante las herramientas de *profiling* de *Valgrind*, y generar un fichero en *Excel*, con las funciones deseadas.

La versión final permite elegir una serie de *bit streams* a través de un fichero XML, seleccionando las escalabilidades para cada *bit stream*, y dejarla decodificando, una vez que el entorno a terminado, se genera un Excel con los resultados que genera Valgrind, filtrando las funciones definidas en otro archivo XML.

El entorno se ha testeado con distintos *bit streams*, y a excepción de aquellos que generan un error, debido a fallos en la decodificación del *bit stream* propias del decodificador, el entorno permite configurar y realizar una sesión de *profiling* generando archivos *Excel* con los resultados de análisis.

5.2. Requisitos previos

Puesto que PySVCVal está desarrollado en Python, es necesario disponer de algunas librerías externas, además de alguna herramienta externa, creando el entorno de análisis adecuado para interactuar entre ellas. Por lo tanto, existen una serie de aplicaciones y requisitos previos mínimos, como disponer de las herramientas de *profiling*. Debido a que no todos resultan tan evidentes, a continuación se detallan los requisitos y aplicaciones que necesita el entorno, y siempre que su licencia lo permita, se acompañan junto a este TFM en el CDROM adjunto.

A continuación se detallan los requisitos y aplicaciones que necesita el entorno:

- ***Sistema Operativo Linux*:** El entorno se ha diseñado bajo el sistema operativo *Ubuntu 12.04*, aunque cualquier otra versión de *Linux* no debería dar problemas, siempre y cuando sea capaz de ejecutar la versión de Python adecuada.
- ***Python 2.7.3*:** Para el desarrollo del entorno se ha usado el lenguaje Python en su version 2.7.3, y aunque no se ha probado con versiones inferiores, los desarrolladores de las librerías aseguran que el código podría ejecutarse a partir de la versión 2.6.
- ***Librería xlsxwriter*:** Para generar la salida en Excel, se necesita en Python una librería externa que permita crearlas, en este caso se trata de xlsxwriter, ya que el resto de librerías utilizadas en el entorno se encuentran en el lenguaje.
- ***BitStreamExtractorStatic.exe*:** Es una herramienta que pertenece al conjunto de aplicaciones de referencia en el estándar SVC y desarrolladas bajo el nombre de *JSVM Software* por el JVT. Está escrito en C++ y normalmente se distribuye el código fuente de las aplicaciones a través del sistema de control de versiones CVS (*Concurrent Versions System*), por lo que se debe compilar en función de la plataforma. Se puede generar un binario para plataformas, e incluye las instrucciones si se desea compilar en *Linux* con *gcc compiler*.
- ***XML Viewer*, *XML Notepad* o similar:** El entorno de análisis utiliza archivos en formato XML, en el que configurar el número de *bit streams* o las funciones que se desean analizar, y aunque los archivos XML se pueden editar mediante un editor de texto plano, en ocasiones es preferible utilizar este tipo de herramientas que facilitan la gestión de los archivos.
- ***Open SVC Decoder 1.12*, en modo debug:** El decodificador utilizado es Open Svc Decoder, y es necesario que se encuentre en el directorio donde se encuentra la herramienta, con el nombre `svcd`, y que esté compilado en modo *debug*.
- ***Valgrind 3.8.1*:** El entorno ha sido testeado con la version 3.8.1 de Valgrind, aunque se pueden utilizar versiones posteriores, siempre y cuando la herramienta *Cachegrind* no haya sufrido modificaciones en los archivos de salida.
- **Permisos de escritura:** Además del suficiente espacio en disco duro para realizar las sesiones de *profiling*, el entorno de análisis utiliza el directorio en donde se ejecuta para crear archivos de configuración y de resultados, por lo que el entorno debe tener los permisos adecuados que le permitan crear y borrar archivos en el directorio donde se encuentra instalado.

5.3. Funciones del entorno

En este apartado del capítulo se detallan las operaciones, o acciones de relevancia en el entorno, de forma que el usuario obtenga una visión global del funcionamiento, y adquiera la destreza necesaria para aprovecharlo de forma eficiente.

5.3.1. Inicio del entorno

Cuando se inicia el entorno, se realizan una serie de tareas iniciales que permiten establecer si el entorno puede iniciarse correctamente, como la existencia de *BitStreamExtractorStatic.exe*, el archivo *bitstreams.xml* y el archivo *functions.xml*.

En primer lugar se inicializan las variables generales del entorno, que corresponden con los directorios de las aplicaciones que utiliza, y en segundo lugar verifica que existan los archivos XML.

Si alguno de los pasos anteriores falla, el entorno informará del error con el mensaje del error asociado, mientras que si todo ha ido correctamente, el entorno procederá a realizar la decodificación de los archivos indicados en el archivo *bitstreams.xml*.

5.3.2. Configurar el entorno

Existen dos archivos básicos que intervienen en la configuración del entorno, *bitstreams.xml* y *functions.xml*. En el primero se indica el listado de los *bit streams* que se desean decodificar,mientras que el segundo permite gestionar los bloques funcionales que se deben buscan en las tareas de *profiling*, ambos se deben consultar o modificar manualmente, aunque para facilitar el trabajo, se ha creado una herramienta externa (*PyGenBitSVC*), que genera el archivo *bitstreams.xml* en función de los *bit streams* añadidos en un directorio del entorno *PySVCVal*.

5.3.3. Añadir y eliminar un *bit stream*

El entorno permite que el usuario defina una lista de *bit streams* a los que se desea realizar las tareas de *profiling*, esta lista de *bit streams* se encuentra en el archivo denominado *bitstreams.xml*.

La sintaxis usada para definir un *bit stream* es el siguiente:

```
<Bitstream nombre="archivo" D="X" T="Y" Q="Z" />
```

Donde *archivo* corresponde con el nombre del fichero que contiene el *bit stream*, X corresponde con el nivel de escalabilidad espacial seleccionado en la decodificación, Y con el nivel de escalabilidad temporal y Z con el nivel de escalabilidad temporal.

PySVCVal, comprobará que las escalabilidades seleccionadas pueden usarse para decodificar el *bit stream*, mediante la herramienta *BitStreamExtractorStatic.exe*.

En el caso de que las escalabilidades seleccionadas no pudieran usarse para un determinado *bit stream*, se obviará dicho *bit stream* y se continuará con el siguiente.

Importante: **Todos los *bit streams* deben estar en un directorio llamado 264 dentro del directorio en el que se encuentre el entorno.**

5.3.4. Inicio de una sesión de *profiling*

En este apartado se describen los pasos generales y los procesos que sigue el entorno para iniciar una sesión de *profiling* entendiendo que se han adquirido los conocimientos previos en cuanto a los requisitos y la configuración necesaria del entorno.

En primer lugar el entorno comprueba que existen todos los archivos necesarios, es decir, comprueba que existe el archivo *BitStreamExtractorStatic.exe*, el archivo *bitstreams.xml* y el archivo *functions.xml*

Una vez realizada esta comprobación, se procesa el primer *bit stream* con las escalabilidades seleccionadas, ejecutando Open SVC Decoder mediante *Valgrind*.

Finalizada la decodificación, el entorno dejará un archivo de salida con el nombre fichero_X_Y_Z.txt con el resultado de la sesión, donde X,Y,Z corresponden con las escalabilidades seleccionadas para ese *bit stream*.

Este archivo de salida se procesa con *cg_annotate*, y el resulltado se analiza con el entorno buscando las funciones definidas en *functions.xml*.

En la figura 5.1 se incluye un diagrama de flujo con los pasos básicos del entorno.

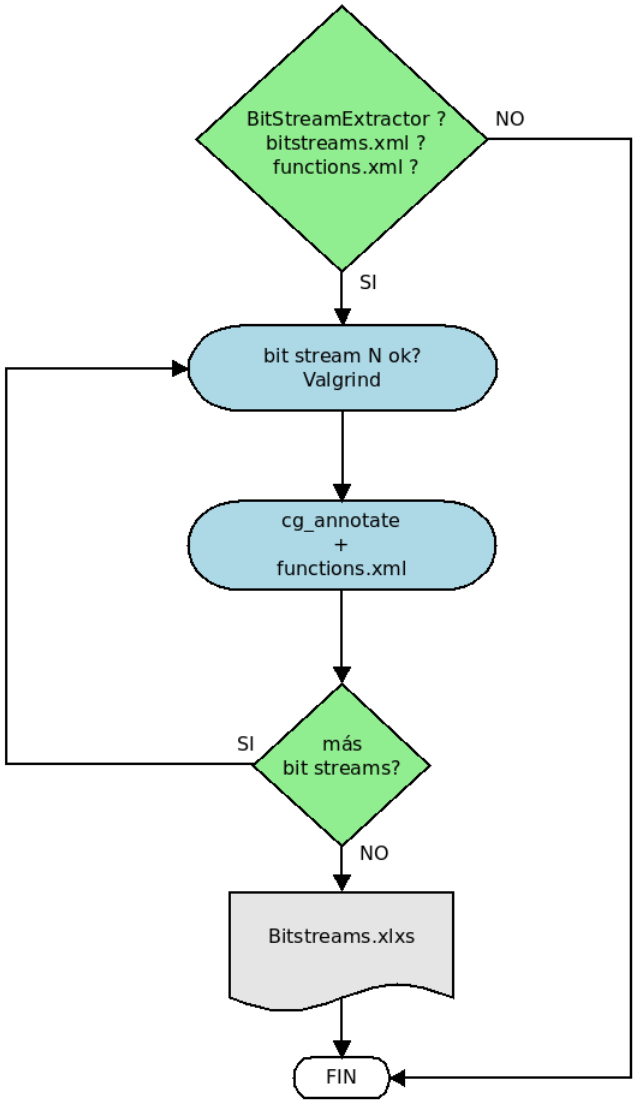


Figura 5.1: Diagrama de flujo del entorno

5.4. Los archivos del entorno

A continuación se incluye un resumen de los dos archivos principales, explicando los aspectos más relevantes que se deben tener en cuenta cuando se ejecuta el entorno.

5.4.1. El archivo *bitstreams.xml*

Tal como ya se ha comentado, en *bitstreams.xml* se incluirán todos los *bit streams* que se deseen analizar, indicando las escalabilidades seleccionadas para cada *bit stream* , en la figura 5.2 se incluye un ejemplo. Este archivo debe encontrarse en el mismo directorio en el que se ejecuta el entorno.

El entorno comprobará si las escalabilidades seleccionadas para cada *bit stream* están dentro de los rangos permitidos para ese *bit stream*, y aquellas escalabilidades que excedan

```
<?xml version="1.0" encoding="utf-8"?>
<Bitstreams>
<Bitstream nombre="SVCHMTS-2.264" D="1" T="0" Q="0" />
<Bitstream nombre="video_4.264" D="0" T="4" Q="1" />
<Bitstream nombre="SVCBMT-7.264" D="0" T="2" Q="1" />
<Bitstream nombre="news2D_2T_2Q_512_20_50_100.264" D="1" T="1" Q="1"
    />
<Bitstream nombre="stefan2D_2T_2Q_512_20_50_100.264" D="1" T="1" Q="
   1" />
<Bitstream nombre="SVCBCTS-1.264" D="2" T="2" Q="0" />
<Bitstream nombre="SVCBST-14.264" D="2" T="2" Q="0" />
<Bitstream nombre="SVCBMST-3.264" D="1" T="4" Q="0" />
<Bitstream nombre="video_5.264" D="1" T="4" Q="0" />
<Bitstream nombre="SVCHMTS-1.264" D="1" T="2" Q="1" />
<Bitstream nombre="SVCBS-4.264" D="1" T="0" Q="0" />
<Bitstream nombre="video_3.264" D="1" T="4" Q="0" />
<Bitstream nombre="SVCBM-2.264" D="0" T="0" Q="1" />
<Bitstream nombre="PARK_I-3D-3T-3Q.264" D="2" T="2" Q="1" />
<Bitstream nombre="SVCBS-8.264" D="1" T="4" Q="0" />
<Bitstream nombre="SVCBS-7.264" D="1" T="0" Q="0" />
<Bitstream nombre="SVCBCTS-3.264" D="2" T="0" Q="0" />
<Bitstream nombre="SVCBMT-6.264" D="0" T="2" Q="1" />
<Bitstream nombre="SVCBCT-1.264" D="1" T="4" Q="0" />
<Bitstream nombre="video_2.264" D="2" T="4" Q="0" />
<Bitstream nombre="SVCBST-13.264" D="1" T="2" Q="0" />
<Bitstream nombre="BUS_2D-2T-2Q.264" D="1" T="1" Q="1" />
<Bitstream nombre="SVCBCTS-2.264" D="2" T="2" Q="0" />
<Bitstream nombre="SVCBST-20.264" D="1" T="1" Q="0" />
<Bitstream nombre="coastguard2D_2T_2Q_512_20_50_100.264" D="1" T="1"
    Q="1" />
<Bitstream nombre="SVCBS-3.264" D="1" T="0" Q="0" />
</Bitstreams>
```

Figura 5.2: Ejemplo del archivo *bitstreams.xml*

los niveles máximos, serán ignorados.

5.4.2. El archivo *functions.xml*

En el archivo *functions.xml* se definen los bloques funcionales, y las funciones que pertenecen a cada bloque funcional, de manera que el entorno tenga en cuenta el uso de memoria de cada función, y se obtenga el total para cada bloque funcional.

El archivo debe seguir la siguiente sintaxis:

```
<Block Name="bloque_funcional" fn="nombre_funcion"..... />
```

Donde cada bloque funcional es un elemento del archivo XML, el atributo *Name* contendrá el nombre del bloque funcional que luego aparecerá en la hoja de cálculo, y en *fn* se

indicarán las funciones que pertenecen a cada bloque funcional.

En la figura 5.3 se incluye un ejemplo de un archivo *functions.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<Blocks>
<Block Name="IQ_IT" f0="ict_4x4_residual_C" f1="rescale_4x4_dc_chr"
   f2="rescale_4x4_dc_lum"
 f3="rescale_4x4_dc_residual" f4="fill_caches_motion_vector_B"
 f5="fill_caches_motion_vector_B_full_ref" f6="get_base_B_mv" f7="
    ict_8x8"
 f8="rescale_8x8_residual" />
  <Block Name="IL_MOTION_PREDICTION" f0="sample_interpolation" f1="
     SampleInterpolation8x8"
 f2="write_back_motion_C" f3="write_back_motion_cache_B_full_ref_C"
    />
  <Block Name="DEBLOCKING_FILTER" f0="filter_mb_svc" f1="
     Loop_filter_avc"
   f2="GetBoudaryStrenght_C" f3="HorizontalEdgeFilter" f4="
      VerticalEdgeFilter" />
  <Block Name="IL_DEBLOCKING_FILTER" f0="filter_mb" f1="filter_mb_B"
      />
  <Block Name="CAVLD" f0="residual" f1="ComputeCurrentCoeff" f2="
     decode_cabac_mb_cbp_luma"
 f3="mb_cabac_P_partionning" f4="P_cabac" f5="residual_block_cabac"
    />
  <Block Name="RESIDUAL_UPSAMPLING" f0="Upsample_residu" />
  <Block Name="INTRA_UPSAMPLING" f0="upsample_mb_luninance" f1="
     upsample_mb_chroma" />
</Blocks>
```

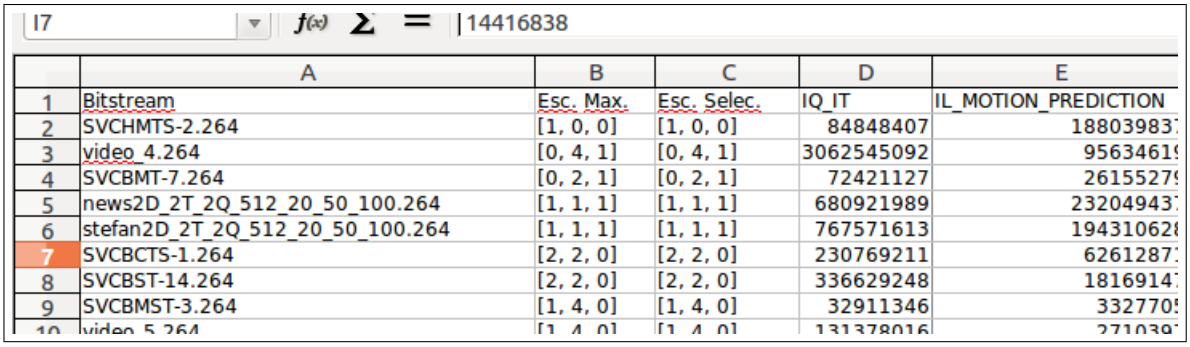
Figura 5.3: Ejemplo del archivo *functions.xml*

La funciones deben declararse tal y como se encuentran en el código respetando mayúsculas y minúsculas, de esta manera se evitan problemas en la identificación con las funciones que tengan nombres y que contengan el nombre de la función que se desea analizar, pero que tenga objetivos distintos.

5.4.3. El archivo de salida *Bitstreams.xlxs*

Los resultados de las sesiones de *profiling* se obtienen como salida en un archivo *xlxs* llamado *Bitstreams.xlxs*. El entorno **borrará cualquier archivo que se encuentre en el directorio y tenga el mismo nombre**, por lo que borrará los resultados previos que se hayan generado con el entorno.

A continuación, en la figura 5.4 se incluye un ejemplo:



I7 | 14416838

	A	B	C	D	E
1	Bitstream	Esc. Max.	Esc. Selec.	IQ_IT	IL_MOTION_PREDICTION
2	SVCHMTS-2.264	[1, 0, 0]	[1, 0, 0]	84848407	18803983
3	video_4.264	[0, 4, 1]	[0, 4, 1]	3062545092	9563461
4	SVCBMT-7.264	[0, 2, 1]	[0, 2, 1]	72421127	2615527
5	news2D_2T_2Q_512_20_50_100.264	[1, 1, 1]	[1, 1, 1]	680921989	23204943
6	stefan2D_2T_2Q_512_20_50_100.264	[1, 1, 1]	[1, 1, 1]	767571613	19431062
7	SVCBCTS-1.264	[2, 2, 0]	[2, 2, 0]	230769211	6261287
8	SVCBST-14.264	[2, 2, 0]	[2, 2, 0]	336629248	1816914
9	SVCBMST-3.264	[1, 4, 0]	[1, 4, 0]	32911346	332770
10	video_5.264	[1, 4, 0]	[1, 4, 0]	131378016	271039

Figura 5.4: Ejemplo del archivo *Bitstreams.xlxs*

5.5. El *script PyGenBitSVC*

Con el objetivo de facilitar el tratamiento de los *bit streams*, se ha desarrollado un *script* para generar de manera automática el archivo *bitstreams.xml*.

Soporta tres parámetros, **-BL**, **-D**, **-T** y **-Q**

Si no se especifica ningún parámetro *PyGenBitSVC* incluirá en el archivo *bitstreams.xml* todos los *bit streams* que se encuentren en el directorio 264 dentro del directorio donde se encuentra el *script*, utilizando las máximas escalabilidades para cada *bit stream*.

Con el argumento **-BL** el *script* incluirá todos los *bit streams* que se encuentren en el directorio 264, seleccionando como opciones decodificar sólo la capa base para cada *bit stream*, es decir, los niveles para cada una de las escalabilidades será cero.

Con el argumento **-D**, el *script* seleccionará la máxima escalabilidad espacial para todos los *bit streams*, seleccionando cero para el resto de las escalabilidades.

El argumento **-T** incluirá todos los *bit streams* que se encuentren en el directorio 264, seleccionando la máxima escalabilidad temporal para cada *bit stream*, dejando el resto a cero.

Por último el argumento **-Q**, creara un *bitstreams.xml*, seleccionando la máxima escalabilidad de calidad para cada *bit stream*, dejando el resto de escalabilidades a cero.

5.6. Observaciones

Existen dos aspectos importantes que conviene señalar en el funcionamiento del entorno.

En primer lugar el tiempo empleado en obtener los resultados puede resultar elevado,

pues además de utilizar el decodificador compilado en modo *debug* y con el argumento *O0*, el decodificador se ejecuta en conjunto con Valgrind, que también ralentiza la labor de decodificación. Como ejemplo, un *bit stream* de unos 15 segundos tardará aproximadamente unos 20 minutos en decodificarse, utilizando *Valgrind* y el compilador en modo *debug*.

Otro aspecto importante esta relacionado con los fallos que se puedan producir con la decodificación de *bit streams* erróneos o que puedan contener fallos, en este caso es necesario que el usuario cancele la decodificación, pues no existen métodos para detectar si un *bit stream* es erróneo, o no, por lo tanto se aconseja decodificar primero aquellos *bit streams* que se deseen analizar, con el objetivo de evitar este tipo de fallos. Actualmente se está trabajando en la posibilidad de incluir algún tipo contador temporal que permita detectar este tipo de fallos, y cancelar automáticamente la decodificación del *bit stream* defectuoso.

5.7. Conclusiones

El desarrollo de este entorno facilitará las labores de investigación del Open SVC Decoder de manera significativa, el tiempo que se emplea en decodificar un *bit stream* variando las opciones de decodificación, recopilar los datos de salida de Valgrind, y trasladarlos a una hoja de cálculo para su gestión, no es una tarea trivial. Por lo tanto el desarrollo de un entorno que permita llevar a cabo esta labor de manera automática, facilitará enormemente el trabajo de análisis del código, con el objetivo final de estudiar las transacciones y establecer una arquitectura que pueda implementar el decodificador de manera eficiente.

Una de las ventajas principales del entorno es que no se limitan ni los bloques funcionales, ni las funciones que pertenecen a cada bloque. Esta característica permite especificar otros bloques funcionales distintos, cambiando el tipo de estudio, por ejemplo, si se deseara realizar un estudio específico y detallado de un bloque en particular, o si se precisara el estudio de una función en concreto, en ese caso bastaría con renombrar el bloque deseado e incluir las funciones que se deseen analizar en el archivo de funciones (*functions.xml*). Este tipo de características, también permite no establecer en un primer estudio las funciones, o modificarlas por si se han encontrado errores en su definición.

Capítulo 6

Resultados, Conclusiones, y líneas futuras

6.1. Introducción

Para finalizar, se exponen los resultados obtenidos tras el desarrollo del presente Trabajo Fin de Máster en relación con los objetivos planteados, así como las conclusiones. Asimismo, se proponen mejoras del entorno y diversas líneas de investigación para futuros trabajos.

6.2. Resultados

6.2.1. Pruebas con Valgrind

El objetivo principal de este TFM, es el de investigar si una herramienta como *Valgrind*, basada en arquitecturas x86, puede servir para realizar un estudio del rendimiento en un sistema empotrado, en concreto, lo que se buscaba era poder estudiar las transacciones de datos entre las funciones más importantes que pertenecen a cada uno de los bloque funcionales del Open SVC Decoder.

Para probar la relación que existe entre los datos aportados por *Valgrind*, y las exigencias de datos de las funciones, es conveniente realizar antes unas pequeñas pruebas con algún código sencillo en C, que pueda dar una idea del comportamiento de *Valgrind* a medida que aumentamos los requisitos de las funciones en el código, y estudiar dicha relación.

Con este propósito se ejecutó en *Valgrind* el código que se muestra en la figura 6.1, en el que se definen tres funciones, una que realiza con una operación sencilla con un matriz de i elementos, otra función que llama a la primera función, y una última a la que se le pasa el puntero de la matriz, en lugar de hacer la operación i veces desde *main*.

```
#include <stdio.h>
#include <stdlib.h>

int funcion1(int dato_entrada){

        int salida;
        salida = dato_entrada + 1;
        return salida;
}

int funcion2(int dato_entrada){
        int salida;
        salida = funcion1(dato_entrada) + 3;
        return salida;
}

void funcion3(int *matriz){
        int i;
        for (i=0; i<100; i++) matriz[i]=matriz[i]+1;
}
int main(void){
        int i;
        int *puntero;
        int entrada[100];
        \hspace{14cm}
        //asignamos valores al vector entrada

        for (i=0; i<100; i++) entrada[i]=i;

        // llamamos a funcion2 que llama a funcion1

        for (i=0; i<100; i++) entrada[i] = funcion2(entrada[i]);

        //llamamos a funcion3 pasando puntero

        funcion3(entrada);

        for (i=0; i<100; i++) printf ("Entrada[%d]=%d\n",i,entrada[i
           ]);

        return 0;
}
```

Figura 6.1: Código de pruebas en C para Valgrind

Para comprobar los resultados de Valgrind se fue aumentando i desde 100 hasta 1000000, y obteniendo los valores **Dr** y **Dw** (lecturas y escrituras en caché) de cada función, como se puede observar en la figura 6.2:

Funcion/i	100	1000	10000	100000	1000000
Funcion1	400	4000	40000	400000	4000000
Funcion2	400	4000	40000	400000	4000000
Funcion3	703	7003	70003	700003	7000003

Figura 6.2: Resultados de Valgrind para código de pruebas (lecturas de caché)

Con estos resultados se observa que el comportamiento de consumo de memoria, si aumentamos el número de datos que procesa cada función sigue un comportamiento lineal.

En el caso de la función 3, que trabaja con un puntero, se observa un incremento de los accesos debido a que tiene que tratar con el puntero, pero es un incremento que se mantiene constante, y que en cualquier caso no afecta en exceso al resultado final.

6.2.2. Resultados del decodificador

Para realizar las pruebas con el decodificador se seleccionaron principalmente diez secuencias de vídeo utilizadas en el ámbito de la investigación, News, Stefan, Park, Bus, City, Flower, Foreman, Mobile, Akiyo y Coastguard.

A continuación se muestran los resultados para cada bloque funcional de la decodificación, usando todas las escalabilidades para cada *bit stream*:

Bitstream	IQ_IT	IL_MOTION_PREDICTION	DEBLOCKING_FILTER
News	680921989	232049437	510247359
Stefan	767571613	194310628	462518980
Park	617768432	188283312	348263844
Bus	511658887	178118639	323786016
City	574818335	181980572	336888840
Flower	877878169	142392188	230819667
Foreman	590305062	166595084	387891008
Mobile	655657905	163718661	412926487
Akiyo	581004791	197422752	436483251
Coastguard	670978523	185790467	436547582

Figura 6.3: Tabla de resultados de acceso a memoria parte 1

Bitstream	IL_DEBLOCKING_FILTER	CAVLD	RESIDUAL_UPSAMPLING	INTRA_UPSAMPLING
News	45593479	128337095	22170055	1421821920
Stefan	40339033	124219900	20949685	1542815832
Park	30647900	111263188	18512668	1583740384
Bus	31231010	104028810	16806185	1025933975
City	30518320	107481220	17490743	1330424046
Flower	53245387	134042299	11186725	1598221367
Foreman	33363722	98084172	17532649	1204944582
Mobile	35899983	113798212	17980118	1235615833
Akiyo	39695670	110065662	18793698	1179495025
Coastguard	38839382	119093035	19810673	1316143650

Figura 6.4: Tabla de resultados de acceso a memoria parte 2

A continuación se muestran los resultados para cada bloque funcional de la decodificación, usando todas las escalabilidades para cada *bit stream*:

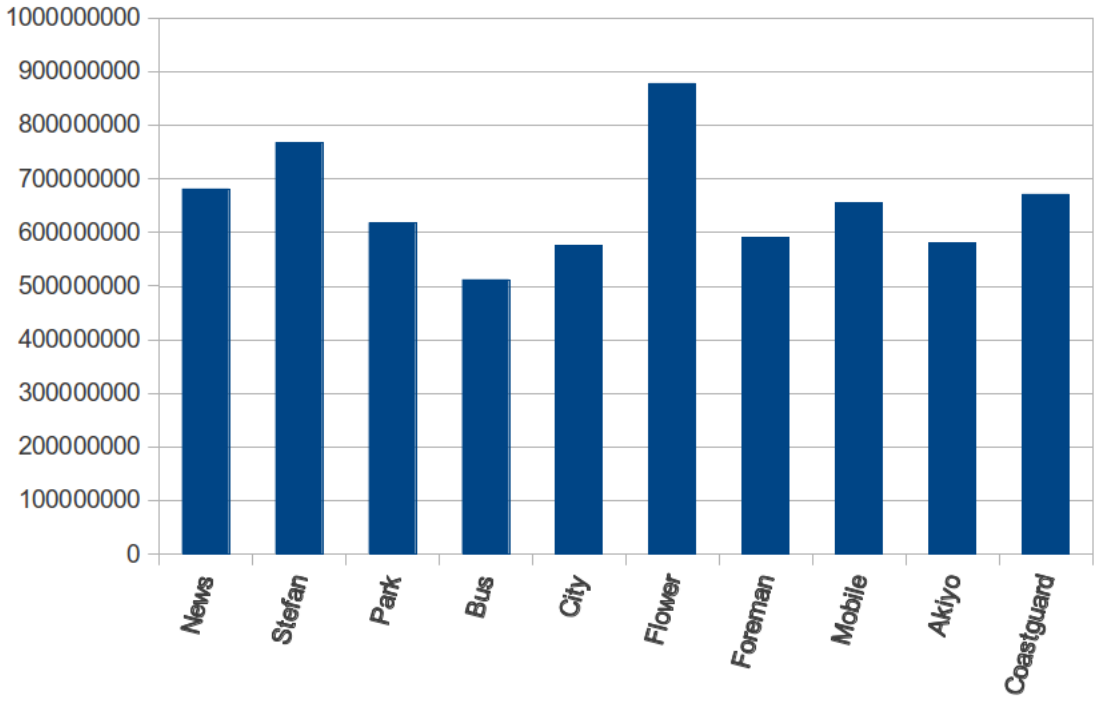


Figura 6.5: Accesos a memoria del bloque IQ/IT

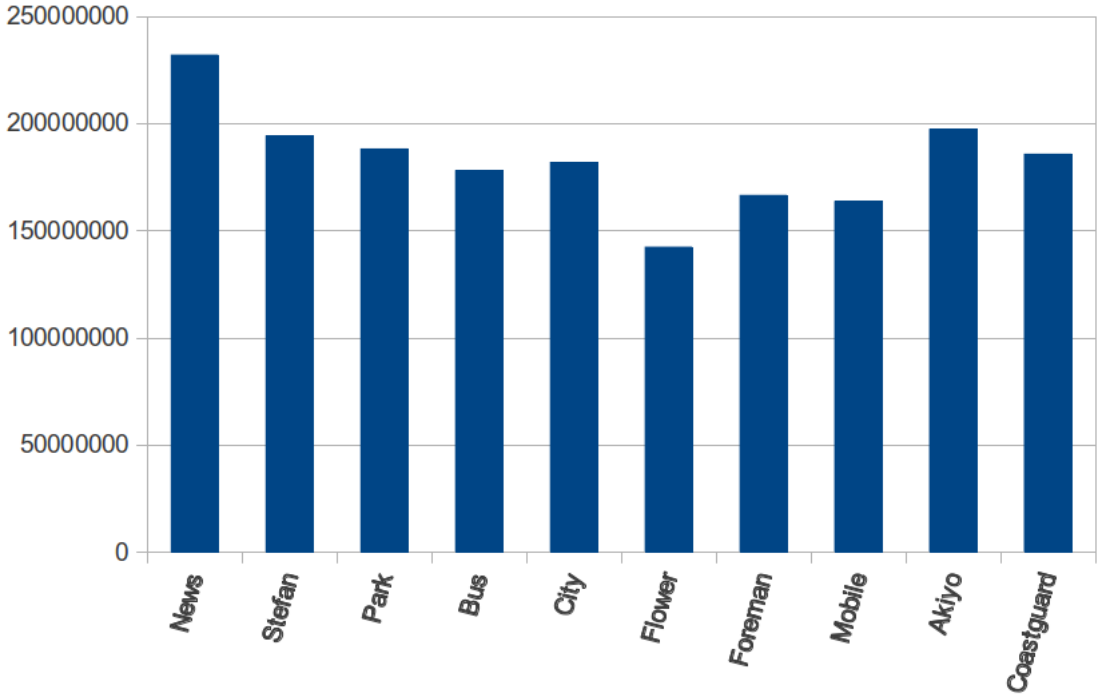


Figura 6.6: Accesos a memoria del bloque IL MOTION PREDICTION

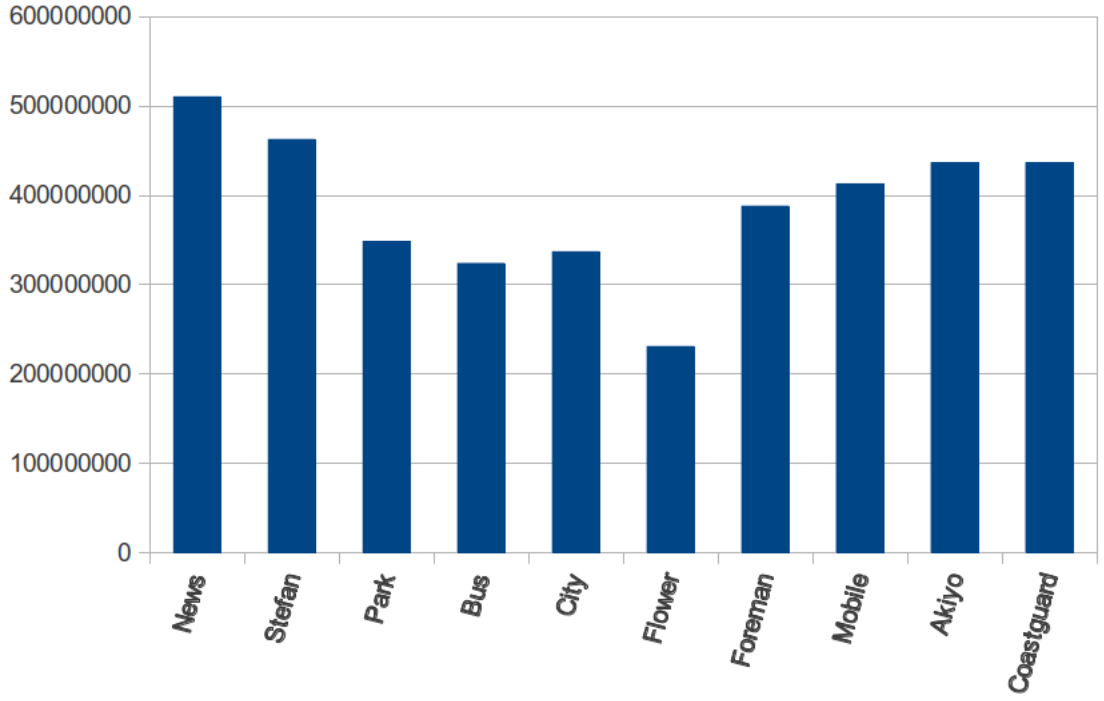


Figura 6.7: Accesos a memoria del bloque DEBLOCKING FILTER

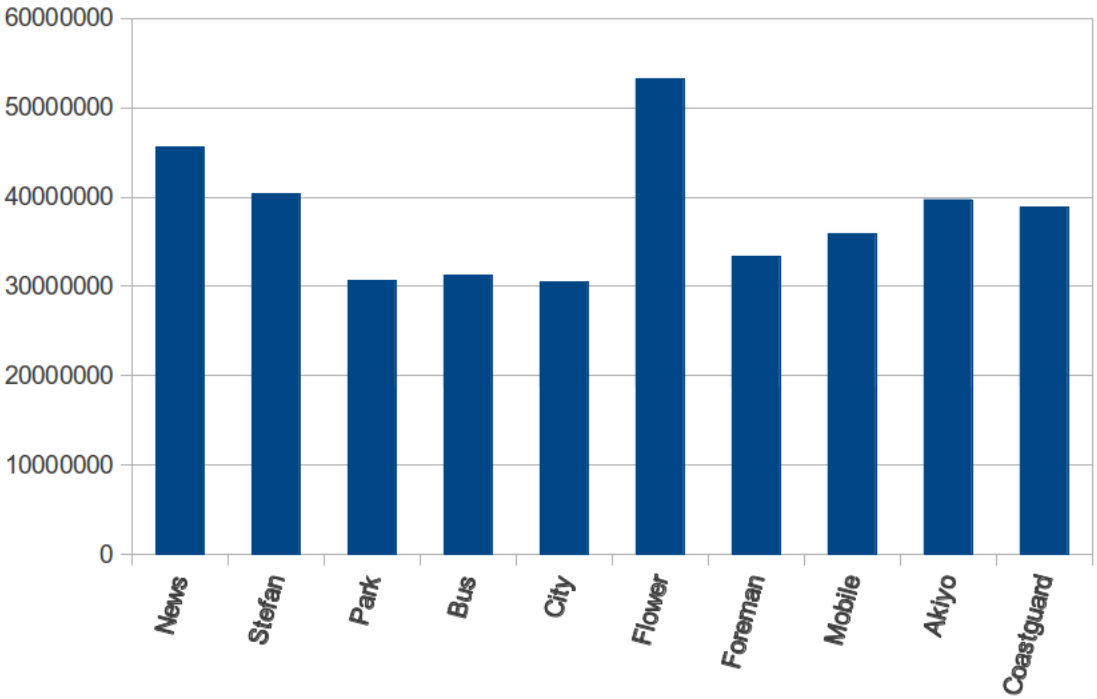


Figura 6.8: Accesos a memoria del bloque IL DEBLOCKING FILTER

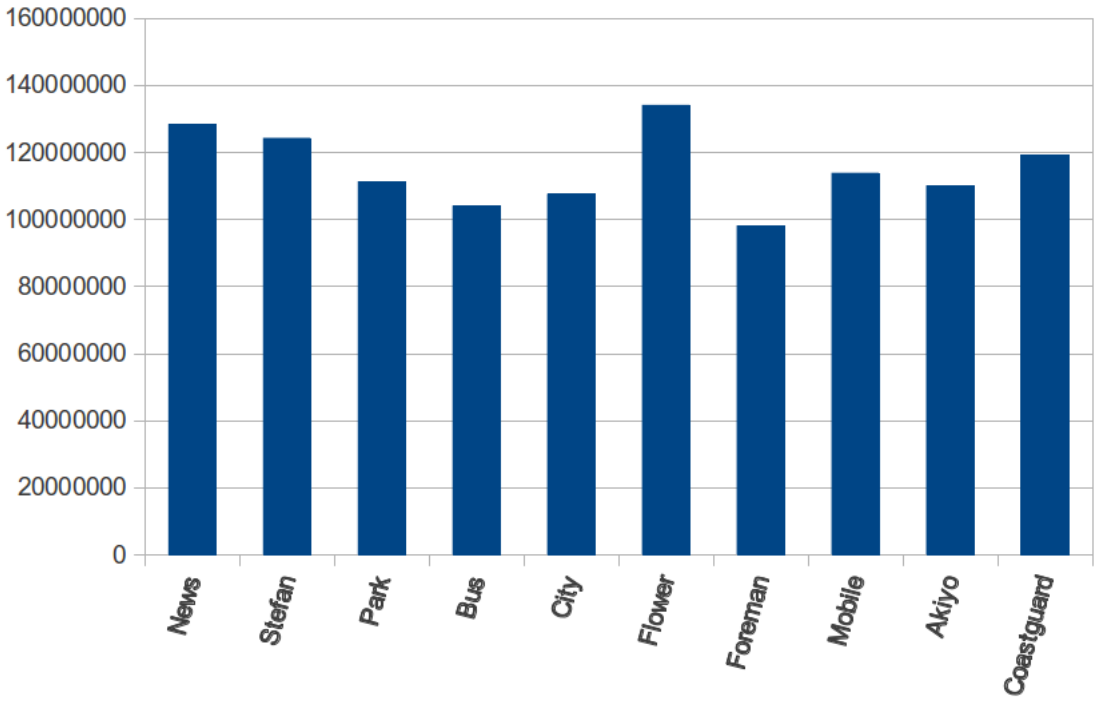


Figura 6.9: Accesos a memoria del bloque CAVLD

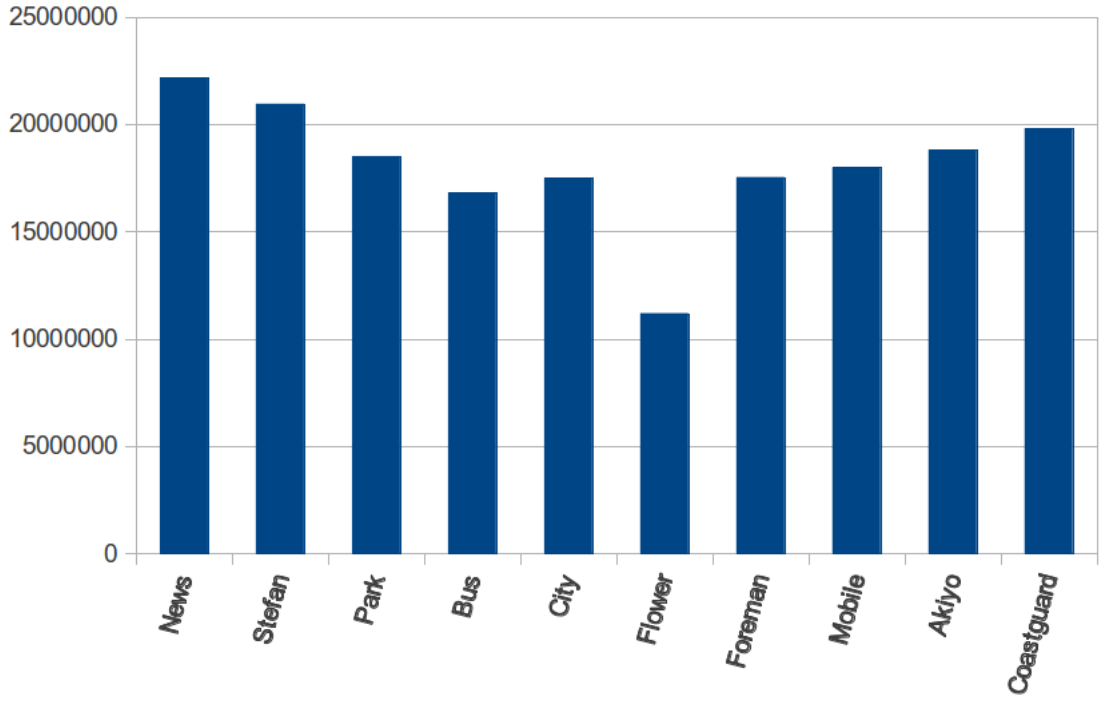


Figura 6.10: Accesos a memoria del bloque RESIDUAL UPSAMPLING

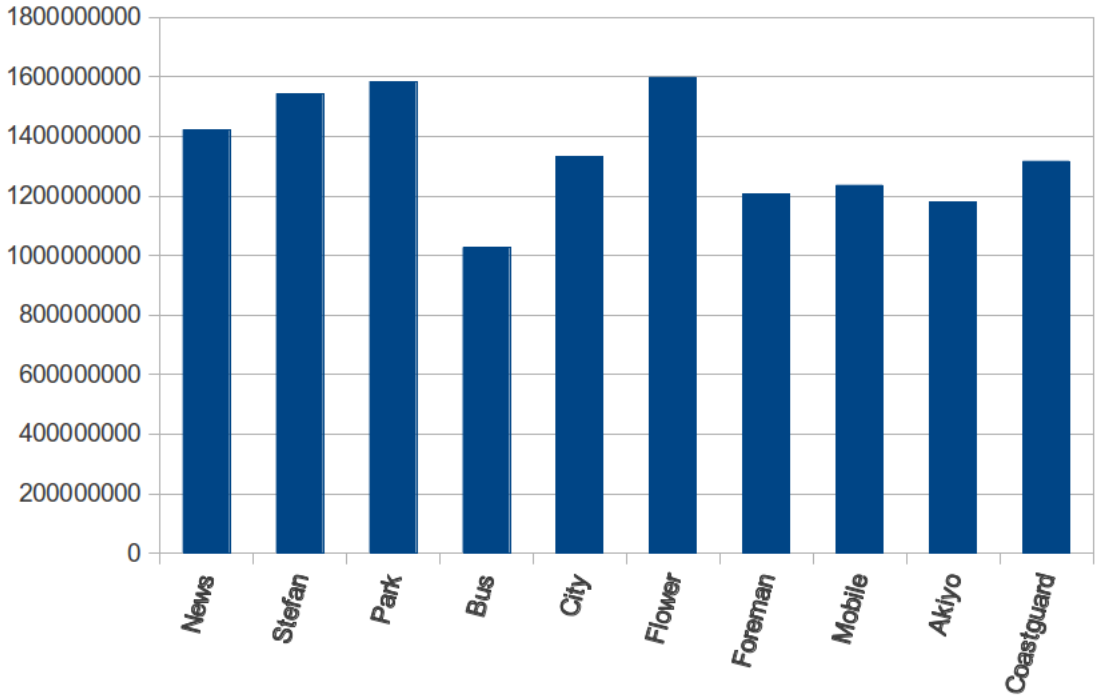


Figura 6.11: Accesos a memoria del bloque INTRA UPSAMPLING

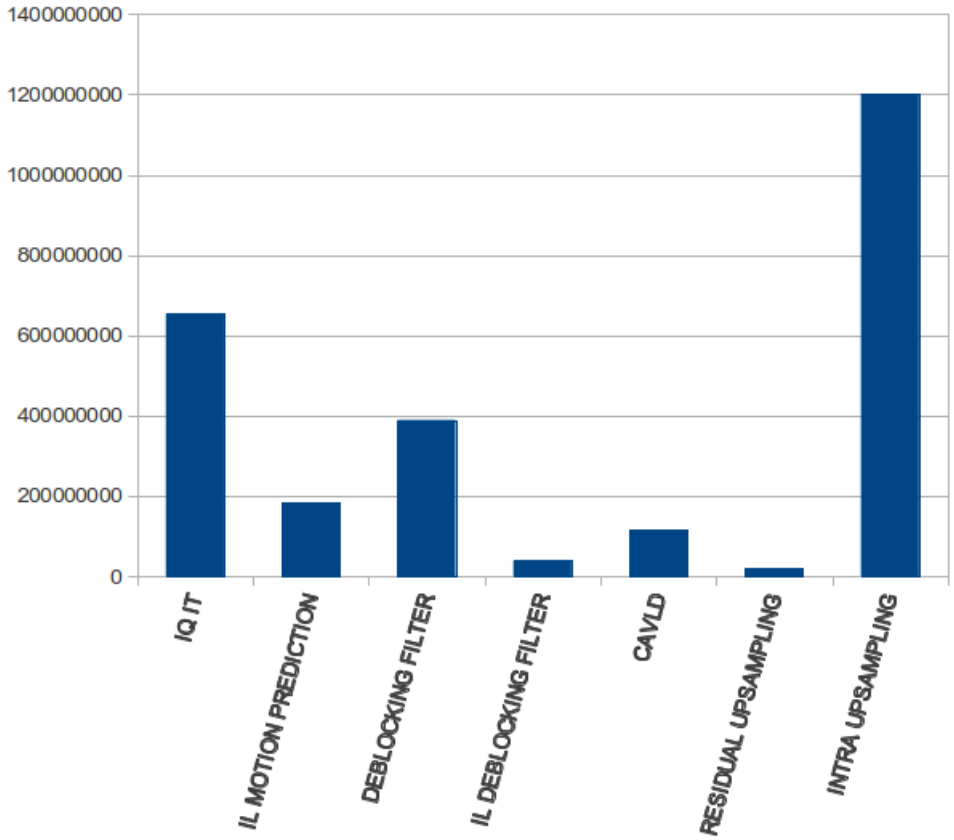


Figura 6.12: Comparativa de los accesos a memoria de cada bloque

Al contrario que con el consumo de CPU estudiado en el pasado, el DEBLOCKING FILTER no es el qu emás recursos consume, sino que tal y como se aprecia en los gráficos, existen grandes exigencias de memoria en el bloque INTRA UPSAMPLING cuando se activan todas las escalabilidades, en contraposición del uso de dicho bloque con H.264, donde no aparece.

El siguiente bloque que presenta grandes exigencias de memoria es el referente al módulo IQ/IT, seguido por el DEBLOCKING FILTER.

La secuencia *flower* contiene secuencias con altos niveles de entropía, lo que se se ve reflejado en la figura 6.8 y en la figura 6.5.

El mayor consumo de memoria se encuentra en el módulo INTRA UPSAMPLING, lo que sugiere que es un módulo que se utiliza en todos los tipos de escalabilidad, esto se confirmará también en el resto de pruebas, donde se utilizan en exclusiva algunos tipos de escalabilidad.

A continuación se incluye una comparativa decodificando sólo la capa base, y decodificando todas las capas para cada *bit stream*. También se incluye una comparación final, que da una idea de la diferencia de los costes de memoria entre H.264 y SVC.

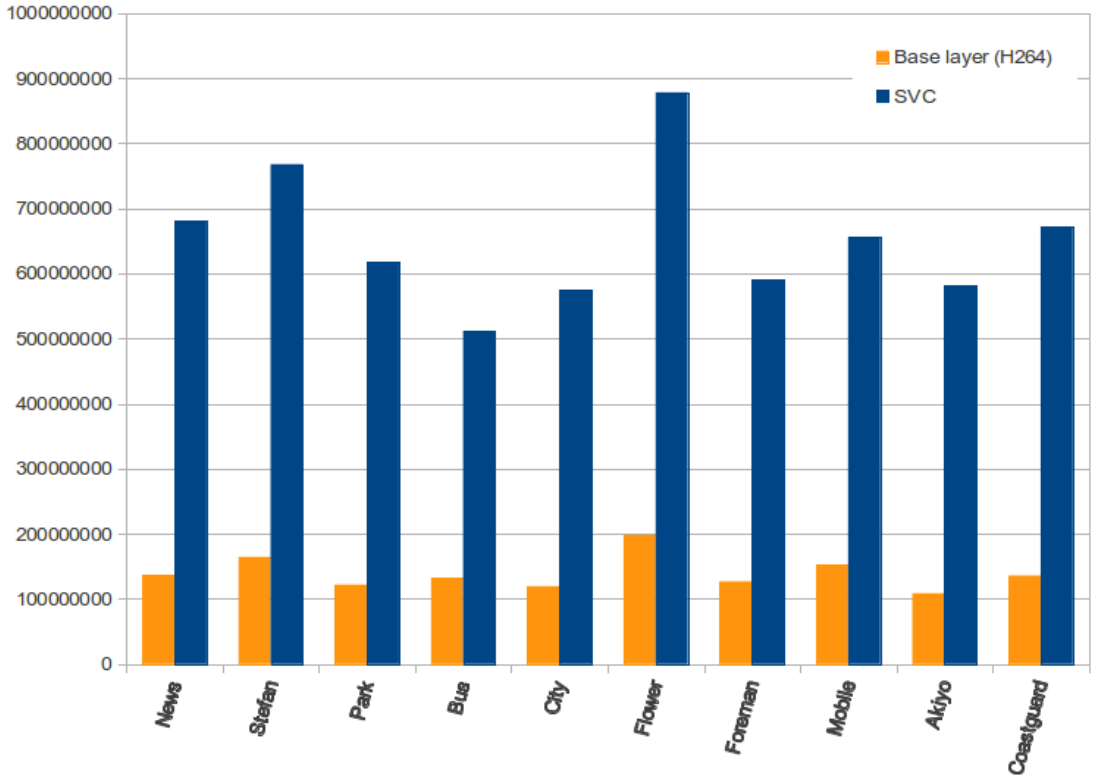


Figura 6.13: Comparativa del acceso a memoria del bloque IQ/IT

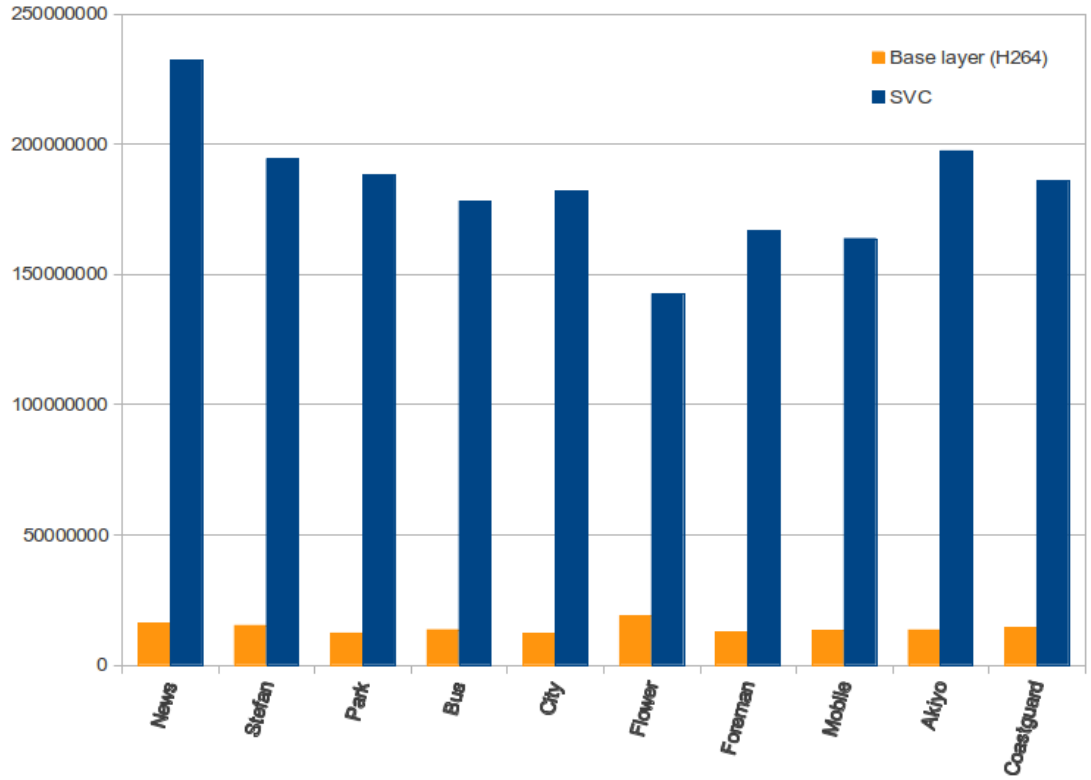


Figura 6.14: Comparativa del acceso a memoria del bloque IL MOTION PREDICTION

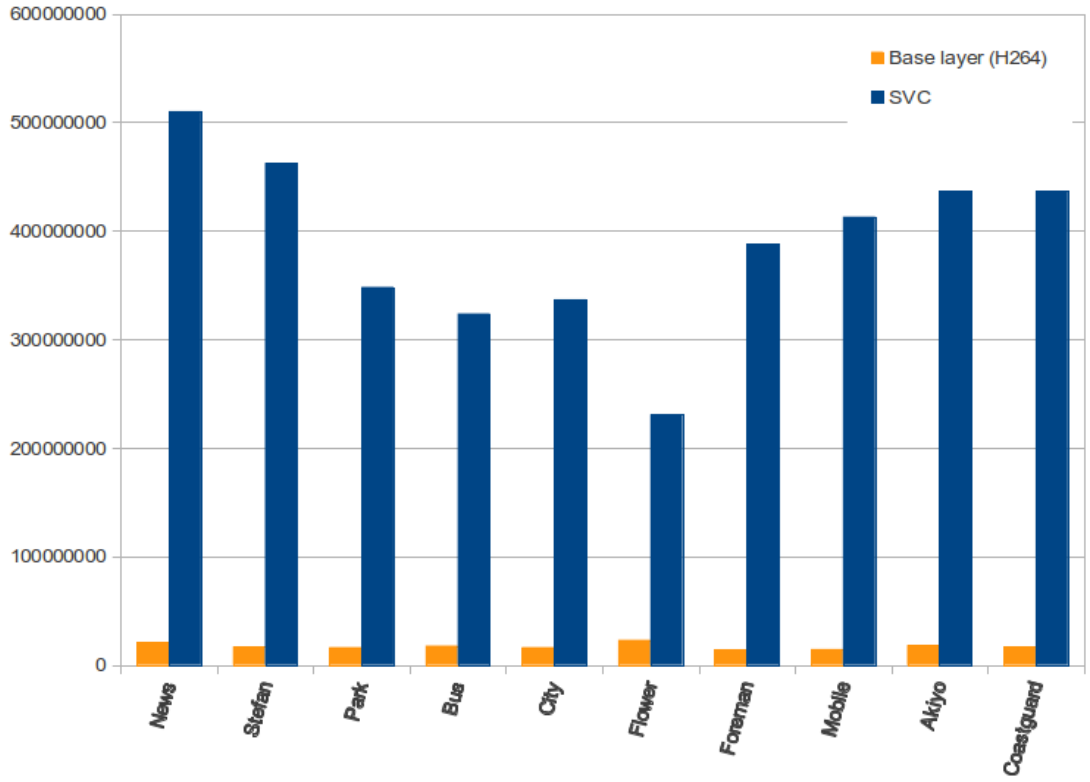


Figura 6.15: Comparativa del acceso a memoria del bloque DEBLOCKING FILTER

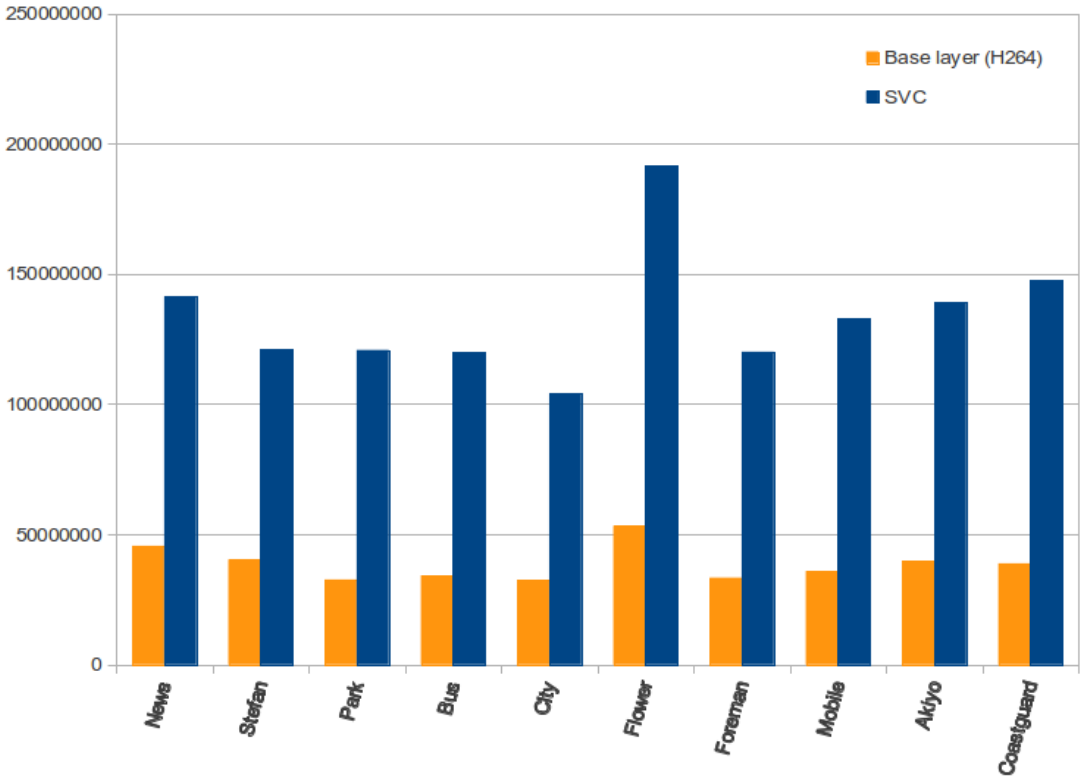


Figura 6.16: Comparativa del acceso a memoria del bloque IL DEBLOCKING FILTER

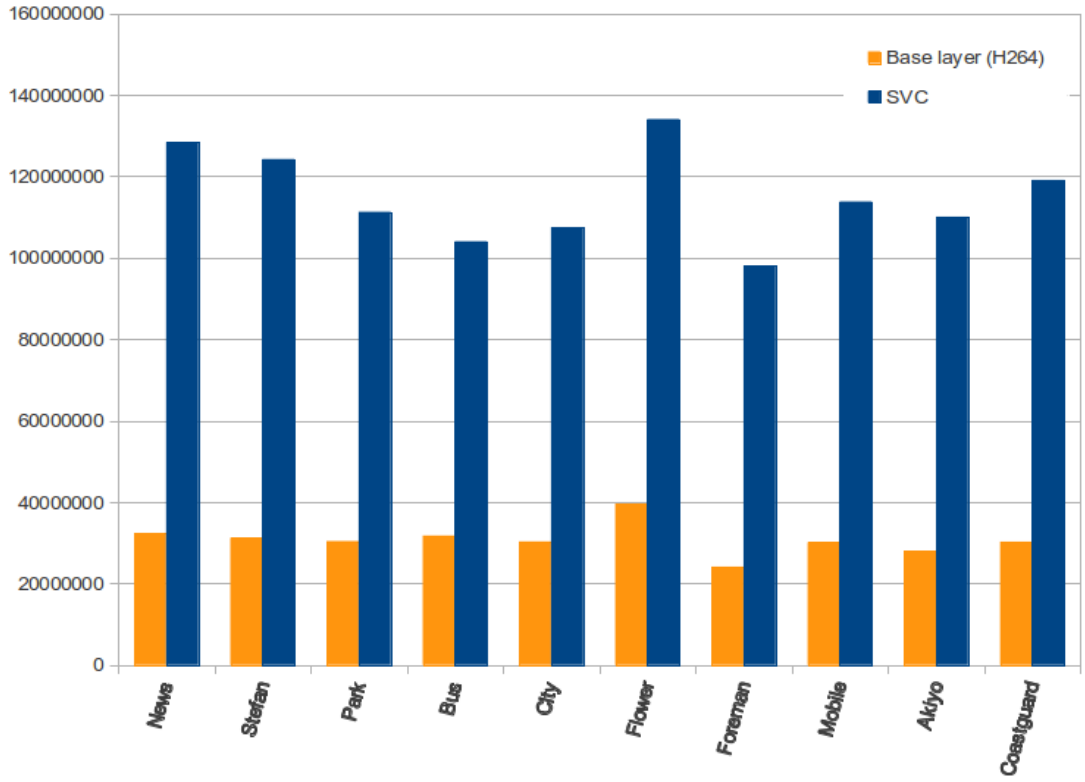


Figura 6.17: Comparativa del acceso a memoria del bloque CAVLD

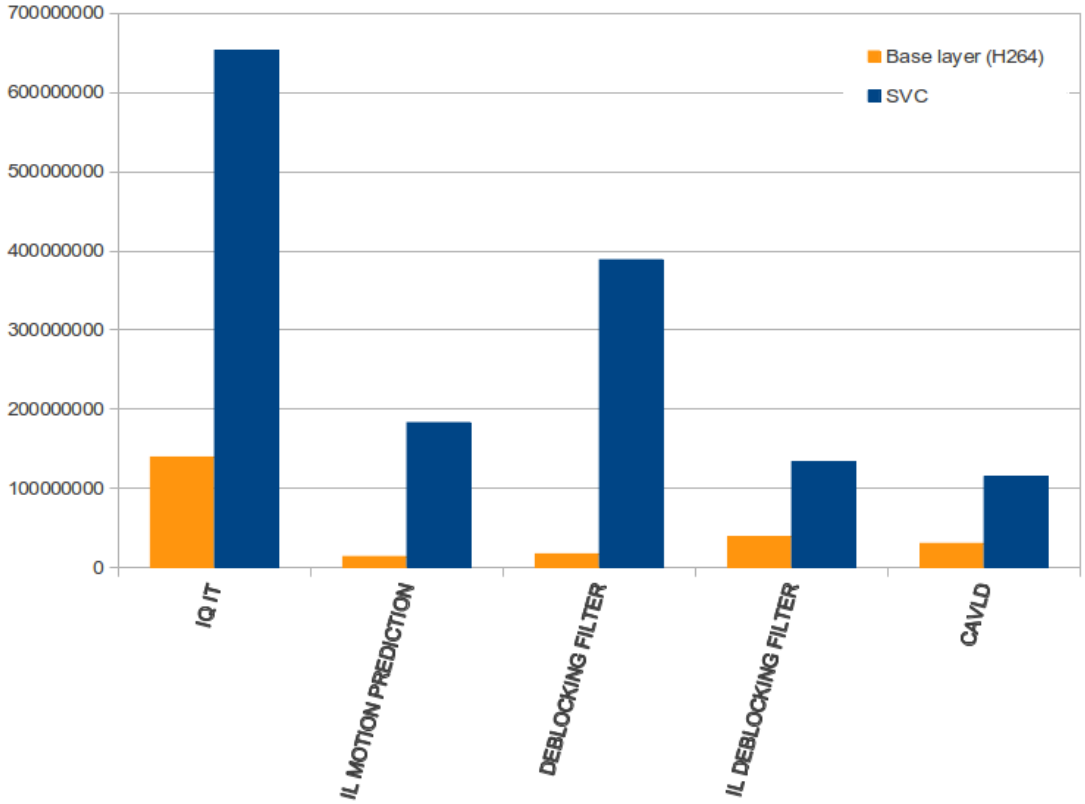


Figura 6.18: Comparativa del uso de memoria de los módulos entre H.264 y SVC

Además de comprobar la diferencia que existe en requisitos de memoria por parte de SVC, se aprecia en las comparativas que es el módulo IQ/IT el que precisa de un estudio relevante del uso de la memoria, pues es significativamente exigente en relación al resto de módulos.

Tal como se esperaba, el DEBLOCKING FILTER es otro de los módulos que deben tenerse en cuenta con las exigencias de memoria, pues, aunque ya se conocía gracias a estudios previos sus exigencias en cuanto a consumo de CPU, queda demostrado que en SVC consume gran cantidad de memoria en comparación con su consumo en H.264.

En esta comparativa faltan los módulos RESIDUAL UPSAMPLING e INTRA UPSAMPLING, ya que dichos módulos únicamente intervienen cuando se activa la escalabilidad, siendo cero en H.264.

Para finalizar se incluye una comparativa del uso de memoria por parte de los módulos principales entre los distintos tipos de escalabilidad, realizando una media de los diez *bit streams* seleccionados, lo relevante en este caso, es comparar qué tipo de escalabilidad es la más exigente.

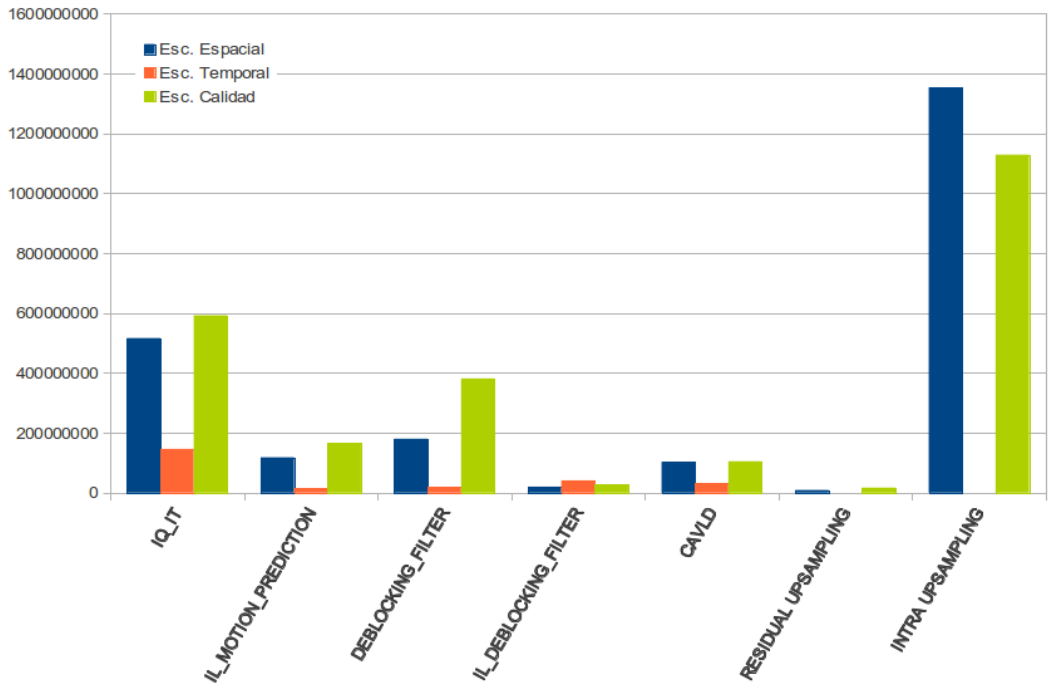


Figura 6.19: Comparativa del uso de memoria entre las distintas escalabilidades

6.3. Conclusiones

Las características finales del entorno de análisis cumplen con los objetivos fijados que pueden resumirse en los siguientes hitos:

- **Posibilidad de realizar sesiones de *profiling* en modo *batch* o procesamiento por lotes**: el entorno permite realizar sesiones de *profiling* a una lista de *bit streams* teniendo en cuenta las opciones seleccionadas por el usuario para la decodificación de cada *bit stream*, sin el control o la supervisión directa del usuario.
- **Informe de los resultados en *Excel***: Los resultados se ofrecen en formato Excel, lo que garantiza que puedan ser consultados tanto en *linux* como en *windows*.
- **Flexibilidad en la configuración de las funciones que pertenecen a los bloques funcionales del decodificador Open SVC Decoder**: el entorno ofrece la posibilidad de configurar los bloques funcionales y las funciones que pertenecen a cada bloque funcional, lo que permite no depender de un estudio inicial de las funciones del decodificador, o modificarlas según el tipo de estudio que se desee realizar del decodificador.
- **Uso de una herramienta de profiling para estudio de rendimiento en sistemas empotrados**: Se ha demostrado que existe una relación directa entre los resultados obtenidos con *Valgrind*, y el consumo que se puede esperar en un sistema empotrado por parte de las funciones.

Las tareas llevadas a cabo durante el desarrollo del presente TFM pueden resumirse en los siguientes puntos:

- **Estudio de las herramientas de *profiling***: en un estudio previo al inicio del desarrollo del entorno se adquirieron los conocimientos básicos en relación a las herramientas de *profiling*. En este sentido, se estudiaron las distintas herramientas que ofrecía *Valgrind*, y una vez seleccionada, estudiar las posibilidades de la herramienta y comprobar si el tipo de resultados era válido para extrapolar sus resultados a un sistema empotrado.
- **Conocimientos del decodificador Open SVC Decoder:** para implementar correctamente las sesiones de *profiling* sobre el decodificador Open SVC Decoder, es necesario conocer los bloques funcionales en el que se divide un decodificador SVC, y conocer qué funciones dentro del código pertenecen a cada uno de los bloques.

- **Verificación y pruebas**: el entorno se probó con distintos *bit streams* que suministró la *División de Diseño de Sistemas Integrados* (DSI) perteneciente al *Instituto Universitario de Microelectrónica Aplicada* (IUMA), lo que permitió comprobar el correcto funcionamiento del entorno, donde surgieron nuevos requisitos a cumplir, y nuevas características que se implementaron a medida que se realizaban las pruebas.

Cabe destacar que este trabajo de investigación es una continuidad de un estudio que comenzó con el Proyecto Final de Carrera, y el desarrollo de una aplicación en *Windows* bajo C#, que utilizaba las herramientas de *profiling* de *Visual Studio*, llamadas también *perfomance tools*.

Con dicha aplicación se pudo analizar el coste computacional por parte de los distintos módulos del Open SVC Decoder, y realizar un análisis de las funciones principales de cada módulo. En el Anexo A de este TFM se incluye una copia de un *paper* presentado en un congreso, con algunos resultados de la aplicación.

Esta investigación continuará y formará parte de la Tesis Doctoral, con el objetivo de analizar los resultados obtenidos y proponer una arquitectura eficiente que permita implementar el decodificador.

6.4. Líneas Futuras

A continuación se relacionan diversas líneas de mejora, desarrollo, e investigación para el futuro desarrollo de un entorno de análisis en el que se ejecuten las herramientas de *profiling*:

- **Utilizar una herramienta de *profiling* alternativa**: es interesante disponer de resultados con otras herraminetas de *profiling* con las que poder comparar los resultados generados con *Valgrind*, esto permitiría aumentar la precisión de los resultados.
- **Depurar las excepciones del decodificador**: en ocasiones se producen errores en la decodificación, en algunos casos se debe a un *bit stream* erróneo, y en otros puede deberse a fallos en el decodificador. Por lo tanto, sería deseable implementar el código necesario que detecte los errores en la decodificación, y actuar en consecuencia, por ejemplo, descartando esa decodificación y continuando con el siguiente *bit stream*, de manera que el entorno sea totalmente independiente.
- **Análisis exhaustivo del decodificador**: aunque se han estudiado y detectado las funciones principales que pertenecen a cada bloque, se hace necesaria una labor de

investigación en profundidad sobre el decodificador. Para esto se sugieren dos alternativas, una de ellas consiste en instrumentalizar el código. Otra alternativa consiste en realizar ejecuciones del código paso a paso. El objetivo final de este estudio es tener la certeza de que ninguna función relevante ha quedado sin analizar.

Bibliografía

[1] Cisco Visual Networking Index, White Paper "Hyperconnectivity and the Approaching Zettabyte Era", Cisco VNI, 2013.

[2] Vasudev Bhaskaran, Konstantinos Konstantinides, "Image and Video Compression Standars. Algorithms and Architectures, Second Edition", Kluwer Academic Publishers, 2000.

[3] ITU-T, Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, (ITU-T Rec. H.264/ISO/IEC 14 496-10 AVC), Mar. 2003.

[4] ITU-T, Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, (ITU-T Rec. H.265/ISO/IEC HEVC), Apr. 2013.

[5] Heiko Schwarz, Detlev Marpe, Member, IEEE, and Thomas Wiegand, "Overview of the Scalable Video Coding Extension of the H.264/AVC Standard", IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, VOL. 17, NO. 9, SEPTEMBER 2007.

[6] Open SVC Decoder, http://sourceforge.net/projects/opensvcdecoder/, IETR/INSA of Rennes, 2010. [Última visita: 3 de Diciembre de 2010]

[7] IETR/INSA of Rennes, http://www.insa-rennes.fr/ietr-laboratory. [Última visita: 1 de Julio de 2013]

[8] Heiko Hübert and Benno Stabernack, "Profiling-Based Hardware/Software Co-Exploration for the Design of Video Coding Architectures", IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, VOL. 19, NO. 11, NOVEMBER 2009.

[9] Heiko Hübert, "A Memory, Performance and Energy Profiler Targeting RISC-Based Embedded Systems for Data-Intensive Applications", 2009

[10] Python http://www.python.org/ [Última visita: Julio de 2013]

[11] Eclipse Kepler Release http://www.eclipse.org/. [Última visita: 10 de Junio de 2013]

[12] Allen B. Downey "Think Python"http://www.greenteapress.com/thinkpython/. [Última visita: Julio de 2013]

[13] xlsxwriter Library, http://xlsxwriter.readthedocs.org/en/latest/ Última visita: Julio de 2013]

Anexo A

Profiling Tool for the Performance Analysis of Scalable Video Decoding

Abelardo Baez Quevedo, Gustavo M. Callicó Sebastian López, José López y Roberto Sarmiento
Research Institute for Applied Microelectronics (IUMA), University of Las Palmas de Gran Canaria (ULPGC)
Campus Universitario de Tafira, Las Palmas (Spain)
{ abaez, gustavo, seblopez, lopez, roberto}@iuma.ulpgc.es

***Abstract*— Before implementing a complex video system using an electronic embedded system it is highly desirable to known in advance where the system bottlenecks are located and what is the distribution of the computational load when the options available in the video decoder change and/or the features of the input video sequences are different. This paper exposes the development of a profiling tool intended to help in the performance analysis of decoding video sequences using the H.264 Scalable Video Coding (SVC) standard. The tool incorporates a Graphical User Interface (GUI) that allows the system designer to easily interface with several advanced profiling tools and options. Using the profiling tool, the results show how the computational load changed based on the intrinsic characteristics of the video sequence and in the scalable options selected. Due to the huge number of functions that form part of the SVC video decoder, a set of modules that better describe the internal structure of the decoder has been defined. The assignation of functions to modules is open to the designer and can be changed at any time to accommodate changes in the system. Additionally, a database to better manage the profiling results has been incorporated. All these features make the created profiling environment a helpful tool for the system designer to make better decisions about the architectural load distribution, based on the modules defined inside the tool.**

Keywords — Scalable Video Coding, profiling, performance analysis.

I. INTRODUCTION

The Scalable Video Coding (SVC) is an Annex of the H.264/AVC standard [1]-[2], which extends the original standard with new tools designed to efficiently support temporal, spatial and quality (SNR) scalability developed by the Joint Video Team (JVT). The JVT is a group of experts from ITU-T Video Coding Experts Group (VCEG) and ISO/IEC Moving Pictures Experts Group (MPEG) created to develop the H.264/AVC standard.

The Open SVC Decoder (OSD) [3] is an open source library created at the IETR/INSA of Rennes that implements a SVC decoder written in C language. It has been developed and tested over different platforms: x86 processors, PDAs and DSPs, making it interesting for embedded systems.

A profiling tool could be basically defined as an analysis environment that measures the performance of a piece of code. This performance is measured in several different ways, being the more frequent ones the CPU load and the memory usage. Profiling tools give detailed information about where the bottlenecks are located in a code or which functions are using larger amounts of memory.

In this work, the Visual Studio Profiling Tools (VSPTs) have been used [11]. VSPTs let system developers measure, evaluate, and target performance-related issues in their code. These tools are fully integrated into the IDE (Integrated Development Environment) to provide a seamless and approachable user experience. The main problem encountered is that VSPT are a set of different standalone tools that require the constant attention of the developer, and they do not constitute an integrated automated environment where the designers can setup several profiling sessions.

The IAOpenSVC (IAO) is a program written and developed in C# intended to automate profiling sessions of Open SVC Decoder using the profiling tools provided by Visual Studio 2010. The IAOpenSVC tool includes a friendly GUI to manage the profiling sessions.

This paper explains the developing of IAOpenSVC and its features designed to automate the profiling process. Section II explains the basics of the Scalable Video Coding standard and the main scalability types than can be found in the Open SVC Decoder. Section III describes the main features of the profiling tools together with the basic features of Visual Studio 2010 Performance Tools (VS2010PT). Section IV describes the main features of IAOpenSVC. Section V analyses some profiling results obtained with IAOpenSVC. Finally, the conclusions are summarized in Section VI.

II. SCALABLE VIDEO CODING STANDARD AND OPEN SVC DECODER

In video coding, the term scalability means that parts of a coded video can be extracted in such a way that the resulting sub-stream forms another decodable bit-stream for the decoder, which represents the source content in a reduced reconstruction quality compared to the original bit-stream. This technique is different to simulcast, where a video content is first independently coded into different bit-streams with different resolutions or qualities and then all these bit-streams are simultaneously transmitted. Simulcast is not efficient since there is much redundancy among the bit-streams.

Scalable video coding is not a brand new research area [2]. Previous coding standards, i.e. H.262, MPEG-2 Video, H.263,

and MPEG-4 Visual already included certain degree of scalability. However, before H.264/SVC, none of the video coding standards provided full scalability and these scalable profiles of those standards have been never used extensively. The main reason for not using these scalable profiles is that these features involve a loss in coding efficiency and increase the decoder complexity as compared to simulcast.

SVC was approved in October 2007 as Amendment 3 of the Advanced Video Coding (AVC) standard ISO/IEC 14496-10 (also published as ITU-T Rec. H.264) [4]. The Moving Picture Experts Group (MPEG) Joint Video Team (JVT) in collaboration with the International Telecommunication Standardization Sector (ITU-T) developed a Scalable Video Coding (SVC) standard. As a part of the standardization process, the Joint Scalable Video Model (JSVM) [5] reference software has been developed as well.

The main goal of SVC standard [6] is to obtain a unique high-quality bit-stream that contains one or more subset bit-streams that can be decoded separately without affecting significantly the coding efficiency and reconstruction quality similar to the existing H.264/AVC with the same quantity of data. Compared with scalable profiles in previous video coding standards the coding efficiency of SVC is greatly improved [7].

The SVC bit-stream is structured in several levels or layers of information, consisting of a base layer and several enhancement layers. Each enhancement layer has a base layer, from which it reuses parameters such as motion vectors or residuals as prediction. Each SVC base layer either has its own base layer or is layer zero. For compatibility reasons, SVC provides a base layer which is totally H.264/AVC compliant. In that sense, terminal devices that only implement H.264/AVC would still be able to present the lowest quality available in the SVC stream. Scalability modes enabled by the enhancement layer information include temporal (increase of frame rate), spatial (increase of picture resolution) and fidelity (increase of quantization accuracy) scalability. These three types of scalability can be combined or used separately into a single bit-stream.

For example, with temporal scalability, several frames can be chosen for decoding and obtain from them a lower frame rate than the original one. It is possible when a sequence is divided into a Group of Pictures (GOP) to organize it into a hierarchical structure as shown in Fig. 1, where the four frame rates can be obtained in the decoder. For each T_k, a bit-stream can be obtained by eliminating frames in higher layers than 'k'.

With spatial scalability several spatial resolutions can be chosen at decoding time, and with the fidelity scalability, also called SNR or quality scalability, it is possible to select between several quality levels in the decoding flow.

In the SVC amendment, temporal and quality scalabilities should be supported by decoders without any restrictions. Only the spatial scalability is limited according to the defined profile. The decoder can select a combination of scalability levels associated, for example, a decoder can select some temporal scalability level with other quality scalability level combination, under one spatial scalability level. For that reason, it is important to make several decode tests to guarantee what is the part in the code that demands more resources.

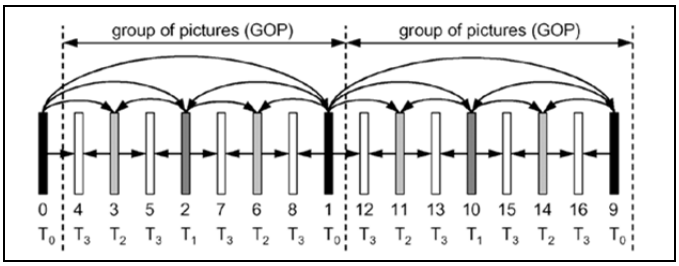


Figure 1. Example of of temporal scalable bit-stream.

The Open SVC Decoder (OSD) [3] is an open source library created at the IETR/INSA of Rennes that implements a SVC decoder written in C language. It has been developed and tested over different platforms: x86 processors, PDAs and DSPs. OSD implements only the scalable baseline profile. For our case of study, this profile is the most suitable for our proposals due to the limited hardware resources and displays required [8]-[9]. In addition to baseline profile, other restrictions has been applied, as no considering CABAC, disabling the use of B-slices in the bit-streams and using only the constrained baseline profiles identifiers number 66 and 83. OSD has been modified to isolate some blocks of the decoder for performance analysis and to redirect the output from the display to a YUV file.

Additionally, the OSD has been tested over the conformance sequences provided on the JVT site [10]. OSD is highly optimized, as shown in Table I, where OSD is up to 50 times faster than JSVM reference software in its version 9.16.

TABLE I. COMPARISON BETWEEN OPEN SVC DECODER AND JSVM REFERENCE SOFTWARE.

Sequences	Decoding Time (s)		
	JSVM	OSD	Speed Up
SVCBST-1	31.2	0.87	35
SVCBST-2	23.3	0.87	26
SVCBST-14	137	2.69	50
SVCBST-15	50	2.11	23

III. VISUAL STUDIO PERFORMANCE TOOLS

In the field of embedded systems, profiling tools (PTs) are used to analyze the performance of a certain code. Sometimes they can be found integrated in the IDE of a compiler, and in other cases they are just provided as a set of standalone tools, specifically designed to some programming languages. C and C++ are the most common languages analyzed, as they are the more extended languages in the field of embedded systems, but it is possible to find some specific PTs for many different languages.

There are several PTs in the market, some of them are commercial products and some other are provided free of charge. For example, Valgrind [12] is a widely used free profiling tool that supports several different programming languages. Nevertheless, Valgrind has two handicaps that

prevent it from being used in our research environment. The first one is that the test-bench is designed to work with an executable OSD under Windows developed with VS2010; the second one is that Valgrind is focused only in memory management, whereas CPU usage is need in this profile. However, the use of Valgrind is planned to be used in a close future, in order to compare the results obtained with Valgrind against the results obtained with Visual Studio 2010 Performance Tools.

In PTs, the source code is often instrumented, which means that the profiling tool needs to insert some additional code to analyze the performance, allowing simulation data gathering.

In that sense, PTs can monitor the code thanks to the code previously inserted. For that reason, the code to be analyzed must be arranged before the analysis task. Fortunately, this task must be performed only the first time that the code is going to be analyzed with the PT.

Normally, PTs need to run some programs in the background, starting them in the initializing of the profiling session, and stopping them when the profiling is finished. After the monitoring programs have stopped, some PTs need to run another tool to analyze the results obtained, normally writing the results in a file.

VS2010PT uses this schema; firstly the code to be analyzed is instrumented; secondly, a monitor tool starts to collect timing information at the entry and exit of every function in the code to be analyzed. When the code under analysis finishes, the monitor is stopped and the result is read in the IDE.

VS2010PT uses different methods to perform profiling: CPU Sampling, Instrumentation, .NET Memory Allocation, Concurrency – Resource Contention and Concurrency –, and Concurrency Visualizer. For this research work Instrumentation has been used as it evaluates all methods and functions in the code under analysis. CPU Sampling method has the drawback that only collects CPU usage, and it is important in this research also to obtain the memory usage during the process execution.

Once the Instrumentation method has been selected, and before starting the profiling task, it is necessary to specify which data will be collected, as CPU usage, memory usage, etc. For this purpose VS2010PT can collect performance data generated by the operating system or performance data generated by the CPU, namely CPU counters.

These CPU counters depend on the computer's CPU and on the way used to store the count of hardware-related events. For example, L2References or L2Misses are CPU counters related with L2 Cache. Finally, once the types of data have been selected, it is necessary to choose between Inclusive Values, that collect the resources (CPU usage, memory usage, etc.) of a function and its child functions, or Exclusive Values, that exclude its child functions.

VS2010PT has two main methods to perform the profiling operations. The first one makes use of a graphical interface, and the second one uses the included command-line tools. In the graphical method, the instrumentation of the code and the start and finish processes of the monitors are automated by means of a wizard in the IDE. In the command-line method, all the steps to deploy the profiling must be defined, running every tool in a proper sequence order.

In both cases, the PT demands the attention of the user, and if the process needs to be repeated, there is not a pre-defined method to program several profiling sessions and gather all the data automatically.

In the deep analysis of a video decoder this situation has been revealed to be very important, even more when using a SVC decoder, where the decoder needs to be tested using different bit-streams (encoded video sequences), with different scalability levels, and with different scalability combinations. It is important to keep in mind that the computational load of a scalable decoder will vary with the different selected scalability levels, even when using the same bit-stream.

IV. IAOpenSVC, A Profiling Enviroment for Open SVC Decoder

IAOpenSVC (IAO) is an interface written in C# using the Visual Studio 2010 command-line profiling tools under Windows XP, which helps in the profiling sessions of OSD. The current operative version is the 0.9b. The IAO allows the designer adding several bit-streams to automate the profiling sessions, repeating the process to obtain accurate results. The collected data results are stored in XML format, and can be connected with Excel to create several datasheets. IOA also allows to storage and manage the data in a custom database.

Besides the VS2010PT, IOA also makes use of several additional tools to automate the profiling sessions, as the "Bit Stream Extractor" from JSVM tools, and MySQL to storage the profiling data.

The use of MySQL instead of MS-SQL Server, has been motivated by the need of having the data stored in an expanded database system that can be easily exported. MySQL can be executed in a wider range of platforms and operative systems than the MS SQL Server.

In summary, IAOpenSVC was developed under the following constrains and conditions:

- Use of .NET 4.0 Framework.
- Visual Studio 2010 Performance Tools.
- BitStreamExtractorStatic.exe (JSVM Tool).
- MySQL Server 5.1 & MySQL Connector .NET 6.2.2.

IAO is not restricted to use certain versions of the tools employed. To solve this problem, a feature to configure all the external programs has been added in the program. However, it is necessary to use at least the same or a higher version of the initially used versions.

IAO is composed of two central windows: in the main window, shown in Fig. 2, it can be selected the bit-streams to decode, set the decoder options and configure the profiling options. In the database window, shown in Fig. 3, it is possible to query the database to inspect the data recollected, and to generate the reports in XML format or in the MS-Excel datasheet format.

In the main window it is possible to select the bit-streams to decode, the decoder options (scalability levels) for each bit-stream added, the number of profiling pass, and the counters to analyze in the profiling sessions.

The main flow starts by selecting a bit-stream to decode. Once selected, the program will show a list in the main window with all the layers detected in the bit-stream. These layers have been obtained using the BitStreamExtractor tool. The designer can follow the top-down scalability levels adding the desired options for the bit-stream by selecting the levels required in the decoder. Different levels can be selected for each one of the added bit-stream.

In the next step the designer can select one or more profiling session for all the bit-streams in the initial list. It is highly advisable to add the option for repeating a profiling session in order to obtain more accurate results. Using several profiling passes assures some degree of independence against the circumstantial load of the operative system. The program will add in the database only the average results, but the program will repeat the process up to 10 times. Once all the profiling passes are finished, the program computes and stores the arithmetic mean of all the results.

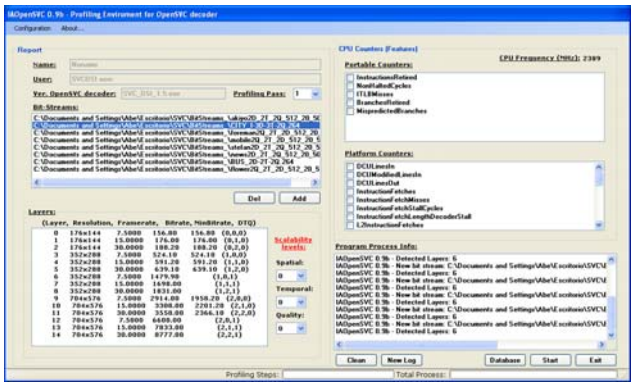


Figure 2. IAOpenSVC Main window.

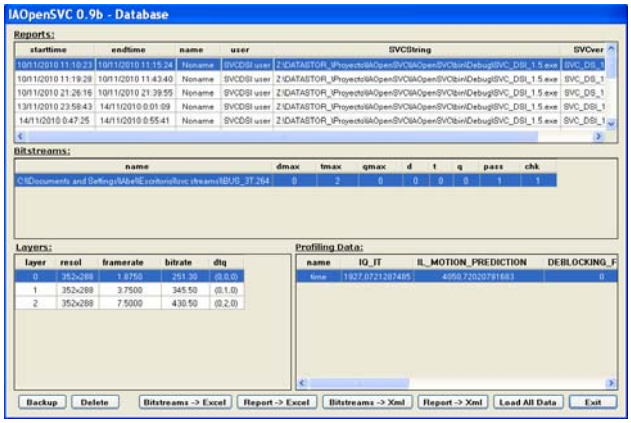


Figure 3. IAOpenSVC database window.

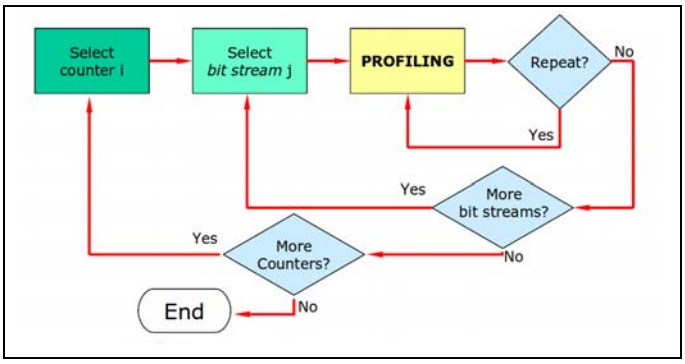


Figure 4. Proccess diagram in IAOpenSVC.

```
<?xml version="1.0" encoding="utf-8"?>
<Blocks>
  <Block Name="IQ_IT" f0="_ict_4x4_residual_C" f1="
     rescale_4x4_dc_chr" f2="rescale_4x4_dc_lum" f3="
     rescale_4x4_dc_residual" />
  <Block Name="IL_MOTION_PREDICTION" f0="_sample_interpolation" f1="
     _SampleInterpolation8x8" />
  <Block Name="DEBLOCKING_FILTER" f0="_filter_mb_svc" />
  <Block Name="IL_DEBLOCKING_FILTER" f0="_filter_mb" />
  <Block Name="CAVLD" f0="_residual" />
  <Block Name="BITSTREAM_PROCESSING" f0="GetNalBytesInNalUnit" />
  <Block Name="RESIDUAL_UPSAMPLING" f0="_Upsample_residu" />
  <Block Name="INTRA_UPSAMPLING" f0="_upsample_mb_luninance" f1="
     _upsample_mb_chroma" />
</Blocks>
```

Figure 5. Function.xml example.

Finally, the type of data to be obtained in the process must be selected. By default, IAO storages the CPU usage and selects the “Inclusive Values” for all the types of data, included CPU counters. The program detects the CPU counters available in the computing platform (usually a Personal Computer) in which IAO is running, and shows two lists, one with the portable counters, and another one with the platform counters.

Once the bit-streams, the profiling pass, and the CPU counters have been selected, the profiling process will initiate by simply pressing the start button. A diagram of the full process can be seen in Fig. 4.

IAO has also a list named “Program Process Info” in which information is shown about the profiling process, for example, when the MySQL Server is started, when the program is decoding a bit-stream, when it is looking for the functions in the results, or when it incorporate data to the database.

IAO is highly configurable, not only through the main window making use of the configuration option in the bar menu, but also by means of several external XML files.

When IAO finishes a profiling pass, it looks for the relevant functions in the data recollected by the VS2010PT and stores the results in a “.vsp” file that IAO converts to XML with a specific tool found in VS2010PT. These functions can be defined in the XML file located in the working directory of the IAO. This file contains all the functions that the user wants to collect from the profiling. It is necessary to remember that IAO obtains only the “Inclusive Values” of the functions. The user can define as many blocks as desired, and can also add all the

functions as the designer wants into every specific block. A function can also be located in more than one block. That is the case of some library functions that are called from several different higher level functions.

A file called "*functions.xml*" has one section named *Blocks*, and one element per each block named *Block*, with attributes *Name*, that contains the name of the block, and attributes *fn* that contains the *n* functions associated to the *Block*. An example of the file "*functions.xml*" can be seen in Fig. 5.

There are some other external files that IAO uses, but "*functions.xml"* is the more important one. For example, IAO uses "*conf.xml"* that contains all the important paths: the VS2010 path, the MySQL path and the IAO path. This file is automatically generated if IAO cannot find it, but the system allows the user to overwrite the paths through an option (Configuration) in the main window of IAO. The file "*init.txt"* is used by the IAO to setup the root password for MySQL. In all the case, IAO creates those files in the case that it cannot find them in the application path.

To create reports, IAO provides two different ways. The first one is to export the results in a XML file, and the second one is to connect with a MS-Excel datasheet and export the results.

IAO lets the user to export results at any time by using the database, and it also incorporates the possibility of exporting data from a single session or data of several bit-streams.

V. RESULTS OF PROFILING SESSIONS WITH IAOPENSVC

To test IAO, several profiles using some of the common videos used in the video research community were performed. In Fig.6 to Fig.10 are shown some of the tests performed with IAO to check the computational load of each block defined in OSD as a function of the selected scalabilities. The sequences used for test have been Bus, City and Park. In Table II are shown the encoder parameters used for these simulations. Due to the fact that spatial and SNR scalabilities exhibit similar characteristics (although not total computational costs) only the results for spatial scalability are shown.

As a reference, in Fig.6 is shown a comparative between these three video sequences using only the base layer resolution, which it is equivalent to use just H.264/AVC. In this case, the resolution is the minimum in all the scalabilities, i.e. 176x144 pixels and 7.5 fps. As it was expected, those modules related with inter-layer predictions exhibit no computational cost: RESIDUAL UPSAMPLING, IL DEBLOCKING FILTER and INTRA UPSAMPLING. It is worth to mention that the computational cost of Park is higher due to its superior amount of texture and motion. The sequence City is just the opposite case.

In Fig.7 it is shown a comparative using only spatial scalability, in order to assess the impact in every block. In this case RESIDUAL UPSAMPLING and INTRA UPSAMPLING present some computational cost. As the computational cost of these blocks is quite small (an average of 0.35% and 0.45% of the total computational cost) to be appreciated in Fig.7, the numerical costs have also been displayed.

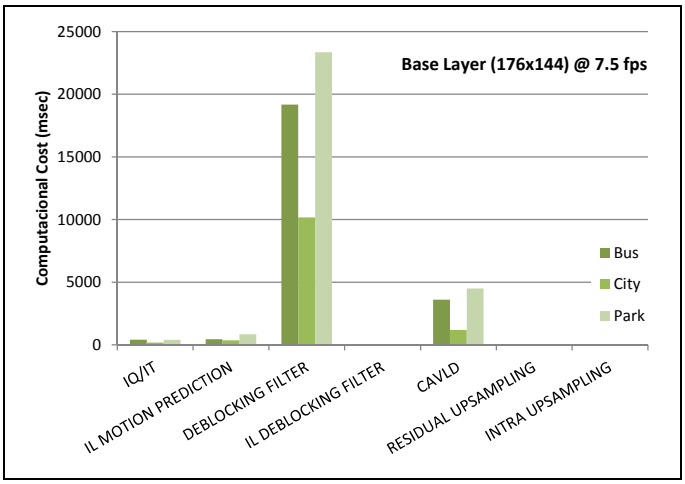


Figure 6. Decoding Base Layer.

TABLE II. SUMMARY OF ENCODER PARAMETERS.

SPATIAL SCALABILITY:	QCIF, CIF
TEMPORAL SCALABILITY:	7.5, 15 fps
FRAMES TO ENCODE:	200
ENTROPY ENCODER:	CAVLC
OTHER RESTRICTIONS:	DISABLEBSLICES, PROFILE_IDC = 66 | 83

Temporal scalability results can be seen in Fig.8, and supposes an average of 21.29% of the computational cost of the spatial scalability, mainly due to the use of lower spatial resolution frames, even taken into account that now the number of output frames has been doubled. In the case of temporal scalability, the inter-layer prediction of the deblocking filter supposes no computational cost.

When using all the scalabilities the resulting sequence exhibits the higher resolution, i.e. 352x288 pixels @15 fps. The results are shown in Fig. 9, and the average values for the three sequences are shown in Table III. It is clear that the deblocking filter and the CAVLD are the blocks with higher computational cost and therefore the main candidates to be implemented in hardware or using some type of accelerators.

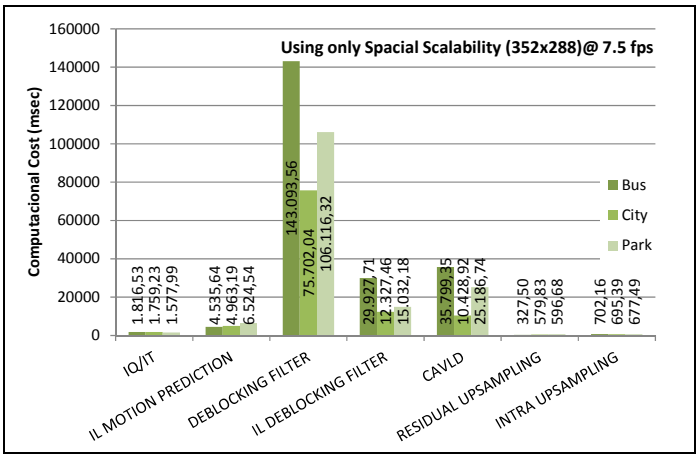


Figure 7. Using Spacial Scalability.

TABLE III. AVERAGE COMPUTATIONAL COST (THREE SEQUENCES) OF EACH BLOCK USIGN ALL THE SCALABILITIES.

Block	Average computation
IQ/IT	1.84%
IL MOTION PREDICTION	4.00%
DEBLOCKING FILTER	56.82%
IL DEBLOCKING FILTER	8.86%
CAVLD	27.71%
RESIDUAL UPSAMPLING	0.30%
INTRA UPSAMPLING	0.47%

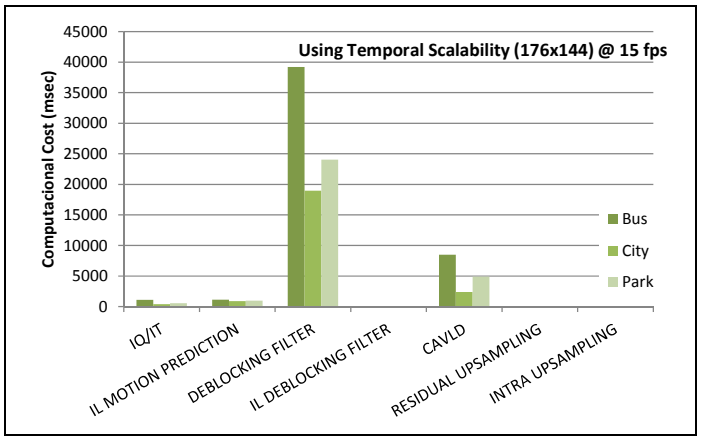


Figure 8. Using Temporal Scalability.

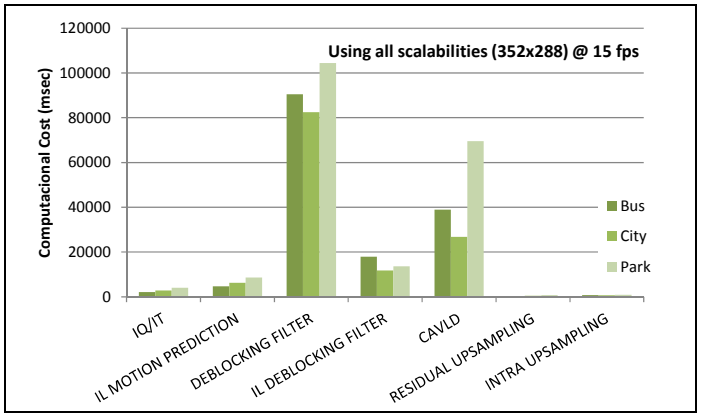


Figure 9. Computational cost using all scalabilities.

Finally, in Fig. 10 can be seen the average results over all the blocks for each scalability. The spatial scalability is the one with higher computational cost, followed by the base layer and the temporal scalability.

VI. CONCLUSIONS

The use of profiling tools can be of great help for designers at the time of properly distribute the computational load of complex systems using embedded systems. In this paper a profiling environment for scalable video decoding has been presented. The results show that the higher computational load in concentrated on the deblocking filter and in the CAVLD, especially for spatial scalability.

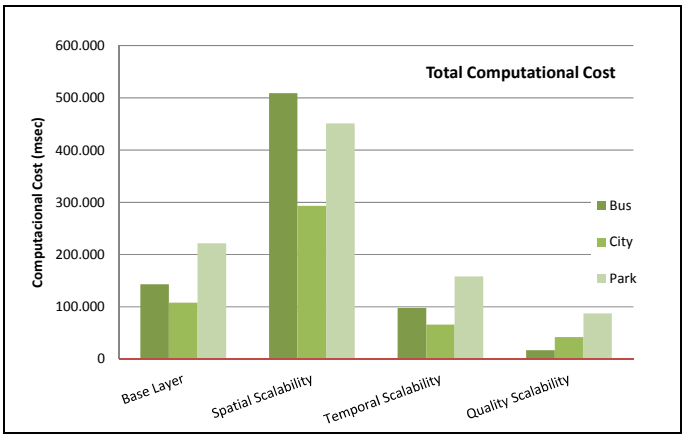


Figure 10. Total computacional costs.

ACKNOWLEDGMENT

This work is supported by the Spanish Ministry of Economy and Competitiveness (*Ministerio de Economía y Competitividad*) as part of the I+D+I Plan 2012-2014 (Research+Development and Innovation) support program in the context of Dynamically Reconfigurable Embedded Platforms for Networked Context-Aware Multimedia Systems (DREAMS) project (TEC 2011-28666-C04-04).

REFERENCES

[1] Wiegand, T., Sullivan, G.J., Bjontegaard, G., Luthra, A., "Overview of the H.264/AVC video coding standard," Circuits and Systems for Video Technology, IEEE Transactions on , vol.13, no.7, pp.560-576, (2003).

[2] Mrak, M., Sprljan, N., Izquierdo, E., "An overview of basic techniques behind scalable video coding," Electronics in Marine, 2004. Proceedings Elmar 2004. pp. 597- 602, (2004).

[3] M. Blestel and M. Raulet, "Open SVC decoder: a flexible SVC library," Proceedings of the international conference on Multimedia (MM '10). ACM, New York, NY, USA, 1463-1466, (2010).

[4] ITU-T and ISO/IEC JTC 1, ITU-T Recommendation H.264 - ISO/IEC 14496-10(AVC), "Advanced Video Coding for Generic Audio Visual services, Amendment 3: Scalable Video Coding," (2007).

[5] Joint Scalable Video Model JSVM-9.16, Available in CVS repository at Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen.

[6] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the Scalable Video Coding extension of the H.264/AVC standard," IEEE Trans. Circuits Syst. Video Technol., vol. 17, no. 9, pp.1103–1120, Sept. 2007.

[7] Alfonso D., Gherardi M., Vitali A., Rovati F., "Performance analysis of the scalable video coding standard," Packet Video, pp.243-252, 2007.

[8] Horowitz, M., Joch A., Kossentini F., and Hallapuro A., "H.264/AVC baseline profile decoder complexity analysis. IEEE Transactions on Circuits and Systems for Video Technology, 13(7):704–716, (2003).

[9] Maiti, S.N., Gupta, A., Piccinelli, E.M., and Saha, K., "Real-time SVC Decoder in Embedded System," In Proceedings of SIGMAP. (2009).

[10] Joint Video Team, "Conformance testing," http://wftp3.itu.int/av-arch/jvt-site/bitstream_exchange/SVC/, (2008).

[11] Visual Studio 2010: Analyzing Application Performance by Using Profiling Tools, http://msdn.microsoft.com/en-us/library/z9z62c29.aspx, (2012).

[12] Valgrind Home Page, http://valgrind.org/, (2012).