Facultad de Ciencias

Proyecto Fin de Carrera

Co-simulación HW/SW de una plataforma MPSoC Multi-SO

HW/SW co-simulation of a MPSoC Multi-OS platform

Para acceder al Título de

INGENIERO EN INFORMÁTICA

Autor: Rodrigo Fernández Rodríguez

Diciembre – 2012

Facultad de Ciencias

INGENIERÍA EN INFORMÁTICA

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por: Rodrigo Fernández Rodríguez
Director del PFC: Héctor Posadas Cobo y Eugenio Villar Bonet
Título: "Co-simulación HW/SW de una plataforma MPSoC Multi-SO"
Title: "HW/SW co-simulation of a MPSoC Multi-OS platform"

Presentado a examen el día:

para acceder al Título de

INGENIERO EN INFORMÁTICA

Composición del Tribunal:
Presidente (Apellidos, Nombre): Michael González Harbour
Secretario (Apellidos, Nombre): María del Carmen Martínez Fernández
Vocal (Apellidos, Nombre): Rafael Menéndez de Llano Rozas
Vocal (Apellidos, Nombre): Pablo Sánchez Barreiro
Vocal (Apellidos, Nombre): Roberto Sanz Gil

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: Vocal	Fdo.: Vocal
Fdo.: Vocal	Fdo.: El Director del PFC

Índice

1 Resúmenes en Castellano y en Inglés

1.1 Co-simulación HW/SW de una plataforma MPSoC Multi-SO

El objetivo de este trabajo ha sido trabajar en la generación de una herramienta de simulación de sistemas embebidos de alto nivel que permita simular sistemas que combinen distintos sistemas operativos sobre diversas plataformas HW. Más en concreto, el objetivo es partir de una herramienta previa y extendiderla para soportar sistemas que combinen dos sistemas operativos sobre tres arquitecturas distintas, generando un demostrador apropiado a tal efecto.

El trabajo se ha centrado en extender la plataforma de simulación SCoPE desarrollada por el Grupo de Ingeniería Microelectrónica (GIM) de la Universidad de Cantabria (UC) para realizar la simulación de sistemas embebidos multiprocesador System-on-Chip (MPSoC) con plataformas software que combinan distintos sistemas operativos. Además se han analizado las capacidades de simulación y estimación de prestaciones de este tipo de sistemas con múltiples sistemas operativos con diferentes arquitecturas que implican diferentes formas de comunicación entre los sistemas operativos y los componentes del software de aplicación, como pueden ser el uso de una red NoC o zonas de memoria compartida.

Para comprobar las capacidades de simulación y estimación se ha implementado una aplicación compleja que realiza la captura y codificación de vídeo y audio. Los datos se obtienen gracias a unos periféricos cuyos modelos se han implementado y añadido a la plataforma hardware, junto con sus respectivos controladores. Esta parte de captura de datos se realiza sobre un sistema operativo POSIX. Además, las tareas de compresión tanto del vídeo como del audio (a los formatos H.264 y mp3 respectivamente) se repartirán entre componentes ejecutados sobre sistemas operativos POSIX, que se encargará de la compresión de audio, y Win32, que se encargará de la compresión de vídeo.

Para desarrollar la plataforma que permita realizar las simulaciones se ha hecho uso de un plug-in desarrollado para SCoPE que permite la simulación de código de aplicación que utilice la API Win32. Este plug-in ha sido desarrollado de una forma inicial para una versión anterior del simulador. Por ello en primer lugar se ha realizado la adaptación del plug-in a la última versión disponible; así como realizado una prueba de su correcto funcionamiento con una aplicación de cierta envergadura para complementar los test que se han hecho por separado de las diferentes funcionalidades que este plug-in implementa.

Por último, se han comparado las simulaciones y estimaciones obtenidas con la infraestructura desarrollada con otras proporcionadas por otras herramientas como OVP.

1.1.1 Palabras Clave

Multi-SO, Co-simulación HW/SW, Sistemas Embebidos, MPSoC.

1.2 HW/SW co-simulation of a MPSoC Multi-OS platform

The aim of this work has been to work on the generation of a simulation tool for high-level embedded systems to simulate systems combining different operating systems on different HW platforms. More specifically, the goal is to start from a previous tool and extend it to support systems combining two operating systems over three different architectures, generating a test suitable for this purpose.

In this project, the SCoPE simulation tool, developed by Microelectronic Engineering Group (GIM) from University of Cantabria (UC), has been extended in order to support simulations of Multiprocessor System-on-Chip (MPSoC) embedded systems running software platforms combining several operating systems. Furthermore, during the proyect it has analyzed the simulation and performance estimation capabilities for systems including different operating systems on different hardware architectures. These combinations implies applying different communication strategies among operating systems and software application components, as Network-on-Chip communication or shared memory areas.

In order to test simulation and performance estimation capabilities, a complex application which performs audio and video capture and encoding has being implemented. Data is retrieved through peripheral models implemented and added to hardware platform for that purpose together with their respective device drivers. This data retrieval is performed on a POSIX operating system. Also, video and audio encoding tasks (to H.264 and mp3 formats respectively) are performed among components executed over POSIX operating systems, in charge of audio encoding, and Win32, in charge of video compression.

To develop a platform capable of performing the proposed simulations, a plug-in developed for SCoPE, which allows to simulate application code using Win32 API, has being used. This plug-in was initially developed for a previous version of SCoPE. Because of that, an adaptation to latest SCoPE version was needed; as well as a test with a complex application to ensure proper results of the plug-in, complementing other tests used to check individual features of the plug-in.

Finally, several tests have been performed in order to compare simulations and performance estimation results obtained with this infrastructure to others results obtained with other tools like OVP.

1.2.1 Palabras clave Inglés

Multi-OS, HW/SW Co-simulation, Embedded Systems, MPSoC.

2 Introducción y objetivos

2.1 Introducción

La evolución de los sistemas embebidos lleva al diseño de sistemas cada vez más complejos. Los nuevos sistemas embebidos complejos suelen constar de múltiples procesadores que se integran habitualmente en un mismo circuito integrado (Multi-Processor-System-on-Chip o MPSoC). Además, estos sistemas pueden incluir periféricos, co-procesadores, o incluso redes completas Network-on-Chip (NoC). Como consecuencia del aumento de las capacidades de la plataforma HW, los sistemas embebidos permiten soportar mayores funcionalidades, con lo que también el software que incluyen estos sistemas se vuelve más complejo.

El aumento de las capacidades de las plataformas HW y el incremento de la funcionalidad soportada han conllevado un incremento en la utilización de sistemas operativos complejos. Estos sistemas operativos proporcionan los recursos software necesarios para implementar y gestionar un mayor número de aplicaciones más complejas y con mayor funcionalidad. Un ejemplo de estos sistemas operativos son Linux, Windows o Android.

Como consecuencia de esta evolución, dichos sistemas embebidos con mejores prestaciones han hecho posible que surjan plataformas que combinan diferentes sistemas operativos. Esto proporciona a los diseñadores un mayor número de recursos con los que responder a requisitos como velocidad de respuesta, throughput, soporte multimedia [4,8], una mejora en la seguridad del sistema [5], o facilitar la reutilización de código [6].

Sin embargo, el desarrollo de sistemas embebidos más complejos no solo requiere de plataformas HW/SW con mayores capacidades. Las técnicas de diseño de dichos sistemas también han de evolucionar. En este sentido, se hace necesaria la utilización de herramientas de simulación que nos permitan verificar el correcto funcionamiento de la plataforma así como obtener estimaciones de métricas de rendimiento de dichos sistemas complejos, reduciendo el esfuerzo y el tiempo necesarios para su diseño.

Estas herramientas son especialmente importantes en etapas tempranas de diseño de sistemas embebidos donde no se suele disponer de la plataforma física sobre la que probar el software. Además, en los sistemas embebidos, en los cuales el tiempo de ejecución, el tamaño, el consumo energético, etc. son un factor determinante, al permitir elegir adecuadamente un hardware u otro. Esta es una decisión sumamente importante, la cual condiciona el resto del diseño del producto debido a la necesidad de ajustar el precio del producto final cumpliendo las restricciones en cuanto a rendimiento, tamaño y consumo. El uso de nuevas herramientas permiten evaluar diferentes soluciones y así elegir la que mejor se adapte a las características y requisitos de nuestro diseño. Además, estas herramientas, permiten depurar, analizar y optimizar el software antes de tener la plataforma física.

Para cumplir estos objetivos es importante obtener al principio del proceso de diseño un equilibrio entre tiempos de simulación y precisión de las estimaciones. Sin

embargo, muchos simuladores sacrifican el tiempo de simulación en favor de la precisión, como es el caso de los simuladores de conjunto de instrucciones (Instruction Set Simulators o ISS). Esto también implica modelos de hardware más complejos y por lo tanto requiere mayor esfuerzo en el desarrollo de estos modelos, lo cual repercute en el proceso de diseño de un sistema embebido incrementando el trabajo y el tiempo invertidos en este. Por ello el uso de estas herramientas debe ser limitado a etapas posteriores del proceso de diseño.

Otra técnica de simulación, que reduce los tiempos de ejecución de los ISS, es la virtualización de la plataforma. La virtualización permite ejecutar un sistema operativo en un procesador virtual sin necesidad de modificaciones del código. Estas técnicas también pueden simular plataformas distintas a la del host mediante traducción dinámica del código binario cross-compilado a código binario de la plataforma host. Sin embargo, estas técnicas tienen actualmente limitaciones para modelar correctamente aspectos como las cachés y los consumos energéticos [7]. Además, estas técnicas necesitan esfuerzos considerables para generar los modelos que permitan evaluar las distintas opciones de implementación.

Por el contrario, las técnicas de simulación nativa permiten modelar la ejecución del software en la plataforma host sin necesidad de emuladores intermedios; modelando los efectos en el rendimiento de ese software en una plataforma HW/SW específica distinta de la del host. Para ello se realiza una anotación de información de rendimiento en el código fuente original. De este modo, cuando este código anotado es ejecutado de forma nativa, se obtienen estimaciones globales del rendimiento del sistema simulado con unos tiempos de simulación menores que los obtenidos mediante ISSs o traducción binaria. Además, la utilización de modelos de sistemas operativos en estas herramientas de simulación nativa facilita la simulación de códigos de aplicación. Estos modelos evitan el tener que portar un sistema operativo a una plataforma concreta para realizar las simulaciones.

Uno de los problemas que nos encontramos en las actuales herramientas de simulación basadas en las diferentes técnicas expuestas es que a día de hoy no están orientadas a soportar simulaciones en las cuales la plataforma software comprenda varios sistemas operativos. De este modo surge la necesidad de herramientas que den soporte a dicho tipo de simulaciones. Por ello, en este proyecto se ha realizado una extensión de la herramienta de co-simulación nativa SCoPE para poder realizar simulaciones de plataformas con más de un sistema operativo. Hasta ahora SCoPE se ha utilizado para simular plataformas con un único tipo de sistema operativo. Sin embargo, por sus características, puede ser modificado para realizar el tipo de simulaciones propuestas.

Además, la infraestructura desarrollada se ha probado mediante una aplicación multimedia compleja desarrollada a tal efecto. La utilización de una aplicación multimedia para probar la infraestructura implementada se debe a que una de las principales áreas de aplicación de los sistemas embebidos es la captura y procesado de audio y vídeo. Un ejemplo de esto son las cámaras de vigilancia, teléfonos móviles, cámaras de tráfico, o sistemas de comunicación de vídeo o voz sobre IP.

Esta aplicación se ha probado sobre una plataforma software formada por un

sistema operativo Linux y un sistema operativo Windows. Se han elegido estos dos sistemas operativos por ser ejemplos representativos de sistemas operativos complejos que nos podemos encontrar en sistemas embebidos como los mencionados anteriormente. Así, las diferencias entre los sistemas operativos Windows y Linux hacen especialmente interesante el uso de ambos en una misma plataforma hardware como ejemplo de utilización de la infraestructura desarrollada.

2.2 Objetivos

El objetivo principal de este proyecto es extender la herramienta de simulación nativa SCoPE para que permita realizar co-simulaciones HW/SW de plataformas software compuestas por más de un sistema operativo diferente. Además se ha comparado SCoPE con una herramienta comercial similar.

Se pretende simular un sistema operativo de tipo Win32. Estos sistemas Windows reducidos basados en la API Win32 como por ejemplo Windows CE, son habitualmente integrados en algunos sistemas embebidos. Para su simulación nos hemos basado en una versión en inicial de un plug-in Win32 para SCoPE, el cual nos proporciona compatibilidad con el interfaz Win32. Conjuntamente con este sistema operativo Win32 se va a simular un sistema operativo de tipo POSIX. Esta API es la interfaz nativa implementada por SCoPE, y sobre la cual se ha desarrollado el modelo de sistema operativo que SCoPE proporciona por defecto.

La infraestructura resultante debe ser capaz de simular código de aplicación que implique diferentes funcionalidades distribuidas entre un sistema POSIX y un sistema Win32 sobre una misma plataforma hardware. Además, por una combinación de flexibilidad y esfuerzo requerido, se ha decidido que esta infraestructura permita la comunicación entre ambos sistemas operativos de dos modos, ya sea por red o mediante memoria compartida; así como proporcionar una forma de compilar, ejecutar, y depurar el software de aplicación sobre las plataformas HW/SW propuestas. Además no sólo debe ser capaz de simular este tipo de plataformas, sino también proporcionar estimaciones de rendimiento fiables del software ejecutándose sobre las plataformas HW/SW que se simulen en cada caso.

La segunda parte de este proyecto consiste en la implementación de una aplicación compleja cuyo objetivo es probar el correcto funcionamiento de la infraestructura desarrollada previamente. Para ello, la aplicación debe contar con una parte de código POSIX que realice una codificación de audio en formato WAVE a formato mp3 y una parte Win32 que realice la codificación de vídeo en formato YUV QCIF a formato H.264. Se ha elegido este tipo de aplicación por ser un caso representativo de uno de los campos de aplicación de sistemas embebidos. Esta aplicación capturará los datos de vídeo y audio a través de sendos periféricos, por lo que también se deben implementar los modelos de dichos periféricos y sus respectivos controladores.

El último objetivo de este proyecto es la comparación de la infraestructura desarrollada con la herramienta comercial de co-simulación HW/SW OVP. Esta comparativa busca mostrar que las soluciones basadas en técnicas de simulación nativa como SCoPE proporcionan mejores resultados y requieren menos esfuerzo por parte

del desarrollador que otras soluciones basadas en técnicas de traducción binaria.

3 Estado del arte

En esta sección se presentan diversas técnicas de simulación de sistemas embebidos, así como las herramientas y software utilizados en este proyecto.

3.1 Técnicas de simulación

Una de las principales técnicas de simulación y análisis de rendimiento utilizadas a lo largo de los años es la simulación de conjunto de instrucciones (ISS), aunque en los últimos tiempos está siendo sustituida por la traducción binaria. Los ISS proporcionan un gran nivel de detalle, sin embargo requieren de elevados tiempos de ejecución. En las primeras etapas del proceso de diseño no es necesaria una gran precisión en los tiempos de ejecución o consumos, sino que es preferible obtener buenos tiempos de simulación. Por lo tanto, estas etapas iniciales de diseño se ven beneficiadas al utilizar otras técnicas de simulación, ya que nos permiten incrementar la velocidad de simulación con respecto a los ISS. Las principales alternativas en este momento se basan en la virtualización, la traducción binaria y la simulación nativa.

La virtualización es una técnica que puede reemplazar el uso de ISS para la simulación de código binario [9]. Entre otras, alguna de las primeras aproximaciones se han centrado en virtualizar sistemas operativos, esto es, la creación de un entorno virtual ejecutando un sistema operativo en otra plataforma software. Se han desarrollado diversas tecnologías en este sentido, como la virtualización, la para-virtualización o la emulación [9,10]. Vmware [11,12] y Microsoft Virtual PC [13] proporcionan virtualización a nivel de la capa de abstracción hardware (HAL). Estas herramientas virtualizan todos los componentes hardware de modo que se puedan simular varias instancias de sistemas operativos en una única máquina física con un procesador x86. Sin embargo esta técnica no esta diseñada para simular otras plataformas diferentes a la del host, por lo que no resulta muy útil para simular sistemas embebidos.

Otra estrategia de virtualización cuya aceptación está creciendo últimamente es la virtualización del procesador. Estas técnicas modifican el código binario generado para procesador que se quiere simular, de forma que se puede ejecutar código compilado para otro procesador y plataforma en el computador nativo. Esto se hace normalmente mediante traducción dinámica de código binario. Las herramientas más importantes de este tipo son QEMU [14], OVP [15] y FastModels [16] de ARM. QEMU y OVP soportan diferentes arquitecturas, como x86, ARM o SPARC. La virtualización de procesadores es significativamente más rápida que los ISS [9], por ello esta técnica está ganando aceptación para crear modelos virtuales funcionales de un sistema. Sin embargo, esta técnica todavía tiene ciertos problemas que resolver en cuanto a la obtención de métricas de rendimiento de los sistemas, como son la estimación de consumos energéticos o modelados de cachés [7].

Además, tanto las técnicas de traducción binaria como los ISS necesitan esfuerzos considerables para generar los modelos que permitan evaluar las distintas opciones de

implementación. Los modelos de plataformas hardware utilizados por estas técnicas son complejos y requieren un esfuerzo y tiempo para su desarrollo superior a otras técnicas como la simulación nativa. En cuanto a las plataformas software, también requieren un mayor esfuerzo, ya que en muchos casos será necesario portar a la plataforma destino el sistema operativo que se vaya a utilizar.

Por último, como se comentó en la introducción, la simulación nativa es una técnica de modelado y simulación que proporciona estimaciones con una alta velocidad de simulación y un bajo esfuerzo en el desarrollo del modelo. Se han propuesto dos tipos de aproximación a la simulación nativa. En la primera se abstrae la plataforma hardware, de modo que todo el software, incluyendo el sistema operativo es anotado con información de rendimiento [17]. Esta técnica tiene el problema de que no siempre vamos a disponer del código fuente del sistema operativo para poder anotarlo, como es el caso de algunos sistemas basados en Windows. La segunda aproximación consiste en abstraer la plataforma software, de modo que en lugar de ejecutar el sistema operativo completo se utiliza un modelo de ese sistema operativo. Esta estrategia es utilizada por simuladores como RTK-Spec TRON [18] o la herramienta SCoPE, ambas basadas en SystemC.

3.2 Uso de sistemas operativos combinados

Como se ha dicho en la introducción, la tendencia en la evolución de nuevos sistemas embebidos cada vez más complejos permite que surjan plataformas software que combinan más de un sistema operativo. Existen diferentes motivaciones para este tipo de sistemas como pueden ser la mejora de la seguridad, el incremento del throughput, aumento de la velocidad de respuesta, o la reutilización de código. En este sentido, la utilización de múltiples sistemas operativos en una plataforma permite utilizar diferentes códigos de aplicación que utilizan APIs y recursos de diferentes sistemas operativos y de este modo evitar la necesidad de portar dichos códigos a otro sistema operativo, lo cual puede ser muy costoso en el caso de aplicaciones complejas. Por otra parte, el tener varios sistemas operativos nos proporciona un cierto aislamiento de los diferentes componentes de una aplicación, dependiendo de la técnica utilizada para combinar dichos sistemas operativos [5]. Este aislamiento es una ventaja de cara a la seguridad de dichos sistemas ya que un ataque a un componente de la aplicación no tendría porqué afectar al resto de componentes.

Para realizar la implementación de plataformas software que combinen más de un sistema operativo se han propuesto varias aproximaciones. Una de estas aproximaciones consiste en enlazar dos kernels formando un sistema operativo híbrido [24]. Esta técnica consigue combinar la funcionalidad de ambos kernels, pero con un coste en ingeniería elevado ya que requiere de importantes modificaciones en el código de ambos kernels. Además se puede dar el caso de no disponer del código fuente del sistema operativo, en cuyo caso esta técnica no sería válida.

Otra aproximación consiste en utilizar una capa de virtualización sobre la que se ejecutan los diferentes sistemas operativos, como hace por ejemplo la herramienta de virtualización SPUMONE [8]. En este caso, la capa de virtualización se encarga de gestionar las interrupciones y el uso de recursos por parte de los sistemas operativos.

Aquí se pueden diferenciar dos tipos de virtualización: virtualización completa o paravirtualización. En el primer caso las modificaciones requeridas en el sistema operativo son pocas o nulas, mientras que en el segundo caso se requiere una mayor modificación en los sistemas operativos. En cualquier caso, la virtualización hace que las modificaciones necesarias en los sistemas operativos sean mínimas.

En el caso de que los diferentes sistemas operativos no tengan que compartir recursos hardware se pueden utilizar directamente sin necesidad de modificaciones ni virtualización alguna.

3.3 Herramientas

A continuación se describen los distintos lenguajes y herramientas utilizados en el proyecto. En primer lugar se describe el lenguaje SystemC como infraestructura de modelado de sistemas embebidos, así como la herramienta de co-simulación nativa SCoPE, la cual ha sido extendida en este proyecto, junto con la versión inicial del plug-in que da soporte a la simulación de software desarrollado para la API Win32. Los diferentes elementos hardware de la plataforma que se simula mediante SCoPE se modelan en SystemC, el cual implementa un núcleo de simulación que es la base sobre la que se ha construido SCoPE .

También se describen las características básicas de la herramienta OVP, la cual será utilizada para realizar comparaciones con los resultados obtenidos mediante SCoPE.

3.3.1 SystemC

SystemC es un lenguaje de descripción de sistemas. Este lenguaje permite modelar sistemas HW/SW complejos. Como resultado de su utilización se obtiene una especificación ejecutable de un sistema que permite probar su funcionamiento.

SystemC proporciona recursos para simular tanto la funcionalidad como la plataforma HW de los sistemas en desarrollo. Sin embargo, a diferencia de los lenguajes de descripción de hardware (HDLs) como VHDL o Verilog, SystemC no es un nuevo lenguaje, sino que es una extensión de C++, implementada como un conjunto de librerías. En muchas ocasiones el diseño de sistemas VLSI implica una simulación inicial hecha en C o C++, normalmente orientada a realizar pruebas de concepto, y seguida por una traducción a un HDL. SystemC reduce este proceso combinando estos dos pasos en uno solo [19]. Al estar basado en C, facilita la integración de los equipos de trabajo SW y de diseñadores con experiencia en programación en el proceso de co-diseño HW/SW. Además, SystemC proporciona mejor soporte a la orientación a objetos, heredado del lenguaje C++ del que deriva. SystemC también proporciona diferentes niveles de abstracción para los modelos hardware.

Los componentes principales de SystemC son los siguientes:

- Módulos. Son los bloques básicos. Modelan diferentes componentes del sistema.

- Puertos. Estos elementos forman parte de los módulos y permiten la comunicación de éstos con otros elementos.

- Procesos. Son los principales elementos de computación. Los procesos SystemC son elementos concurrentes fuertemente controlados por el núcleo de ejecución de SystemC con objeto de implementar una simulación basada en eventos y tiempo correcta.

- Canales. Son los elementos que permiten la comunicación entre los distintos componentes de la plataforma. Pueden ser de diferentes tipos: señal, búffer, mutex, etc.

- Interfaces. Son los elementos que conectan los canales a los puertos.

- Eventos. Estos elementos son los que permiten la sincronización entre los diferentes procesos que se están ejecutando en el modelo.

SystemC es desarrollado y soportado por OSCI (Open SystemC Initiative), la cual es una organización sin fines de lucro, compuesta por una gran cantidad de compañías, universidades y personas a título individual.

La herramienta SCoPE hace uso de SystemC para modelar los distintos componentes de las plataformas que se simulan. Esto es transparente para el usuario, por lo que no se requieren conocimientos de SystemC para realizar simulaciones básicas. Sin embargo, puesto que los componentes hardware están modelados en este lenguaje, la implementación de nuevos modelos de periféricos o componentes hardware implementados por parte del usuario deberá hacerse en este lenguaje.

3.3.2 SCoPE

SCoPE es una herramienta de simulación orientada a la simulación de sistemas HW/SW MPSoC desarrollada por el Grupo de Ingeniería Microelectrónica de la Universidad de Cantabria (GIM/UC). Esta herramienta combina la facilidad de creación de modelos de plataforma con técnicas de modelado nativo de ejecución de software que permiten obtener estimaciones de rendimiento de una forma rápida además de verificar la funcionalidad del software y así facilitar su desarrollo. Para obtener estos resultados se utilizan técnicas de anotación en código nativo conjuntamente con modelos en alto nivel basados en transacciones (Transaction Level Modeling o TLM) de la plataforma hardware del sistema a simular.

SCoPE proporciona un conjunto de herramientas para realizar modelados de diferentes tipos de plataformas, tanto plataformas simples formadas por un procesador, un bus y una memoria; como sistemas complejos formados por MPSoCs conectados mediante NoCs como el mostrado en la Figura 1.

Esta herramienta proporciona velocidades de simulación cercanas a la ejecución del software en el propio PC. Esto se debe a que, a diferencia de la traducción binaria, el código ejecutado está compilado para la plataforma host sobre la se realiza la simulación. Sobre ese código nativo se realiza la instrumentación con información de

prestaciones de rendimiento. Con esto se puede ejecutar el código directamente en el host y obtener estimaciones de prestaciones sobre la plataforma que se esté simulando.

SCoPE esta especialmente orientado a etapas tempranas de diseño, siendo especialmente de utilidad para las siguientes tareas:

- Explorar las mejores configuraciones mediante la simulación en conjunto del software y el hardware en un mismo modelo, pudiendo de este modo analizar el rendimiento de diferentes configuraciones tanto software como hardware y observar el efecto que puedan tener el uso de diferentes componentes hardware o configuraciones software en el diseño.

- Ayudar al diseño del software proporcionando una plataforma virtual donde se puede analizar el comportamiento del software y su interacción con los diferentes elementos hardware así como sus prestaciones. Esto permite comenzar el desarrollo del software sin necesidad de tener un prototipo físico, ya que podemos probarlo sobre dicha plataforma virtual. Además evita tener que adaptar un sistema operativo a la plataforma destino, ya que SCoPE proporciona un modelo de sistema operativo abstracto que podremos utilizar sobre cualquier plataforma hardware.

La Figura 1 muestra en detalle cómo se estructura una simulación y nos permite entender mejor cómo funciona SCoPE. En ella se puede observar cómo el código anotado se simula sobre el modelo de la plataforma HW/SW, ejecutándose tanto el modelo como el software anotado sobre el host.

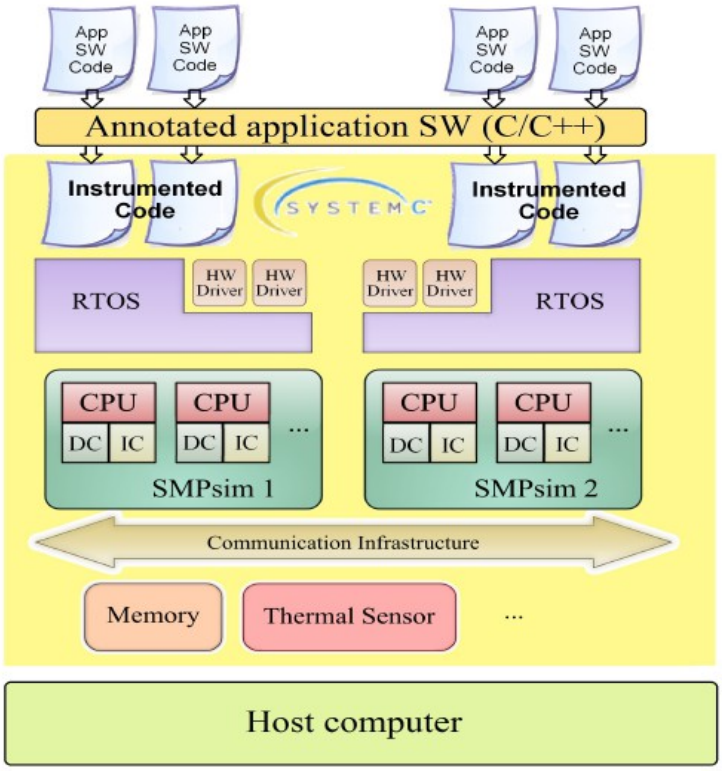

Figura 1: Estructura de un modelo de sistema completo

Desde un punto de vista técnico, SCoPE es una extensión de SystemC implementado mediante una librería C++. Por un lado simula y modela el rendimiento de código C/C++ sobre modelos de alto nivel de sistemas operativos, y por el otro realiza la co-simulación de este código con modelos de hardware implementados en SystemC.

A continuación se describe de forma más detallada la herramienta SCoPE. Para ello, en las siguientes sub-secciones se explica la estructura de SCoPE, la forma de realizar la anotación de información de prestaciones sobre el código nativo, cómo modela SCoPE sistemas operativos y hardware y cómo se realiza una simulación mediante SCoPE.

3.3.2.1 Estructura de SCoPE

SCoPE está compuesto por dos partes principales: un núcleo de simulación, que proporciona la infraestructura básica de simulación, y una serie de plug-ins que se conectan a este.

El núcleo de simulación consta de los siguientes componentes:

- Estimaciones aproximadas y modelado de tiempos de ejecución software.
- Modelo aproximado de caché de instrucciones.
- Modelo SMP basado en el standard POSIX.
- API POSIX.
- Infraestructura HAL basada en el modelo de dispositivos de caracteres Linux.

Mientras que los plug-ins que hay actualmente son los siguientes:

- Estimación de software optimizada
- Modelado de cachés de instrucciones, datos y L2
- Descripción de plataforma IP-XACT
- API uC/OS
- API Win32
- Análisis térmico
- Modelado de redes
- TCP/IP

- Linux de Tiempo Real

3.3.2.2 Anotación del software de aplicación

SCoPE realiza la estimación de prestaciones de tiempo y consumo mediante simulación nativa. Como ya se ha dicho previamente, la simulación nativa consiste en la instrumentación de código con información de prestaciones de dicho código en una plataforma diferente a la plataforma sobre la cual se va a ejecutar.

Para modelar el rendimiento del software de aplicación se analiza el código fuente y se estima el número de instrucciones de cada bloque básico de código. A partir de esta información se anota información de prestaciones de tiempo y consumo en el código de forma que ese código instrumentado nos pueda dar información relativa a su rendimiento cuando sea ejecutado. A continuación ese código instrumentado se compila de forma nativa para el computador en el que se quiere simular [23].

La aplicación `scope-g++` es una extensión de `gcc/g++` que realiza esta anotación en el código y genera los ficheros binarios de forma automática, tal y como se muestra en la Figura 2. En primer lugar se analiza el código y se identifican los bloques básicos de código. Esa información se utiliza para anotar en el código fuente información de prestaciones de dichos bloques de código. Una vez hecho esto se instrumenta el código para modelar las cachés y se genera el fichero ejecutable.

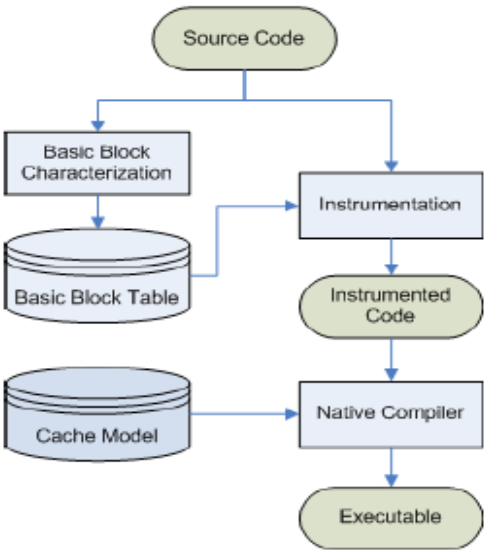

Figura 2: Proceso de estimación

El compilador `scope-g++` también se encarga de modificar las llamadas a funciones del sistema de forma que sean redireccionadas a las llamadas del modelo de sistema operativo de SCoPE en lugar de al sistema operativo del host.

`scope-g++` requiere los mismos flags necesarios para una compilación mediante

`gcc/g++`, además de los flags específicos de SCoPE, los cuales modifican parámetros del simulador como el procesador a simular, la API utilizada por el software de aplicación, el lenguaje del código fuente, el método de estimación a utilizar, etc.

Las anotaciones realizadas por SCoPE se hacen de modo que no interfieren con el uso de herramientas de depuración como `gdb`, facilitando el depurado del código.

3.3.2.3 Modelado de Sistemas Operativos

SCoPE modela el comportamiento de un sistema operativo, incluyendo la concurrencia entre diferentes threads, paralelismo entre procesadores, planificación y sincronización.

SystemC utiliza una política de planificación no expulsora, por lo que SCoPE utiliza su propio planificador que permite utilizar diferentes políticas basadas en prioridades.

Para modelar los sistemas operativos, SCoPE implementa las librerías necesarias para esta tarea. Puesto que SCoPE está diseñado para ejecutarse sobre un sistema Linux permite que sean utilizadas diferentes librerías del sistema como `string.h`, `math.h`, etc.; siempre y cuando estas no sean llamadas a funciones bloqueantes, que modifiquen la planificación o dependan de temporizadores.

También implementa funciones básicas para el soporte de controladores de dispositivos de caracteres así como de manejo de interrupciones basándose en las proporcionadas por el kernel de Linux 2.6, con lo que facilita el uso y desarrollo de controladores reales.

Además del soporte original a la API POSIX, SCoPE da soporte a otras APIs:

- uC/OS El soporte de esta API se realizó en colaboración con Marvel Hispania.
- Win32 Realizado por GIM, pero se encuentra en una fase inicial y está poco probado.

3.3.2.4 Modelado de la plataforma Hardware

Como ya se ha dicho anteriormente, los elementos hardware de la plataforma se modelan mediante SystemC.

Para facilitar la tarea de modelar diferentes sistemas, SCoPE incluye varios elementos hardware genéricos:

- Bus basado en TLM2 para la comunicación entre diferentes elementos hardware y transferencia de interrupciones.
- DMA para realizar transferencias de grandes bloques de datos.
- Módulo de memoria para la simulación de cachés y transferencias mediante

DMA.

- Interfaz de red que actúa como conector de los nodos a la NoC del sistema.
- Interfaz hardware que permite añadir fácilmente nuevos periféricos.

3.3.2.5 Simulación mediante SCoPE

Para realizar una simulación utilizando esta herramienta se deben seguir una serie de pasos:

- Compilar el software de aplicación. Para ello se deberá utilizar `scope-g++`, el cual se encargará de realizar el procesado e instrumentación adecuados. Además de los flags que requiera el código deberemos incluir el flag -c para que el resultado sea un fichero objeto que posteriormente enlazaremos con el resto de elementos de la simulación.
- Crear y compilar los periféricos. Para crear los elementos hardware que queramos conectar al modelo del bus mediante el estándar TLM2, SCoPE proporciona una serie de wrappers que facilitan esta labor. Los wrappers proporcionados se corresponden a modelos de periféricos de tipo maestro, maestro/esclavo y esclavo. Estos modelos de periféricos se deberán compilar con `g++` de forma que luego puedan ser integrados en la plataforma hardware a simular.
- Crear y compilar la arquitectura del sistema. Para generar una descripción del sistema deberemos crear un fichero `sc_main.cpp`, el cual deberá contener la función `sc_main`. Dicha función es llamado por SystemC y es la que se ejecutará en primer lugar cuando se realice la simulación. En esta función se definen tanto la plataforma hardware como la infraestructura software y el software de aplicación. La estructura básica de este fichero es la siguiente:

```
#include "sc_scope.h"
int sc_main(int argc, char **argv){
      Crear la infraestructura software.
      Cargar el software de aplicación y drivers.
      Describir plataforma hardware.
      sc_start(time);
      Destruir componentes creados previamente.
      return 0;
}
```

Figura 3: Formato básico del fichero sc_main.cpp

Una vez hecho esto se compila con `g++`, incluyendo los ficheros de cabeceras de SCoPE y SystemC

- Enlazar los componentes de la simulación. Por último se deberán enlazar los ficheros objeto del “sc_main”, el software de aplicación y los componentes hardware, junto con las librerías de SCoPE y SystemC; así como el resto de librerías necesarias para ejecutar el código de aplicación. De este modo se generará un fichero ejecutable que nos permitirá realizar la simulación de nuestro sistema.

- Realizar la simulación. Una vez generado el fichero ejecutable basta con ejecutarlo para que se realice la simulación de nuestra plataforma HW/SW, al final de la cual obtendremos los resultados de estimación de prestaciones con un formato como el del ejemplo de la Figura 4. En dicha figura podemos ver que se muestran las estimaciones de tiempos de ejecución, tanto de la aplicación completa como individuales por threads. También se proporciona información del número de cambios de contexto y de instrucciones ejecutadas así como de uso de las cachés y el bus, de consumos de las cachés y del procesador, o el número de interrupciones

```
RTOS:  0
      Number of m_processes created:  1
      Number of m_processes destroyed:  1
      Mean process duration (process start-process end):38.1546sec
      Last SW execution time:  38.1546 sec
      Process PID: 4
            Thread TID: 5, name: , User time: 37698321085 ns
      Total User time:  37.6983 sec
      Total Kernel time:  0.456269 sec

  processor_cpu_0_rtos_0_0_0
      Number of thread switches:  23818
      Number of context switches: 3817
      Running time: 36172127485 ns
      Use of cpu: 18.0861%
      Instructions executed: 4159792126
      Instruction cache misses: 1546714
      Data cache hits: 1405935899
      Data cache misses: 565561
      Data cache write backs: 0
      Core Energy: 8.31958e+09 nJ
      Core Power: 41.5979 mW
      Instruction Cache Energy: 1.25412e+10 nJ
      Data Cache Energy: 4.24043e+09 nJ
      Instruction Cache Power: 62.7062 mW
      Data Cache Power: 21.2022 mW
      Bus access time:  1982462850 ns
      Bus transfers:  79442046 bytes
      Idle time:  161835409665 ns
      Stall time:  0 ns
      Number of interrupts:  20001

  Bus HAL_0
      Bytes transferred: 79442046
```

Figura 4: Ejemplo de resultados de estimación de SCoPE

3.3.3 Plug-in Win32

SCoPE dispone de diferentes plug-ins que extienden sus funcionalidades. Uno de ellos es el plug-in "Win32", del cual se dispone de una versión inicial. Este plug-in añade un nuevo interfaz de sistema operativo, dando soporte para la simulación del software de aplicación que haga uso de la API de Win32.

Para proporcionar capacidades de simulación de código Win32, esta extensión utiliza y acopla el software de emulación WINE en la estructura de SCoPE. De este modo WINE se encarga de la gestión de threads y elementos de sincronización mientras que el plug-in traduce el comportamiento de esos elementos a POSIX para integrarlos en la simulación. Con esto WINE nos facilita la simulación de código Win32, además de proporcionando implementaciones de las diferentes librerías de Windows.

3.3.3.1 WINE

WINE es un proyecto open source que permite la ejecución de programas compilados para plataformas Windows en plataformas de tipo UNIX. Para ello implementa versiones de las principales librerías de sistema Windows creando una capa software que permite emular un sistema Windows sobre una plataforma UNIX.

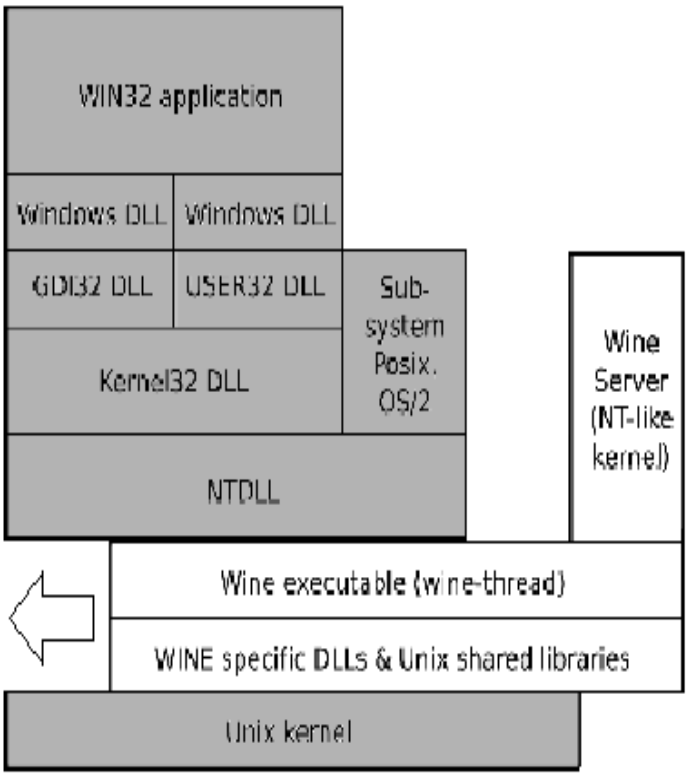

Figura 5: Arquitectura Windows NT + WINE

Este emulador está basado en una arquitectura Windows NT como podemos ver en la Figura 5, donde se representan en blanco los módulos añadidos por WINE sobre la arquitectura Windows NT para actuar de intermediario entre ella y el host UNIX.

El elemento principal es Wine Server, que hace las veces de kernel de Windows NT y que se ejecuta como un proceso UNIX independiente. Wine Server se encarga de gestionar las llamadas de sistema Win32 y proporcionar servicios de gestión de threads, sincronización y comunicación entre procesos (IPC) a las aplicaciones Win32 ejecutadas sobre WINE.

3.3.3.2 SCoPE + WINE

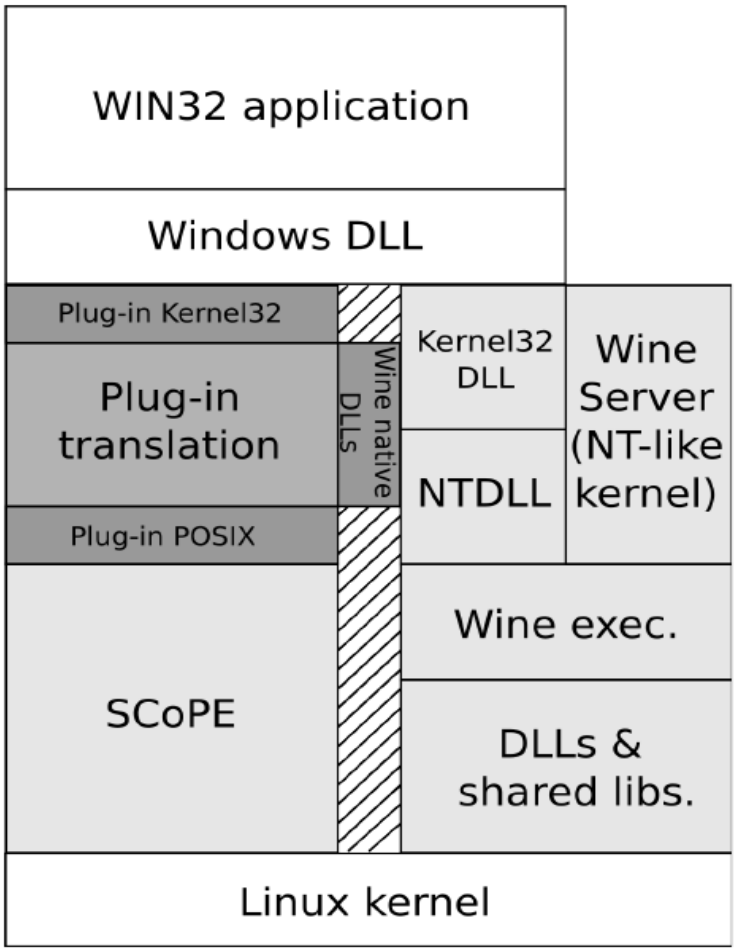

Figura 6: Arquitectura SCoPE + plug-in Win32

La arquitectura resultante de la combinación de SCoPE con WINE mediante la adaptación a través del plug-in se puede observar en la Figura 6. En ella se muestra cómo se conectan los diferentes elementos y capas software.

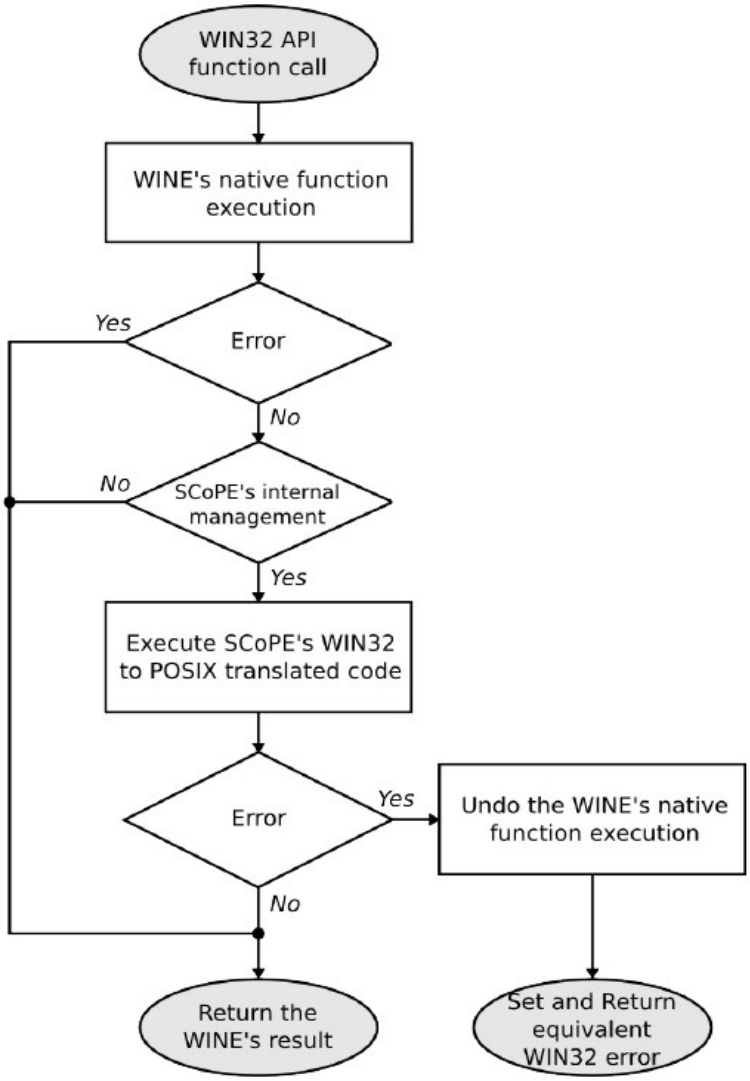

Figura 7: Gestión de llamadas a funciones Win32

Este plug-in proporciona traducción directa de las funciones de gestión de threads, procesos y sincronización al interfaz POSIX nativo de SCoPE. Estas funciones traducidas a POSIX se ejecutan bajo la supervisión de WINE, quien también ejecuta esas funciones sin traducir, garantizando de este modo un correcto funcionamiento de dichas funciones. El resto de funciones que no hayan sido parseadas y traducidas a POSIX serán gestionadas directamente por WINE, dejando a SCoPE únicamente la tarea de realizar las estimaciones de prestaciones al ejecutar la simulación. La Figura 7 nos permite entender mejor cómo se realiza esta gestión de llamadas de sistema Win32 por parte de SCoPE y WINE.

3.3.3.3 Uso del plug-in

Este plug-in requiere un proceso de compilación distinto al usado habitualmente con WINE, el cual está pensado para ser ejecutado directamente sobre un host Linux, y por lo tanto no es adecuado para su ejecución sobre SCoPE. En el nuevo proceso de

compilación se generan las librerías WINE de forma que cuando compilemos una aplicación Win32 para simularla con SCoPE se incluirán estas librerías junto con las librerías de SCoPE, así como el código a simular compilado en un único ejecutable. Seguidamente se realiza la inicialización de WINE una vez se halla realizado la inicialización del entorno de co-simulación nativa como se muestra en la Figura 8.

Para compilar el código de aplicación deberemos tener en cuenta tanto las opciones necesarias para generar un binario Win32, como las requeridas por SCoPE. Entre ellas deberá estar el flag `--scope-api=win32` para indicarle que el código hace uso de dicha API.

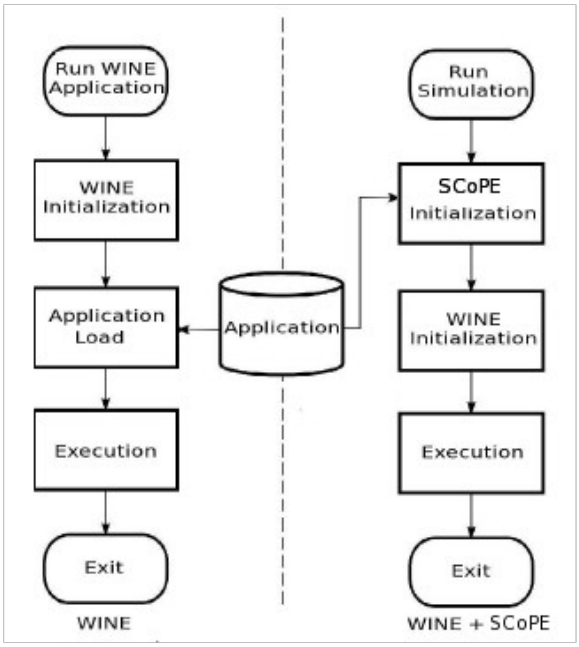

Figura 8: Ejecución sobre WINE vs. ejecución sobre SCoPE + plug-in Win32

En la Figura 8 se compara el proceso de ejecución de una aplicación Win32 sobre WINE y la ejecución de dicha aplicación en una simulación mediante SCoPE.

La versión inicial del plug-in se desarrolló para una versión anterior de SCoPE. Esta versión no es compatible con la versión actual de SCoPE debido a los cambios que se han realizado en el simulador desde la creación de este plug-in. Esto ha hecho que fuera necesaria una modificación en la forma en que el plug-in se integra con el simulador. Además, puesto que era una versión inicial, se han corregido diversos bugs que se han encontrado.

3.3.4 Open Virtual Platform (OVP)

Open Virtual Platform (OVP) es una herramienta comercial enfocada a la simulación mediante virtualización de la plataforma y traducción binaria del código de

aplicación. Esta herramienta facilita las labores de verificación y depurado de sistemas embebidos.

En este proyecto se ha utilizado OVP para realizar una comparativa con la herramienta de simulación SCoPE y de este modo poder analizar los resultados obtenidos con ambas herramientas, así como del esfuerzo requerido para crear plataformas HW/SW en cada una de ellas.

Esta herramienta utiliza SystemC para crear modelos detallados de la plataforma sobre la que se vaya a ejecutar el software que se quiera simular. Por el contrario, SCoPE realiza una simulación nativa utilizando modelos de alto nivel de la plataforma, con lo que debería proporcionar mejores velocidades de simulación que OVP.

OVP ha sido desarrollado principalmente por Imperas, aunque la lista de compañías y organizaciones implicadas en su desarrollo está en continuo crecimiento. Entre ellas se pueden destacar MIPS, Cadence o la Universidad de Washington.

Los principales componentes de esta plataforma son los siguientes:

- Modelos OVP: Son modelos en SystemC de diferentes plataformas hardware entre los que se destacan modelos detallados de diferentes procesadores para sistemas embebidos (MIPS, ARM, Motorola, etc). Estos se dividen en dos tipos:
    - Hardware Virtual Platforms: Estos modelos de alto nivel son muy detallados y proporcionan precisión de ciclo.
    - Software Virtual Prototypes: Estos modelos son menos detallados que las Hardware Virtual Platforms y proporcionan precisión de instrucción. Sin embargo proporcionan mayor velocidad de simulación.
- APIs OVP: Son un conjunto de APIs escritas en C/C++ que permiten al usuario crear diferentes modelos de procesadores, periféricos y plataforma.
- OVPsim: Es el núcleo de simulación implementado en C/C++. Se proporciona en forma de librería que se integra con la plataforma a simular. Incluye una serie de wrappers para C/C++, SystemC y SystemC TLM que permiten conectarlo a los modelos OVP. Permite la simulación de ejecución de binarios de código de aplicación proporcionando de este modo resultados con precisión de instrucción.

Esta plataforma está disponible tanto para sistemas Windows como Linux.

Imperas también proporciona diferentes tipos de herramientas de simulación avanzada multi-core, verificación automática de software, depurado en plataformas multi-core, etc. A diferencia de la plataforma OVP estas herramientas proporcionadas por Imperas son de pago. Actualmente otras compañías están desarrollando sus propias herramientas que se integran con la plataforma OVP, pero todavía no están disponibles.

3.4 Códigos de aplicación

En este proyecto se han utilizado codificadores de audio y vídeo para generar una en la aplicación de prueba con la que demostrar el funcionamiento de la infrastructura desarrollada. Estos codificadores se basan en los algoritmos mp3 y H.264 respectivamente. A continuación se hace un breve resumen de la tecnología detrás de estos formatos, además de explicar que codificadores se han utilizado.

3.4.1 Códec mp3

El formato de audio mp3 es un formato de audio comprimido con pérdidas desarrollado por el Moving Picture Experts Group (MPEG) en el año 1993 y definido por los estándares ISO/IEC 11172-3 e ISO/IEC 13818-3, MPEG-1 y MPEG-2 respectivamente.

Esta codificación se basa principalmente en eliminar frecuencias no audibles por el oído humano así como en eliminar redundancias estadísticas y sensoriales, es decir, relacionadas con la percepción del sonido por el oído humano.

No hay un algoritmo estándar para realizar la compresión al formato mp3, pero el estándar define la metodología mostrada en la Figura 9 que recomiendan seguir a los algoritmos de compresión que se implementen. Esta figura se refiere al estándar MPEG-2, que añade la primera y segunda etapa de filtros respecto a lo definido en el estándar MPEG-1.

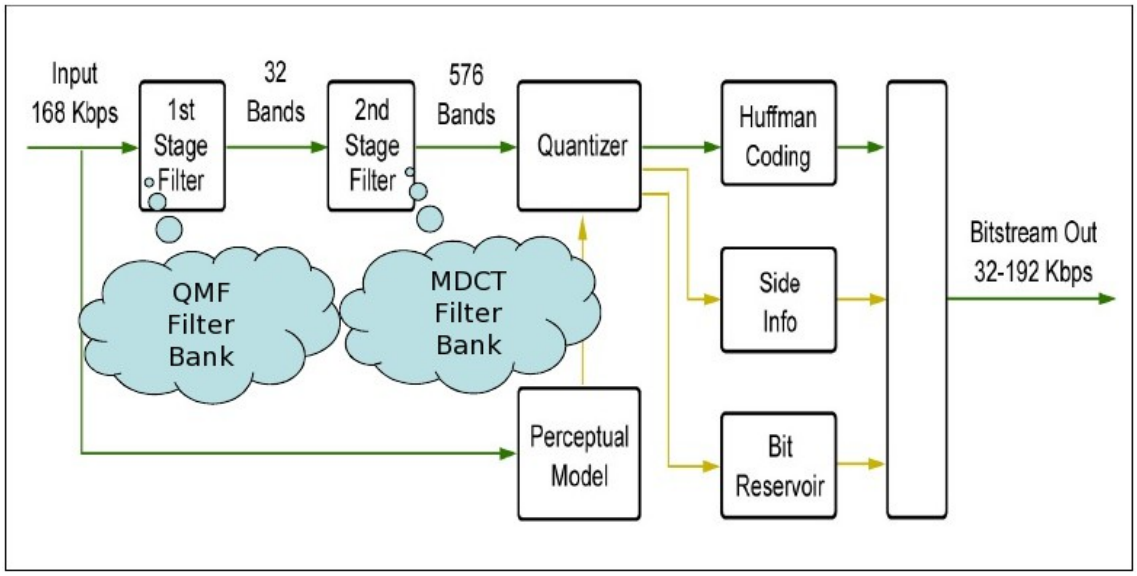

Figura 9: Flujo de codificación mp3

Para este proyecto se ha utilizado un algoritmo de compresión ligero que podría encontrarse en cualquier sistema embebido. Por ello se ha decidido utilizar la versión 0.1.4 del software Shine, que es un codificador mp3 creado por Gabriel Bouvigne, el mismo autor de LAME-mp3, uno de los códecs mp3 más extendidos a nivel PC. A diferencia de LAME, el objetivo de Shine es proporcionar un codificador sencillo que

facilite la tarea de ser integrado por otros desarrolladores en proyectos que requieran un codificador mp3, como es nuestro caso.

Shine está diseñado para utilizarse desde consola de comandos y puede ser compilado tanto para sistemas Windows como para sistemas Linux. Proporciona codificación de audio en formato WAVE a mp3 permitiendo especificar diferentes tasas de entre 32 y 320 kbps así como diferentes modelos psicoacústicos, estándar MPEG-1 o MPEG-2, y otras opciones.

Este software se ha modificado para adaptarlo a la aplicación de prueba implementada en este proyecto. Esta adaptación ha requerido la modificación de la inicialización del códec así como otras modificaciones necesarias para la integración del códec en la aplicación.

3.4.2 Códec H.264

El formato H.264 o MPEG-4 parte 10 es un formato de codificación de vídeo desarrollado por el Moving Pictures Expert Group (MPEG) y la International Telecommunications Union (ITU) en 2003.

La codificación de vídeo en formato H.264, así como su predecesor MPEG-2, se basa principalmente en no incluir todos los cuadros de vídeo en el flujo de datos, sino que ciertos cuadros se predicen a partir de otros. De este modo se diferencian tres tipos básicos de cuadros: I, P y B, los cuales a su vez de dividen en macrobloques y bloques sobre los que se realiza la correlación entre cuadros. Los tres tipos básicos de cuadros se utilizan del siguiente modo:

- Cuadros I (Intra): Contienen una imagen completa que suele estar comprimida en formato JPEG y es utilizada para la predicción de los cuadros de tipo P y B. Sirven también de punto de entrada aleatorio a un flujo de vídeo. Estos cuadros son los que necesitan mayor cantidad de bytes para ser codificados.

- Cuadros P (Predicted): Son obtenidos por predicción a partir de otros cuadros previos y necesitan de una imagen de referencia anterior para ser decodificados.

- Cuadros B (Bi-predicted): Estos se obtienen a partir de una combinación del cuadro anterior y el posterior, ya sean de tipo I o P. Por ello necesitan de dos imágenes de referencia I para ser decodificados.

Además los cuadros B y P pueden contener algún macrobloque codificado de forma que se eviten errores de predicción en secuencias largas entre cuadros I.

En la Figura 10 podemos observar cual es el esquema de codificación para el estándar H.264.

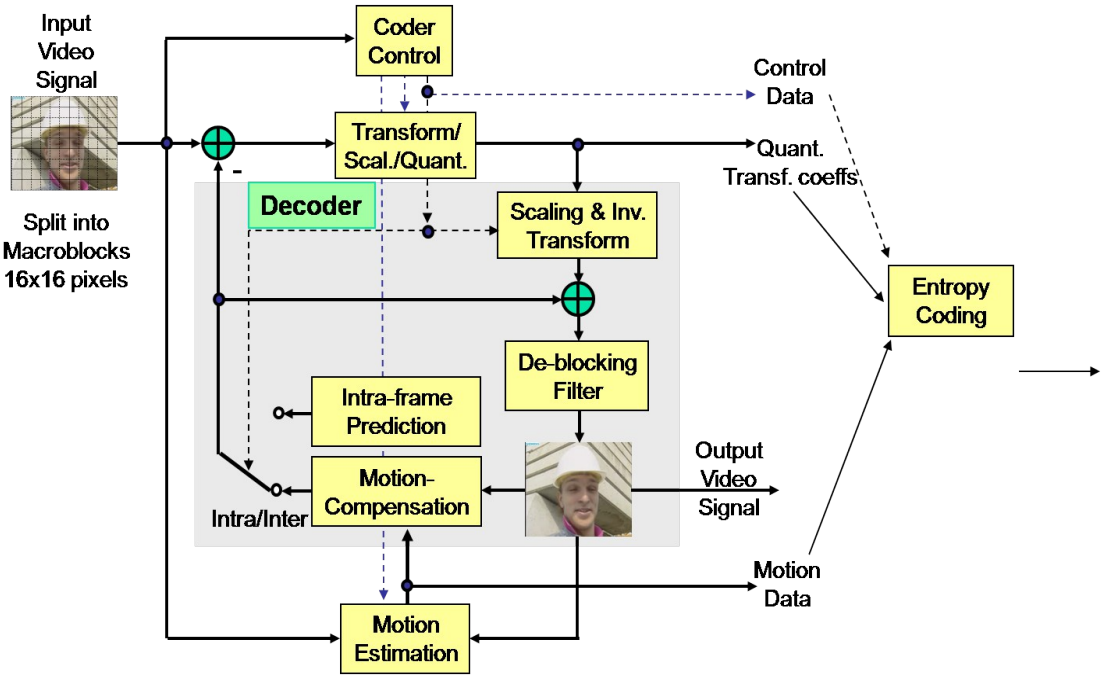

Figura 10: Esquema de compresión H.264

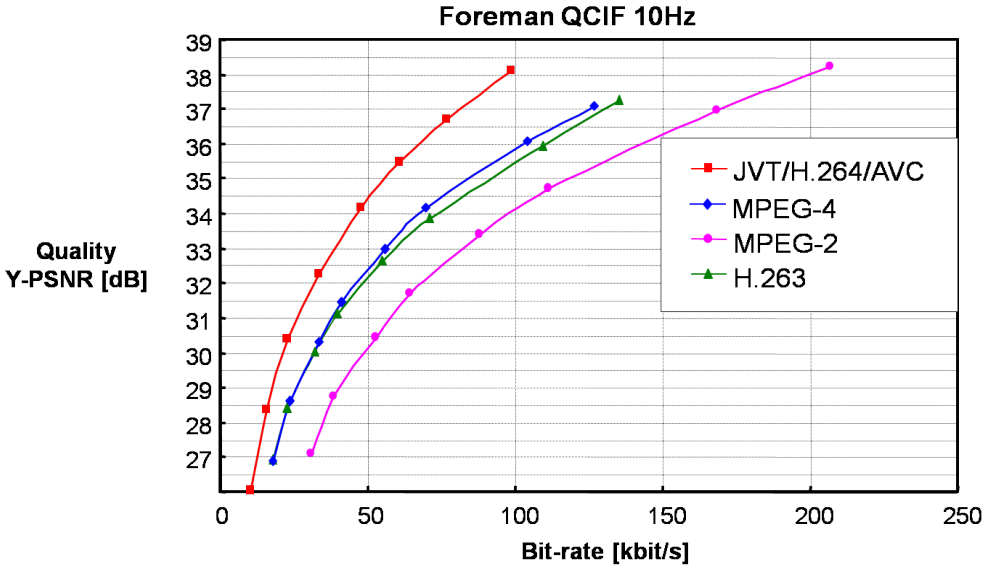

Figura 11: Comparativa formatos de vídeo

Este formato proporciona una calidad de imagen razonable a bajas tasas de bits por segundo, como se puede observar en la Figura 11. Gracias a esto y a su baja complejidad, que implica mayor facilidad de implementación, se ha convertido en el

estándar de vídeo más utilizado actualmente por la industria, superando a su predecesor MPEG-2 y a otros formatos similares como Theora o WebM. Es un formato especialmente adoptado en Internet para la transmisión de vídeo gracias a sus bajas tasas de bits, siendo utilizado por la gran mayoría de sitios dedicados a la transmisión de contenidos en vídeo como pueden ser YouTube o Vimeo.

Para nuestra aplicación de prueba hemos utilizado una versión del códec H.264 para Windows proporcionado por el propio ITU. Este códec convierte un vídeo en formato sin comprimir YUV en un flujo de vídeo comprimido H.264. Permite realizar la caracterización del flujo de salida a través de las opciones que lee de un fichero de configuración, en el cual se le deberán indicar además las características del vídeo de entrada y del vídeo de salida. Este fichero se debe encontrar en la misma carpeta que el ejecutable a menos que se indique otra cosa. Se puede también especificar si el formato de salida será un fichero o un flujo de datos de paquetes RTP.

Las modificaciones realizadas a este software se han centrado en la inicialización del códec, así como en la captura y sincronización de datos.

4 Implementación de la infraestructura propuesta

En este capítulo pasamos a describir el trabajo desarrollado durante el proyecto. En primer lugar describiremos la infraestructura software desarrollada para realizar la simulación de los sistemas embebidos con distintos sistemas operativos y en segundo lugar la aplicación desarrollada para probar el correcto funcionamiento de dicha infraestructura.

4.1 Infraestructura de simulación

La primera parte del desarrollo de este proyecto ha consistido en la extensión de la herramienta de co-simulación nativa SCoPE para poder realizar simulaciones de plataformas software que combinen diversos sistemas operativos. Esto ha requerido de varios pasos.

En primer lugar se ha adaptado el plug-in Win32 para poder ser utilizado en la versión actual de SCoPE. En segundo lugar se han propuesto diferentes técnicas de comunicación entre los distintos sistemas operativos. Por último se han creado los scripts necesarios para generar los tipos de simulaciones propuestos.

4.1.1 Adaptación del plug-in Win32

Como ya se ha dicho anteriormente, para realizar la simulación de código que haga uso de la API Win32 hemos hecho uso del plug-in de SCoPE Win32. Sin embargo, este plug-in ha requerido de ciertas modificaciones y adaptaciones para poder ser utilizado. A continuación describimos las modificaciones que se han realizado, que se han centrado en la adaptación de la versión inicial del plug-in a una versión actual del simulador y en la corrección de bugs.

En primer lugar se ha realizado la adaptación del plug-in Win32 a la última versión de SCoPE, que es la que utilizaremos para llevar a cabo el desarrollo del proyecto. Esto es necesario ya que SCoPE ha sufrido una serie de modificaciones desde que se implementó esta versión inicial del plug-in, con lo que se ha quedado obsoleto y la última versión disponible del plug-in ya no funciona con las últimas versiones del simulador. Esto no quiere decir que la funcionalidad básica del plug-in ya no sea válida, sino que ha requerido ciertas modificaciones respecto a la forma en la que se integra con SCoPE.

Originalmente, para instalar el plug-in no se disponía de un script que realizara esta tarea automáticamente por lo que se ha recurrido a copiar manualmente los diferentes directorios y ficheros del plug-in en los lugares apropiados. Esto ha requerido de un análisis en profundidad de los diferentes componentes del plug-in, de forma que pudiésemos situarlos en los lugares adecuados. Esto ha sido necesario ya que la estructura de directorios de SCoPE ha sufrido diversas modificaciones y por lo tanto las indicaciones de instalación de que disponíamos no eran válidas.

Se ha modificado el compilador de SCoPE (`scope-g++`), para que acepte las opciones específicas del plug-in y realice las operaciones correspondientes a esas opciones.

Las cabeceras incluidas en los ficheros del directorio `{SCOPE_DIR}/scope/rtos/low_level` también han tenido que ser modificadas por problemas de incompatibilidad. Para ello se ha sustituido en todos los ficheros en los que aparece la cabecera `<asm/sigcontext.h>` por `<signal.h>` solucionando de este modo los problemas de compatibilidad que nos generaba el plug-in.

También se ha modificado el script de compilación de SCoPE para que incluya el plug-in Win32 correctamente, incluyendo los directorios apropiados

Para compilar WINE se ha aplicado un parche que corrige un problema con la librería Freetype. Este parche corrige un bug de la versión 1.0.1 de WINE que provocaba que si se compilaba en un sistema con una versión superior a la 2.3.8 de Freetype, el compilador mostraba un error en que indicaba que no se había declarado `FT_MultFix` y una serie de errores derivados de este.

Los scripts de generación de simulaciones que proporciona el plug-in han requerido ciertas modificaciones. Se han modificado para incluir las opciones necesarias así como referencias a las nuevas localizaciones de los directorios de ficheros de cabeceras relativos al plug-in. También se le han añadido referencias a las nuevas librerías de SCoPE `libscope_plus.so`, la cual añade mejoras a la anterior `libscope.a`; y la librería `libcaches.a`, la cual proporciona los modelos y técnicas de estimación de cachés de datos e instrucciones.

Otro elemento que ha requerido modificaciones es el fichero `syscalls.cfg`, en el cual se definen las llamadas a funciones de las distintas APIs que están implementadas en SCoPE y deben ser sustituidas con las funciones correspondientes de SCoPE. En este fichero se han añadido las funciones de la API Win32 que implementa el plug-in.

Además, el plug-in ha requerido de algunas modificaciones internas, así como de la corrección de diversos bug que se han encontrado y que afectaban al correcto funcionamiento del plug-in en ciertos casos.

4.1.2 Comunicación entre sistemas operativos

La comunicación entre los diferentes sistemas operativos es uno de los elementos clave en este tipo de simulaciones, ya que permiten que los diferentes sistemas interactúen entre sí.

En este proyecto se proponen dos aproximaciones diferentes al modelado de la comunicación entre diferentes sistemas operativos:

- La primera está basada en transferencias a través de red. Esta aproximación nos permite modelar comunicaciones entre sistemas en los que los diferentes

sistemas operativos se encuentren en diferentes nodos conectados mediante una red.

- Una segunda aproximación se basa en modelar la utilización de zonas de memoria compartida y se aplicará en plataformas en las que los diferentes sistemas operativos tengan acceso a una misma memoria RAM.

4.1.2.1 Comunicación por Red

La primera técnica que se propone para modelar la comunicación entre múltiples sistemas operativos, es la comunicación por red. Esta se aplicaría en plataformas en las que contemos con diferentes nodos conectados entre sí mediante una red, como por ejemplo la de la Figura 12.

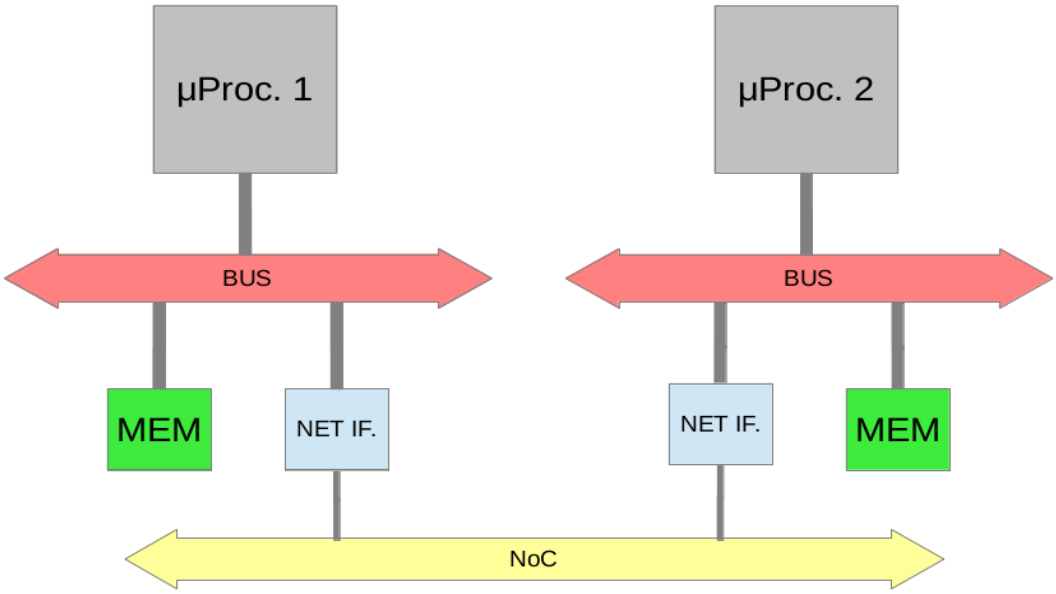

Figura 12: Ejemplo de plataforma con dos nodos conectados por red

El modelo de red de SCoPE nos permite conectar los diferentes nodos de la plataforma mediante una topología de malla tridimensional. Para ello, cuando definimos la plataforma hardware especificamos las coordenadas *x*, *y* y *z* que va a tener un cierto interfaz de red dentro de la topología que estamos definiendo. Este modelo únicamente tiene en cuenta los saltos necesarios para llegar desde el nodo origen al nodo destino sin aplicar protocolos de enrutado o tener en cuenta colisiones o búferes intermedios. Esto, sin embargo, provoca que la sobrecarga en la ejecución provocada por la simulación de la red sea mínima.

En este proyecto se propone un modelo de comunicación basado en transferencia por red a nivel Ethernet. La utilización del protocolo Ethernet reduce el tamaño de los paquetes de red, además de reducir las tareas de cómputo, respecto a otros protocolos como TCP-IP. Esto resulta útil en un sistema embebido, donde hay restricciones en cuanto a recursos disponibles y a consumo.

De este modo se propone crear manualmente los paquetes de datos Ethernet siguiendo el esquema de la Figura 13 y escribirlos directamente en el dispositivo de red. Esto también implica que no se tiene control de flujo y el usuario es responsable de reordenar los paquetes en destino.

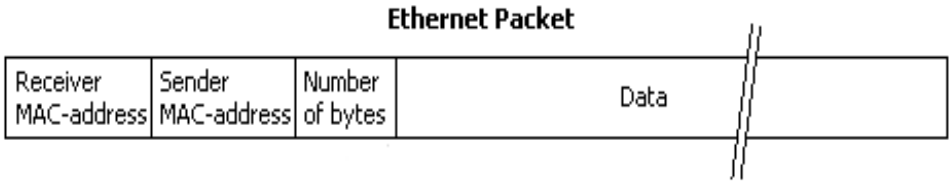

Figura 13: Formato de paquete Ethernet

SCoPE proporciona una HAL basada en Linux. Esto implica que no podemos cargar controladores específicos de Win32. Sin embargo, puesto que en SCoPE la implementación de la HAL es independiente de la API que se está ejecutando sobre un modelo de sistema operativo concreto, podemos cargar el controlador de red TUN/TAP para Linux que nos proporciona SCoPE. De este modo, únicamente se requiere abrir el fichero del dispositivo de red como se haría en Linux y podemos leer y escribir sobre él. Las llamadas bloqueantes siguen funcionando igualmente ya que independientemente del interfaz de sistema operativo que se esté utilizando el encargado de la planificación es el mismo modelo genérico en cualquier caso. Y las transferencias de datos tanto por el bus como por la red son igualmente tenidas en cuenta a la hora de realizar las estimaciones de prestaciones.

Si además se quisiera modelar la comunicación por red a nivel TCP/IP habría que tener en cuenta ciertos detalles. Por ejemplo, para el interfaz POSIX contamos con una implementación de la pila TCP/IP mediante el software lwIP, el cual se ha portado a SCoPE previamente. Éste proporciona acceso a la red a nivel TCP/IP. Sin embargo no se ha portado esta implementación de la pila TCP/IP para el interfaz Win32 por lo que no contamos con acceso a la red a nivel IP. Portar este software al interfaz Win32 podría llevar bastante trabajo.

4.1.2.2 Comunicación mediante memoria compartida

La segunda técnica de comunicación propuesta es la comunicación mediante áreas de memoria compartida. Esta se aplica a cualquier arquitectura en la que se disponga de uno o más procesadores ejecutando distintos sistemas operativos siempre y cuando estos procesadores tengan acceso a una misma memoria, es decir, que estén conectados al mismo bus.

Esto en una plataforma hardware real puede implicar, por ejemplo, que haya dos sistemas operativos con sus respectivas áreas de memoria a las cuales tienen acceso exclusivo y otro área de memoria al cual ambos tengan acceso, como podría ser el caso de la Figura 14.

Para modelar este tipo de comunicación con memoria compartida se ha tenido en cuenta el modo en el funciona el modelo de memoria que proporciona SCoPE. Este

modela las escrituras y lecturas en memoria asignándolas un coste en instrucciones y tiempo y anotando esta información en el código fuente. Este modelo hace que no se vea afectado el comportamiento de las variables a la hora de ejecutar el código de aplicación.

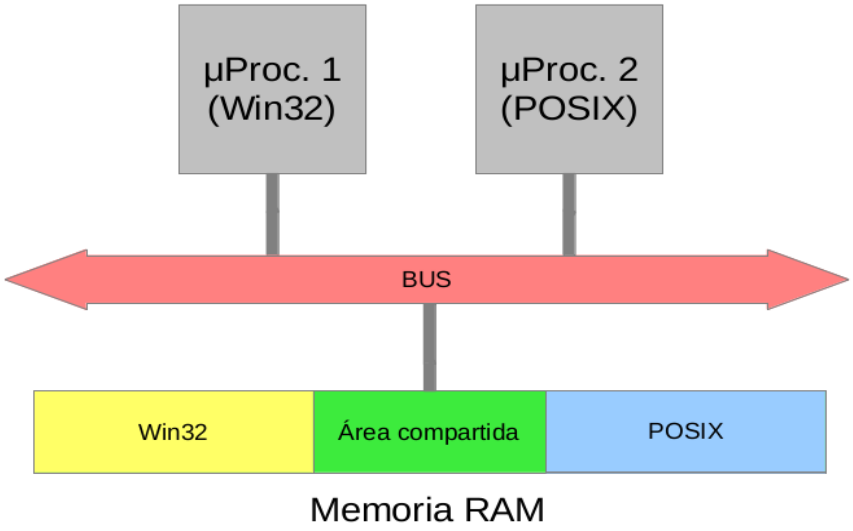

Figura 14: Plataforma con memoria compartida

De este modo, una variable global será visible desde cualquier parte del código de aplicación, independientemente del sistema operativo sobre el que se esté ejecutando dicho código. Por lo tanto, esa variable podrá ser accedida tanto para lectura como para escritura por el código que se ejecute sobre cualquiera de los sistemas operativos que conformen la plataforma software. Esto quiere decir que para modelar una zona de memoria compartida podemos utilizar una variable global.

Utilizando esta técnica podemos modelar zonas de memoria compartida que permiten la comunicación entre diferentes sistemas operativos. Sin embargo no disponemos de funciones de sincronización para variables compartidas entre diferentes sistemas operativos, por lo que también será responsabilidad de programador el asegurarse de que los accesos a estas variables se hagan de forma correcta.

El hecho de que una variable global sea visible por defecto desde cualquier punto del código que se simule no quiere decir que deba ser así siempre, sino que la visibilidad de dicha variable puede ser restringida. Por ejemplo, si estamos ejecutando la misma pieza de software simultáneamente en varios nodos, querremos que las variables globales sean exclusivas de ese nodo. Para esto tenemos dos opciones que podemos usar dependiendo del caso:

- Librerías protegidas. Si compilamos cada pieza de software como una librería protegida, de modo que el ámbito de las variables globales de ese código se verá restringido al código que haya dentro de la librería.

- Modificador `__thread`. Si declaramos una variable global con el modificador `__thread`, esta será exclusivamente visible dentro del thread en que se ha

creado. Esta técnica permite combinar el uso de variables globales visibles desde cualquier lugar del código de aplicación con variables globales exclusivas del nodo en el que se declaren.

Podremos utilizar cualquiera de las opciones según lo requiera la plataforma y consideremos oportuno.

4.1.3 Generación de la simulación de la plataforma HW/SW

SCoPE ofrece una serie de scripts genéricos de compilación mediante la herramienta `make` que permiten generar las diferentes simulaciones a partir del código fuente de la aplicación a simular y el fichero de definición de la plataforma. El objetivo de estos scripts es abstraer al usuario del proceso de generación de las simulaciones, el cual reviste cierta complejidad y requiere de un conocimiento en profundidad del proceso mediante el cual SCoPE compila la aplicación y genera el ejecutable que contiene la plataforma HW/SW junto con el código de aplicación listo para ser simulado.

Sin embargo estos scripts sólo permiten generar una simulación en la que el software de aplicación utilice una única interfaz de sistema operativo o API. Por lo tanto deberemos crear un script que nos permita compilar y generar una simulación en la cual se utilicen diferentes interfaces de sistema operativo.

Para ello, en primer lugar se ha analizado el proceso de generación de de una simulación así como los requisitos de cada parte de esta generación. Este proceso se divide en 4 partes:

- Compilación del software de aplicación mediante `scope-g++`.
- Compilación de los modelos de periféricos mediante `g++`.
- Compilación de la plataforma HW/SW mediante `g++`
- Enlazado de los binarios generados anteriormente para generar la simulación.

Para realizar la compilación del software de la aplicación hay que tener en cuenta que dependiendo de la API que se utilice, el compilador requiere diferentes opciones. Por lo tanto hemos dividido la compilación del software de aplicación en dos partes, una por cada API. En primer lugar utilizamos diferentes opciones del flag `--scope-api`, siendo `posix` o `win32` dependiendo del caso. Además al compilador se le indican los directorios en los que se encuentran los ficheros de cabeceras específicos de cada sistema operativo además de los genéricos de SCoPE que son incluidos en ambos casos.

Esto resulta en los siguientes comandos de compilación, en los cuales se incluye el flag `scope-api` en las variables `SCOPE_FLAGS_POSIX` y `SCOPE_FLAGS_WIN32`, y las variables `SCOPE_POSIX_INC_DIR` y `SCOPE_WIN32_INC_DIR` hacen referencia a los directorios donde se encuentran los

ficheros de cabeceras de POSIX y Win32 respectivamente.

```
# Parse and compile software application files
# POSIX
$(OBJECTS_POSIX):
	$(SCOPE_CXX) $(CFLAGS) $(COBJ) \
	$(SCOPE_FLAGS_POSIX) $(SCOPE_INC_DIR) \
	$(SCOPE_POSIX_INC_DIR) $< -o $@ -lm

# WIN32
$(OBJECTS_WIN32):
	$(SCOPE_CXX) $(CFLAGS) $(COBJ) \
	$(SCOPE_FLAGS_WIN32) $(SCOPE_INC_DIR) \
	$(SCOPE_WIN32_INC_DIR) $< -o $@ -lm
```

Figura 15: Comandos de compilación de software de aplicación

Las variables `OBJECTS_POSIX` y `OBJECTS_WIN32` hacen referencia a los ficheros objetos generados una vez compilado el software de aplicación y que han sido definidos previamente. Estas variables se generan en base a las variables `SOURCES_POSIX` y `SOURCES_WIN32`, las cuales a su vez son obtenidas automáticamente por el script. Para ello requiere que los ficheros de código fuente de Win32 y POSIX estén en las carpetas `win32` y `posix` respectivamente. Si queremos indicarle al script que nuestras fuentes están en diferentes directorios lo podemos hacer mediante los flags `--WIN32_DIR={DIRECTORIO}` y `--POSIX_DIR={DIRECTORIO}` al ejecutar el `make`.

Para compilar de forma automática los modelos de periféricos, en el caso de que los haya, establecemos el requisito de de que los nombres de los ficheros de dichos periféricos se ajusten al patrón `uc_{NOMBRE_PERIFERICO}_hw.cc` de forma que el script los pueda identificar mediante una búsqueda basada en ese patrón y así compilarlos.

```
# Compile peripherals
$(PERIPHERALS).o : (PERIPHERALS).cc
	$(CXX) $(CFLAGS) $(COBJ) $(SCOPE_INC_DIR) $< -o $@
```

Figura 16: Comando de compilación de los periféricos

El último paso de compilación que realizamos es el del fichero `sc_main.cpp`, el cual contiene la definición de la plataforma HW/SW sobre la que se simulará nuestro código de aplicación. En primer lugar un detalle que tenemos que tener en cuenta al definir la plataforma software es que la función que ejecutará el código Win32 requiere ejecutar la función `__win32_plugin_create_main_thread()` antes de llamar a nuestro código de aplicación para así inicializar WINE y el resto de la infraestructura

de la API de Win32. Para compilar la plataforma utilizaremos `g++` al cual le deberemos indicar los directorios de cabeceras de SCoPE como se indica a continuación.

```
sc_main.o : sc_main.cpp
	$(CXX) $(CFLAGS) $(COBJ) $(SCOPE_INC_DIR) $< -o $@
```

Figura 17: Comando de compilación de la plataforma HW/SW

Finalmente enlazamos todos los binarios intermedios que hemos obtenido al compilar los distintos compones con el siguiente comando:

```
run.x: \
$(OBJECTS_POSIX) $(OBJECTS_WIN32) sc_main.o $(PERIPHERALS).o -lm
	$(CXX) $(DEBUG) -pg $(OPT) $(SCOPE_INC_DIR) \
	$(SCOPE_LIB_DIR) $^ -o run.x $(SCOPE_LIB) -lgcov
```

Figura 18: Comando de enlazado para generar el ejecutable de la simulación

4.2 Aplicación multimedia desarrollada

Una vez tenemos la infraestructura que nos permitirá simular plataformas con múltiples sistemas operativos, hemos procedido a crear una simulación con la que probar el correcto funcionamiento de dicha infraestructura.

En primer lugar se describe la aplicación que se ha creado para probar esta infraestructura. Seguidamente se describen los modelos de los periféricos que se han utilizado y los controladores asociados a dichos periféricos. Finalmente se describen las diferentes plataformas hardware sobre las que se ha simulado esta aplicación.

4.2.1 Software de aplicación

La aplicación que se ha desarrollado para probar la infraestructura consta de una parte de captura de datos de audio y vídeo y compresión de ese audio a mp3 sobre un sistema POSIX y una parte de compresión de vídeo a H.264 sobre un sistema Win32.

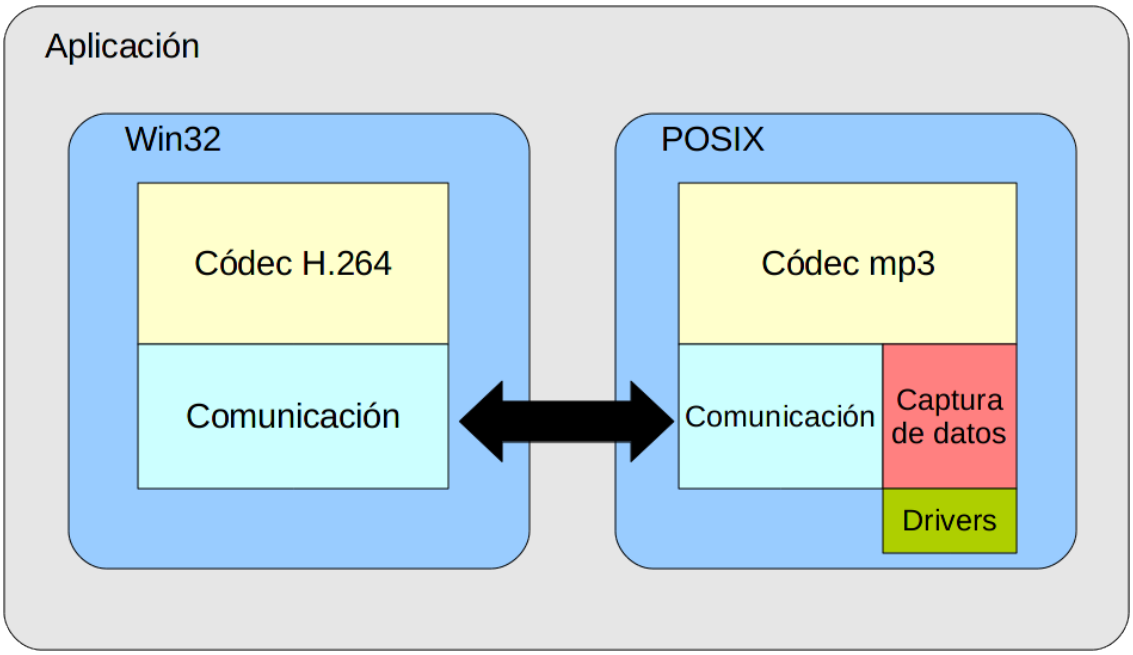

Figura 19: Estructura de la aplicación multimedia

En la Figura 19 se puede observar la estructura general de la aplicación multimedia desarrollada, y cómo se dividen los diferentes componentes de la aplicaicón entre el sistema operativo de tipo Win32 y el de tipo POSIX, además de cómo se comunican entre sí dichos componentes.

En esta sección describiremos la compresión del audio, la compresión del vídeo y la comunicación entre los componentes.

4.2.1.1 Audio

La primera parte de la aplicación de prueba consiste en un codificador de audio, el cual realiza la codificación al formato mp3 a partir de datos en formato WAVE. A continuación pasamos a describir los pasos que se han llevado a cabo para integrar este codificador en la aplicación de prueba.

Como ya se ha dicho anteriormente, para este proyecto se ha buscado un algoritmo de compresión ligero. Por ello se ha decidido utilizar la versión 0.1.4 del software Shine, que es un codificador mp3 creado por Gabriel Bouvigne.

Este codificador no puede ser utilizado directamente por nuestra aplicación, ya que está pensado para ejecutarse como una aplicación independiente. Sin embargo se ha integrado en nuestra aplicación realizando alguna modificación, principalmente en la forma de inicializar el codificador. Seguidamente describimos las modificaciones llevadas a cabo en éste software para adaptarlo a nuestro código.

Lo primero que se ha hecho es eliminar la función `parse_command`, la cual se encarga de recibir la línea de comando con la que se ha ejecutado el codificador y analizarla para configurar el fichero de entrada, de salida así como las diferentes

opciones de codificación que se van a aplicar posteriormente. Como en nuestro caso no se ejecuta este software desde línea de comandos, hemos trasladado esa funcionalidad a la función principal, en la cual especificamos cuales son los ficheros de entrada, en este caso el fichero del dispositivo de captura de audio `/dev/mic`, y de salida, al que llamaremos `test.mp3`. El resto de opciones se establecen por defecto mediante la función `set_defaults`.

A continuación se procede a abrir el fichero de entrada de datos y se lee la cabecera de datos WAVE. Esta lectura se hace de una forma poco eficiente y excesivamente complicada ya que va leyendo byte a byte y comparando los datos con lo que espera leer. Esto puede ser debido a que se le puede pasar como fichero de entrada cualquier tipo de fichero y nada garantiza que sea del tipo correcto. En nuestro caso, puesto que leemos los datos del periférico, en principio tenemos garantizado que lo que vamos a leer son datos válidos y puesto que conocemos cual es el formato de la cabecera de datos, el cual se indica en la Figura 20, hemos modificado la lectura de esa cabecera.

Para ello, en lugar de leer campo a campo, lo que se ha hecho es crear una estructura de datos `wave_header` con el mismo formato que la cabecera de forma que con leer los 44 bytes de la cabecera y escribirlos en la dirección de memoria en la que comienza la estructura de datos se rellenan todos los campos y reducimos el número de lecturas sobre el dispositivo a una únicamente.

También se ha modificado cómo calcula el tamaño de la cabecera y el número de muestras de audio que se van a codificar. Para realizar este cálculo recurría a una serie de llamadas a `fseek` y `ftell`, cuando se puede obtener este dato fácilmente a partir de los datos de la cabecera.

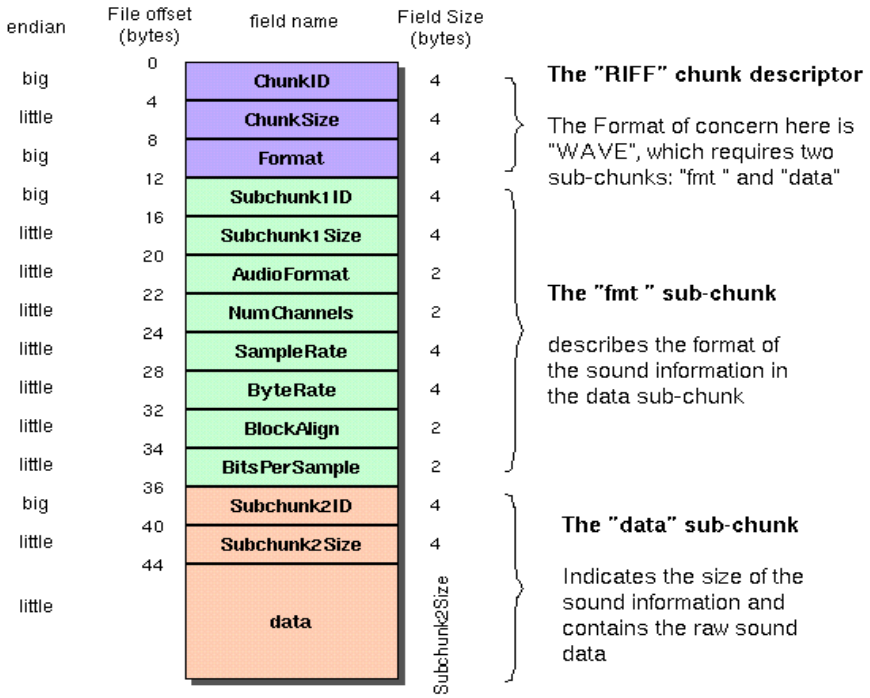

Figura 20: Formato WAVE estándar

A continuación se ha adaptado el código para que lea estos valores de la nueva estructura de datos creada. Y puesto que en POSIX se utiliza una llamada a la misma función para leer de un fichero y para leer un periférico de caracteres a través de su fichero de dispositivo, no requiere realizar ninguna modificación en cómo se leen los datos de audio por parte del código. Únicamente hemos tenido en cuenta que de que no debe hacer llamadas a funciones sobre el fichero que sean distintas de las de apertura, cierre, escritura y lectura, que son las que nos va a proporcionar el controlador.

4.2.1.2 Vídeo

Como ya se ha explicado, para nuestra aplicación de prueba hemos utilizado el códec H.264 proporcionado por el propio ITU. Este códec convierte un vídeo en formato sin comprimir YUV en un flujo de vídeo comprimido H.264. Sin embargo, al igual que el codificador de audio, éste está pensado para ejecutarse de forma independiente y leer los datos de un fichero de entrada. Por ello ha requerido ciertas modificaciones para adaptarlo a la aplicación de prueba que se ha desarrollado.

En primer lugar se ha modificado la forma de captura de datos, ya que será el sistema POSIX el que se encargue de capturar estos datos y enviárselos al sistema Win32 para su posterior codificación. Para ello se ha eliminado el código en que se abría y se cerraba el fichero de entrada de datos.

Además se ha creado una implementación de una triple cola en la que se almacenan los tres componentes de los cuadros de vídeo a medida que se van recibiendo y de donde los lee el códec a medida que va codificando el flujo de vídeo. También se ha sincronizado el acceso a estas colas mediante tres objetos `CriticalSection` que garantizan un acceso seguro a las colas por parte del códec y del thread que recibe los datos de vídeo.

Otro elemento de sincronización que se ha añadido es el uso de tres semáforos que bloquean el thread del códec en el caso de que no haya datos que leer. De ese modo, el códec espera a que esos datos sean recibidos y colocados en sus respectivas colas para leerlos y codificarlos.

También se han definido los valores `TRUE` y `FALSE`, así como el tipo booleano `bool` que la librería de C utilizada por nuestro compilador no tenía definidos.

4.2.1.3 Comunicación

Con esta aplicación queremos probar los dos tipos de comunicación propuestos:

- Comunicación por red

En este caso hemos implementado un thread que se encarga de ir leyendo los datos de vídeo del dispositivo `/dev/cam` y de enviárselos al códec de vídeo por red.

Como ya hemos dicho anteriormente, para comunicar un sistema POSIX y uno Win32 por red mediante SCoPE, hay que componer manualmente los paquetes Ethernet y escribirlos en el periférico de red.

Los componentes de los cuadros de vídeo tienen un tamaño de 25.344 bytes en el caso del componente Y, y 6.336 en el caso de los componentes U y V. Esto implica que hayamos tenido que dividir cada componente en diversos paquetes, puesto que el *payload* o tamaño de los datos de un paquete Ethernet es de 1500 bytes como máximo.

La comunicación a nivel Ethernet no tiene control de flujo, por lo que hemos añadido a cada paquete un número indicando su orden para que sean reordenados en destino y reconstruir correctamente el componente completo del cuadro de vídeo antes de ser insertados en las colas de componentes que lee el códec. Además el nodo Win32 que recibe los datos envía un paquete a modo de señal ACK al nodo POSIX cada vez que termina de recibir, reordenar y encolar un componente completo para que proceda a enviar el siguiente componente.

- Comunicación por zonas de memoria compartida

Como se ha propuesto anteriormente, para comunicar dos sistemas operativos mediante zonas de memoria compartida, se han utilizado variables globales que actúan como tales zonas de memoria.

En este caso se han creado cinco variables globales: una por cada componente de vídeo, otra para indicar que un cuadro está completo y una última para indicar que ese cuadro ha sido leído.

El thread POSIX lee las tres componentes de vídeo de un cuadro, las escribe en las variables compartidas y actualiza la variable que indica que hay un nuevo cuadro preparado escribiendo el número del cuadro actual. Una vez el thread Win32 comprueba que hay un nuevo cuadro lo lee e indica que ya lo ha leído. Cada vez que el thread POSIX va a escribir un nuevo cuadro, reinicia la variable que indica que el cuadro ha sido leído.

En este caso no disponemos de funciones u objetos que nos garanticen una correcta sincronización entre ambos sistemas para realizar la lectura y escritura de las variables compartidas. Por ello se ha utilizado una técnica de *polling* para saber cuando un cuadro nuevo está disponible y cuando éste ha sido leído.

4.2.2 Periféricos de audio y vídeo

Para la simulación de la aplicación de prueba, los datos de vídeo y audio son obtenidos a partir de sendos periféricos que forman parte de la plataforma hardware modelada. Sin embargo, en un principio no disponemos de los modelos de los periféricos que se van a encargar de realizar la captura de audio y vídeo respectivamente. Por lo tanto se han implementado los periféricos necesarios.

Para ello se ha hecho uso de las herramientas que nos proporciona SCoPE y que nos permiten crear modelos de periféricos en SystemC que incluyan las funcionalidades básicas de los periféricos que estamos modelando.

En este caso, los periféricos que vamos a implementar serán del tipo esclavo, y por lo tanto no tendrán soporte para interrupciones. Para ello deberemos utilizar el wrapper

`uc_hw_if`. Para un periférico de este tipo, SCoPE proporciona un wrapper que nos permite generarlo extendiendo la clase `UC_hw_if` y redefiniendo las funciones `read(ADDRESS addr, DATA data, unsigned int size)` y `write(ADDRESS addr, DATA data, unsigned int size)` como se muestra en la Figura 21. De este modo permiten definir en el cuerpo de esas funciones, las cuales se corresponden con las funciones de lectura y escritura de los registros de datos y de control respectivamente, la interacción de esos dispositivos con el bus.

```
#ifndef _UC_IO_HW_H_
#define _UC_IO_HW_H_

#include "sc_scope.h"

class uc_IO_hw : public UC_hw_if
{
 sc_time response_time;

public :

  uc_IO_hw(sc_module_name mod_name, unsigned int begin, unsigned int end, int ret);

  int read(ADDRESS addr, DATA data, unsigned int size);
  int write(ADDRESS addr, DATA data, unsigned int size);

};

#endif /* _SLAVE_H_ */
```

Figura 21: Cabecera de periférico de ejemplo uc_IO_hw.h

Periférico de vídeo:

Este periférico tiene un funcionamiento relativamente básico. Permite capturar audio en formato YUV QCIF. En este formato cada imagen es de tamaño Quarter-CIF, es decir, 176x144 pixels y está compuesta por 3 componentes diferentes:

- Componente Y: Contiene la luminosidad de la imagen o *luma* en blanco y negro. Su tamaño es el mismo que el de la imagen.

- Componentes U y V: Contienen la crominancia de la imagen, es decir, la información sobre el color de la imagen. En este caso su tamaño es un cuarto del tamaño de la componente Y.

Para simular la captura de vídeo mediante una cámara lo que hace es leer el vídeo del fichero `coastguard.yuv`, el cual contiene un vídeo en formato YUV QCIF.

En primer lugar se han creado los correspondientes ficheros de código y cabeceras (`uc_cam_hw.cc` y `uc_cam_hw.h`). Una vez hecho esto, se ha pasado a codificar la funcionalidad del periférico. Para ello en primer lugar se realiza la inicialización en el constructor de la clase. Esta inicialización se lleva a cabo cuando se crea la instancia del periférico en el fichero `sc_main.cpp` donde se define la plataforma hardware. En este caso lo que se hace es realizar la apertura del fichero que utilizaremos como fuente

para nuestro flujo de vídeo y mostramos un mensaje de error en caso de no poder abrir el fichero.

A continuación se ha definido en las funciones `read` y `write` la funcionalidad del periférico. Como queremos mantener el modelo simple, en la función `read` lo único que hacemos es leer del fichero tantos bytes como se nos indique en los parámetros de la función. Mientras tanto, en el modo `write` lo que hacemos es utilizarlo como función de reinicio. Para ello en caso de que se escriba un 1 en el periférico reiniciamos la lectura del fichero con un `fseek` a la posición 0. Para cualquier otro valor de escritura no hacemos nada.

Periférico de audio:

Para este periférico realizaremos un modelo muy similar al de vídeo. En este caso leeremos el audio del fichero `test.wav` en formato WAVE. Los ficheros de audio WAVE están formados por una cabecera de tamaño variable, aunque normalmente es de 44 bytes, y los datos de audio a continuación divididos en muestras.

De este modo, procedemos igual que con el periférico de vídeo, con la diferencia de que en este caso gestionamos el reinicio del periférico de diferente modo. Es decir, como en este caso el flujo de datos tiene una cabecera que en un dispositivo real sería generada por el propio periférico o por el controlador, dependiendo del caso, lo que hace es definir dos tipos de reinicio. Uno cuando se escriba el valor 1 en el periférico, en el cual no se volverían a leer los datos de la cabecera sino que situaríamos el punto de lectura al principio de los datos de audio; y otro cuando se escriba el valor 2, en el cual el reinicio sería completo y se situaría el punto de lectura al principio del fichero leyendo de este modo la cabecera de nuevo. Al igual que en el periférico de vídeo, en el constructor realizamos la inicialización que en este caso también consiste en la apertura del fichero que utilizamos como fuente de nuestro flujo de datos de audio.

En este caso los ficheros generados son `uc_mic_hw.cc` y `uc_mic_hw.h`.

4.2.3 Controladores de audio y vídeo

Una vez tenemos los periféricos que nos permiten simular la captura de datos de vídeo y audio desde nuestra plataforma, necesitamos unos controladores para esos periféricos que nos faciliten su utilización desde el software de aplicación. SCoPE nos proporciona una implementación de la HAL de Linux que nos permite cargar y utilizar controladores de dispositivos de bloque.

No se ha pretendido crear unos controladores excesivamente complejos, sino que únicamente se han implementado las funciones básicas que nos permiten el acceso a los periféricos. Por ello nos hemos centrado en las partes básicas que deben contener estos controladores y que son las siguientes:

- Función de inicialización.
- Función de finalización.

- Estructura de datos `file_operations`.
- Funciones de apertura y cierre (`open` y `close`).
- Funciones de lectura y escritura (`read` y `write`).

En la función de inicialización lo que se ha realizado en ambos casos es asignarles los números *major* y *minor*, seguidamente creamos el fichero del dispositivo con la función `uc_mknod` y finalmente registramos el dispositivo junto con sus funciones mediante la función `register_chrdev`. Como se puede observar no utilizamos la función `mknod`, sino que utilizamos una versión de esta que nos permite conectar el controlador al modelo de HAL que nos proporciona SCoPE. El hecho de que haya que utilizar esta versión de la función `mknod` es debido a que los controladores no se compilan mediante `scope-g++` sino mediante `g++`. Esto implica que no se hace un preprocesado del código, que es donde al compilar el código de aplicación se sustituyen las llamadas a funciones del sistema por las llamadas equivalentes a funciones nativas de SCoPE. De este modo si queremos utilizar en SCoPE un controlador implementado previamente para un sistema real es posible que se necesite realizar modificaciones en las llamadas a ciertas funciones del sistema.

```
int mic_init(void)
{
      dev_t dev_n = MKDEV(4, 0);
      uc_mknod("/dev/mic", S_IFCHR, dev_n);
      register_chrdev(4, "/dev/mic", &mic_fops);

      return 0;
}

void mic_cleanup(void)
{
      unregister_chrdev(MIC_MAJOR, "/dev/mic");
}
```

Figura 22: Funciones de inicialización y finalización del driver del micrófono

En la función de finalización lo único que hacemos es eliminar el fichero del dispositivo borrándolo del registro mediante la función `unregister_chrdev`.

A continuación definimos la estructura de datos `file_operations` en la que indicamos las funciones que este controlado implementa y que estarán disponibles en el espacio de usuario para ser utilizadas por el software de aplicación. En ella indicamos que éste únicamente contará con las funciones "`open`, `close`, `read` y `write`" asociándolas con las funciones correspondientes del controlador tal y como se muestra en la Figura 23.

Seguidamente procedemos a definir las funciones `open` y `close`, las cuales estarán vacías y únicamente retornarán un `int` con valor 0. Esto es debido a que no es necesario que tengan ninguna funcionalidad pero es necesario que estén definidas para poder abrir y cerrar los dispositivos, y de este modo ser utilizados por el software de aplicación.

```
static struct file_operations mic_fops = {
      NULL,
      NULL,
      mic_chr_read,
      mic_chr_write,
      NULL,
      NULL,
      mic_chr_open,
      NULL,
      mic_chr_close
};
```

Figura 23: Estructura "file_operations" del controlador del micrófono

Una vez tenemos las funciones de apertura y cierre procedemos a definir las funciones de escritura y lectura. Para esto en primer lugar realizamos la función de escritura, ya que únicamente tendrá que escribir un valor en en el registro de control de los periféricos. Para realizar esta escritura hacemos uso de la función `uc_iowriteBurst` que nos proporciona SCoPE y que simula escrituras de datos en un periférico a través del bus y de este modo se incorporan a las estimaciones globales los resultados de estimación de estas escrituras en el periférico.

```
static ssize_t cam_chr_write(struct uc_file * file, const char *
buf, size_t count, loff_t *pos) {
      int char_wrote;
      char_wrote = uc_iowriteBurst((void**)&buf, count, (void*)
IO_START + 0x2F);
      return char_wrote;
}
```

Figura 24: Función de escritura del controlador del micrófono

A continuación definimos las funciones de escritura de los controladores. Estas funciones son similares en ambos casos, leyendo los datos del periférico mediante la función `uc_ioreadBurst`, la cual modela las lecturas de datos de los registros de un periférico.

Por último compilamos los controladores. Para ello se han copiado en el directorio de controladores de SCoPE y se ha ejecutado el script de compilación que se proporciona a tal efecto. Una vez hecho esto ya están disponibles para ser cargados

desde la simulación.

4.2.4 Arquitectura hardware

En este proyecto se han definido tres arquitecturas diferentes sobre las que se ha simulado la aplicación que hemos desarrollado previamente.

En primer lugar definimos la arquitectura que llamaremos “plataforma A”, la cual está compuesta por dos procesadores conectados a sendos buses y sus respectivas memorias RAM. Ambos buses tienen conectados además un interfaz de red cada uno que les permite conectarse a una red común. Además uno de los nodos tiene conectados dos periféricos: una cámara que se encarga de capturar vídeo, y un micrófono que se encarga de capturar audio. En la Figura 25 podemos observar un esquema básico de esta arquitectura propuesta.

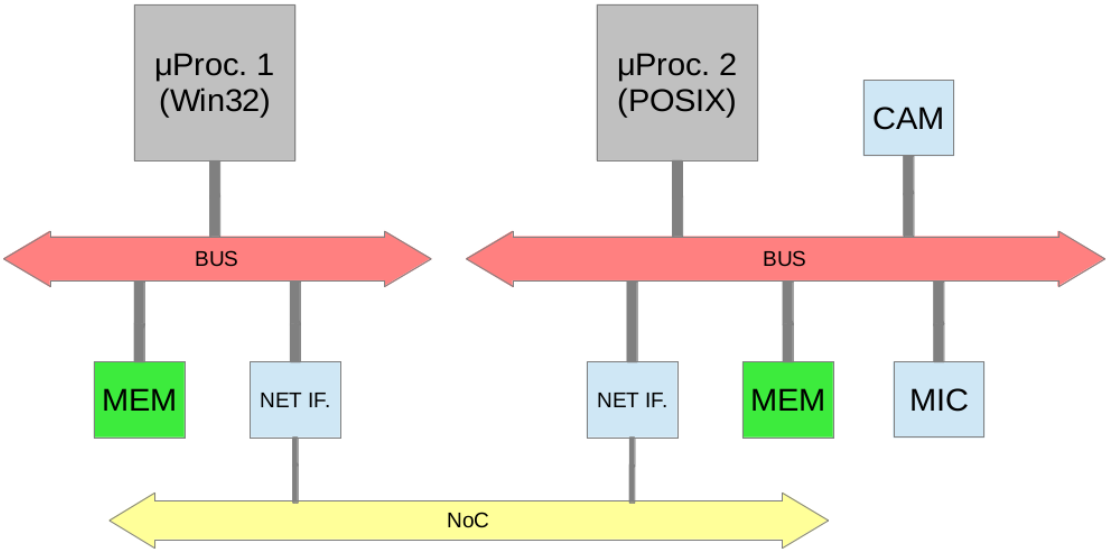

Figura 25: Arquitectura de plataforma A

En esta plataforma la comunicación entre los diferentes procesadores se realiza a través de la red. Para ello SCoPE nos proporciona una implementación de un interfaz de red genérico. Este interfaz maneja las peticiones de escritura y lectura generadas por cada nodo transformando los datos recibidos en las estructuras apropiadas del modelo de red.

La siguiente arquitectura que se propone, y que denominaremos “plataforma B”, es una arquitectura en la que también contamos con dos procesadores, pero en este caso ambos estarán conectados al mismo bus, el cual deberán compartir al igual que la memoria RAM. Puesto que en esta arquitectura se puede realizar la comunicación entre los procesadores mediante la memoria prescindiremos de la red y por lo tanto de los interfaces de red. Esta arquitectura contará asimismo con los periféricos de captura de video y audio con los que contaba la “plataforma A”. En la Figura 26 podemos ver el esquema de la plataforma propuesta.

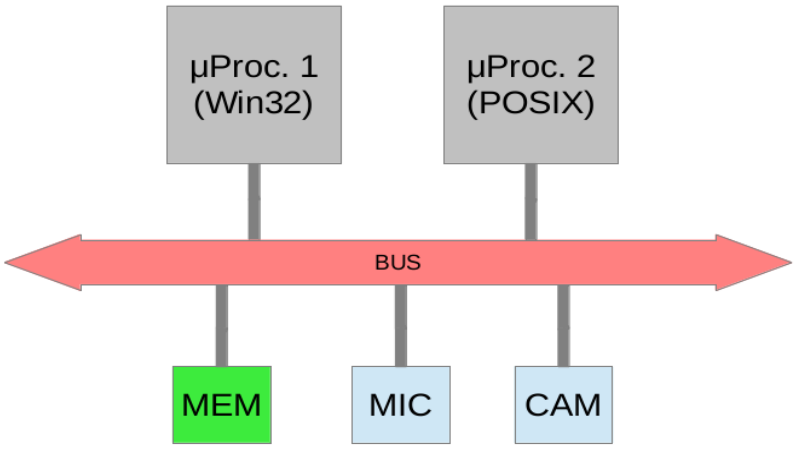

Figura 26: Arquitectura de la plataforma B

Por último se propone una tercera arquitectura, la cual denominaremos "plataforma C". Ésta será la más sencilla de las tres, y sólo dispondrá de un procesador sobre el cual se ejecutarán ambos sistemas operativos. También dispone, al igual que la plataforma B, de una memoria RAM y los dos periféricos de captura de datos. Podemos ver esta arquitectura en la Figura 27.

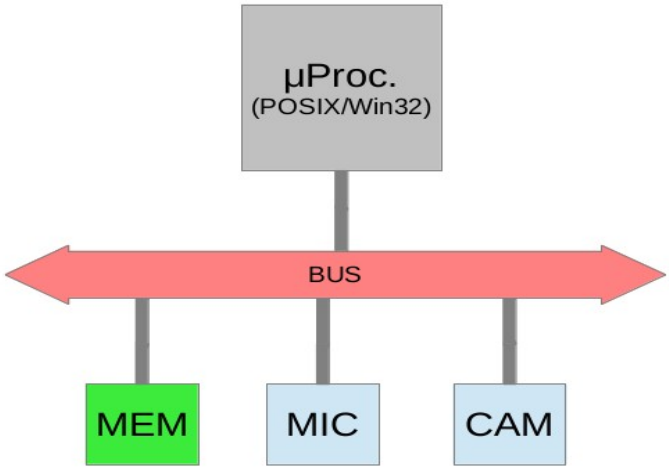

Figura 27: Arquitectura de la plataforma C

Los procesadores que se utilizarán en las tres plataforma son procesadores ARM 926T, mientras que el bus será un bus basado en la especificación AMBA (Advanced Microcontroller Bus Architecture), la cual ha sido desarrollada por ARM y es ampliamente utilizada en sistemas SoC que utilizan procesadores ARM.

5 Evaluación y pruebas

Para evaluar la infraestructura desarrollada se ha utilizado la aplicación de prueba que se ha implementado previamente. También se han realizado una serie de pruebas para comparar SCoPE con la herramienta OVP.

5.1 Simulación de las plataformas propuestas

En este apartado analizamos los resultados obtenidos de la simulación de la aplicación de captura y codificación de audio y vídeo sobre las diferentes plataformas HW/SW propuestas.

En primer lugar se comprobó que todos los modelos de plataformas HW/SW, junto con el código de aplicación correspondiente compilaban correctamente. Una vez hecho esto sólo restaba ejecutar las simulaciones para obtener las estimaciones de rendimiento y posteriormente analizar esos datos.

Antes de simular la aplicación completa, se han simulado independientemente el codificador de audio y el de vídeo. Esto nos ha permitido ver que tanto la simulación de código POSIX como la de código Win32 funcionan correctamente. Estas dos simulaciones se han hecho sobre un modelo de plataforma hardware básica compuesta por un procesador, un bus y una memoria. Los datos obtenidos de estas simulaciones también nos son de utilidad para compararlos con los resultados de estimaciones de la aplicación completa y comprobar si las estimaciones obtenidas son coherentes.

La Tabla 1 contiene los resultados más relevantes de estimaciones de prestaciones de dichos códigos. En ella se muestran las estimaciones de tiempos de ejecución, número de instrucciones ejecutadas, utilización de las cachés de datos e instrucciones, utilización del bus, así como consumos de los diferentes componentes. Como se puede observar, la codificación de vídeo tiene un mayor coste computacional que la codificación de audio. Esto se debe a la mayor cantidad de datos necesarios para representar el flujo de vídeo y a la mayor complejidad del algoritmo de codificación H.264 respecto al algoritmo de codificación mp3. Y como es lógico, esto también repercute en un mayor consumo y una mayor utilización del bus por parte del codificador H.264. También podemos observar que en proporción al número de instrucciones ejecutadas, el codificador mp3 produce un número de fallos en caché mucho menor que el codificador H.264.

	Mp3 (POSIX)	H.264 (WIN32)
Tiempo de ejecución	16,2018 s	51,7270 s
Tiempo de usuario	15,9944 s	50,6800 s
Instrucciones ejecutadas	1.727.735.550	5.012.486.600
Consumo del procesador	43,1934 mW	148,0050 mW
Misses caché de instrucciones	164.399	4.763.920
Consumo caché de instrucciones	64,8723 mW	224,5320 mW
Hits caché de datos	755.796.977	16.669.071.861
Misses caché de datos	1.144.472	4.763.920
Write-backs caché de datos	1.574	7661
Consumo caché de datos	28,9146 mW	76,7380 mW
Bytes transferidos por el bus	47.118.240	303.673.760

Tabla 1: Estimaciones de prestaciones de los codificadores de audio y vídeo

Una vez simulados por separado los componentes de la aplicación pasamos a realizar la simulación de la aplicación completa. En este caso sobre la plataforma A, la cual consta de dos nodos conectados por red. La ejecución se realizó correctamente y los resultados obtenidos los podemos ver en la Tabla 2.

	Mp3 (POSIX) [Proc. 0]	H.264 (WIN32) [Proc. 1]
Tiempo de ejecución	64,7175 s	65,0000 s
Tiempo de usuario	16,9054 s	47,7374 s
Instrucciones ejecutadas	1.817.693.359	4.703.040.256
Consumo del procesador	55,9290 mW	144,7090 mW
Misses caché de instrucciones	216.160	4.133.767
Consumo caché de instrucciones	84,0266 mW	217,5340 mW
Hits caché de datos	783.130.040	1.566.261.315
Misses caché de datos	1.258.151	4.647.546
Write-backs caché de datos	1.665	7.433
Consumo caché de datos	36,9187 mW	75,1490 mW
Bytes transferidos por el bus	55.524.142	296.393.218

Tabla 2: Estimaciones de prestaciones de la plataforma A

Estos resultados son coherentes con las estimaciones obtenidas de las diferentes partes del código por separado. El procesador 0, en el cual se ejecuta el codificador

mp3 y la captura de datos de los periféricos presenta un mayor número de instrucciones ejecutadas, consumos, etc. Esto es debido al hecho de que ahora no sólo realiza la codificación de vídeo, sino la captura de los datos y el envío de los datos de vídeo por red. Sin embargo observamos que para la parte del codificador de vídeo H.264 obtenemos unas estimaciones inferiores a las obtenidas ejecutándolo independientemente. Esto es debido a que cuando se simuló el código H.264 por separado se hizo con la versión original sin modificar. En esta versión se ha modificado la forma en que el código obtiene los datos de vídeo, además de alguna otra optimización realizada en el código, lo cual repercute en mejores resultados.

A continuación realizamos la simulación de la plataforma B, en la cual ambos sistemas operativos comparten el mismo bus y la misma memoria, por lo que realizan la comunicación mediante un área de memoria compartida. El resultado es el que podemos ver en la Tabla 3.

	Mp3 (POSIX) [Proc. 0]	H.264 (WIN32) [Proc. 1]
Tiempo de ejecución	60,2900 s	63,5173 s
Tiempo de usuario	16,9043 s	48,7481 s
Instrucciones ejecutadas	1.817.656.488	4.791.738.151
Consumo del procesador	45,4414 mW	119,7930 mW
Misses caché de instrucciones	215.491	4.241.609
Consumo caché de instrucciones	68,2699 mW	181,8110 mW
Hits caché de datos	783.124.981	1.611.165.894
Misses caché de datos	1.257.864	4.764.682
Write-backs caché de datos	1.571	7.629
Consumo caché de datos	29,9961 mW	62,8011 mW
Bytes transferidos por el bus	356.957.036	

Tabla 3: Estimaciones de prestaciones de la plataforma B

En este caso, las estimaciones de rendimiento obtenidas son similares a las obtenidas en el modelo de plataforma con comunicación por red. Esto se debe a que las modificaciones en el código son mínimas. Únicamente se cambia la forma de comunicarse a un modelo de comunicación por memoria compartida. Los tiempos de ejecución son similares ya que, tanto en la comunicación por red como en la comunicación por memoria compartida, los datos están disponibles para el código más rápido de lo que son procesados. Por lo tanto en este ejemplo no se aprecia el impacto que pueda tener el utilizar un tipo de comunicación u otro en otro código en el cual la disponibilidad de los datos por parte del código sea más crítica. Sin embargo, este ejemplo nos sirve para comprobar el correcto funcionamiento del modelo de comunicación propuesto.

Finalmente realizamos la simulación de la plataforma C, en la cual ambos sistemas

operativos se ejecutan sobre el mismo procesador. Los resultados obtenidos se muestran en la Tabla 4.

	Mp3 (POSIX)	H.264 (WIN32)
Tiempo de ejecución	68,1958 s	
Tiempo de usuario	16,9065 s	48,7455 s
Instrucciones ejecutadas	6.609.391.821	
Consumo del procesador	165,2340 mW	
Misses caché de instrucciones	4.460.882	
Consumo caché de instrucciones	250,0830 mW	
Hits caché de datos	2.394.290.039	
Misses caché de datos	6.016.702	
Write-backs caché de datos	9.186	
Consumo caché de datos	92,7942 mW	
Bytes transferidos por el bus	356.861.804	

Tabla 4: Estimaciones de prestaciones de la plataforma C

En este último caso, también comprobamos que los datos son coherentes con los resultados obtenidos anteriormente. Se puede observar que el tiempo de ejecución es superior en este caso. Esto es debido a que ambos sistemas operativos están compartiendo un mismo procesador y por lo tanto esto afecta a sus prestaciones en cuanto a tiempo de ejecución. Sin embargo, si analizamos los tiempos de ejecución efectivos por separado del código POSIX y el código Win32 vemos que son similares a los obtenidos anteriormente. Los consumos apenas se ven incrementados, al igual que el uso de las cachés o del bus.

5.2 Comparativa con otras herramientas (OVP)

Por último, se ha realizado una comparación entre SCoPE y OVP. Esta comparativa no se ha podido realizar desde el punto de vista de simulación de plataformas software con múltiples sistemas operativos. Esto es debido a que OVP no da soporte a este tipo de simulaciones. Por ello se han simulado diversos fragmentos de código, tanto en OVP como en SCoPE. La principal motivación de esta prueba es comparar SCoPE con una herramienta comercial y verificar que las estimaciones proporcionadas por SCoPE son similares a las proporcionadas por una herramienta de simulación mediante traducción binaria.

Para ello se han realizado una serie de simulaciones de diferentes aplicaciones sobre SCoPE y sobre OVP. Sobre SCoPE se han realizado dos simulaciones por cada aplicación, utilizando el cross-compilador `arm-none-linux-gnueabi-gcc` en la primera, y `arm-elf-gcc` en la segunda. Normalmente se ha utilizado este último,

pero para la comparativa con OVP se ha utilizado el primero también debido a que el sistema operativo Linux ejecutado por el modelo de OVP requiere que el código sea compilado con este. Para ello ha habido que modificar el analizador de código ensamblador de SCoPE para que este realizara las anotaciones de información de prestaciones de forma correcta.

La versión de OVP utilizada es la 1.4.13. En ella se ha cargado el modelo de la plataforma hardware comercial ARM Integrator CP, la cual consta de los siguientes componentes:

- SoC CM9x6
  - ARM926 Core
  - Controlador de Interrupciones
  - UART
  - Memoria RAM de 128 MBytes
- Placa base
  - Controlador LCD
  - Interfaces Ratón/Teclado
  - Memoria Flash

Sobre esta plataforma se ha cargado un Linux en cuya imagen se han incluido los diferentes códigos a simular.

Para obtener el número correcto de instrucciones de la aplicación a analizar, en primer lugar se ha realizado la simulación de arranque y apagado automáticos de la plataforma. De ese modo obtenemos el número de instrucciones que necesita la plataforma para arrancar y apagarse. Una vez obtenido este valor podemos restarlo al total obtenido de las simulaciones con código de aplicación, en las cuales se arranca el sistema, se ejecuta la aplicación, y se apaga el sistema. De este modo se obtiene el número de instrucciones ejecutadas por la aplicación.

En primer lugar se ha simulado la parte de codificación de mp3 de la aplicación de prueba. Se ha eliminado la captura de datos mediante periféricos. En segundo lugar se ha extraído la función `filter_subband` del codificador y se ha simulado independientemente. Además se han simulado el benchmark Linpack y un código que calcula la sucesión de Fibonacci. En los resultados de la Tabla 5 se muestran el número de instrucciones simuladas por cada herramienta para los distintos códigos de prueba.

	Codif. mp3	filter_subband	Linpack	Fibonacci
OVP	6.855.608.799	3.087.849.593	332.441.460	509.594.902
SCoPE (1)	6.663.837.555	3.564.198.184	317.047.743	589.582.378
SCoPE (2)	1.727.735.562	1.067.749.898	289.404.288	716.654.732

Tabla 5: Número de instrucciones simuladas por OVP y SCoPE

(1) Compilado con `arm-none-linux-gnueabi-gcc`
(2) Compilado con `arm-elf-gcc`

Con esto, se obtienen los siguientes porcentajes de error con SCoPE respecto a OVP:

	Codif. mp3	filter_subband	Linpack	Fibonacci
Error (%)	2,80	15,43	4,63	15,70

Tabla 6: Porcentajes de error de SCoPE (2) respecto a OVP

En la Tabla 7 se muestran los tiempos de simulación para los diferentes códigos de aplicación, tanto con SCoPE como con OVP. En la Tabla 8 se muestran los tiempos de ejecución de los mismos códigos en SCoPE pero en este caso modelando las cachés de datos e instrucciones.

	Codif. mp3	filter_subband	Linpack	Fibonacci
OVP	29,84	13,20	11,95	12,85
SCoPE (1)	22,21	11,74	1,66	0,88
SCoPE (2)	7,21	4,24	1,62	3,17

Tabla 7: Tiempos de simulación de SCoPE y OVP en segundos

	Codif. mp3	filter_subband	Linpack	Fibonacci
SCoPE (1)	31,81	14,56	1,63	1,03
SCoPE (2)	18,81	5,75	1,91	3,60

Tabla 8: Tiempos de simulación de SCoPE con modelado de cachés

Como indican estos resultados, las estimaciones proporcionadas por SCoPE son suficientemente precisas como para ser utilizadas en etapas tempranas de diseño. Respecto a los tiempos de ejecución mostrados en la Tabla 7, todos han sido más rápidos con SCoPE que con OVP. Sin embargo, en el codificador mp3 y la función `filter_subband` los tiempos de simulación han sido similares. Esto se debe a una particularidad del compilador.

A diferencia del compilador `arm-elf-gcc`, `arm-none-linux-gnueabi-`

`gcc` utiliza llamadas a funciones de la API ARM EABI para realizar ciertas operaciones en punto flotante. El problema de esto es que si esas operaciones son codificadas por el compilador en código ensamblador directamente, sin realizar una llamada a otra función de una librería, ese número de instrucciones está incluido en la anotación del bloque de código al que corresponda. De este modo, cada vez que se ejecute ese bloque de código se suman el total de instrucciones del bloque. Sin embargo, al realizar llamadas a esa API, la cual no se puede anotar directamente, hay que indicarle a SCoPE que cuando encuentre esa llamada sume las instrucciones correspondientes. Esto produce un overhead mayor en bucles con muchas iteraciones y pocas instrucciones, de las cuales la mayoría producen llamadas a esas funciones. Ese es el caso de la función `filter_subband`, y por extensión del codificador mp3, ya que esa es la función que corresponde al 39,57% de la ejecución de dicho codificador.

Sin embargo SCoPE tiene otras ventajas, ya que además del número de instrucciones y el tiempo de ejecución, proporciona estimaciones de utilización de cachés, consumo energético, uso del bus, tiempo por thread o número de interrupciones. Además los modelos utilizados por OVP son más complejos que los de SCoPE, por lo que requiere mayor trabajo implementar un nuevo modelo de un procesador o periférico para OVP. SCoPE también evita tener que portar el sistema operativo a la plataforma que se quiera simular, lo cual supone un ahorro en costes.

6 Conclusiones

En este proyecto se ha extendido la herramienta SCoPE para que ésta realice co-simulaciones HW/SW de sistemas embebidos cuya plataforma software conste de más de un sistema operativo. Para ello se ha adaptado el plug-in Win32, que da soporte a dicha API, a la versión más reciente de SCoPE y se ha probado su correcto funcionamiento con una aplicación de prueba. También se han propuesto dos métodos de comunicación entre los diferentes sistemas operativos, mediante red y memoria compartida, y se han creado los scripts necesarios para generar las simulaciones para el tipo de plataformas propuesto.

Además se ha implementado una aplicación compleja de prueba que nos ha permitido probar la infraestructura desarrollada. Esta aplicación consiste en la captura y codificación de datos de audio y vídeo. Los diferentes componentes de la aplicación se han mapeado sobre un sistema operativo POSIX y otro Win32 ejecutándose sobre diferentes plataformas hardware. De este modo se han utilizando y probado las dos técnicas de modelado de comunicación propuestas, dependiendo de las características de las plataformas simuladas.

La aplicación de prueba captura los datos de audio y vídeo mediante dos periféricos implementados a tal efecto, con lo que se ha probado el funcionamiento de modelos de periféricos en la infraestructura desarrollada. De este modo, también se ha probado el funcionamiento de los controladores de dichos dispositivos, así como el controlador del interfaz de red, el cual se ha podido utilizar también en el sistema Win32 pese a ser un controlador Linux.

Finalmente, una vez comprobado el correcto funcionamiento de la infraestructura

con la aplicación de prueba, se han llevado a cabo una serie de pruebas con diferentes códigos de aplicación POSIX sobre SCoPE y la herramienta comercial OVP. Con ello se ha realizado una comparativa en la que se ha observado que SCoPE proporciona unas estimaciones de número de instrucciones ejecutadas similares a las proporcionadas por SCoPE, y en el mismo o menor tiempo de simulación por regla general. Además, SCoPE proporciona una serie de estimaciones que OVP no proporciona, como son los usos de cachés o del bus, o el consumo energético.

Otro aspecto que diferencia a ambas herramientas es el esfuerzo requerido para crear los modelos. Tanto la creación de modelos de procesadores como de periféricos requiere un esfuerzo superior en OVP. En primer lugar hay que estar familiarizado con las distintas APIs que OVP proporciona para modelar dichos componentes. Además estos modelos requieren un elevado nivel de detalle que implica conocer en profundidad el funcionamiento interno de los componentes a modelar. SCoPE, por el contrario, facilita esta labor ya que en el caso de los modelos de procesadores basta con especificar las características de rendimiento de dicho procesador. Respecto al modelado de periféricos, SCoPE proporciona una serie de wrappers en SystemC, de modo que únicamente debemos de modelar la funcionalidad del periférico mediante las funciones de lectura y escritura sobre dicho periférico.

También se ha observado que para los códigos probados, el compilador soportado por OVP genera código binario con peores prestaciones que utilizando el compilador `arm-elf-gcc`. Esto indica que, al menos en este caso, es preferible no utilizar la API EABI.

Por último, si se el código de aplicación que queremos simular necesita un sistema operativo, en el caso de OVP deberemos proporcionarlo nosotros. Esto puede requerir portar dicho sistema operativo a la plataforma destino, con la consiguiente inversión de esfuerzo y tiempo. Por el contrario SCoPE utiliza un modelo abstracto del sistema operativo, de modo que nos permite ahorrarnos ese esfuerzo.

7 Trabajo futuro

Pese a haberse logrado los objetivos propuestos en este proyecto, hay otras extensiones y consideraciones que pueden ser interesantes de cara a un trabajo futuro. A continuación se exponen algunas de estas ideas.

En primer lugar, el trabajo que se va a realizar de forma más inmediata es la mejora de la compatibilidad con el cross-compilador `arm-none-linux-gnueabi-gcc` y otros compiladores que utilicen la API ARM EABI (Embedded Application Binary Interface). Como ya se ha visto en las pruebas, el uso de esta API por parte del compilador para realizar operaciones de punto flotante puede ralentizar las simulaciones mediante SCoPE. Por ello se va a mejorar el modo en que SCoPE anota información de prestaciones en estas llamadas para hacerlo de forma más eficiente.

Una extensión interesante sería el modelado de una capa de abstracción del hardware (HAL) basada en Windows. Actualmente SCoPE únicamente proporciona un modelo de HAL basado en Linux. Aunque este modelo permite utilizar controladores de Linux desde código Win32, sería interesante poder cargar controladores específicos de Windows.

Asimismo, otra ampliación que puede resultar útil es la implementación de una pila TCP/IP para la API Win32. Para esto se puede portar el software `lwIP` que ya se portó a la API POSIX de SCoPE o utilizar una implementación diferente. De cualquier modo es interesante de cara a disponer de otro método de comunicación por red.

También puede ser interesante el extender la herramienta para modelar software de virtualización. Algunas de las soluciones propuestas para la utilización de diversos sistemas operativos en un sistema embebido pasan por utilizar un software de virtualización que abstraiga a dichos sistemas operativos de la plataforma hardware subyacente y gestione los recursos hardware. Evidentemente, esta virtualización tiene un impacto en el rendimiento del sistema embebido, impacto que SCoPE no tiene en cuenta y por tanto no se ve reflejado en las estimaciones que proporciona.

Otra labor a tener en cuenta puede ser la implementación de una versión de la librería GDI32.dll. Esta librería gráfica de Windows proporciona los recursos necesarios para crear ventanas y objetos 2D. En este momento, el plug-in Win32 de SCoPE permite utilizar esta librería a través de WINE, pero las llamadas a funciones de dicha librería no son tenidas en cuenta en el proceso de estimación de prestaciones de SCoPE.

8 Índice de figuras y tablas

8.1 Índice de figuras

8.2 Índice de tablas

9 Bibliografía y Referencias

[1] Grupo de Ingeniería Microelectrónica. Dpto. de Tecnología Electrónica y de Sistemas y Automática (TEISA). Universidad de Cantabria. http://www.teisa.unican.es/gim.

[2] Grupo de Ingeniería Microelectrónica (GIM/UC) y Design of Systems on Silicon (DS2): "SCoPE User Manual", Julio 2010 (versión 1.1.5).

[3] G. Caballero: "The Win32 API plug-in for SCoPE, User Manual". (Draft v1.0).

[4] A. Kale, P. Mittal, S. Manek, N. Gundecha & M. Londhe: "Distributing Subsystems across Different Kernels running simultaneously in a Multi-Core Architecture", International Conferenece on Computational Science and Engineering, IEEE, 2011.

[5] H. Jeong & S. Lee: "Dynamic CPU Resource Allocation for Multicore CE Devices Running Multiple Operating Systems". International Conference on Consumer Electronics, IEEE, 2012.

[6] T. Nakajima, Y. Kinebuchi, A. Courbot, H. Shimada, T. Lin & H. Mitake: "Composition Kernel: A Multi-core Processor Virtualization Layer for Highly Functional Embedded Systems". Pacific Rim International Symposium on Dependable Computing, IEEE, 2010.

[7] M. Becker, G. Di Guglielmo, F. Fummi, W. Mueller, G. Pravadelli & T. Xie: "RTOS-Aware Refinement for TLM2.0-based HW/SW Designs". Proc. of the Design, Automation and Test Conference, IEEE, 2010.

[8] Y. Kinebuchi, T. Morita, K. Makijima, M. Sugaya & T. Nakajima: "Constructing a Multi-OS Platform with Minimal Engineering Cost", Proc. of the International Embedded Systems Symposium, Septiembre 2009.

[9] S. Nanda & T. Chiueh, "A survey of virtualization technologies", Stony Brook University, Tech. Rep. TR-179, Febrero 2005.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt & A. Warfield: "XEN and the art of virtualization", Proc. of the Symposium on Operating Systems Principles, ACM, 2003.

[11] Vmware: "Vmware Products", http://www.vmware.com/products (Online, Julio 2012).

[12] J. Sugerman, G. Venkitachalam & B. Lim: "Virtualizing I/O devices on Vmware workstation's hosted virtual machine monitor," Proc. of the USENIX Annual Technical Conference, 2001.

[13] Microsoft: "Microsoft Windows Virtual PC",

http://www.microsoft.com/windows/virtual-pc/ (Online, Julio 2012).

[14] QEMU: "QEMU Homepage", http://wiki.qemu.org/Main_Page, (Online Julio 2012).

[15] OVP: "OVP Homepage", http://www.ovpworld.org/, (Online, Julio 2012).

[16] N. Romdan: "ARM FastModels – Virtual Platforms for Embedded Software Development", I.Q. V.7, N.4, 2008.

[17] A. Bouchhima, P. Gerin, F. Pètrot: "Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation", Asia and South-Pacific Design Automation Conference, IEEE, 2009.

[18] M.A. Hassar, K. Sakanushi, Y. Takeuchi, & M. Imai: "RTK-Spec TRON: a simulation model of an ITRON based RTOS kernel in SystemC", Proc. of Design, Automation, and Test in Europe, 2005.

[19] S. Edwards: "Languages for Embedded Systems". Kluwer, 2000.

[20] G. Bonanome: "Hardware Description Languages Compared: Verilog and SystemC", Department od Computer Science, Columbia University, 2001.

[21] W. Kanda, Y. Yumura, Y. Kinebuchi, & K. Makijima: "SPUMONE: Lightweight CPU Virtualization layer for Embedded Systems", IFIP International Conference on Embedded and Ubiquitous Computing, IEEE/IFIP, 2008.

[22] H. Posadas, L. Díaz, E. Villar: "Fast data-cache modeling for nativeco-simulation", Asia and South-Pacific Design Automation Conference, IEEE, 2011.

[23] H. Posadas, F. Herrera, P. Sánchez, E. Villar, & F. Blasco: "System-level performance analysis in SystemC". Proc. of the Design, Automation and Test Conference, 2004.

[24] P. Mantegazza, E. Dozio, & S. Papacharalambous: "RTAI: Real Time Application Interface, vol 2000". Specialized Systems Consultants, Inc., Seattle, 2000.

[25] Accellera: "SystemC Homepage", http://www.accellera.org/home/, (Online, Julio 2012).

[26] WINE: "Wine user guide," http://www.winehq.com/site/docs/wineusr-guide/index. (Online, Julio 2011).

[27] ISO/IEC: "Information technology – Coding of moving pictures and associated audio for digital storage media at up to 1,5 Mbits/s – Part 3: Audio", ISO/IEC 11172 Part 3, 1993.

[28] ISO/IEC: "Information technology – Coding of moving pictures and

associated audio information – Part 3: Audio", ISO/IEC 13818 Part 3, 1998.

[29] ISO/IEC & ITU-T: "Information technology – Coding of audio-visual objects – Part 10: Advanced Video Coding", ISO/IEC 14496 Part 10, 2003.

[30] G. Bouvigne: "Shine MP3 encoder", http://gabriel.mp3-tech.org/, (Online, julio 2012).

[31] ITU-T: "Codificación de vídeo avanzada para los servicios audiovisuales genéricos", http://www.itu.int/rec/T-REC-H.264/es, (Online, Julio 2012).

[32] S. Wilson, "WAVE PCM soundfile format", https://ccrma.stanford.edu/courses/422/projects/WaveFormat/, (Online, Julio 2012).

[33] A. Calderón, & N. Castillo: "Why ARM's EABI matters", http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Why-ARMs-EABI-matters/, (Online, Julio 2012).

[34] J. Corbet, A. Rubini, & G. Kroah-Hartman: "Linux Device Drivers, Third Edition", O'Reilly, 2005.