Facultad de Ciencias

HERRAMIENTAS DE SIMULACIÓN HARDWARE SOBRE ARM

(Hardware Simulation Tools for ARM Architectures)

Trabajo de Fin de Grado

para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Luis Pérez Moya

Director: Pablo Prieto Torralbo

Junio – 2019

Resumen

El vertiginoso avance de la tecnología, soportado en gran medida por los recursos de computación disponibles, convierte su estudio y desarrollo en un área de investigación fundamental. Dado el actual nivel de complejidad de cualquier procesador comercial, esta tarea se ha vuelto extremadamente complicada. Evaluar nuevas alternativas para mejorar las cotas de rendimiento de un procesador supone un gran esfuerzo de investigación, siendo el coste de "prototipado" hardware inasumible cada vez que se requiere poner a prueba una posible modificación. Debido a esta limitación surge la simulación por software que, esencialmente, consiste en emular el hardware a través de un programa, permitiendo al usuario observar el comportamiento de un sistema que no necesariamente exista. Esta metodología evita disponer de un prototipo cada vez que se requiera evaluar un cambio en el procesador.

A pesar de sus evidentes ventajas, el proceso de simulación de sistema completo es complejo y tedioso. Para obtener resultados con un nivel de precisión razonable y una funcionalidad adecuada, es preciso un trabajo previo con las herramientas de simulación que dista de ser trivial. El objetivo principal del presente Trabajo de Fin de Grado ha consistido en desplegar un simulador de sistema completo sobre la plataforma de desarrollo Nvidia Jetson TX2[1], preparándolo para simular una arquitectura ARM (actualmente en auge).

Las tareas principales realizadas durante el proyecto han consistido en el despliegue e instalación de la plataforma de trabajo (Jetson TX2), la instalación del simulador Gem5 preparado para desarrollar sobre una arquitectura ARM, la puesta a punto tanto del *kernel* de la máquina como del *framework* de simulación para poder hacer uso de las extensiones de virtualización propias del ISA del procesador (agilizando de manera muy significativa el proceso de simulación) y la evaluación del *working-set* de varias aplicaciones pertenecientes a la suite de *benchmarks* SPEC CPU2017.

[1] La plataforma de desarrollo Nvidia Jetson TX2 utilizada en este TFG ha sido donada por NVIDIA Corporation a través de su programa NVIDIA GPU Grant.

Abstract

The huge advances on technology, mostly driven by available computing resources, makes its study and development a fundamental research area. Given the current level of complexity of any commercial processor, this task has become increasingly complicated. Testing new possibilities to improve performance levels involves a great deal on research, the cost of hardware prototyping being unbearable each time a possible modification is required to be tested. Due to this limitation, software simulation arises. This essentially consists of a program that allows the user to observe the behavior of a system that does not necessarily exist. This process would avoid having a prototype every time a change is required.

Despite its obvious advantages, the complete system simulation process is complex and tedious. In order to obtain results with a reasonable level of precision and adequate functionality, it is necessary to work previously on the simulation tools, which is far from trivial. The main objective of this End of Grade Project consists on deploying a complete system simulator on the development platform Nvidia Jetson TX2, adapting it to simulate an ARM architecture (now in its peak).

The main tasks carried out during the project have consisted in the deployment and installation of the working platform (Jetson TX2)[2], the installation of the Gem5 simulator (including the build process that allows to develop on ARM), the fine-tuning of both the machine kernel and the simulation framework to be able to make use of the virtualization extensions of the processor's ISA (significantly speeding up the simulation process) and the evaluation of the working-set of several applications belonging to the SPEC CPU2017 benchmarks suite.

[2] The development plattform Nvidia Jetson TX2 employed in this work has been donated by NVIDIA Corporation through their GPU Grant program.

Tabla de contenido

1 Introducción

Actualmente, resulta evidente la implicación de los computadores en los significativos cambios tecnológicos y socioculturales experimentados durante los últimos 50 años. Desde el primer computador digital totalmente electrónico hasta el último *Smartphone*, la evolución tecnológica es abrumadora, en aspectos tales como rendimiento, tamaño, coste, etc. Es más que probable que la evolución de los computadores mantenga este ritmo vertiginoso en un futuro cercano, espoleada por la creciente dependencia de la capacidad de cálculo en cada vez más ámbitos de nuestra vida diaria.

Dos de los pilares básicos que han contribuido a la evolución del computador han sido el desarrollo tecnológico, capaz de proporcionar cada vez más transistores a menor precio [1], y la arquitectura de computadores, a cargo de transformar la tecnología en rendimiento [2]. Esta rama de la informática consiste en la selección e interconexión de componentes hardware para crear computadores capaces de cumplir determinados objetivos en relación con el coste, la funcionalidad y el rendimiento.

La utilización de soluciones de prototipado en el área de Arquitectura de Computadores presenta unos costes difícilmente asumibles, forzando el empleo de metodologías alternativas para llevar a cabo labores de investigación. Una de las principales alternativas son las herramientas de Simulación, que permiten la evaluación de los diseños de hardware sin necesidad de construir sistemas físicos. Un simulador ofrece un término medio entre costes y precisión a la hora de medir los resultados del sistema objetivo. Dicho sistema no tiene por qué existir físicamente. Evitando la necesidad de trabajar con hardware real se eliminan los altos costes asociados con el prototipado hardware, tanto económicos como temporales.

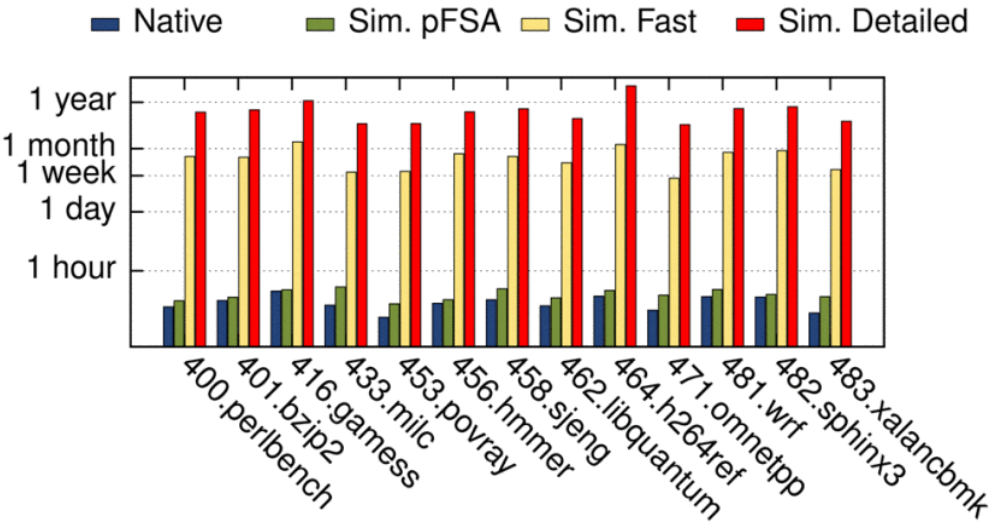


Figura 1. Tiempo de ejecución de aplicaciones del *benchmark* SPEC CPU 2006. Ejecución Nativa (*Native*) y simulación con Gem5 con diferentes niveles de detalle (*Sim Fast* y *Sim Detailed*). Imagen extraída de [3].

A pesar de sus evidentes ventajas, las herramientas de simulación no están exentas de desafíos significativos. El más relevante consiste en el frágil equilibrio entre precisión y coste computacional. Esto implica que, a mayor precisión de la simulación (se entiende como el nivel de detalle con el que se describe el hardware), mayor es el coste computacional, ya que habría que simular más elementos. La Figura 1 pone de manifiesto este efecto, mostrando el tiempo requerido para simular una aplicación completa con diferentes niveles de detalle en la simulación. Como puede observarse, simular 30 minutos de ejecución con un elevado nivel de detalle requeriría aproximadamente un año de ejecución, valor completamente prohibitivo.

En un intento por dar respuesta a su creciente complejidad, los simuladores han ido ofreciendo diferentes opciones para reducir costes de cómputo, ya sea por una simulación parcial del hardware [4][5][6][7], ejecución *multithread* [8][9] o simulaciones del sistema completo limitadas a una región específica de la aplicación en ejecución por medio de puntos de control [10]. En este último caso, con el fin de reducir el coste computacional de las herramientas de simulación de sistema completo, algunos f*rameworks* hacen uso de metodologías combinadas, donde la simulación detallada se limita a una región de interés y el resto del tiempo se sacrifica precisión para mejorar la velocidad de simulación. Uno de los métodos de aceleración utilizados consiste en hacer uso de simulación asistida por virtualización [3], más concretamente mediante *kvm* (*Kernel-based Virtual Machine*). En este modo de simulación la ejecución de instrucciones se delega a la máquina real, sincronizando el estado de la máquina virtual y el sistema simulado periódicamente. Este mecanismo proporciona velocidades de simulación cercanas a la máquina real, dado que los elementos de la micro-arquitectura del procesador no se simulan. Esta técnica, que supone una reducción importante en el coste computacional de la simulación, impone algunas limitaciones en su uso, siendo la principal que para poder simular un sistema con ayuda de virtualización con aceleración *hardware*, la arquitectura del sistema host tiene que ser la misma que la del sistema simulado.

El grupo de investigación *Computer Engineering* de la Universidad de Cantabria (http://www.ce.unican.es) trabaja actualmente en la realización de propuestas micro-arquitecturales para procesadores basados en la arquitectura de 64 bits ARMv8 [11], mostrando su interés en adaptar sus herramientas de simulación a este tipo de arquitectura. Dicho proceso de adaptación define el objetivo principal del trabajo realizado, que ha culminado con un *framework* de simulación funcional para arquitecturas ARM que el grupo utilizará en futuros trabajos de investigación.

A lo largo de este documento se irá describiendo el proceso seguido, desde la puesta en marcha e instalación de un equipo con arquitectura ARM, la instalación de la herramienta de simulación Gem5 [12] en dicho equipo y su adaptación para que sea completamente funcional sobre esta arquitectura y

permita la simulación con aceleración por virtualización, hasta la realización de una prueba de concepto sencilla que verifique su funcionamiento.

1.1 Contexto y antecedentes de trabajo

Como se ha comentado en la Introducción, los prototipos *hardware* como método de evaluación no son una opción realista en el área de trabajo de este TFG. Es por esto que se adoptan metodologías como la simulación o el *profiling hardware*. En las metodologías basadas en simulación se describe el hardware a través de código que imita su funcionamiento, mientras que las técnicas de *profiling* se basan en la recopilación de información relacionada con el rendimiento de una aplicación o sistema a través de los contadores hardwarede una máquina real, accesibles mediante la unidad de monitorización de rendimiento (Performance Monitoring Unit). Esta metodología tiene la enorme ventaja de que es posible trabajar sobre hardware real (toda CPU actual cuenta con una PMU y multitud de eventos hardware accesibles), con lo que esto supone para la velocidad de ejecución de las aplicaciones.

Las ventajas descritas han incrementado el interés sobre las metodologías basadas en *profiling* hardware en los últimos años. Sin embargo, el *profiling* cuenta con una clara desventaja, dado que solo permite evaluar el sistema real sobre el que actúa. Esto impide evaluar propuestas basadas en modificaciones arquitecturales o incluso arquitecturas totalmente diferentes, siendo la simulación *software* la única alternativa adecuada a la hora del desarrollar nuevas propuestas arquitecturales.

Actualmente, existen gran cantidad de herramientas de simulación [13], cada una con diferente nivel de detalle en el modelado hardware. Las herramientas de simulación más simples se limitan al modelado de elementos aislados del hardware, haciendo uso de trazas de ejecución para su posterior evaluación. Un ejemplo de este tipo de herramientas es el *framework* de evaluación de predicción de saltos utilizado en competiciones académicas durante los últimos años [14]. Se trata de un simulador que se limita a implementar un algoritmo de predicción de saltos via software y evaluarlo a través de trazas de la suite SPEC CPU2006. En el lado opuesto a este tipo de herramientas, la técnica más compleja aparece con los simuladores de sistema completo. Estos simuladores llegan a tal nivel de detalle que un software de un sistema real puede ejecutarse sobre el simulador sin ninguna modificación.

De acuerdo a los datos obtenidos en las publicaciones más relevantes del área de Arquitectura de Computadores de los últimos años [15], gran parte de los equipos de investigación se decantan por simuladores de sistema completo, con un coste computacional elevado pero con alto nivel de precisión. Gem5 [12] es un referente dentro de este tipo de simuladores, altamente extendido y reconocido. Este simulador, implementado en su mayoría en C++, implementa varios modelos de CPU (desde una arquitectura muy simple con 1 ciclo por instrucción hasta una superescalar con ejecución fuera de orden, similar a la

presente en un procesador comercial actual) y soporta la simulación de sistema completo con diferentes ISAs (Alpha, ARM, SPARC, MIPS, POWER, RISC-V, x86). Dada su versatilidad, esta herramienta es usada de manera activa por el grupo de investigación *Computer Engineering*, siendo por esta razón la herramienta de simulación utilizada en este proyecto.

A pesar de sus ventajas y versatilidad, el *framework* de simulación Gem5 no está exento de una problemática específica que requiere un trabajo previo de preparación de la herramienta. Dichos problemas se detallan en el siguiente apartado y dar solución a los mismos ha supuesto el grueso del trabajo realizado, como se describirá a lo largo de este documento.

1.2 Motivación

Actualmente, Gem5 es una de las herramientas de simulación más versátiles, contando entre sus principales cualidades con la de simular un sistema completo (capaz de ejecutar un sistema operativo sin modificar). Esta herramienta implementa diversos modelos de CPU, cada uno con un nivel de detalle distinto en la descripción del hardware subyacente. Dichos modelos ofrecen diferentes alternativas en el *tradeoff* entre precisión y coste computacional, como se muestra en la Figura 1. El modelo de mayor precisión de Gem5 (*detailed*) solamente es capaz de ejecutar 0,1 Millones de instrucciones por segundo, lo que supone un slowdown de aproximadamente 4 órdenes de magnitud. En la versión actual de la herramienta, el modelo *detailed* simula al completo un procesador superescalar y fuera de orden basado (en parte) en una arquitectura de CPU similar al Alpha 21264. Este procesador simulado cuenta con un *pipeline* de 5 etapas: *Fetch, Decode, Rename, Issue/Execute/Writeback, Commit*. El elevado nivel de detalle ofrecido por este modelo conlleva un gran coste computacional ya que, al tratarse de un modelo software del hardware subyacente, se genera mucha carga de cómputo al reproducir con exactitud todos sus elementos micro-arquitecturales.

Debido al elevado coste computacional de este proceso de simulación, Gem5, implementa diversas técnicas mencionadas previamente en la sección 1.1. Entre este conjunto de técnicas aparece una muy significativa sobre la cual se hará hincapié a lo largo del proyecto; simulación con soporte KVM [3]. La simulación bajo KVM está implementada como un módulo de Gem5 y permite hacer uso del soporte para virtualización que ofrecen los últimos *kernels* como medio para llevar a cabo la simulación. Con esta técnica, el simulador ejecuta las instrucciones directamente sobre el procesador físico, el cual actúa con Gem5 como una máquina virtual.

El uso de KVM en Gem5 requiere que el equipo *target* a simular implemente la misma arquitectura que el *host* físico sobre el que se ejecuta el simulador, obteniendo una velocidad de simulación próxima a la real. Sin embargo, este proceso no ofrece información sobre el comportamiento del hardware ni permite cambios micro-arquitecturales, por lo que hay que

plantear como unificar el modelo de CPU *detailed* (que ofrece la mayor precisión) con el uso de KVM (que ofrece el mayor rendimiento). Esto se resuelve haciendo uso de *checkpoints*. En primera instancia se buscará la región de interés del programa a evaluar y se llegará a ella de la manera más rápida posible, haciendo uso de KVM. En ese punto se creará un *checkpoint* y, a partir de ahí, se pasará a ejecutar en modo *detailed*, ahorrando un tiempo de simulación enorme, el cual es innecesario.

En la actualidad una de las arquitecturas más extendidas es x86. Si se quiere hacer uso de KVM para agilizar el proceso, Gem5 debe trabajar con x86. Esto implica un problema ya que el repertorio de instrucciones de x86 es muy amplio y por desgracia Gem5 no lo tiene implementado al completo. Esto puede ocasionar problemas a la hora de ejecutar aplicaciones específicas que hagan uso de este tipo de instrucciones que no han sido implementadas.

La alternativa que se ha encontrado es trabajar con un ISA diferente. En este caso se ha apostado por una arquitectura, ARM, que en la actualidad está tomando un gran protagonismo debido a que forma parte de la mayoría de los dispositivos móviles y de sistemas embebidos. Además, es una de las arquitecturas más eficientes en términos de energía. Desde el punto de vista de las herramientas de simulación, esta arquitectura cuenta con la ventaja de que la gran mayoría de sus instrucciones están implementadas en Gem5.

Para trabajar con la arquitectura ARM, además de poder hacer uso de KVM para agilizar la simulación, es necesario hacer una preparación previa de la herramienta, así como de la propia plataforma de trabajo. Esto implica adaptar la plataforma para que pueda operar con KVM, así como adaptar la herramienta Gem5 para que trabaje sobre la arquitectura en cuestión y pueda hacer uso de esta funcionalidad. El proyecto consistirá en la realización de estos trabajos y en la comprobación de que todo funciona correctamente y de que se obtienen los resultados esperados.

1.3 Objetivos y Fases del Proyecto

El objetivo del proyecto será la utilización de la plataforma hardware proporcionada (NVIDIA Jetson TX2 [16]) para trabajar con Gem5. La placa Jetson TX2 cuenta con procesadores con arquitectura ARM, y será necesario comprobar el funcionamiento de KVM. Sobre dicha plataforma se instalará, compilará y ejecutará el simulador de sistema completo Gem5. Se hará especial énfasis en probar si el modelo de virtualización basado en *kernel* (KVM) es válido para esta plataforma [17], así como poner en marcha las funcionalidades requeridas para la evaluación de aplicaciones. Una vez finalizado el proceso, se llevará a cabo una prueba de concepto, analizando el *working-set* de varias aplicaciones de la suite SPEC CPU2017 [18].

El resto del documento, reflejo de las fases definidas para el proyecto (y también reflejo del orden cronológico del trabajo llevado a cabo), se define a continuación:

- En el Capítulo 2 se presentan las herramientas hardware y software sobre las que se ha trabajado en el proyecto. Desde la plataforma TX2, pasando por *kernel* y Gem5, hasta el software empleado en la prueba de concepto.
- El Capítulo 3 describe la puesta en marcha de la plataforma sobre la que se va a trabajar (NVIDIA Jetson Tx2), instalando todos los componentes necesarios para su correcto funcionamiento, y estableciendo un modelo de comunicación entre el equipo *host* y la plataforma sobre la que se hará el proyecto. Se incluye una prueba de concepto para familiarizarse con el desarrollo sobre la plataforma, desplegando el *framework* de aprendizaje automático TensorFlow y realizando medidas de rendimiento.
- El Capítulo 4 detalla el despliegue de Gem5 sobre la placa TX2. Esto implica diversas consideraciones y trabajo adicional, como la modificación del *kernel* que viene por defecto en la placa, así como la compilación y preparación de Gem5, para proporcionar compatibilidad con KVM.
- Finalmente, el Capítulo 5 describe la prueba de concepto llevada a cabo. Haciendo uso de la suite de *benchmarks* SPEC CPU2017, se crearán cargas de trabajo a través de simulación asistida por virtualización y se hará un estudio del *working-set* de estas aplicaciones como prueba de correcto funcionamiento de la plataforma.
- Se cerrará el documento con un Capítulo dedicado a las conclusiones y a las líneas de trabajo futuras.

2 Hardware y Software utilizado

Se dedica este capítulo a describir todas las herramientas con las que se ha trabajado a lo largo del proyecto. El tiempo dedicado al aprendizaje de cada una de ellas ha supuesto una parte relevante del proyecto, dado su nivel de complejidad.

2.1 NVIDIA JetsonTX2

Todo el desarrollo del trabajo ha sido realizado sobre la placa NVIDIA Jetson TX2 [16]. Esta plataforma integrada de bajo consumo es idónea para el desarrollo por su arquitectura basada en ARM. Su extensa comunidad de desarrolladores y documentación proporciona una fuente de consulta de gran ayuda. Actualmente, los servidores implementados con arquitecturas ARM son escasos y presentan un precio elevado en relación a la capacidad de cálculo que ofrecen. Dado que la utilidad de esta arquitectura se limita a agilizar el proceso de simulación para la creación de *checkpoints*, gracias al dispositivo elegido se consiguen los mismos resultados a bajo coste. Gracias a su soporte para virtualización hardware (ARM Cortex-A57) y a contar con un repertorio de instrucciones disponible en Gem5 (ARM-v8), la Jetson TX2 nos permitirá hacer uso del soporte KVM para simulación de un sistema completo ARM.

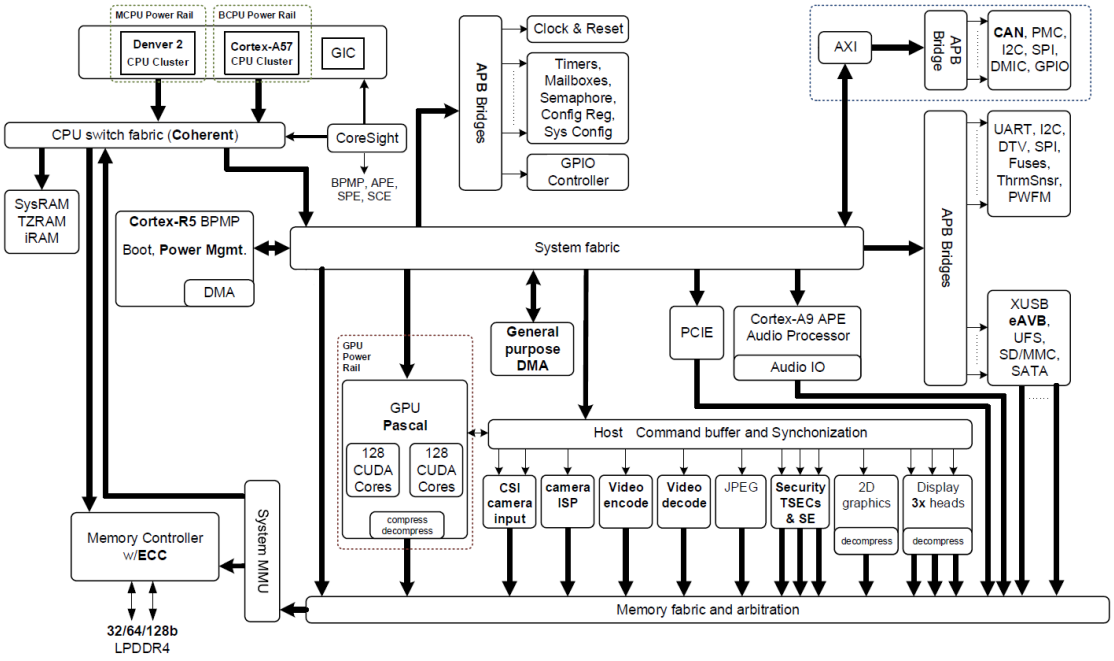


Figura 2. Esquema arquitectural de la placa Nvidia Jetson TX2.

Como se muestra en el esquema de la Figura 2, la placa NVIDIA Jetson TX2 cuenta con una GPU NVIDIA Pascal de 256 CUDA cores, una CPU de 64 bits ARMv8 con 6 cores y 8 GB de memoria LPDDR4. La CPU combina dos núcleos NVIDIA Denver con cuatro núcleos ARM Cortex-A57. Se trata de dos arquitecturas de procesador superescalares que hacen uso del mismo

repertorio de instrucciones, pero presentan importantes diferencias micro-arquitecturales. El procesador Denver implementa una arquitectura en orden con un número elevado de vías (7) y hardware específico para la optimización dinámica de código (evitando el paso por las etapas de *fetch* y *decode* a aquellas instrucciones optimizadas). Los procesadores Cortex-A57 presentan una estructura de superescalar más convencional, con ejecución fuera de orden y *fetch/retire* de 3 instrucciones por ciclo. En el caso concreto de este trabajo, ha sido necesario hacer uso de estos últimos procesadores, ya que implementan una parte crucial para el desarrollo, la capacidad de virtualización con aceleración hardware.

Para llevar a cabo labores de desarrollo, NVidia pone a disposición de los usuarios un paquete software denominado JetPack SDK. A través de dicha herramienta ha sido posible configurar la plataforma con los siguientes elementos:

- Imagen más reciente del sistema operativo Linux LT4 [19], modificado para su ejecución sobre una arquitectura ARM.
- Conjunto de herramientas de desarrollo tanto para el equipo que actuará como anfitrión en el proceso de instalación del sistema como para la placa Nvidia Jetson TX2, incluyendo herramientas de cálculo paralelo destinadas a GPUs NVIDIA (CUDA) y librerías orientadas a redes neuronales profundas (cuDNN [20]).
- Bibliotecas, APIs, muestras y documentación necesarias para poner en marcha el entorno de desarrollo.
- Plataforma de ejecución para labores de inferencia en redes neuronales profundas (TensorRT).

2.2 Gem5

Gem5 [12] es una herramienta de simulación de sistema completo con un elevado nivel de detalle, que permite la ejecución, sobre el sistema simulado, de un Sistema Operativo comercial sin modificar. Gem5 es el resultado de la unión de dos simuladores: m5 y GEMS. Está programada en su mayor parte en C++ y Phyton, se distribuye bajo licencia BSD y evoluciona siguiendo un modelo de desarrollo colaborativo (repositorio git alojado en Google Open Source[3]).

El simulador Gem5 implementa varios modelos de CPU, heredados del simulador m5, con diferentes niveles de detalle. Es capaz de simular desde una arquitectura escalar y en orden sin modelado de timing (1 instrucción = 1 ciclo siempre) hasta una arquitectura superescalar con ejecución fuera de orden y especulativa con modelado de timing (tanto de los elementos del pipeline como

3 https://gem5.googlesource.com/public/gem5

de la jerarquía de memoria). De la misma forma dentro de la herramienta conviven varios modelos de memoria con diferente nivel de detalle; uno más sencillo y rápido heredado del simulador m5 y otro más preciso, pero más costoso, heredado del simulador GEMS y denominado Ruby. Gem5 Soporta la mayoría de los ISAs comerciales (Alpha, ARM, SPARC, MIPS, Power, RISC-V y x86).

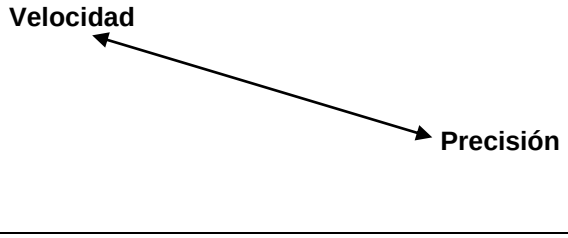
<table>
<tr><th colspan="2">Procesador</th><th colspan="3">Sistema de Memoria</th></tr>
<tr><td rowspan="2">Modelo de CPU</td><td rowspan="2">Modo de funcionamiento</td><td rowspan="2">Clásico</td><td colspan="2">Ruby</td></tr>
<tr><td>Simple</td><td>Garnet</td></tr>
<tr><td>KVM</td><td>SE</td><td rowspan="6" colspan="3"> 
</td></tr>
<tr><td rowspan="2"></td><td>FS</td></tr>
<tr><td>Atomic</td><td>SE</td></tr>
<tr><td rowspan="2"></td><td>FS</td></tr>
<tr><td>O3</td><td>SE</td></tr>
<tr><td rowspan="2"></td><td>FS</td></tr>
</table>

Figura 3. Comparativa Velocidad-Precisión en función del modo elegido en Gem5.

Gem5 cuenta además con dos modos de funcionamiento: Emulación de llamadas al sistema (SE o *syscall emulation*) y Sistema Completo (FS o *Full-System*). En modo SE, Gem5 solamente ejecuta código de usuario, sin incluir el sistema operativo y emulando las llamadas al sistema (por ejemplo, *read*()). De manera alternativa, el modo *full-system* simula un entorno *bare-metal* adecuado para la ejecución de la pila software completa (S.O. + Aplicación), incluyendo soporte para interrupciones, excepciones, niveles de privilegios, dispositivos de E/S, etc.

En comparación con el modo SE, el modo FS ofrece un nivel de detalle y una precisión mayor, a cambio de un incremento notable en la complejidad de uso y en el coste computacional de la simulación. Cabe por ello destacar que el resto del proyecto ha sido desarrollado haciendo uso de este modo de funcionamiento

Entre los modelos de CPU proporcionados por Gem5, aquellos utilizados en el desarrollo del trabajo han sido **KVM**, ***Atomic y O3.*** Cada uno de estos modelos se sitúa en un punto distinto en lo referente al *trade-off* precisión-Coste computacional. A continuación, se detallan los aspectos básicos de cada modelo de CPU indicado:

- **Atomic** es el modelo de CPU más sencillo. Modela un *core* que ejecuta una instrucción por ciclo, sin modelar el timing de los accesos a la jerarquía de memoria (son atómicos). Es posible utilizarlo junto a una jerarquía de memoria con distintos niveles o en solitario. Este modelo resulta adecuado en aquellas situaciones donde el *timing* del *pipeline o de la jerarquía de memoria* no tienen importancia, como en las fases de warm-up de una aplicación, o en la medida de tasas de fallos (*miss rate)* sobre los distintos nivele de cache.

- **O3 (detailed)** También conocido por *detailed*, es un modelo de CPU que simula toda la funcionalidad y el timing de un procesador escalar fuera de orden (*Reorder Buffer*, dependencias entre instrucciones, unidades funcionales, accesos a la memoria y las etapas del *pipeline)*. Gran parte de los componentes del *pipeline* son altamente parametrizables, y permiten a O3 simular gran variedad de arquitecturas superescalares.
- **Kvm** (*Kernel-based Virtual Machine*) es un modelo que hace uso de simulación asistida por virtualización con soporte hardware, liberando así a la herramienta del modelado de la funcionalidad de las instrucciones y del *timing*. Este modelo no simula la ejecución de instrucciones en absoluto, que ejecuta kvm en el host, y tan solo la memoria se mantiene en el simulador, siendo que la máquina kvm se mapea sobre ella. Gracias a esto, es capaz de conseguir una velocidad de ejecución de las instrucciones prácticamente nativa, siendo una solución infinitamente más ágil que el modelo simulado más sencillo (atomic). Por esta razón, este modelo permite alcanzar regiones de interés lejanas (en tiempo de ejecución) donde incluso *atomic* presenta dificultades en cuanto a tiempo requerido (ver Figura 1).

Las labores a realizar con Gem5 en este proyecto han sido múltiples, y se describirán con detalle a lo largo del documento. Incluimos en este punto un breve resumen de las mismas:

- Instalación y compilación de la herramienta de simulación sobre la plataforma Jetson TX2.
- Ajustes sobre el modelo de CPU KVM para su correcto funcionamiento sobre un ISA ARM, así como para implementar la metodología de trabajo basada en la creación de *checkpoints*, descrita en la sección 1.2.
- Creación de imagen de disco y *checkpoints* para trabajar con las aplicaciones del *benchmark* SPEC CPU17, descrito en la siguiente sección.
- Realización de pruebas de concepto utilizando los modelos de procesador más detallados.

2.3 Benchmarking - SPEC CPU 2017

Los rápidos avances tecnológicos asociados al computador han tenido como consecuencia principal su presencia actual en casi cualquier ámbito de nuestra vida diaria. Esta heterogeneidad en el uso del computador hace casi imposible determinar con precisión el tipo de software que ejecutará a lo largo de su vida útil. Con el objetivo de estandarizar de alguna forma las medidas de rendimiento de la gran diversidad de procesadores comerciales disponibles en la actualidad, se han desarrollado herramientas que permiten la comparación de diferentes arquitecturas haciendo uso de un grupo reducido de aplicaciones representativas de entornos mucho más amplios. Estas herramientas,

denominadas *benchmarks*, están diseñadas para imitar un tipo particular de carga de trabajo en un componente o sistema y proporcionar resultados detallados para su posterior análisis.

Dado el elevado número de áreas y aplicaciones disponibles en la actualidad, es extremadamente complejo el diseño de un *benchmark* representativo de cualquier entorno de ejecución. Por esta razón, las herramientas de benchmarking han evolucionado necesariamente hacia entornos de ejecución específicos. Entre los muchos disponibles, algunos *benchmarks* intentan recopilar software de uso habitual en equipos domésticos y laborales, mientras que otros se enfocan a entornos de altos requerimientos computacionales (High Performance Computing o HPC) o entornos distribuidos (Cloud Computing). Adicionalmente, existen *benchmarks* orientados a arquitecturas específicas (como las GPUs) o a áreas de trabajo concretas (Bio benchmark). La tabla 5 agrupa algunos de los principales *benchmarks* asociados a cada una de las categorías definidas previamente.

Tabla 1. Ejemplos de benchmarks en función de su área de trabajo.

Entorno	**Benchmark Suites**
Doméstico y Workstation	SPEC [18], Parsec [21]
High Performance Computing	NPB [22], Linpack [23], Graph500 [24]
Cloud Computing	Cloudsuite [25], YCSB [26], HiBench [27]
GPU	Parbloil [28], Rodinia [29]
Bio-informatics	BioBench [30]

Para evaluar el *framework* de simulación Gem5 con arquitectura ARM y poner a prueba el trabajo realizado, se ha optado por usar una de las suites más extendidas de entorno de escritorio: SPEC CPU 2017. Este conjunto de *benchmarks* basados en aplicaciones reales es utilizado con asiduidad en trabajos de investigación en el área de Arquitectura de Computadores, y el grupo de investigación *Computer Engineering* ha adquirido la licencia para trabajar con la suite.

La suite CPU 2017 está desarrollada por la *Standard Performance Evaluation Corporation* (SPEC), una organización sin fines de lucro fundada en 1988 para establecer *benchmarks* de rendimiento estandarizados que sean objetivos, significativos y claramente definidos. Los miembros de SPEC incluyen proveedores de hardware y software, universidades e investigadores.

El conjunto de *benchmarks* SPEC CPU2017 contiene todos los paquetes intensivos en CPU de última generación de SPEC, estandarizados por la industria, utilizando cargas de trabajo desarrolladas a partir de aplicaciones de usuario reales, como intérpretes de Perl, compiladores de C, compresión de vídeo... etc. Estas aplicaciones están programadas en lenguajes muy frecuentemente usados: C, C++ y Fortran.

Este conjunto de *benchmarks* se divide en dos grandes categorías INT y FP, donde los de tipo INT son aplicaciones que se centran en evaluar el rendimiento del equipo ejecutando instrucciones que trabajan con números enteros. Contrario a esto, se encuentran las aplicaciones de categoría FP que buscan determinar el rendimiento en base a la ejecución de instrucciones de punto flotante. La Tabla 2 y la Tabla 3 muestran la clasificación de las diferentes aplicaciones en función de la suite a la que pertenecen, indicando su lenguaje y uso.

Tabla 2. Aplicaciones SPEC CPU de tipo INT.

SPEC INT 2017	Lenguaje	Descripción
PERLBENCH	C	Intérprete de Perl
GCC	C	Compilador GNU C
MCF	C	Planificador de rutas de transporte
OMNETPP	C++	Simulador de Redes de computadores
XALANCBMK	C++	Conversor de XML a HTML via XLST
X264	C	Compresión de Video.
DEEPSJENG	C++	Inteligencia Artificial: tree search(Ajedrez)
LEELA	C++	Inteligencia Artifical: Monte Carlo tree search (Go)
EXCHANGE2	Fortran	Inteligencia Artificial: recursive solution generator (Sudoku)
XZ	C	Compresión de datos.

Tabla 3. Aplicaciones SPEC CPU de tipo FP.

SPEC FP 2017	Lenguaje	Descripción
BWAVES	Fortran	Modelado de explosiones
CACTUBSSN	C++, C, Fortran	Resolución de problemas físicos relacionados con la relatividad
NAMD	C++	Simulador de sistemas moleculares.
PAREST	C++	Diagnósticos médicos a partir de imagen.
POVRAY	C++,C	Simulador de trazado de rayos.
LBM	Fortran, C	Simulador de dinámicas de fluidos.
WRF	C++	Predicción meteorológica.
BLENDER	C++,C	Simulador de la atmósfera.
CAM4	Fortran, C	Simulador de la atmósfera.
IMAGICK	C	Manipulación de imágenes.
NAB	C	Dinámicas moleculares.
FOTONIK3D	Fortran	Simulaciones electromagnéticas.
ROMS	Fortran	Simulador de océanos a nivel regional.

Cada aplicación del *benchmark* tiene dos implementaciones: Speed y Rate, que buscan medir el rendimiento del equipo de manera diferente. Por un lado, la implementación Speed busca medir el tiempo que tarda el sistema en ejecutar una única copia del *benchmark*, obteniendo así el rendimiento del sistema a partir del tiempo que tarda en ejecutar la aplicación. Esta implementación permite al usuario elegir, opcionalmente, el número de *threads OpenMP* que se ejecutarán. Por otro lado, en la implementación Rate se ejecutan múltiples copias de una misma aplicación de manera concurrente, lo cual sirve para medir el rendimiento del sistema en términos de cantidad de trabajos completados por unidad de tiempo (*throughput*). En esta implementación el uso de *OpenMP* esta desactivado.

SPEC proporciona el *framework* de trabajo completo en forma de imagen ISO donde se encuentra el código fuente de los *benchmarks*, los conjuntos de datos usados por estos, así como un conjunto de herramientas para compilar, ejecutar, validar e informar sobre los *benchmarks*.

A lo largo del proyecto, se analizará el código de determinadas aplicaciones contenidas en la suite SPEC CPU 2017. El tipo de metodología utilizado con el simulador requiere un trabajo previo sobre las aplicaciones que se detalla principalmente en el capítulo 5 de este documento. Se trata, básicamente, de identificar la región de interés de dichas aplicaciones y anotar el código para que el simulador sea capaz de identificar dicha fracción de código. Las aplicaciones se ejecutarán haciendo uso de KVM hasta alcanzar la región de interés, donde se procederá a hacer un cambio del modo de CPU; de KVM a *detailed*. Este cambio se conseguirá gracias a la creación de un *checkpoint*, que se genera gracias a la ejecución de instrucciones propias de Gem5 dentro del código de las aplicaciones usadas como referencia.

3 Primeros pasos con la plataforma Jetson TX2

A lo largo de este apartado se describirá el proceso de configuración que se ha llevado a cabo para poner en marcha la placa NVidia Jetson TX2. Todo este proceso ha sido realizado con ayuda de la herramienta de desarrollador proporcionada por el fabricante; Nvidia JetPack. Finalmente, realizaremos una prueba de concepto sencilla para verificar su correcto funcionamiento.

Como primer paso, el software de desarrollo debe ser instalado sobre un PC que tomará el rol de anfitrión (Host), al cual se conectará la placa (Target). Haciendo uso de esta herramienta, se procede a instalar en la placa un sistema operativo Linux modificado por NVIDIA (Linux for Tegra, L4T [19]) para optimizar el rendimiento de la plataforma y aprovechar todas las funcionalidades que presenta el dispositivo. Gracias al software de desarrollo esta instalación se puede llevar a cabo de manera sencilla, poniendo en manos del usuario la elección de los componentes clave del sistema operativo que van a ser instalados.

Uno de los aspectos clave del proceso de configuración es el modo de comunicación entre el equipo host y el dispositivo. Entre las dos opciones disponibles, conexión indirecta a través de un router/swith o directa mediante cable de red, se ha optado por usar la primera. La conexión directa presenta varias desventajas relacionadas con la conexión a internet del target:

- En este modo es necesario que el host disponga de dos interfaces de red; una para conectar con el target y otra para conectar el host a un router/switch que proporcione una conexión a internet.
- Adicionalmente, se requiere configurar un servidor DHCP en el equipo host para que el target pueda tener acceso a internet en el caso de que esto sea requerido.

A pesar de que durante gran parte del desarrollo se ha trabajado directamente sobre la placa, es recomendable tener un enlace con el equipo host preparado para realizar algunas tareas, como compilaciones cruzadas o cambios sobre la memoria flash de la placa. Finalizado el proceso de instalación ya es posible arrancar el dispositivo y trabajar sobre él como si fuese un equipo de sobremesa común. Una vez arrancada la placa, se realizó la prueba de concepto que se detalla en la siguiente sección.

3.1 CPU vs GPU con Aplicaciones de Machine Learning

Como primera toma de contacto con el dispositivo se ha instalado y probado el *framework* de *Machine Learning* de Google, TensorFlow[4], para analizar el rendimiento del dispositivo frente a otras plataformas por medio de la ejecución de un *benchmark*.

A pesar de la existencia de métodos de instalación automatizada de TensorFlow (instalación de versiones compiladas con *pip* o contenedores), se ha optado por la compilación e instalación manual del f*ramework* sobre la placa. Gracias a ello se ha podido parametrizar de forma exhaustiva el proceso de instalación para optimizar el rendimiento de ejecución en nuestro dispositivo. La comunidad de desarrolladores de NVIDIA ha sido una fuente de ayuda indispensable para llevar a cabo este proceso.

Como paso previo a la instalación se resolvieron las dependencias necesarias (Java, Python, Python-Pip, Python-Wheel y Bazel) y se incrementó el tamaño de swap disponible al mínimo requerido (8GB). Esto se llevó a cabo por medio de un *swapfile* que se desmontó y eliminó al finalizar el proceso de compilación. Java se usa para instalar Bazel, compilador utilizado por TensorFlow, mientras que pip se utiliza para instalar los paquetes .whl (ejecutable de TensorFlow) generados por Bazel.

A continuación, se obtiene la última versión de TensorFlow del repositorio oficial y se lleva a cabo el proceso de compilación. Bazel muestra una serie de variables para complementar la compilación e instalar solo los paquetes necesarios. Para llevar a cabo nuestra prueba de concepto, generaremos 3 compilaciones de TensorFlow distintas (archivos .whl), cada una destinada a la ejecución en un hardware diferente:

- Gpu.whl: preparado especialmente para ser ejecutado en la GPU NVIDIA Pascal de la Jetson. Compatibilidad CUDA, incluye librería cuDNN y entorno de ejecución TensorRT.
- Arm.whl: Compilación para isa ARMv8, para poder ejecutarse en los *cores* Cortex-A57. Incluye extensiones vectoriales (NEON)
- X86.whl: compilación x86, para poder ser ejecutado en un procesador Intel Xeon X5650. Incluye extensiones vectoriales (AVX).

Para evaluar el desempeño de la Jetson Tx2 frente a otro *hardware* se ha optado por una implementación de red neuronal profunda de tipo convolucional (CNN) entrenada con el dataset CIFAR-10 [31] para clasificar imágenes RGB de 32x32 píxeles en 10 diferentes categorías: avión, automóvil, pájaro, gato, ciervo, perro, rana, caballo, barco, camión**.** Trabajando con una red neuronal previamente entrenada, se ha analizado a qué velocidad es capaz dicha red de clasificar las imágenes en cada una de las plataformas sobre las que se ha

[4] https://www.tensorflow.org

ejecutado TensorFlow. Los resultados obtenidos se muestran en la Figura 4, donde se representa el número de imágenes por segundo que es capaz de procesar cada plataforma de ejecución.

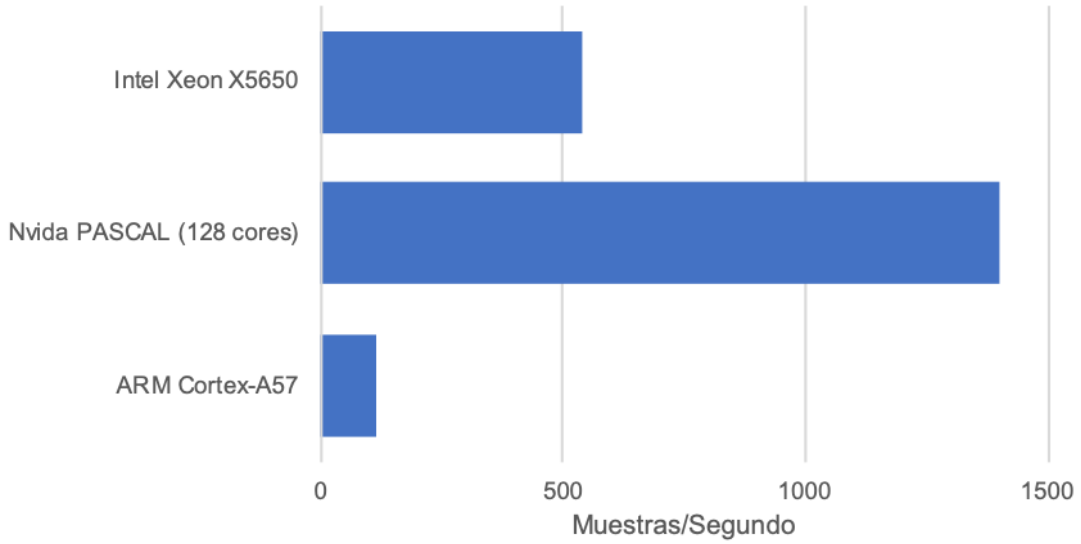


Figura 4. Comparativa de rendimiento. Red convolucional sobre diferentes plataformas.

Como se observa en la gráfica, el procesador ARM es el sistema que peor rendimiento ha obtenido. Se trata de un procesador orientado a un consumo de potencia moderado, y este tipo de configuración limita su rendimiento. En el otro extremo, la GPU de la placa de desarrollo (Nvidia Pascal) obtiene el mayor rendimiento en el *benchmark*. A pesar de que la GPU de la plataforma Jetson solamente cuenta con 2 “Streaming Multiprocessor” (en comparación a los 60 existentes en una GPU Tesla P100), es capaz de mejorar los resultados del procesador de la misma placa (ARM Cortex A57) así como los obtenidos por un procesador Intel Xeon X5650.

3.2 Configuración Avanzada

Como se ha mencionado previamente en la sección 2.1, la placa Jetson TX2 cuenta con dos tipos de CPU distintas: 4 *cores* CortexA57 y 2 *cores* Denver 2. Solamente los *cores* CortexA57 implementan soporte para kvm, por lo que una de las primeras tareas de configuración ha consistido en garantizar la ejecución en este tipo de *cores*. El sistema operativo Linux proporciona diferentes mecanismos para controlar la ubicación de la ejecución de un programa, como son el comando *taskset* o la limitación del *scheduling* de procesos a un subconjunto de *cores* a través de systemd. Nvidia también proporciona herramientas propias para este tipo de tareas, que permiten configurar diferentes modos de funcionamiento de la placa. Dado que esta herramienta ofrece mayores garantías de funcionamiento (los *cores* no utilizados se apagan), ha sido la empleada. A través del comando “nvpmodel”, se ajusta el rendimiento de la placa, activando o desactivando diferentes grupos de *cores*.

La Tabla 4muestra los diferentes modos de funcionamiento, de los cuales se ha escogido el Modo “Max-P ARM” para trabajar en el resto del TFG.

Tabla 4. Modos de Funcionamiento de la plataforma Jetson TX2.

Mode	Mode Name	Denver 2	Freq.	ARM A57	Freq.	GPU Freq.
0	Max-N	2	2.0GHz	4	2.0GHz	1.30GHz
1	Max-Q	0		4	1.2GHz	0.85GHz
2	Max-P Core-All	2	1.4GHz	4	1.4GHz	1.12GHz
3	Max-P ARM	0		4	2.0GHz	1.12GHz
4	Max-P Denver	2	2.0GHz	0		1.12GHz

4 Gem5 Sobre ARM

En la sección anterior se ha visto cómo poner en marcha la placa NVidia Jetson TX2. Además, se ha comparado su rendimiento frente a otros dispositivos, lo cual ha servido para familiarizarse con la placa y con el proceso de instalación de nuevas aplicaciones y funcionalidades sobre esta. A partir de los resultados obtenidos, se puede observar que la placa ofrece el rendimiento justo, pero suficiente como para llevar a cabo una simulación acelerada, en este caso de Gem5.

A lo largo de este capítulo, se mostrará cómo llevar a cabo el despliegue del software de simulación Gem5 sobre la placa para que pueda llevar a cabo simulaciones con soporte KVM. En primer lugar, se lleva a cabo una modificación en el *kernel* de la placa para habilitar KVM que, por defecto en el sistema proporcionado por el fabricante, se encuentra deshabilitado y así poder hacer uso de esta herramienta en la simulación. Posteriormente, se pasa a compilar Gem5 sobre la placa, haciendo los cambios necesarios para resolver todas las dependencias que puedan aparecer, así como las modificaciones pertinentes en el simulador para que pueda hacer uso de KVM sobre una arquitectura ARM, previamente habilitado en la placa. Finalmente se hacen unas pruebas para validar el correcto funcionamiento del simulador haciendo uso de esta característica, con el fin de solventar posibles errores que aparezcan en tiempo de ejecución.

4.1 Ajustes previos

Una parte crucial de este proyecto es habilitar la posibilidad de usar las extensiones hardware para acelerar la simulación en la plataforma Jetson TX2. Los *cores* ARM alojados en la CPU de la placa cuentan con el repertorio de instrucciones ARMv8.0-A, el cual implementa extensiones de virtualización **¡Error! No se encuentra el origen de la referencia.**. En la última arquitectura ARMv7 se incorpora el nivel de privilegios EL2 para la ejecución del hipervisor y en las primeras arquitecturas ARMv8 se añaden las extensiones de virtualización hardware, eliminando la necesidad de para-virtualización y permitiendo la ejecución de máquinas virtuales con sistemas operativos sin modificar **¡Error! No se encuentra el origen de la referencia.**.

La virtualización en los sistemas basados en ARMv8 está organizada en base a niveles de privilegios, como se muestra en la Figura 5. En EL2 se ejecuta el hipervisor que controla la ejecución de las máquinas virtuales. Los niveles EL1 (*kernel* del sistema operativo, código privilegiado) y EL0 (código no privilegiado) se dejan para las instancias de la máquina virtual. La traducción de direcciones se realiza en dos etapas, siendo que el hipervisor tiene su propia tabla de páginas para las traducciones propias y las traducciones de las direcciones “físicas” de las máquinas virtuales. El TLB incorpora además un identificador de

máquina virtual (VMID), lo que elimina la necesidad de vaciar el TLB al cambiar de máquina. Aun cuando ARM ha seguido añadiendo mejoras en las subsiguientes arquitecturas ARMv8 (ARMv8.2 y ARMv8.3), algunas de las cuales permiten la ejecución del *kernel* del anfitrión en EL2, reduciendo los cambios de contexto en virtualizaciones como KVM, el procesador con el que trabajamos, Cortex A57, no incluye estas características, como muestra la Figura 5.

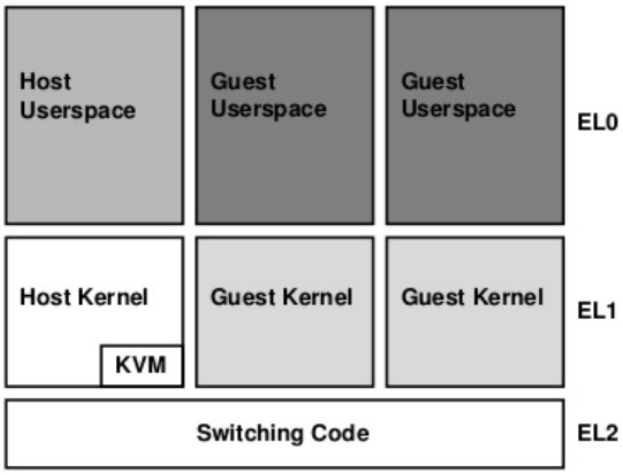


Figura 5. Jerarquía de privilegios en ARMv8.0-A

Gem5 está preparado para hacer uso de virtualización con el fin de acelerar partes del proceso de simulación a través de una máquina virtual basada en KVM, una tecnología de virtualización *open source* integrada en el propio *kernel* desde Linux 2.6.20 y que requiere de extensiones de virtualización hardware. KVM convierte a Linux en un hipervisor de tipo 1 (*Hosted*). Cada máquina virtual se implementa como un proceso regular de Linux, programada por el planificador estándar de Linux con hardware virtual dedicado como tarjeta de red, adaptador de gráficos, CPU, memoria y discos. La Figura 6 muestra un esquema de funcionamiento de KVM.

Por defecto, el *kernel* que se instala en la placa Nvidia Jetson Tx2 no posee soporte para KVM, sin embargo, el procesador que ésta utiliza (ARM Cortex-A57) hace uso del repertorio de instrucciones ARMv8, que es capaz de usar las extensiones de virtualización. Habilitar KVM en la placa, por tanto, conlleva compilar de nuevo el *kernel* del sistema para que incluya el módulo de virtualización KVM.

Otro inconveniente importante para hacer uso de esta funcionalidad en la placa es que, por defecto, el *Device Tree* del dispositivo no ha sido portado correctamente desde la versión usada en la Nvidia Jetson Tx1. El *Device Tree* es una estructura de datos que describe los componentes hardware de un equipo en particular de modo que el *kernel* del sistema operativo pueda utilizar y gestionar dichos componentes, incluyendo la CPU o CPUs, la memoria, los buses y los periféricos. Los equipos con arquitectura x86 generalmente no utilizan árboles de dispositivos, sino que se basan en varios protocolos de

configuración automática para descubrir el hardware. Sin embargo, en ARM los árboles de dispositivos han sido obligatorios para todas las nuevas SoC desde 2012 [32].

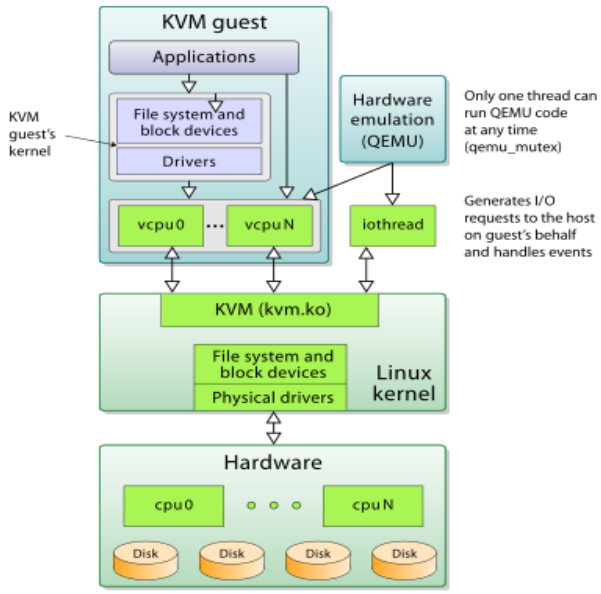


Figura 6. Esquema de funcionamiento de KVM.

Aun cuando el núcleo de Linux soporta la descripción del árbol de dispositivos estático junto con el sistema operativo, los sistemas que utilizan *Device Tree* suelen hacer uso de una descripción almacenada en una ROM. El propósito es mover una parte significativa de la descripción del hardware fuera del binario del *kernel*, y dentro de un DTB (*Device Tree Blob*), el cual es entregado al *kernel* por el cargador.

Originalmente las distribuciones de Linux para ARM incluían un gestor de arranque personalizado para placas específicas. Sin embargo, esto supone un problema para los creadores de estas distribuciones debido a que una parte del sistema operativo debe ser compilada específicamente para cada variante de placa, o actualizada para soportar nuevas placas. Algunos SoCs modernos, como es el caso de la placa de la que se hace uso, tienen un gestor de arranque proporcionado por el vendedor con un árbol de dispositivos en un chip separado del sistema operativo.

El hecho de que no se haya ajustado correctamente la especificación del *Device Tree* al ser portado de la primera iteración de la placa (Jetson TX1) hasta la que se está usando en este trabajo (Jetson TX2) implica, entre otras cosas, que los registros de virtualización a los que apunta el *Device Tree* son erróneos, dado que esta placa cuenta con un procesador diferente al de la primera iteración. Es necesario aplicar un parche [33] sobre los archivos del kernel y del DTB para solucionar este problema y hacer uso de las extensiones de virtualización ofrecidas por el procesador.

El cambio dentro de los archivos del DTS (*Device Tree Source*) que se muestra en el recuadro inferior es el que añade valores nuevos al controlador de

interrupciones, que son dos direcciones de registros claves para que el *kernel* pueda iniciar en modo hipervisor. Una vez compilado el nuevo *Device Tree*, se pasa a la placa para el nuevo arranque. Para ello, el SDK que se ofrece con la placa (JetPack) cuenta con una herramienta para poder escribir la memoria que contiene la información del DTB con los nuevos archivos que se le indiquen desde el equipo host. Con los cambios aplicados corrigiendo el *Device Tree*, junto con la recompilación del *kernel*, se podrá hacer uso de virtualización KVM en la plataforma.

```
$>diff -burp tegra186-soc-base-orig.dsti tegra186-soc-base.dsti
--- tegra-soc-base-orig.dsti 2017-07-20 09:41:09.000000000 +0200
+++ tegra-soc-base.dsti 2018-01-14 14:30:57.289320962 +0100
@@ -1122,8 +1122,11 @@
      compatible = "arm,cortex-a15-gic";
      #interrupt-cells = <3>;
      interrupt-controller;
-     reg = <0x0 0x3881000 0x0 0x00001000
-           0x0 0x3882000 0x0 0x00002000>;
+     reg = <0x0 0x3881000 0x0 0x00001000>, /*Z*/
+          <0x0 0x3882000 0x0 0x00002000>, /*Z*/
+          <0x0 0x3884000 0x0 0x00002000>, /*Z*/
+          <0x0 0x3886000 0x0 0x00002000>; /*Z*/
+     interrupts = <GIC_PPI 9 (GIC_CPU_MASK_SIMPLE(4) |
+     IRQ_TYPE_LEVEL_HIGH)>; /*Z*/
      status = "disabled";
```

4.2 Instalación y compilación de Gem5

Se recomienda obtener una copia de Gem5 utilizando Git. La última versión de código fuente de Gem5 está disponible a través del repositorio de googlesource[5] mencionado anteriormente. Para este trabajo, se ha optado por usar una rama específica del repositorio que abarca toda la parte de ARM [34] y que se encuentra en evolución constante gracias a desarrolladores de diferentes compañías (ARM,Broadcom,Nvidia,AMD…), así como usuarios habituales de la plataforma que trabajan sobre esta arquitectura.

Una vez finalizada la descarga del código fuente se pasará a compilar Gem5, en este caso, sobre la plataforma ARM en la que trabajamos. El sistema de compilación de Gem5 está basado en SCons [35],un sistema de código abierto implementado en Python [36]. El archivo SCons principal se llama SConstruct y se encuentra en la raíz del árbol de fuentes. Los archivos SCons adicionales se denominan SConscript y se encuentran en todo el árbol, normalmente cerca de los archivos a los que están asociados.

En Gem5, los objetivos de SCons son de la forma <build dir>/<configuration>/<target>. En este caso la instrucción correspondiente a ejecutar sería:

[5] https://gem5.googlesource.com/

```
$>scons build/ARM/gem5.opt
```

El parámetro <build dir> del destino es una ruta de directorio que termina en "build". Típicamente esto es simplemente "compilar" por sí mismo (como en el ejemplo), pero puede especificar un directorio llamado "build" ubicado en otro lugar. Se supone que todos los objetivos bajo el mismo directorio de construcción se compilan para la misma plataforma host y comparten los mismos ajustes de variables globales. Estos ajustes se almacenan en el archivo build/variables.global.

La <configuración> ("ARM" en el ejemplo anterior) selecciona un conjunto de opciones de construcción en tiempo de compilación que controlan la funcionalidad del simulador, como el ISA utilizado, qué modelos de CPU se incluyen, qué protocolo de coherencia se debe utilizar, etc. Estas variables de configuración se almacenan en archivos separados en el directorio build/variables. Esta ubicación permite limpiar una compilación eliminando el directorio de configuración completamente sin perder los ajustes de las variables.

El objetivo de construcción <target> ("gem5.opt" en el ejemplo anterior) es típicamente el nombre del binario Gem5 a construir, que se corresponde con los vistos en el apartado 2.2.

Una vez compilada la herramienta bajo las indicaciones que se han descrito se pasará a hacer una pequeña prueba para ver el funcionamiento básico de un sistema completo.

4.3 Primeras pruebas con Gem5

Una vez preparado Gem5 para simular ARM haciendo uso de las indicaciones del apartado anterior, se ha optado por tener una primera toma de contacto con la herramienta por medio de la ejecución de un “Hello Word”, el cual se proporciona a través de un archivo junto con los fuentes de la herramienta. La ejecución de este pequeño sistema de prueba se lleva a cabo en el modo Syscall Emulation (ver sección 2.2) por lo que, una vez ejecutado y en el caso de que todo se haya configurado correctamente, se mostrará por la misma consola el mensaje “Hello World” con la consiguiente finalización del programa. Este es el comando ejecutado:

```
$>./build/ARM/gem5.opt configs/example/se.py -c tests/test-
progs/hello/bin/ARM/linux/hello
```

Como se observa, en este modo hay que especificar el archivo binario a simular con el argumento -c. Este archivo binario puede ser enlazado estática o dinámicamente **¡Error! No se encuentra el origen de la referencia.**. También es posible especificar detalles sobre la construcción de la arquitectura, aunque para esta primera prueba se han tomado los valores por defecto.

A continuación, se pasa a una segunda prueba, un poco más compleja: Simular un sistema en modo Full System (FS) haciendo uso de las extensiones de

virtualización de la arquitectura del host gracias a KVM. Previo a la simulación hay que tener preparados los archivos necesarios para poder ejecutar Gem5 en modo FS**.** Si bien, es posible compilar un nuevo kernel, así como un DTB simple y construir una imagen específica, se ha optado, para las primeras pruebas, por obtener los archivos que ofrece la página oficial de Gem5 para emular un sistema completo bajo la arquitectura ARM y hacer una primera toma de contacto. Se analizarán los archivos necesarios para la ejecución a partir del siguiente comando usado para esta prueba:

```
$>./build/ARM/gem5.opt configs/example/se.py --cpu-
type=ArmV8kvmCPU --num-cpus=1 --disk <path imagen> --kernel
<path kernel> --machine-type=VExpress_GEM5_V1 --dtb
armv7_gem5_v1_1cpu.dtb
```

El primer parámetro corresponde al fichero de configuración, en este caso *fs.py*. Este script en Python es el que Gem5 usará para generar el sistema objetivo y automatiza la creación del sistema completo. Aunque es posible crear una versión propia del fichero, se ha usado uno por defecto que proviene de la distribución de Gem5, el cual es suficientemente flexible como para adaptarse al experimento que se va a realizar.

Al trabajar en modo *Full System*, es necesario proporcionar al simulador una imagen de disco y un *kerne*l para el arranque del sistema. La imagen proporcionada a través del parámetro **disk** se corresponde con el sistema de ficheros objetivo sobre el cual se lanzará la simulación y deberá tener preparadas las aplicaciones que se van a ejecutar. Para la realización de este experimento, la imagen ha sido creada tomando como base una distribución de Linux *Debian Jessie* (versión 8), añadiendo los binarios de las aplicaciones y las librerías que éstas necesitan, además del *scripting* necesario para configurar y lanzar la ejecución de las aplicaciones. Adicionalmente, por ser una arquitectura ARM, es necesario aportar un DTB para la creación del *Device Tree*. Aunque es posible dejar que Gem5 genere un DTB automáticamente según la configuración de nuestro sistema, se proporciona un fichero DTB previamente generado por ser lo recomendado.

Por último, y de forma significativa en este proyecto, podemos especificar qué modelo de CPU será usado en esta ejecución, dentro de los que se han expuesto en el apartado 2.2. Para este primer ejemplo, y para probar la funcionalidad implementada, se hace uso del modelo de CPU KVM.

Con la ejecución anterior se lanza una simulación en sistema completo, en la que la salida no se vuelca por salida estándar (como ocurría en modo SE) sino que hace uso de su propio terminal simulado. En consecuencia, Gem5 arranca un servidor de *ssh* y/o *telnet* para que el usuario pueda conectarse e interactuar con el sistema simulado desde otra terminal, el cual se pone en escucha en el puerto 3456. Además de esto, se pone a disposición del usuario una aplicación ya preparada por los desarrolladores (**m5term**) que permite al usuario usar una terminal específica para los sistemas simulados en Gem5 y que solventa posibles errores ocasionados, por ejemplo, por el uso de telnet (líneas de

comandos duplicadas, bloqueo en determinados procesos…). Desde el terminal, es posible interaccionar con la máquina simulada, que corre un sistema operativo comercial. Al ejecutar con tipo de cpu KVM, es posible realizar operaciones sobre la máquina simulada de forma fluida.

Se realiza finalmente una última prueba importante de cara al paso final del trabajo: la creación de *checkpoints* de cargas de trabajo en una arquitectura ARM mediante KVM. Para ello, tenemos que poder ejecutar en el código de una aplicación las instrucciones **m5ops** en una arquitectura ARM.

Las m5ops son unas instrucciones especiales que activan una funcionalidad específica de la simulación al ser detectadas por el simulador. De entre las opciones que ofrecen las **m5ops** se ha puesto especial hincapié en las siguientes debido a que son importantes en el desarrollo de este trabajo:

- **exit**[delay]: Detiene la simulación con n nanosegundos de retardo.
- **m5_work_begin**: Permite definir el comienzo de una zona de trabajo, que en nuestro caso usaremos para delimitar las regiones de interés dentro de la simulación.
- **m5_work_end**: Complementario a la instrucción anterior, permite definir el fin de la región de interés. Con estos dos comandos se evita simular N instrucciones y en su lugar simular M trabajos de contenido conocido.
- **readfile**: lee el archivo especificado en un parámetro de lanzamiento, con formato de script que se ejecutará al arrancar el sistema. Con esta m5op es posible diferenciar distintas ejecuciones haciendo uso de la misma imagen y *kernel.*

Para esta prueba se ha optado por crear un simple programa en C que se encargue de parar la simulación cuando es ejecutado. Para poner en marcha esta funcionalidad, las m5ops se deben compilar previamente. Gem5 promociona los ficheros necesarios (librerías, ficheros fuente, ficheros de cabecera…) para poder compilar un programa que hace uso de las instrucciones m5_ops. Estos se incluyen con su respectivo *makefile* para cada arquitectura.

```
#include <stdio.h>
#include <gem5/m5ops.h>

void map_m5_mem();

int main(void)
{
     printf("Mapping m5 mem\n");
     map_m5_mem();
     printf("hello world\n");
     printf("Closing simulation\n");
     m5_exit(0);
     printf("Simulation did not close correctly\n");
     return(0);
```

```
}
```

En su formato habitual, las m5ops se implementan como una instrucción que no existe en el procesador. Si el procesador que la ejecuta es simulado, cuando intente decodificarla, se dará cuenta de que la instrucción en cuestión no pertenece al ISA y que es una *m5_op*. Cuando se ejecuta en modo KVM la máquina real se encarga de ejecutar la instrucción, lo que origina una excepción que no puede recoger el simulador y por lo tanto la simulación no puede continuar y termina en error. La alternativa consiste en usar una región de la MMIO para mapear las m5ops. Como la memoria del KVM se mapea en la memoria de la máquina simulada, la operación a memoria se ejecuta, y las MMIO son capturadas por Gem5, que se da cuenta de que no es una dirección de ningún dispositivo, si no que esa dirección de memoria que se corresponde con las m5ops. Es necesario modificar la compilación de las m5ops para que se conviertan en instrucciones de memoria, siendo necesario especificar la región de memoria correspondiente dentro de la arquitectura (en nuestro caso *aarch64*). Para ello, se modifica la siguiente línea del *makefile*:

```
CFLAGS=-O2 -I $(JDK_PATH)/include/ -I $(JDK_PATH)/include/linux
\ -I$(PWD)/../../include -march=armv8-a -DM5OP_ADDR=0x10010000
```

El último parámetro (DM5OP_ADDR) que aparece es la dirección de memoria de la MMIO, que además es específica para la arquitectura ARM, ya que en x86 se usa una dirección distinta.

Finalmente, para probar si la funcionalidad es correcta hay que incluir el ejecutable generado al compilar el programa dentro de la imagen que usará el simulador. Una vez lanzado el simulador y con el sistema iniciado se ha entrado a la máquina simulada de manera interactiva, ejecutando el programa desde la propia consola, terminando la simulación de forma correcta cuando el programa llega a la instrucción “**m5_exit**”.

5 SPEC CPU2017 sobre Gem5

Como resultado final del proyecto, debe quedar un sistema de bajo coste capaz de crear cargas de trabajo de arquitectura ARM para el simulador Gem5, haciendo uso de la simulación acelerada con KVM.

A lo largo de este apartado, describiremos el proceso de creación de cargas de trabajo mediante *checkpoints*, preparando las aplicaciones y haciendo uso de la simulación acelerada para llevar al simulador a la región de interés.

Se realizará una prueba de funcionamiento que hace uso de todo lo desarrollado durante el proyecto para generar en un tiempo razonable, cargas de trabajo de la suite SPEC CPU2017, que bajo otras condiciones habrían llevado días. Más tarde, los *checkpoints* generados serán cargados en el simulador para poder realizar experimentos con simulación detallada, y se realizará una prueba de concepto calculando el *working set* de las aplicaciones para las que se ha creado el *checkpoint*.

5.1 Selección de aplicaciones y profiling (ROI)

La metodología de creación de cargas de trabajo mediante *checkpoints* se basa en arrancar la máquina simulada y ejecutar la aplicación hasta llegar a la región que queremos medir (Región de interés o ROI). Generalmente este proceso se hace en un modo de baja precisión del simulador, pues es importante hacerlo rápido, y la exactitud del simulador es irrelevante. En nuestro caso haremos uso de KVM, que permite una ejecución a velocidad casi nativa. Al llegar a la región de interés, la aplicación (previa modificación de su código) debe hacer una llamada a una de las instrucciones **m5ops**, en nuestro caso **work_begin**(), que usaremos para generar los *checkpoints*.

Las cargas de trabajo que se proponen serán creadas de aplicaciones procedentes del paquete de *benchmarks* SPEC CPU2017. Las aplicaciones elegidas para esta prueba de concepto de creación de *checkpoints* son: NAMD, LBM, X264 y NAB. Se ha elegido estas aplicaciones por estar escritas en C, lo que facilita la modificación de su código.

El primer paso que llevar a cabo para poder generar estos *checkpoints* es identificar la región de interés de cada aplicación que se va a medir mediante *profiling*. Esta tarea ha sido llevada a cabo por el grupo de investigación *Computer Engineering* de la Universidad de Cantabria y ha consistido en buscar en cada aplicación un lazo (en el caso de que exista) que comprenda, al menos, el 90% del tiempo de ejecución de la aplicación. El punto de parada que creará el *checkpoint* se encontrará al principio de este bucle, normalmente tras haber completado alguna vuelta, y servirá para, más tarde, simular el momento en el que la aplicación se encuentra en su región de interés.

Una vez identificada la región de interés hay que añadir la instrucción (en este caso instrucciones) que le dirá al simulador que pare la ejecución y guarde el estado de la máquina para, posteriormente, generar el *checkpoint*. Esto se hará por medio de las instrucciones m5ops que se han visto en apartados anteriores. Para ello, se insertarán estas instrucciones dentro del código de cada aplicación de SPEC y posteriormente se compilarán de nuevo.

Como se menciona en el apartado 4.3, para poder ejecutar una m5op dentro de un programa hay que tener en cuenta varios factores. En primer lugar, hay que añadir la referencia, en la propia aplicación, a la librería que proporciona Gem5. Hay que tener en cuenta que habrá que modificar también las directivas de compilación para que enlace la librería correctamente. Esta tarea es sencilla ya que SPEC tiene un archivo makefile, que agrupa todas las directivas de compilación. En segundo lugar, ya dentro del código de cada aplicación, hay que añadir la instrucción `map_m5_mem()` que se encargará de mapear en la memoria del programa la región de las m5ops, necesario para poder ejecutarlas. Esta instrucción debe de ser colocada al principio del programa ya que se tiene que ejecutar antes de cualquier m5op. Posteriormente, hay que introducir en el código las instrucciones `m5_work_begin()` y `m5_work_end()`. La primera de estas deberá ir al principio del bucle de la región de interés y la segunda al final. Esto permitirá, a la hora de lanzar la simulación, indicar el número de trabajos que queremos ejecutar hasta la creación del *chekpoint*. Esto evita calcular el número de instrucciones que queremos que se ejecuten, haciendo el proceso más consistente. Finalmente, será necesario crear una Imagen que pueda ser usada por el simulador Gem5 y que contenga esta versión de SPEC CPU 2017 modificada para la creación de *checkpoints* en las aplicaciones objetivo. El grupo de investigación *Computer Engineering* de la Universidad de Cantabria cuenta con diversos scripts para la creación de imágenes que además, la prepara para su uso con Gem5. Una de estas características es la ejecución de scripts que se pasan al simulador vía fichero de texto y que serán ejecutados nada más ser arrancado el sistema. Esto permitirá automatizar el proceso de creación de *checkpoints* para que, haciendo uso de la misma imagen, se puedan crear cargas de trabajo para todas las aplicaciones que contiene. Una vez creada la imagen, habrá que lanzar Gem5 para ya crear los *checkpoints*. Como se ha mencionado a lo largo del documento, se va a hacer uso de KVM para conseguir acelerar la simulación. Para que esto pueda ser llevado a cabo, es importante tener en cuenta que la arquitectura sobre la que se va a realizar la simulación tiene que ser la misma que la arquitectura simulada. Esto es debido a que las instrucciones a simular se ejecutarán directamente sobre la maquina host. Dado que vamos a generar cargas de trabajo para procesadores ARM, es por eso que se ha usado como máquina host para la simulación la plataforma Jetson Tx2, modificada a lo largo de este trabajo.

En este punto hay que destacar un problema encontrado en la creación de *checkpoints* para ARM en gem5, haciendo uso de la aceleración KVM. Este

problema se produce, aparentemente, porque el simulador no puede generar correctamente el *Device Tree* cuando se hace uso de la aceleración con KVM, por lo que hay que buscar una solución alternativa que no aumente el tiempo de simulación considerablemente. Para ello, se ha optado por arrancar la máquina en modo *atomic*, garantizando la creación del *Device Tree*, y cambiando de modo de CPU a KVM tan pronto como la máquina ha arrancado. Para ello, se ha ampliado la funcionalidad de cambio de CPU en vuelo que incluye Gem5, para que permita cambios de KVM. Adicionalmente, se ha añadido una instrucción **m5_exit** al arranque de la máquina para que actúe como activador del cambio de CPU.

Así pues, el proceso de creación de *checkpoints* queda de la siguiente forma. La simulación del sistema arranca en modo *atomic* para luego, una vez completado el arranque y gracias al script que se pasa como parámetro al simulador, hacer una llamada a la m5op m5_exit. Al encontrar esta m5op, el simulador pasa a ejecutar con la CPU modo KVM haciendo uso de la virtualización con aceleración hardware. El procedimiento propuesto se muestra en la Figura 7.

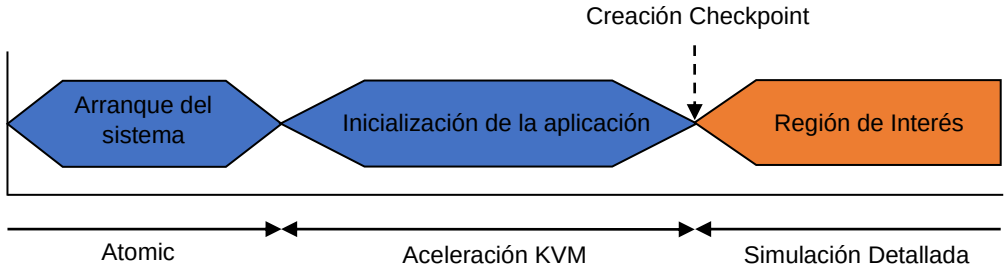


Figura 7. Proceso de generación de checkpoints y posterior simulación.

Finalmente se ha creado un script que corrige los *checkpoints* una vez creados, ya que se han usado dos tipos de “CPUs” en la creación, y Gem5 no interpreta bien los datos del mismo al hacer el proceso de recuperación del *checkpoint*.

Una vez solventados todos estos problemas se podrá automatizar la creación de *checkpoints* para las aplicaciones seleccionadas mediante un sencillo script bash.

5.2 Prueba de Concepto

Una vez creados los *checkpoints* de las diferentes cargas de trabajo que han sido elegidas para la prueba de concepto, se realiza una medida del *working set* de las aplicaciones haciendo uso de los mismos con el fin de comprobar que funcionan correctamente.

Para la realización de los *checkpoints* era necesario hacer uso de la plataforma Jetson TX2, debido a la necesidad de que la arquitectura de host y máquina simulada fuesen la misma. En el proceso de simulación posterior para la medición de *working set* esta restricción no aplica, pues el procesador que ejecuta las instrucciones será completamente simulado. En este caso se hará

uso de una máquina más potente para la prueba, el supercomputador Tres Mares ubicado en la facultad de ciencias de la universidad de Cantabria.

Para la prueba de concepto se ha optado por el modelo O3 (*detailed*) que ofrece este simulador, que simula un procesador superescalar con ejecución fuera de orden. La medida de *working set* se va a obtener midiendo el *miss_rate* para diferentes tamaños de cache de primer nivel. Por lo general, se observa cómo el *miss_rate* decrece de forma significativa cuando una gran parte o la totalidad del *woking set* cabe dentro de la memoria L1 de datos. Como se va a ir variando el tamaño de la memoria L1, se pueden identificar en qué puntos decrece de forma significativa este *miss_rate* y, en función del tamaño que tiene la memoria en ese momento, identificar el *working set* de la aplicación.

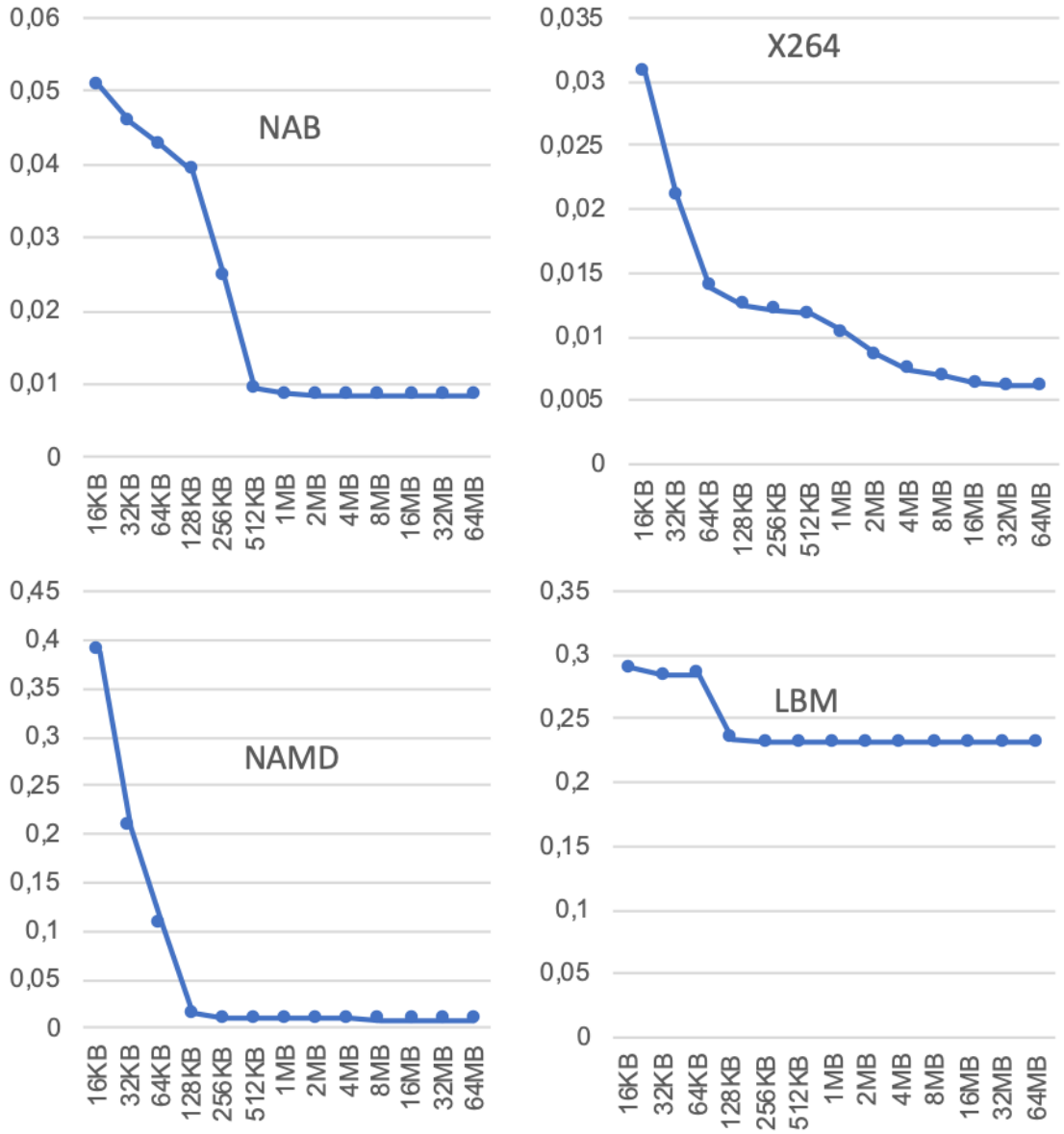


Figura 8. RESULTADOS SPEC.

Para llevar a cabo la prueba, se realizará un barrido de la cache L1 del procesador con distintos tamaños, partiendo de 64kB y multiplicando este valor por dos hasta alcanzar 64 MB. Esto llevará a un total de 44 simulaciones, ya que, habría 11 valores posibles para caché de datos en un total de 4 aplicaciones. Una vez finalizada cada simulación se guardarán los resultados obtenidos de estas y serán evaluados. Ya que se deben de llevar a cabo una gran cantidad de simulaciones y, sobre todo, para conseguir capacidad de repetición, así como asegurar la escalabilidad de las pruebas en caso de que se quieran evaluar más aplicaciones, se ha optado por la creación de un script bastante sencillo que se encargará de ejecutar la misma instrucción adaptada para cada simulación. Esta instrucción hará que Gem5 simule un equipo que cuenta, (además de con una memoria de datos L1 de 64KB a 64MB) con 4GB de memoria RAM, un modelo de CPU O3 (*detailed*), una memoria L2 de 128MB (suficientemente grande para que no haya problemas con los tamaños de L1) y una memoria L1 de instrucciones de 64KB. En cada simulación se buscará ejecutar 1000000000 instrucciones. Una vez terminado todo el proceso de simulación se han obtenido los siguientes resultados que se presentan en la Figura 8 a modo de comparativa.

A la vista de los resultados obtenidos para cada aplicación, se puede determinar qué tamaño abarca el *working set* de cada una. Se puede observar como el *miss-rate* cae entre los 64KB y 512KB, siendo LBM la aplicación que mantiene un *miss-rate* más elevado, por lo que probablemente su *working set* sea superior a los 64MB tomados como límite. Por otro lado, tanto NAB como NAMD terminan con una tasa de aciertos en L1 por encima del 99% y un comportamiento plano, por lo que podemos suponer que el *working set* de estas aplicaciones está en el punto de caída detectado (512KB y 128KB respectivamente).

Dado que la memoria L1 de datos de un procesador actual ronda entre los 32-64Kb, y su L2 en torno a 256KB-1MB, podemos ver que para la mayoría de las aplicaciones medidas, las caches privadas de los procesadores son capaces de absorber una parte importante del *working set*.

6 Conclusiones

Durante el desarrollo de este proyecto se han llevado a cabo los pasos necesarios para preparar un sistema de bajo coste capaz de crear cargas de trabajo de arquitectura ARM para el simulador de sistema completo Gem5. Este simulador cuenta con un mecanismo de aceleración de la simulación mediante la virtualización KVM, idóneo para la creación de cargas de trabajo por ser capaz de ejecutar a velocidad casi nativa, sin embargo, requiere que máquina simulada y host tengan la misma arquitectura.

Por ello, se ha comprobado la idoneidad de la plataforma NVIDIA Jetson TX2, con procesadores ARMv8, como servidor en el que crear las cargas. La plataforma ha sido obtenida a través de una beca de colaboración con NVIDIA. Para familiarizarse y evaluar el desempeño de la Jetson TX2, se ha preparado y ejecutado sobre la misma el *framework* de *machine learning* TensorFlow. Se ha confirmado que, para este tipo de aplicaciones, las GPU tienen un buen comportamiento. Por otro lado, el rendimiento de la CPU ARM ha resultado justo, pero suficiente para la labor que se pretende en este trabajo.

Se ha habilitado en esta plataforma la virtualización KVM, mediante modificación del *kernel* y el árbol de dispositivos. Finalmente se ha configurado el simulador Gem5 para el funcionamiento de su modo de cpu KVM sobre ARM, y se ha realizado una prueba de concepto mediante la generación de *checkpoints* y medidas sobre los mismos.

Gracias al desarrollo de este trabajo, se ha conseguido agilizar un proceso que podría tomar días enteros en llevarse a cabo y que ahora puede reducirse a unas pocas horas. Así mismo este proyecto podrá ayudar a futuros proyectos del equipo de investigación, así como otros grupos con problemas similares.

7 Bibliografía

[1] C. Mack, “The Multiple Lives of Moore’s Law”, IEEE Spectrum, Volume 52, Issue 4, April 2015.

[2] J.L. Hennessy, N.P. Jouppi, “Computer Technology and Architecture: An Evolving Interaction”, IEEE Computer, Volume 24, Issue 9, September 1991.

[3] A. Sandberg, N. Nikoleris, T.E. Carlson, E. Hagersten, S. Kaxiras, D. Black-Schaffer, “Full Speed Ahead: Detailes Architectural Simulation at Near-Native Speed”, 2015 IEEE International Sympsium on Workload Characterization.

[4] D.Wang,B.Ganesh,N.Tuaycharoen,K.Baynes,A.Jaleel,andB.Jacob, "DRAMsim: A Memory System Simulator," ACM SIGARCH Computer Architecture News, vol. 33, no. 4, pp. 100-107, November 2005.

[5] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," 1998, retrieved September 3, 2015 from http://pages.cs.wisc. edu/~markhill/DineroIV/.

[6] M. L. C. Cabeza, M. I. G. Clemente, and M. L. Rubio, "CacheSim: A Cache Simulator for Teaching Memory Hierarchy Behaviour," ACM Special Interest Group on Computer Science Education Bulletin, vol. 31, no. 3, p. 181, September 1999.

[7] A. Faravelon, N. Fournel, and F. PeÃÅtrot, "Fast and accurate branch predictor simulation," in Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition. EDA Consortium, 2015, pp. 317-320.

[8] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation," in Proceedings of ACM International Conference for High Performance Computing, Networking, Storage and Analysis. Seattle, WA, 2011, pp. 1-12.

[9] D. Sanchez, C.Kozyrakis, "ZSim:Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in Proceedings of the 40th Annual International Symposium on Computer Architecture, June 2013, pp. 475-486

[10] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. Automatically Characterizing Large Scale Program Behavior, In the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002), October 2002.

[11] R. Grisenthwaite. ARMv8 Technology Preview. http://www.arm.com/files/downloads/ARMv8_Architecture.pdf, 2011.

[12] N. Binkert, et. Al. “ The gem5 simulator”, ACM SIGARCH Computer Architecture News, Volume 39, Issue 2, May 2011.

[13] A. Akram, L. Sawalha, “A Survey of Computer Architecture Simulation Techniques and Tools”, IEEE Access.

[14] Championship Branch Prediction (CBP-5), 5th JILP on Computer Architecture Competitions.

[15] Adrián Colaso, “Conjugando Herramientas de Simulación con Aplicaciones y Tecnologías Emergentes en Arquitectura de Computadores”, Tesis Doctoral, Universidad de Cantabria.

[16] NVIDIA Jetson Architecture https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

[17] ARM Research Starter Kit: System Modeling Using Gem5 https://raw.githubusercontent.com/arm-university/arm-gem5-rsk/master/gem5_rsk.pdf

[18] The SPEC2017 benchmark suite https://www.spec.org/cpu2017/

[19] Linux LT4, Nvidia, https://developer.nvidia.com/embedded/linux-tegra

[20] Nvidia cuDNN, https://developer.nvidia.com/cudnn

[21] C. Bienia, “Benchmarking Modern Multiprocessors”, Ph. D. Thesis. Princeton University, January 2011.

[22] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, M. Yarrow, “The NAS Parallel Benchmarks 2.0”, NASA Report NAS-95-020

[23] J. Dongarra, “Performance of Various Computers Using Standard Linear Equations Software”, University of Tennessee, CS Technical Report CS-89-85

[24] R.C. Murphy, K.B. Wheeler, B.W. Barrett, J.A. Ang, “Introducing the Graph 500”, Cray User’s Group, 2010.

[25] M. Ferdman et. Al. “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware”, International Conference on Architectural Support for Programming Languages and Operating Systems, 2012

[26] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, “Benchmarking cloud serving systems with YCSB” ACM Symposium on Cloud Computing, 2010.

[27] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, “The HiBench benchmark suite: Characterization of the MapReduce-based data analysis”, International Conference on Data Engineering Workshops, 2010.

[28] J.A. Stratton et. Al. “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throghput Computing” IMPACT Technical Report, University of Illinois at Urbana-Champaign.

[29] S.Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, K. Skadron, “Rodinia: A Benchmark Suite for Heterogeneous Computing”, IEEE International Symposium on Workload Characterization, 2009

[30] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C. Tseng, D. Yeung, “BioBench: A Benchmark Suite of Bioinformatics Applications” IEEE International Symposium on Performance Analysis of Systems and Software, 2005

[31] Krizhevsky, A., Nair, V., & Hinton, G. (2014). The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, *55*.

[32] ARM SoC Linux Support checklist https://www.elinux.org/images/a/ad/Arm-soc-checklist.pdf

[33] "Error getting vgic maintenance irq from DT" for KVM at boot-up

https://devtalk.nvidia.com/default/topic/1006055/jetson-tx2/-quot-error-getting-vgic-maintenance-irq-from-dt-quot-for-kvm-at-boot-up/

[34] Gem5 Arm Development Repository https://gem5.googlesource.com/arm/gem5/

[35] Gem5: Build System http://gem5.org/Build_System

[36] SCons: A Software construction Tool https://www.scons.org/